Privacy

- In addition to "normal" security threats we also need to care about:
  - Correlation
  - Identification
  - Secondary use
  - Disclosure
  - Exclusion
  - Re-Identification

  → Outlined in RFC 6973

- The Request For Comment series (RFC) has been an important way to document Internet Protocol and practices since 1969 when RFC 1 was written

- Consider the example of an email. The header contains a "Received" field that can contain the mail user agent IP address. This is bad for privacy since we can track people by their IPs.

However this header was first implimented to prevent mail-loops from occuring

- There are various privacy processes:
  - Privacy by design
  - Data protection by design and default
  - Data protection impact assessments

- Take the Irish Covidtracker example:

The HSE produced a Google/Apple Exposure Notification (GAEN) covid tracking application

The result wasn't great from a privacy standpoint:
- It depended on Google Play Services (GAEN depends on this)
- It contained a Super-cookie. Most other GAEN trackers didn't have this.

- Privacy by Design (PbD) has 7 foundational principles:

  - Proactive not reactive • Preventative not remedial
  - Privacy as the default setting
  - Privacy embedded into design
  - Full functionality
  - End-to-end security
  - Visibility and transparency
  - Respect for user privacy

- How can designers improve/protect privacy
  - Behave as if your entire family will be all the users
  - Encrypt things in transit and storage
  - Short lived dynamic identifiers is better than long term static identifiers

- Many risks arise due to the existance of vulnerabilities in computer systems.

- All systems have vulnerabilities, the goal is not to remove all of them (oft times impractical) but to control their impact.

  This can be done by reducing their numbers or by isolating parts of the system eg. firewalling.

- Some common vulnerabilities and attacks.
  - Scripting User Agents
  - Buffer Overruns
  - XSS and Injection (eg. SQL Injection)
  - Insecure default settings (eg. setting password to "admin")

- One uncommon but interesting attack is Acoustic side-channel key extraction.

- Systems have users classified as "normal", "administrative" or "root".

Networks have nodes classified as "inside", "outside" and "trusted".

- Some possible bad actors could be a disgruntled employee, crackers, spies, criminals ect.

- Some exploits include:
  - Forcing legitimate users to reveal passwords
  - Social Engineering
  - Attacking the network
  - Installing malware.

- Attacks can be classified as "active" eg. Fabrication, modification, deletion of messages, or "passive" eg. network traffic analysis.

- Many variations of Risk Analysis exist but mostly they resemble:
  - Identifying Assets
  - Identifying Risks and Vulnerabilities
  - Considering probabilities of risks
  - Considering costs/losses as a result of these risks
  - Ranking risks
  - Develop mitigations for highest ranked risks
  - Keep iterating until satisfied or some other criteria is reached.

- Risk is a function of the cost of threats and their probability of occurance. Threats occur when a vulnerability is exploited.

↳ This summarises what Risk is. [Typical exam question]

Security Concepts

- Traditionally security has the CIA model:

Good to think about when asked to design something in the exam

C: Confidentiality

I: Integrity
↳ Preventing unauthorised modifications
Keeping data consistent

A: Authentication / Assurance / Availability

- From the old OSI world we've inherited two concepts:

Security mechanism allows security services to work.

- Security Service: Provides a security function to the system, based on the use of security mechanisms eg. confidentiality

- Security Mechanism: A technique or protocol ect which can be used to provide a service eg. encryption, access control

- Eg. the Access Control mechanism provides the Confidentiality service from the "CIA" model

The Integrity mechanism can be achieved by checksums or digital signatures.

The Availability service can be achieved by using a load balancing mechanism.

It is important to note that multiple mechanisms can impliment the same security service even though the mechanisms work in different ways.

- A lot of security concepts came from military concepts.

In the military, information is classified according to some national security classification scheme.

All users are "cleared" to see information up to a certain level. Users will have to prove their clearance to users files from a safe.

Additional compartments enforce the "need-to-know" principal.

- New files are labeled with the classification of the most sensitive / secret component.

The aggregation of a lot of unclassified information can create a "top-secret" file. Aggregation of data usually increases the sensitivity of the file.

- Sanitization downgrades the label of existing information.

- The "holy grail" in military security is "Multilevel Secure Systems". This seperates users with different clearance levels on the same computer system

In MSS, access control is mandatory (the policy is defined and enforced by the system) as opposed to discretionary

- There is also the idea of confinement and isolation.

Isolation is the idea where we isolate the running process with the minimal set of access rights needed to complete its tasks (principle of least privilage

Confinement is the idea that we need to control any form of communication channel. Eg. people could communicate by leaving files in persistent storage.

- Covert / Side-Channels are hidden means of communication that allows information to be leaked to 3rd parties

Some types of side channels could be timing channels where there is observable differences in system utilisation in time. Or acoustic channels where the computer makes a different noise when running different commands

- MSS also introduced the idea of security perimeters.

- An "attack surface" is all the possible ways in which attacks can be attempted.

A lot of ways to attack a program yields a very large attack surfaces and vice-versa.

- A way we can reduce the attack surface on a system is by limiting the number of useless services that are running.

We can lower the attack surface by having a firewall.

- In operating systems we try to increase security by isolating process and privileges.

- In OS's cross-boundary attacks usually have the most impact. Usually a combination of attacks is effective as well. Eg. Local Privilage Escalation is bad, but it becomes significantly more dangerous if the attacker can perform Remote Code Execution.

- The typical Enterprise Security Model usually has system administrators and network admins set up and manage users and applications

The Distributed Computing Environment (DCE) is an example of this. The DCE gave some framework for a client/sower protocol.

An example of this is DCE Remote Procedure Call (DCE/RPC)

- The model is as follows:

The server has a function $f(a, b)$ that a client wants to call.

Interface Definition Language (IDL) is used to generate client and server "stubs"; f_client(a, b) and f_server(a, b).

The runtime hides the transport from the client and the client call gets sent to the work.

The transport layer does some security stuff (via GSS-API in the case of DCE/RPC)

Kerberos v5 was also used in this

- There are some issues with this 1990's enterprise security model:

  It generally assumed a homogeneous network where everything was supplied by one vendor and was meant to work together. This became more false as time went on.

  Another assumption is that all users and applications were centrally managed. This is unviable for large businesses.

- Periodically people have also tried to develop an API to "hide" security. However this has been fairly unsuccessful for higher level security functions.

  However at lower levels such an API has been more successful eg. MS-CAPI.

- Cryptography makes interoperability harder. It is much easier to get systems to work together in "insecure" mode (ie. no cryptography)

This means that lots of people have turned off security features. The solution to this is to develop with all the security features from the start.

- As the internet became more widespread, the number of reachable hosts grew exponentially.

This highlighted issues with:

- Proxies
- Tunnelling (ie. SQL over HTTP)
- Having a findable end-point on many machines
- The browser security model.

- Customers needed to have some confidence that the system/network they are about to purchase is secure.

- How do we compare the security of different systems?
  They have different authentication mechanisms, different access control mechanisms and different cryptographic algorithms

- The US created the "Orange Book" that was a 7-point scale that defined certain standards for the security of the system. A1 was the most secure classification while D class was the least secure.

- The UK and some other European countries developed their own scale that was inspired by the US Orange Book.

ITSEC separated "Functionality Classes" (the security features) from the "Assurance Class" (ie how well tested is the system)

- Eventually this developed into the "Common Criteria".

This kept previous ideas such as separation of functionality and assurance classes; Target of Evaluation and the Security Target (the end goal of security that the customer wants)

It also included Protection Profiles that defined Security Requirements, Security Objectives. The Protection Profiles were also meant to be independent of implementation

- Some other aspects of assurance mean that we also need to consider how the system is installed, what software was installed and how the machine is administered.

Attacks on how the machine is installed and how the machine is administered is called "Supply chain" attacks

- Assurance doesn't mean that it does what we want. Assurance only shows that the product/system matches a specification.

The owner of the product pays for the evaluation. So even though the testers are government licenced the owner has reasons to be "nice" to the developer.

- There really is no interest in APIs for Network security. Normally the idea here is to secure a network from network based attacks.

- The original internet architecture was assumed to be end-to-end connectivity.

Hence this end-to-end security was the main consideration for those developing the internet

But they were very slow with developing IPSec (IP Security). Meanwhile NAT and firewalls arrived

- The End-to-End Argument still has current consequences. HTTP/2 and QUIC are protocols that attempt to have an E2E encryption as a mandatory mechanism in order to achieve confidentiality but primarily to prevent ossification.

- Network Address Translation is used to hide local IP addresses from the internet. This breaks many end-to-end assumptions.

  Carrier Grade NAT (CGNs) are NATs happening at the ISP level.

- NAT tiny introduced much more complexity into the network.

- IPSec can also be used to set up VPNs that allow a user to appear on a certain network even though they are elsewhere. Traffic is encrypted.

- Until the mid-90's most sites didn't bother filtering traffic. It soon became clear that exposing your internal topology to the open network was not a good idea.

Initially people put filtering rules in border routers
Eg. "no packets with a destination on my inner network are allowed out".

But IP spoofing attacks means that this wasn't sufficient so companies started making firewall protocols.

- Denial-of-Service (DoS) attacks usually aim to consume some type of resources to make a service unavailable.

This has been known as a vulnerability for many years and is often exploited.

DDoS - Distributed Denial-of-Service

- One type of DDoS attack is a TCP SYN flooding attack.

A poor implimentation of a TCP stack means that during a TCP handshake a SYN packet can cause the server to use a bit of memory to store some state of the handshake.

Using IP spoofing we can keep sending SYN packets to the server and consume all the memory on server.

- Identification and authentification is the process of establishing identity and verifying credentials.

- We can establish identity through 4 main methods:
    1) Providing something known
        eg. password, pin ect
    2) Providing something in your possession
        eg. smart card, java ring
    3) Providing something personal
        eg. fingerprint, retina scan
    4) Providing something you did
        eg. a signature

Usually we combine these for some added security
eg. a smart card + ring

- Some people want biometrics to replace passwords and it could be a good idea if:
    - Secure biometrics exist
    - If the reuse implications are acceptable
    - If the context supports the full life cycle

- Biometrics include:
  - Fingerprint scans
  - Retina scan
  - Facial recognition
  - How a person walks

- Fingerprint scanners usually work by capturing an "image" and then performing some feature extraction and then comparing it to some saved data.

- Some poor implementations are prone to "gummy" finger attacks. This is where a fingerprint from a surface could be transfered to a fake finger and then be used to break the security

- The #1 mitigation for all issues is to backup and backup often.

Passwords + Hashes

- The strength of passwords can be weakened through the reuse of passwords and the use of insecure passwords eg 123456 ect.
- The text-box entry style of entering passwords yield themselves to phishing attacks.
- Forcing "hard" password policies usually lead users to try and game the system.

  Forcing password rotations means that users will often create weak/insecure passwords to game the system.

- Passwords can also be great since they are human-memorable as opposed to cryptographic keys.

There isn't any need for any hardware or special software.

They can also link between diverse systems eg a VPN access via Microsoft Active Directory login.

- Often passwords can leak by man-in-the-middle attacks, or by phishing or via key-logging or by breaking into a database

Bruteforce attacks also exist. People can also purchase leaked databases.

- People can use password management systems to help them remember complex passwords.

- People can also avoid using passwords through ssh keys, USB tokens. Also using some form of 2FA.

  Also turn of ssh password authentication on new servers with an ssh source

- Sysadmins should avoid holding the password verifier database but should outsource it to some messgascaler. eg. those "Sign-in with Google Account" button you see.

  They should also ensure best practices for the password verifier database.

  They should also monitor their systems and include some form of banning if an SSH password was entered incorrectly a number of times. eg. fail2ban

- Sysadmins could also force 2FA

- Sysadmins should not enforce password "quality" requirements since this leads to people gaming the system.

Also they shouldn't follow "traditional" password policies eg. A password could be X letters long.

It is far better to have intrusion detection systems and ban IPs after a number of failed login attempts

- Password Verifiers work like this:

A user enters a password (p) and a username (u)

The system transmits p and u to some type of verification system

The verifying system uses "u" to get a password verifier value (pv) from a database (pvdb)

The system then checks if $f(p, pv) =$ "Ok" for some function $f()$ to decide whether or not to let the user in.

You need to be able to replace $f()$ easily without changing all the passwords so the pvdb stores some form of u, f, pv

- The pvdb usually has a lot of u, f, pv entries if people have the same password then they will have the same pv value.

So we use a salt which is a random string that gets stored and allows us to have different pv values. So now we have to store u, f, s, pv

- Longer salts makes dictionary attacks harder.

- One can use a cryptographic hash function to generate the pv value.

- Cryptographic hash functions are one way functions that accept an arbitrary input and produces a fixed size output.

- Hash functions should be:

  1) Collision Resistant: Finding $H(x) == H(y)$ should be hard

  2) Pre-Image Resistance: Given x, such that $H(y) = x$, it should be hard to find y.

  3) 2nd Pre-Image Resistant: Given x, finding y such that $H(x) = H(y)$ is hard.

- Finding collisions is much easier than pre-Image due to the Birthday Paradox.

If we have "n" length output then it takes $2^{n/2}$ work to find a collision but 2^n work to find a pre-Image.

- An ideal password hashing algorithm would be slow to compute, use as much memory as possible and be hard to parallelize.

- MD5 used to be a popular hash function with 128 bit output. But it has been broken with only 2^{18} work needed to find a collision.

Given a pw it is also very easy to reverse using a dictionary attack.

- Dictionary attacks become feasible when we consider that the password has to be human memorable (around 40 bits).

We generate a guess and then if it fails we permute the guess to generate the next one. This is based on some algorithm.

- Dictionary attacks can be made even easier if we know:
  1) The possible alphabet
  2) A set of substrings that could be in the password
  3) The length of the password

- Another form of password checking function "f" is using "p" as an encryption key:

$$pv = e(p, \text{"fixed-string"})$$

- If we use the above method, we can still crack the password using a dictionary attack, or we could leverage a time-memory trade-off using rainbow tables.

- The size of these tables can be 100s of GB.

They are sets of chains of passwords, where we only store the first and last elements of each chain.

> The next element in the chain is derived via a password hash function and a "reduction function"

To use this attack we iterate the reduction/hashing of "pv" until we find a value in some chain. At that point "p" is likely somewhere in the chain and we can begin in that chain till we find "p".

- SHA 256 replaced much of the shortcomings of MD5, but attacks can benefit from parallelism.

  Argon 2i seems to be the new standard for generating pw.

- HaveIBeenPwned is a public api where you can send the first 20 bits of a SHA-1 hash of your password.

  The website will then return all the hashes that match those 20 bits. You can then compare your full hash with all the returned ones to see if your password has been leaked.

Cryptograph Algorithms

- Cryptography mainly consists of a public and secret key and message digest (hash) algorithms
- These algorithms mainly provide encryption, decryption and integrity protection
- Secret Key Cryptography is where both parties have access to the same secret key (eg AES)
- Public Key Cryptography is where there is a public key that is known to everyone and a secret key. (eg RSA)
- Diffie-Hellman allows keys to be shared securely.
- Key Derivation Functions allow one to derive new keys from existing keys.

- Pseudo RNGs allow us to generate some pseudo random bits to generate keys.

- Encryption is where two parties agree on how to transform data.

We use that transform on data that we want to send over an unsafe channel.

Instead of coming up with new transforms, we would rather design a common algorithm and have a secret key.

- The output from these cryptographic functions should be "random looking".

Each output value should have about 50% "1" bits.

Changing one bit of the input should change about half of the output.

Outputs should be uncorrelated regardless of how closely related the input are.

Any subset of the bits should be equally random.

- Any public algorithm is liable to brute force attacks. The main strength of cryptographic functions rely on the limited computational power of the attacker.

- Algorithms also have weaknesses that are independent of key size

- It is always possible to break cryptographic algorithms but we want it to be atleast as hard as trying all the keys.

- The main uses of cryptography include:

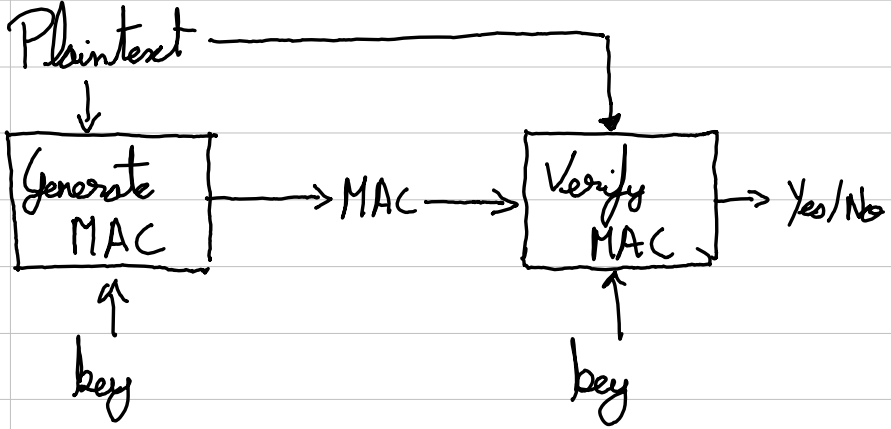
  - Confidentiality for data in transit

- Confidentiality for data at rest (Storing secure data on an insecure medium.

- Data Integrity : Eg. Message Integrity Checksum (MIC)
  Message Authentication Code (MAC)

- Authentication : "Challenge" the other party to encrypt or decrypt a random number.

Message Authentication Code BD:



- DES (Data Encryption Standard):
  - 56 bit key (+8 parity bits) = 64 bit "key"
  - Input and Output are 64 bit blocks
  - Slow to do in software

- IDEA (International Data Encryption Algorithm):
  - 128 bit key
  - Input and Output are 64 bit blocks
  - Designed to be efficient in software
  - Never caught on because it was patented.

- Since the 56 bit key was too short in DES, it was superseded by Triple DES
  - Apply DES 3 times with 3 different keys
  - Input and output are 64 bit blocks
  - Keys are 112 bits or 128 bits

- RC4 was a secret key algo used in browsers:
  - 128 bit key stream cipher
  - Widely used in TLS (up to 2014)

- Advanced Encryption Standard (AES)
  - 2001 US standard to replaces DES
  - Came from a public design and selection process
  - Key Sizes of 128, 196, 256
  - Block Sizes of 128
  - AES is implimented on hardware on many platforms.

- Chacha 20
  - 256 bit key stream cipher
  - Considered as a good replacement for RC4
  - It is faster than AES on processors without AES hardware support.
  - Recommendation is to use chacha 20-poly1305 over Chacha 20

- These are all Secret key Algorithms with symmetric keys since both parties share the same secret key.

- We can also use the XOR operation as an encryption algorithm.

We generate a one-time pad that is the same length as the message:

Encryption: B XOR A = Cipher-text

Decryption: Cipher-text XOR A = B

- Many crypto algos exist for a given purpose. It is very hard to invent new secure algos.

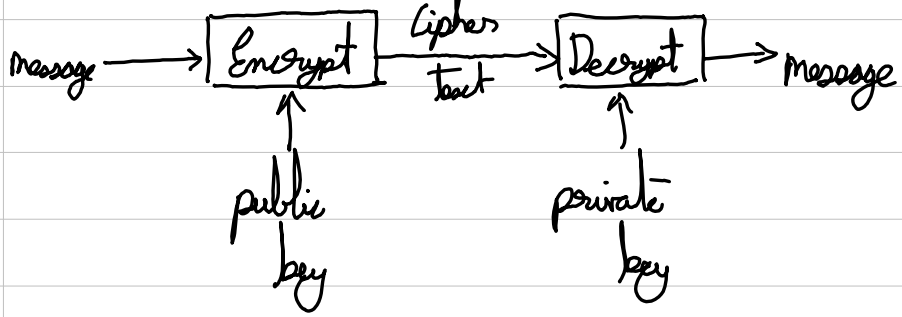
- Ideally for each purpose we would have a recommended algo and a backup algo.

- In Public Key Cryptography we generate 2 keys per user. The Public key and the private key.

The public key is known to everyone. If I want to send a message to Alice, I would encrypt the message using her public key.

Alice can then decrypt the message using her own private key.

- Alice can also generate Digital Signatures using her private key. Bob can then verify this Digital Signature using Alices' public key.



- Message Digest Functions (cryptographic hashes) are non reversible functions. They take an arbitrary sized message and output a fixed size digest.

  It should be hard to find two messages that have the same digest.

- MD2, MD4, MD5 and SHA-1 used to be popular but MD2-5 have been broken and SHA-1 is shown to have severve weaknesses.

  SHA-256 took over and has been shown to be pretty secure.

- True RNGs are based on some physical phenomena eg (noise diodes). This is hard to do in practice for every single device that needs to use RNs.

- Pseudo RNGs generate random numbers based on a seed and some algorithm.

All PRNGs eventually repeats and not all PRNGs are good.

→ Purposefully weakness introduced by NSA

There have been cases where a PRNG was engineered to be deliberately bad eg (Dual EC DRBG)

- We should be careful with PRNGs when creating forks and virtualisation since the new process can share the same PRNG state as the parent process.

- Blindly using PRNG implimentation is bad too since we don't know how random the PRNG is.

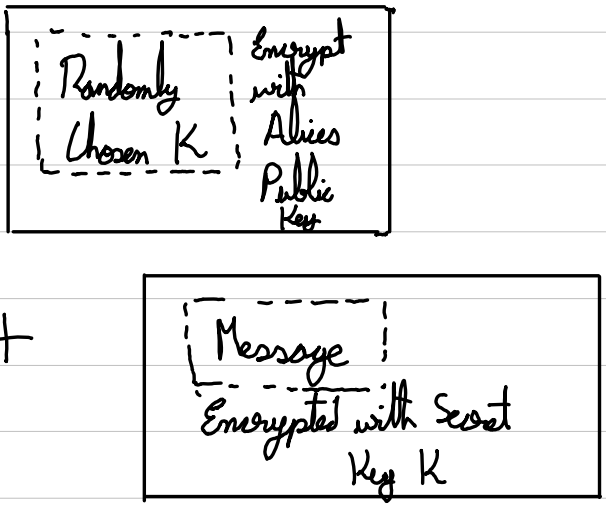
- It is a good idea to have two seperately seeded PRNGs for sensitive and non sensitive purposes

- Public key cryptography is slow compared to secret key cryptography and hashes.

However PKC is often more convenient and secure in setting up keys.

Usually algorithms can be combined to get the advantages of both.

- Eg. A Hybrid Encryption scheme can be like this



- The modern Hybrid Public Key Encryption (HPKE) scheme can be found in RFC 9180

- Passwords should never be used as Secret Keys since an attacker can learn sufficiently enough to guess the password.

- One major problem with secret key cryptography is the issue of key-exchange.

When we get a large number of users, configuring n^2 keys become impractical.

It is better to use something like a Key Distribution Center (KDC)

In KDCs:
- Everyone has a long term Secret Key
- The KDC knows <u>all</u> the Secret keys
- The KDC assigns a session key to any pair who need to talk.

- This is how Kerberos works.

Alice lets the KDC know that she wants to talk to Bob.

The KDC generates a session key "K_{ab}". One copy gets encrypted with Alice's Secret Key to generate $A\{K_{ab}\}$. The other copy gets encrypted with Bob's secret key to generate $B\{K_{ab}\}$.

Both these keys get sent to Alice. She can use her key to decrypt $A\{K_{ab}\}$ to get the session key which she can use to encrypt her message to Bob $K_{ab}\{Message\}$.

She then sends $K_{ab}\{Message\} + B\{K_{ab}\}$ to Bob. Now Bob can read the message and they can now use the common secret session key "K_{ab}" to encrypt all future messages for that session.

- With public key cryptography we make use of a Certificate Authority (CA).

The Certificate Authority signs "certificates" that is a signed message which basically says "I the CA can vouch that this public key PK is Alice's public key."

If everyone has a certificate, a private key and the CA's public key, they can then authenticate anyone elses' public key.

- There are some tradeoffs between KDCs and CAs.

Stealing the KDC database allows the impersonation of all users and the decryption of all previously recorded conversations

Stealing the CA private key allows forging of certificates and hence impersonating users.

However recovering from a CA compromise is easier since everyone doesn't need to create a new private-public key pair. Only the CA needs to generate new keys in order to sign new certificates

The KDC must always be online and needs to have good performance all the time.

However the CA only needs to create certificates when new users join. The CA can be powered down and the certificates will still work.

CAs work better interrealm because you don't need connectivity to remote CAs

Public key cryptography is slower when compared to secret key.

The "revocation problem" in secret key cryptography somewhat mitigate this

- Secret keys can be split into

  - Stream Ciphers: Takes a key and generates a stream of pseudorandom bits

  - Block Ciphers: Takes a key and fixed size input blocks to generate fixed size block outputs

- Shannon proved that the XOR function with a one time pad was unbreakable if you only have the ciphertext. If we reuse the same key for another text, XOR effectively becomes insecure.

- For block ciphers, if we have a small block size we can build a table.

  However if we see the same block multiple times in the cipher text, we can make some inferences about the plaintext

- To avoid this we should probably change the key every $2^{\frac{\text{block size}}{2}}$ blocks. However this isn't always practical for some applications eg. long lived sessions with VPNs.

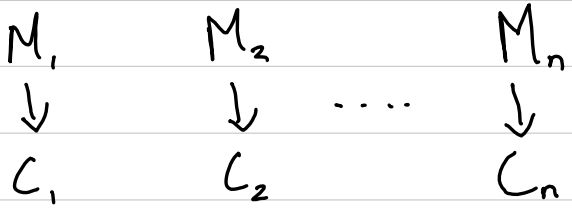
But essentially there is a soft limit on how many blocks we can encrypt with one key.

- Since basic block ciphers encrypt small fixed size blocks, we run into issues with large messages.

- The solution is to cut the message into blocks and encrypt these blocks. This is called "Electronic Code Book" mode (ECB).

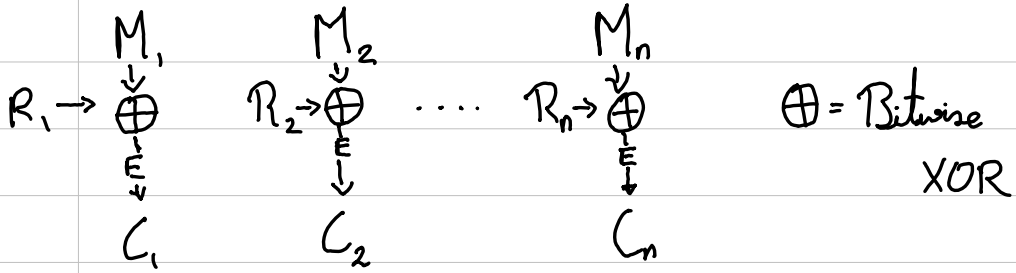
The issue is that repeated plaintext blocks will yield repeated cipher blocks. There are "chain" modes that avoid this (CBC, CFB, OFB)

- ECB works like this:



As said before, if we know that $C_1 = C_4$ then we also know $M_1 = M_4$. One other issue is that we can rearrange the blocks to affect the plaintext.

We could do something like this:

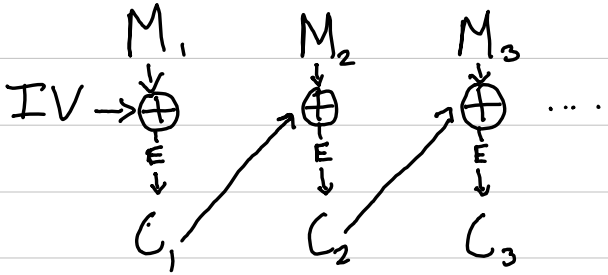


We then transmit $R_1, C_1, R_2, C_2 \ldots R_N, C_n$.

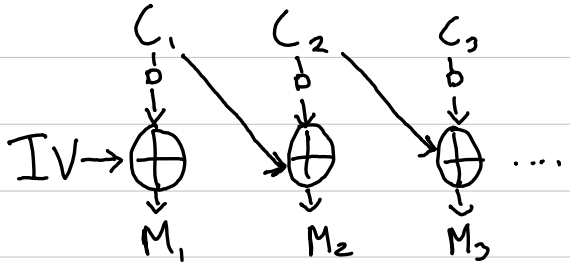
R_n = Random Number

- Here we can still rearrange blocks. Also if two cipher blocks are equal, we can get the XOR of the plaintext blocks by XORing the two related R_ns.

- CBC generates its own R_nS by using previous cipher blocks plus an Initialisation Vector at the start.



Decryption is then:



- There are some issues with CBC. If a block C_x gets lost, then subsequent decryptions will yield garbled text. Also this doesn't guarantee message integrity

- "CBC Residue" uses a key K and generates an integrity check on message M.

We can then do CBC encryption on M using the key K and throw all but the last block. The last block is called the "residue" and is used as an integrity check.

This has the property where if you don't know K you can't generate or verify the Message Authentication Code (MAC) or modify the message without changing the MAC.

- Note that it is possible to generate an arbitrary message with a particular residue if we know the secret key.

1) Create a Message but leave one block blank

2) Use any key K and IV

3) Start from the beginning, doing ordinary CBC residue until we get to the blank block.

4) Start from the end and perform CBC decryption until we get to the blank block.

5) Finally you will be left with two values that need to be XOR'd to yield the blank block.

- We can create a stream cipher from block ciphers. This is what Output Feed Back (OFB) does.

  - We create an IV vector that gets transmitted in clear.
  - $pad_1 = e(IV, key)$
  - $pad_2 = e(pad_1, key)$
  - $pad_3 = e(pad_2, key)$

- We can encrypt/decrypt using bitwise XOR with as many pad bits as needed. These pad values can also be generated in advance.

- OFB can encrypt an arbitrary number of bits

- There are "counter" modes that are of the form

$$C_i = f(key, IV, \text{block number}, p_i)$$

This has the advantage of being able to decrypt an

arbitrary block. This is good for random access file encryption

- Authenticated Encryption with Additional Data (AEAD)

  - This is a general scheme for combining confidentiality and data integrity as a single primitive.

The motivation for this is that many applications / protocols have cleartext headers that go with the ciphertext and ideally we want both to be protected in a single operation with a single key as input.

The general idea is that you encrypt data and provide the additional data. The result is that the ciphertext that includes a "tag." where this tag authenticates both the ciphertext and the additional authenticated data.

Decryption involves taking the cipher text and the tag and returns an error or the plaintext.

In theory we would never get partial decryption.

RFC5116 defines an abstract interface for AEADs. Typical tags are 16 octets, typical IV/nonces are 12 octets.

Most new work makes use of AEAD modes since CBC have been shown to be vulnerable if there is some implimentation errors.

- Most AEADs are hard to understand and hence their drawbacks are even harder to understand. They tend to be quite complex.

- Performance Considerations:
  AES-GCM (AES Galois-Counter mode) allows for super paralleled and pipelined implementations

It is good for high throughput application such as 100 GB ports.

AES-CCM (CCM = Counter with CBC MAC) requires us to know the lengths ahead of time, but it is popular for smaller processors. Used as needed in Wifi and Zigbee.

ChaCha 20-poly 1305 is fast if you don't have hardware support for AES.

All AEAD modes add some overhead with tags and nonces, which can be a pain if you have a lot of small packets or are constrained by bandwidth.

Patent Considerations:

Lots of people like Offset Code Block (OCB) but it never got used due to patents.

AES-GCM, AES-CCM and Chacha20-poly1305 are patent clean, but some patents exists on tricks that are used to speed up implementation

Brittleness Considerations:

While AES-GCM is good we can end up with very poor security if we reuse the same key and nonce.

This is because it ends up behaving like a stream cipher.

When using AEADs we should <u>NEVER</u> reuse the nonces (these are the IVs)

- AEAD summary:

  - Never reuse nonces
  - AES-GCM is a good default mode
  - AES-CCM is widely supported on less capable devices
  - Chacha20-poly1305 is used when there is no h/w support for AES.

- DES:
  - This is a 70's era block cipher.
  - Uses a 56 bit key and 64 bit blocks
  - It has 16 rounds with an Initial Permutation (IP) and a Final Permutation (FP).
  - The Feistal function is run in each round with 32 bits of input with 48 key bits.
  - Decryption is as easy as reversing the Encryption process.
  - The 56 bits of the 64 bit key is used to generate the 48 bit subkeys.

- Each round uses a new subkey. Each time we generate a new 64 bit key, we need to recalculate all 16 subkeys.
  - This overhead can be significant if we change keys often.
  - The Feistel function is key to the security of DES. The 32 bit input gets expanded to 48 bits and gets mixed with a subkey.
    The 48 bit output gets split into 8 blocks of 6 bits. These get passed to S-boxes (Substitution boxes) which replace the 6 bits with 4 bits according to a LUT. The output is then rearranged into 32 bits for the next stage.

- Triple DES (3DES):
  - It is defined as doing EDE with K1, K2 and K3. (Sometimes K1 = K3)
  - It only offers 112 bits of security rather than 168 due to a "meet-in-the-middle" attack.

EDE = Encrypt - Decrypt - Encrypt

EEE = Encrypt - Encrypt - Encrypt

- We use EDE instead of EEE because the initial and final permutations would cancle each other in EEE.
- EDE is also compatable with single DES if $K1 = K2 = K3$

- AES [Advanced Encryption Standard]

The Rijndael Algorithm was selected to be AES by NIST. It is a block cipher that reads in text, processes it in rounds and outputs a cipher text.

Each round is a result of 4 inner transformations. The standard specifies 128 bit blocks and key sizes of 128, 192 or 256 bits

A round is as follows:

- Substitute bytes using an S-box
- Shift the Rows
- Mix the columns
- Add the Round Key

This round gets repeated a bunch of times depending on how large the key is.

The round key gets derived from the secret key based on some algorithm.

The 16 bytes gets arranged into a state array. This array has its bytes substituted based on the rules set out by the SBox. The rows of the state array get cyclically shifted. The first row remains unchanged, the 2nd row gets shifted once, the 3rd row gets shifted twice and the 4th row gets shifted thrice. This then gets multiplied by a coefficient matrix to mix the columns and then finally the state array gets mixed with the round key.

Usually we do 10, 12, 14 rounds for 128, 192 and 256 bit keys respectively.

Hash / Message Digest

- These take an arbitrary size input and generates a fixed size output.

- A good cryptographic hash / message digest should be:

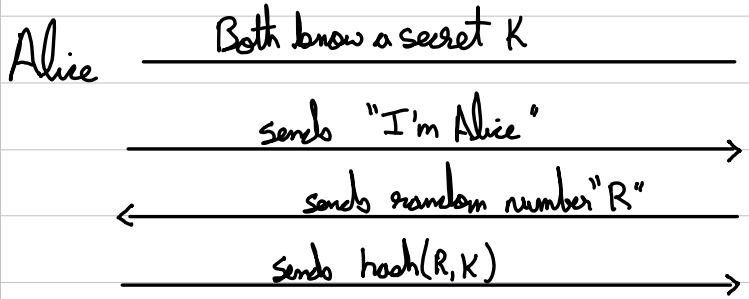
  - One Way. I.e it should be computationally infeasible to find the input for a particular hash value.

  - Collision Resistant. We can't find two inputs that yield the same hash.

  - The outputs should look random

- We can store the digest of files and use that to check if the file was tampered with.

- With a secret key, a hash function can do anything a secret key algorithm can do. This isn't really a good idea though.



This is an example of Bob authenticating that Alice is who she says she is. This works because only Bob and Alice know the secret. Bob can compare the hash with one he computes locally to verify that he is talking to Alice.

- Some more properties a good hash function should have:

  - Pre-Image Resistance: Given a hash value "h" it should be difficult to find any message "m" such that h = hash(m)

- Second Pre-Image Resistant: Given an input m_1, it should be hard to find an input m_2 such that $hash(m_1) = hash(m_2)$

- Collision Resistance: It should be hard to find two messages m_1, m_2 such that $hash(m_1) = hash(m_2)$. This requires a hash value atleast twice as long as that required for pre-image resistance

- Speed: Given an arbitrary size input you should be fast producing an output.

- SHA-1 used to be a good hash until practical collisions were demonstrated in 2017. Then in 2020 the first "Chosen prefix collision" was demonstrated. It has been deprecated by NIST since 2011 in RFC 6194. Git still uses SHA-1 for commit history.

- SHA-2 shared the same Merkle-Damgård design as MD5 and SHA-1. SHA-256 is the reccomended function, it has 64 rounds and produces a 256 bit output. As far as we know it is still secure.

- SHA-3/Keccak was designed to be a successor to SHA-2 since the underlying issue with SHA-1 was the Merkle-Damgård design. SHA3 makes use of a "sponge design". The block transformation function makes use of XORs, NOT and AND operations. It is slower than SHA 256 in S/w.

- We can combine the key with a message and use the entire result as a MAC. However many hash functions have issues with this. Mainly due to "length extention" attacks.

- A length extension attack is where an attacker can use Hash(m.) and the length of m, to

calculate $Hash(m_1 || m_2)$ where m_2 is an attacker controlled message.

When a hash function is used as a MAC with the construction $Hash(secret || m_1)$, the length of the secret and the length of the message is known. So an attacker can add extra information at the end and provide a valid hash, without knowing the secret.

Hash functions based on the Merkle-Damgård design are susceptible to this attack. SHA3 isn't. HMAC (RFC 2104) was proven not to have this issue.

HMAC works like this:

$$HMAC = H(K \text{ XOR } opad, H(K \text{ XOR } ipad, text))$$

where H is either SHA-1 or SHA-256

- HMAC is widely used because it comes with a proof. Assuming the underlying function is collision resistant, if an attacker doesn't know K, he cannot compute the proper digest(K,x) even if he sees an arbitrary (y, digest(K,y))

- Most cryptographers prefer to work with algorithm that have some type of proof. Usually they depend on some underlying assumption eg. Factoring numbers is hard or discrete logarithm are hard.

  However a proof that an algorithm is secure doesn't always mean the implementation is. Implementation errors are more common than algorithm design flaws.

- Hash functions can also be used for One Time Signatures (OTS). To sign a single bit "b" we generate a private key $s_0 || s_1$ and a public key $H(s_0) || H(s_1)$. The signature will be s_0 if "b" is 0 and s_1 if "b" is 1. We can generalise to N-bits

If we ever reuse a private key, we essentially give away the private key. There are now stateless Hash based Signatures that avoid this. A stateless protocol can be SPHINCS+

Public Key Algorithms

- For PKA we want an algorithm with the following properties:

  - It should use 2 numbers "e" and "d". They should also be inverses ie using one reverses the effects of the other.

  - You shouldn't be able to compute "d" from "e".

  - It must be efficient to find a pair of keys and it must be efficient to encrypt and decrypt.

Eg: An insecure PKA:

- Multiplication mod "p" where p is a prime.

Lets choose $p = 127$. We then choose "e" and "d" such that $e \times d \bmod p = 1$. Lets choose $e = 53$ and $d = 12$.

To encrypt a number we multiply by 53 mod 127. To decrypt we multiply by 12 mod 127.

- One issue with this is that it is very easy to compute "d" from "e" by trying all the possible values up to 127. Modular division is very easy and fast.

- Doing this type of crypto means we need to work with Multiprecision libraries since we generally deal with numbers larger than 64 bits.

- RSA:

It was named after its inventors Rivest, Shamir and Adelman. It uses modular exponsion so its functional basis.

We choose a modulus "n" and a public exponent "e". We choose n to be the product of two primes "p" and "q". RSA depends on factoring being hard.

Encryption is as follows:

$$\text{cipher-text} = \text{plain-text}^{e} \bmod n$$

Decryption is as follows:

$$\text{plain-text} = \text{cipher-text}^{d} \bmod n$$

An indepth look into how RSA works:

First we define $\phi(n)$ to be the number of integers $< n$ that are relatively prime to n.

Euler proved that:

$$x^{\phi(n)} \bmod n = 1$$

So $x^{k\phi(n)} \bmod n = 1$ and $x^{k\phi(n)+1} \bmod n = x$

If we can find $d \times e \bmod \phi(n) = 1$ then they would be exponentiative inverses of each other.

Proof:

$$C = M^e \bmod n$$

$$C^d = M^{ed} \bmod n$$

$$= M^{(1+k\phi(n))} \bmod n$$

$$= M \times M^{k\phi(n)} \bmod n$$

$$= M$$

We can compute $\phi(n)$ in the following way:

If p is a prime then $\phi(p) = (p-1)$.

We choose $n = pq$ such that p and q are prime. So $\phi(n) = (p-1)(q-1)$.

Given e, $\phi(n)$ and Euclid's algorithm we can compute "d" such that:

$$d \times e = 1 \bmod \phi(n)$$

[this is essentially a long division problem]

So we can find "d" if we know "e" and $\phi(n)$. But if we don't have the factors of "n" we need to factor it to be able to find $\phi(n)$ which is very hard.

The security of RSA depends on the factoring problem being hard. Once we compute d and e we can forget p and q but in practice we don't.

- Finding large primes is relatively easy. There are a number of tests that we can perform to test the primality of a number.

- One probabalistic test is using Fermat's theorem:

$$x^{p-1} \bmod p = 1 \text{ if } p \text{ is prime}$$

So to test if "p" is prime we pick $x < p$, raise it to the power of $p-1$ mod p. If the answer is not 1 then it is definitley not prime. If the answer is 1 then it is probably prime.

The result can be 1 even if the number is not prime although for the primes we are considering the chances are about 10^{-4}.

- There are deterministic tests that exist that are used for any serious implimentation.

- We implivent exponentiation by performing repeated squaring. The result is that a 1024 bit exponent requires between 1024 or 2048 multiplications instead of a whole lot more with normal exponentiation.

- For RSA Signing we want to apply the private key (d, n) to the message and allow the public key (e, n) to be usable to verify the signature.

We just do the obvious and use a "decrypt" operation on a hash of to sign the message and an "encrypt" operation to verify:

$$\text{Signature: } S = [H(M)]^d$$

Verification: Given M and S, check if $H(M) = S^e$

- It turns out that RSA is secure even if "e" in (e, n) is small like 3 or $2^{16} - 1$.

- Using the Chinese Remainder Theorem we can do the arithmetic mod p and then mod q, and then combine them. We can precompute what "d" is mod p and mod q. Then we can precompute p^{-1} mod q

Here are some RSA threats that are largely avoided by standards:

- If we encrypt a guessable plaintext an eavesdropper can easily verify a guess
- It is trivial to forge a signature if you don't care what you're signing.
- Smooth Numbers
- If we pad on the right with random data someone can choose a padding such that the message will be a perfect cube.

- The smooth threat is as follows:

Suppose we see a signature on m_1 and on m_2. Suppose we can factor the "m" and see lots of signature (thousands). Various

combinations of these factors are likely to give you signatures on various primes and then you can sign anything which has just those primes.

- Smooth numbers are numbers whose prime factors are less than some number. Eg 56 can be factorised as $2 \times 2 \times 2 \times 7$ so we say it is 7-smooth. It is not 5-smooth since $7 > 5$.

Diffie-Hellman Key Agreement

- This allows two individuals to agree on a secret key even though they can only communicate in public.

- Here is how it works

1) Alice chooses a private number and from that calculates a public number.

2) Bob does the same

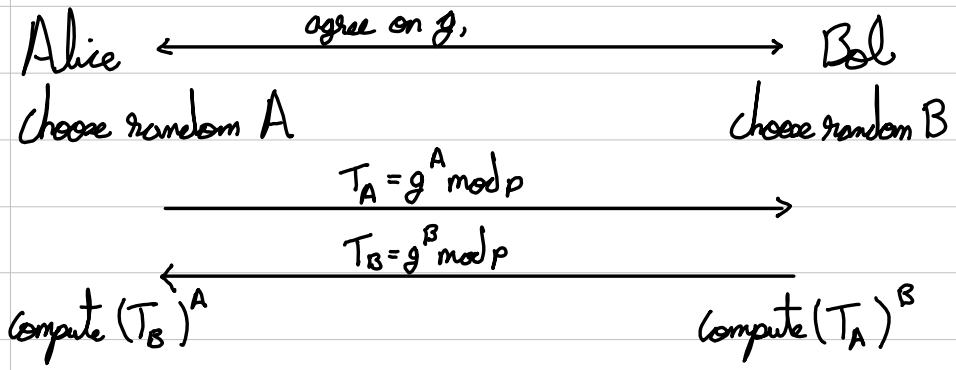
3) Alice and Bob can use each other's public number and their own secret number to compute a shared secret

Most importantly an eavesdropper cannot reproduce the shared secret.

- The security of DH is based on the fact that the discrete log problem is hard.
In DH the dlp is:
Given a prime "p" and a generator "g" and $g^x \bmod p$, what is "x".

- The discrete log problem seems to be slightly harder than the factoring problem for a number of the same length as p.

[Alice and Bob agree on "g" and "p" this can be set out in a standard]



Now they share the same secret $T_B^A = T_A^B = g^{AB} \bmod p$

- However this scheme is liable to a main in the middle attack. A man could sit in the middle and do DH with A and B seperatley. And then he could read all messages between Alice and Bob.

- To avoid this man in the middle attack we can perform signed DHKA.

Alice | Bob

choose random A | choose random B

$[T_A = g^A \bmod p]$ signed with Alice's Private Key →

← $[T_B = g^B \bmod p]$ signed with Bob's Private Key

Verify Bob's signature, compute T_B^A | Verify Alice's signature, compute T_A^B

both agree on $g^{AB} \bmod p$.

- If we have public keys already why would we bother with DH?

DH allows us to have "Perfect Forward Secrecy". PFS prevents an attacker from decrypting a conversation even if they break into both parties after it ends.

- RSA key transport doesn't have this forward secrecy property.

RSA key transport works as follows:

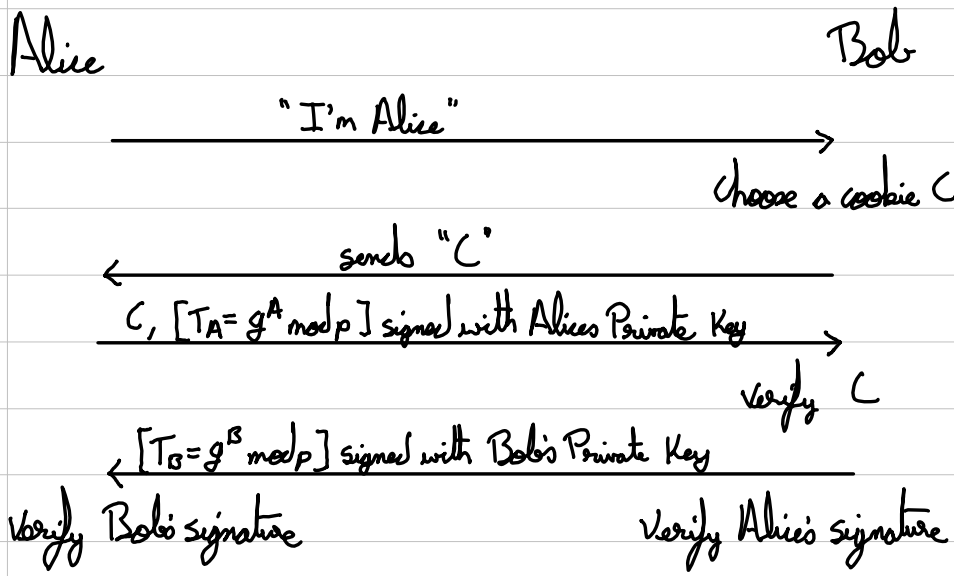
A chooses a secret key "S", she encrypts S with B's public key and sends the cipher text to B.

If an attacker stores all communication between A and B, they can read all messages between A and B if he gets B's private key.

- Today we strongly recommend forward secrecy for all internet security protocols and applications

- DH can be used for some type of DOS attack: if a user just keeps trying to exchange keys with a server.

We can impliment a cookie mechanism into DH to offer some DOS protection.



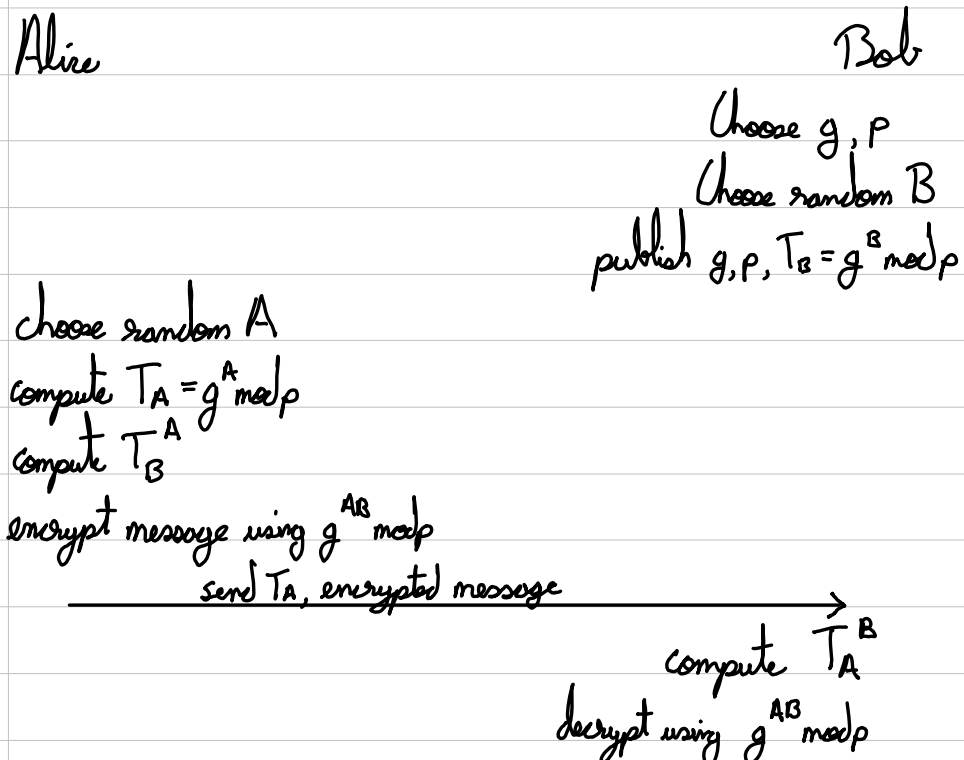
Both now agree on $g^{AB} \bmod p$

Now Bob can avoid doing the computation to generate T_B and verifying Alice's signature until he verifies the cookie. The downside of this is that Bob now needs to store the cookie.

Ideally Bob wants to have a stateless cookie. This is a cookie that Bob can verify without needing to store a state for each connection.

For instance Bob can have a secret S and then create a cookie such as $C = \{IP\ address\}S$

- We can also use DH for encryption:



- The diagram above is called "Ephemeral-Static DH". Alice's value "A" is ephemeral while Bob's value "B" is static.

There is a standard for "Static-Static DH" where both use static DH values. RFC 6278 outline some security considerations

The User Keying Material (ukm) used in Static-Static DH should be changed on a per message basis. If the ukm is not changed then the same keying material will be used across multiple messages which opens the sender to DH attacks such as the "small subgroup" attack.

↑ The ukm is used to seed the KDF

- Static-Static DH also allows wiretapping by whomever knows the private static value ("A" or "B"). This is the case for "centrally" generated private values.

A government agency can use this to look at all outgoing emails.

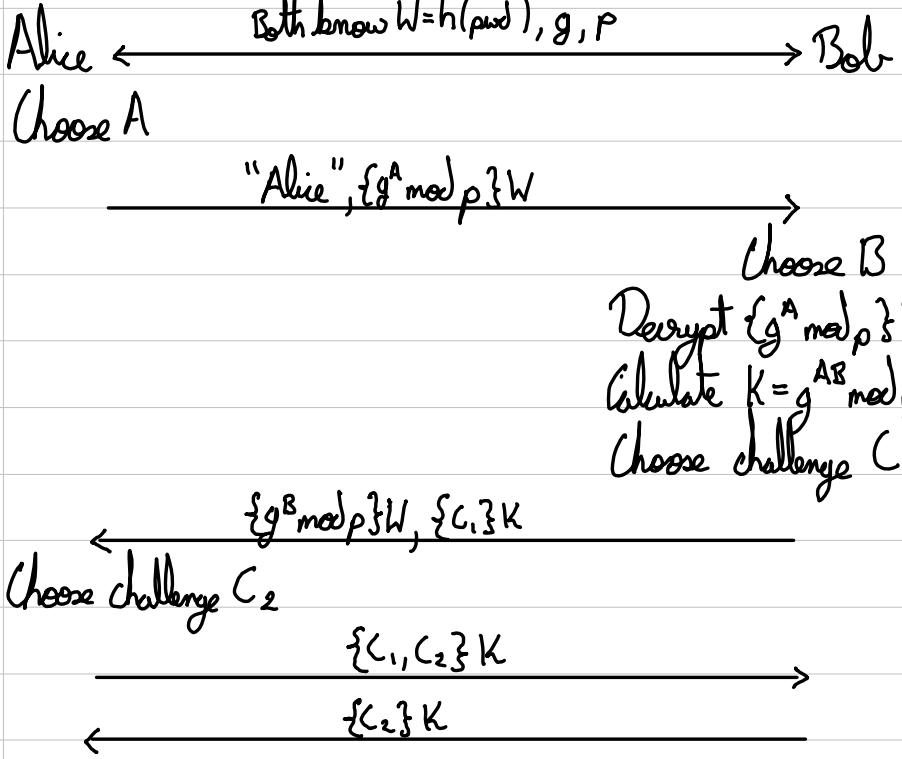
PAKE Protocols

- Password Authenticated Key Exchange protocols is a method in which two or more parties can establish a cryptographic key based on their knowledge of a shared secret.

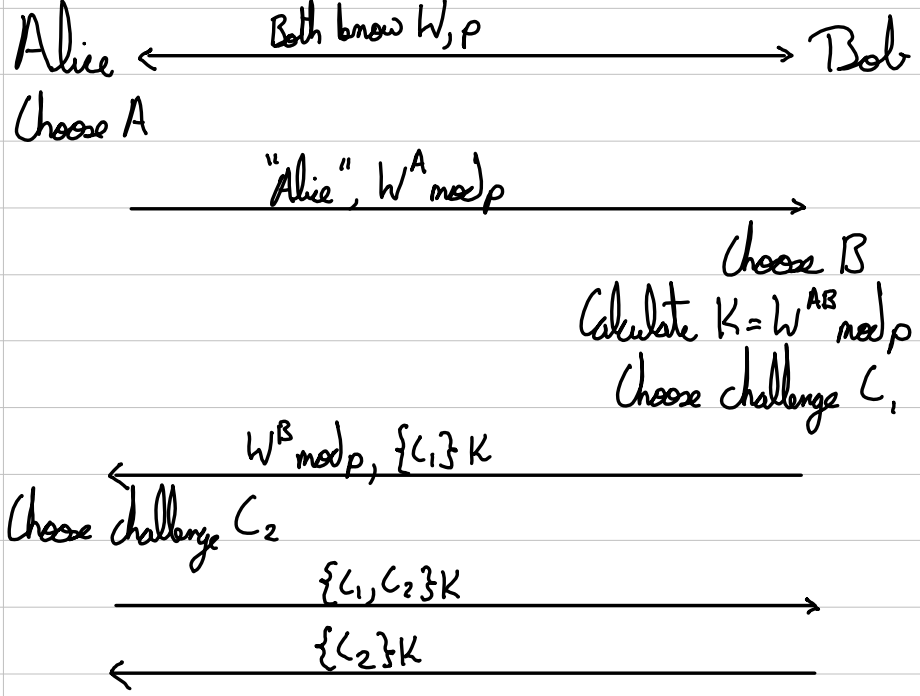
- Using a hashed password as a key is never a good idea since it is vulnerable to a dictionary attack if there is any structure in the ciphertext. PAKE protocols try to avoid using the hashed password as the key.

- There are 3 main protocols to look at, EKE (designed for mutual authentication), SPEKE, PDM

- EKE

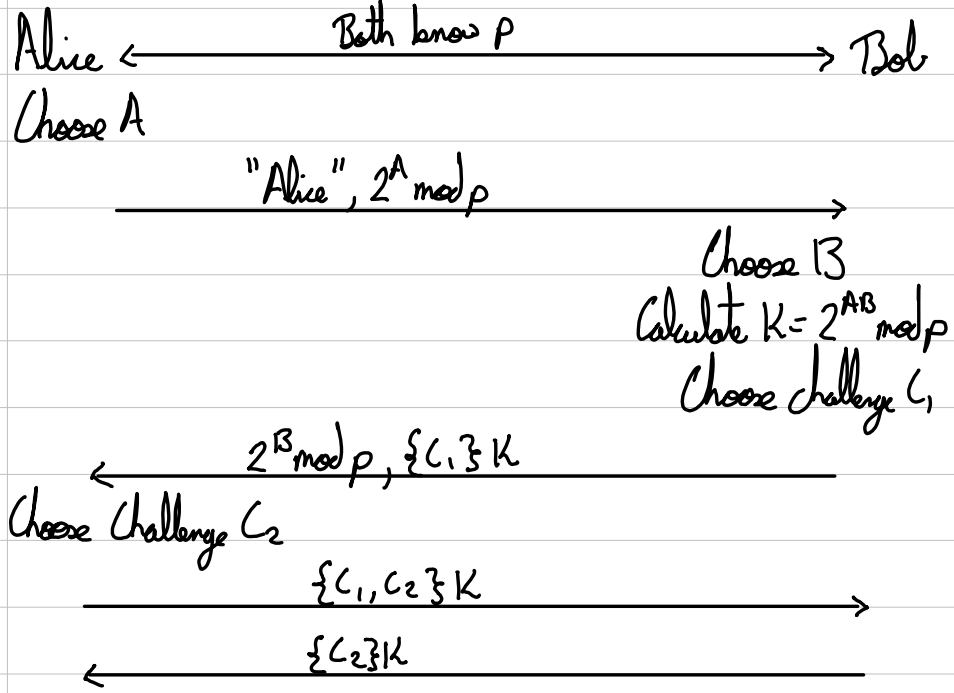

Alice
Both know W=h(pwd), g, p
Bob
Choose A
"Alice", {g^A mod p}W
Choose B
Decrypt {g^A mod p}W
Calculate K=g^AB mod p
Choose challenge C1
{g^B mod p}W, {C1}K
Choose challenge C2
{C1, C2}K
{C2}K

~SPEKE



Only difference to EKE is that W is used as the generator "g"

- PDM (Password Derived Moduli)



The modulus "p" is based on a function of the password.

- Each of these protocols require some shared information in addition to the password. (eg for EKE we need to know both g and p)

  This information is not human memorable and so these values need to be hard coded.

- A possible solution is that the client has software that deals with the protocol and the configuration. This could get messy if the code is proprietary or if the user needs to install multiple software to deal with all the protocols.

  They could also use a token or a smart card that has the code and configuration on it. Issues arise if the token/card is lost or stolen. Can be expensive at scale.

- On the server side we need to have a database and s/w to deal with this. Also account management is problematic.

- Ultimately PAKEs weren't widely used due to patent issues.

Elliptic Curve Cryptography

- Elliptic Curves have the form:

$$y^2 = x^3 + ax + b$$

x and y can be Real or members of a finite group.

- The points along this curve form a group and they have a group operation known as "dot" (or sometimes called addition) that takes two points and returns a third point in the group.

- The main thing about Elliptic Curves and this "dot" operation is that they have the "discrete log problem"

- The discrete log problem can be stated as such for elliptic curves:

  Given P and Q on the curve, it is hard to find a scalar k such that $P = kQ$

- The key sizes in ECC tend to be a lot lower than RSA for equivalent security. A 256 bit key in ECC would need a 3072 bit key in RSA.

- Elliptic Curve Diffie-Hellman has replaced most of RSA key transport and most integer based DH.

- NIST recommended curve P256 but now we dislike this curve since it had poor performance and it is easier to write buggy code for it.

- P256 were replaced with Curve25519 and Ed25519. These were 4x quicker than P256 and it was harder to write buggy code for them.

- Curve448 and Ed448 were created to be a lot more secure.

- Elliptic Curve Digital Signature Algorithm (ECDSA) is a way to sign messages using elliptic curves.

  However if we ever use the same key to sign a message, then we have effectively leaked the private key since it can be recovered from the two signatures.

- It has also been shown that fault injection attacks where an adversary can selectively flip bits can lead to leaking the private key.

Extra Crypto Stuff

- Key Rotations

  Since all symmetric encryption schemes have an upper limit on how many times a key can

be used, it is mandatory to change key periodically.

This also helps in the case an employee gets fired or a hard drive with the key doesn't get decommissioned properly.

- Identity Based Cryptography

The idea here is that you use someone's name as a public key to encrypt something to them.

This works if there is a key generator that is trusted, where the sender has a high integrity copy of the key generator's public key.

Almost all schemes require the key generator to be able to decrypt all traffic (mandatory key escrow)

It functions like RSA group keys

- Secret Sharing

This is where you want to share a secret among N people such that if any "k" of them come together, they can recreate the secret

This is useful if you want to distribute things among partially trusted entities.

This is the idea behind "Shamir Secret Sharing". It uses polynomials. We distribute N points along an x^k order polynomial. So the polynomial can only be reconstructed if "k+1" people come together

- Nothing Up My Sleave

In many crypto schemes, the choice of parameters can greatly affect the security of the algorithm.

The absence of a detailed explanation for the P256 curve that NIST chose raised controversy, since NIST have been known to recommend some parameters that compromise security.

One approach is to use well known constants eg $\sqrt{2}$ or π and generate the free parameters from them. This way we have nothing to hide aka "Nothing Up My Sleeve"

Post Quantum Cryptography

- Cryptographers are studying new public key algorithms that aim to provide good security even in the face of a cryptographically relavent quantum computer

- Grover's algorithm is a more efficient brute-force attack meaning attacks on 128 bit keys only take 2^{64} operations

- Shor's algorithm describes how to factor numbers quickly rendering RSA broken. Variants of Shor's algorithm solve the discrete log problem. meaning ECDH is broken

- While we don't quantum computers now that can implement this algorithm, intelligence agencies like to store ciphertexts in the hope that it can be broken later.

- NIST started a competition in 2017 where they were looking for new Key Encapsulation Methods to replace ECDH and new signature algorithms.

- Kyber (ML-KEM) was an algorithm that won the KEM competition

- Dilithium (ML-DSA) won the digital signature competition

- SPHINCS+ is another algorithm for digital signatures

- However Kyber has some IPR issues which NIST mitigated by paying for licencing. They did also announce a round 4 for KEM algos. SIKE was selected but was broken in August 2022.

- The PQC algos aren't a drop in replacement as they don't satisfy the same size/CPU requirements met by classic public key algos.

- Size/CPU impact varies from algo to algo but mainly these new algos have:

  - Big public keys or signatures which is bad for DNS security

  - Bigger KEM outputs not great if the TLS ClientHello cannot fit in one packet.
    → Place the key in TLS message extensions

Fragmentation of data packets happens

Kyber

- Kyber is based on the "Module Learning With Errors" problem.

  This can be also stated as the "Closest Vector Problem" which tries to find the nearest lattice point (in n-dimensions) to a given point.

- There are many variants, Kyber 512, Kyber 768, Kyber 1024.

- Kyber 512 offers similar security to AES 128. It generates public keys of 800 octets and an encapsulated size of 768 octets.

- The arithmetic in Kyber deals with vectors and matrices of polynomials over a ring.

We set the ring R_q to be:

$$R_q = \frac{\mathbb{Z}_q[X]}{X^n + 1} \quad \text{for } n = 256,\ q = 3329$$

The secret vector "s" is an element of R_q^2 and the error vector "e" is another element of R_q^2

These vectors are "small" (Best to think of them as vectors of polynomials with small coefficients)

We use a matrix A from $R_q^{2\times 2}$. The public key is A and $t = As + e$

- We can encrypt a 256 bit message "m" as a polynomial with one message bit per coefficient.

- To encrypt we choose two error vectors "e_1" and "e_2" and a randomiser vector "r". These are generated everytime we want to encrypt something

To encrypt 'm' using the public key (A, t) we do:

$$u = Ar + e_1$$
$$v = tr + e_2 + \left(\frac{q}{2}\right) m$$

The cipher text is (u, v)

- To decrypt we calculate:

$$= v - su$$
$$= \left[\frac{q}{2}\right] m + er + e_2 - se_1$$

Other than $\left[\frac{q}{2}\right] m$, the other terms are "small" and since each "m" coefficient is either 0 or 1, the result has all the coefficients close to 0 in which case the relavent message bit is 0 or close to $\left[\frac{q}{2}\right]$ in which case its a 1.

Stateful HBS

- A Stateful Hash Based Signature scheme is one where the private state (the value of the private key) changes with every signature operation

- These are quantum resistant so it is worth having for cases like Software Signing where we won't need many signatures and the sizes aren't a big deal.

- But these need a different crypto API since the private keys have a fixed, limited number of uses before they need to be changed. Otherwise signatures can be forged

- Two algorithms were standardised XMSS (RFC 8391) and LMS (RFC 8554)

- SPHINCS+ is internally similar to SHBS like XMSS, however it is stateless.

Statelessness is achieved by selecting from many private values using an r-subset resiliant selection function.

As the private key doesn't change, it's ok to sign 2^{64} messages without significant loss of security.

PQ KEM w/ ECDH

- TLS, SSH IPSec and other classic key exchanges will be extended to allow mixing in of PQ key exchanges in addition to traditional DH exchanges

- However there are some issues with scaling and efficiency.

Each hybrid key exchange group will be sent in the supported groups extension. The messages from the hybrid schemes will be concatenated and is sent in the extension data fields for KeyShareClientHello and KeyShareServerHello.

Quantum Key Distribution

- QKD directly exchanges photons between the endpoints and we can accumulate the key bits based on the received photon state.

- By definition QKD requires no intermidiaries and hence has scaling problems. But it still can be used on specific links for a specific purpose eg. Banks.

Standard Security Protocols

- Standards allow for multi-vendor interoperability as well as encouraging "open-ness".

- Having a good standard can increase security as they can be well tested.

- Some international standards organisations include UN/ITU, ISO.

- Internet standard organisations are IEEE, W3C IETF.

- Open Source Projects : Apache, OpenSSL.

- The ISO/ITU-T have national bodies as their members eg. NIST. They mostly deal with child online protection, cryptographic mechanisms ect.

- The IETF is made up of a group of individuals on the internet

- Some standards can be bad if they are carefully thought out. Eg WEP

WEP

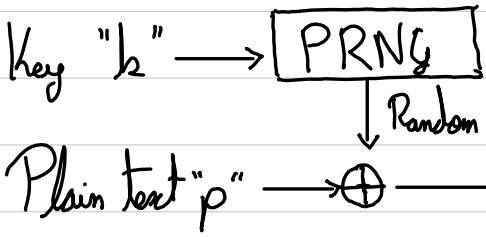
- Wired Equivalent Privacy was a poor security standard designed to bring data confidentiality to Wifi that was similar to wired connections.

- They used the RC4 Encryption algorithm and had a per packet encryption key which was a 24 bit IV concatenated to a pre-shared key.

- WEP allows the IV to be reused with any frame. Data Integrity is provided by CRC-32 of the plain-text data (ICV).

The ICV and the data are encrypted under the per packet encryption key

- The RC4 algorithm is a Vernam Cipher: it works like this:



Decryption works the same way:

$$p = c \oplus b$$

What if we encrypt two things using the same "b"

$$c' = p' \oplus b \qquad c = p \oplus b$$

$$c' \oplus c = (p' \oplus b) \oplus (p \oplus b) = p' \oplus p$$

Essentially we can recover the plain text from the cipher texts. We should never encrypt two different things with the same "b".

- By the birthday paradox, the probability P_n that two packets will share the same IV after n frames is:

$$P_n = P_{n-1} + (n-1)\left(\frac{1-P_{n-1}}{2^{24}}\right) \text{ for } n > 2$$

$$P_2 = \frac{1}{2^{24}}$$

After 4823 packets there is a 50% collision chance. We can use pattern recognition to disentangle the plain text and the ICV will tell us if we have the right plain text

- After only a few hours we can recover all 2^{24} IVs. We can accelerate this process by sending spam into the network

Public Key Infrastructure

- Trusting public keys belong to who they claim they do is hard. So here we push that burden of trust to Certificate Authorities.

- "Trust" here means that the CA is responsible for binding information about an entity with a public key.

- The first PKI standard was X.509 in 1998. The IETF X.509 is in RFC 5280 from 2008 and is the main one used.

- X.509 is old and is disliked by many but no new standard has been sufficiently better to replace X.509.

- Certificates are defined as follows:

```
Certificate = Sequence {
        tbsCertificate,
        signatureAlgorithm,
        signatureValue }
```

Defined using Abstract Syntax Notation (ASN.1)

The tbsCertificate consists of:

- Version (Default is 1)
- Serial Number
- Signature (The Algorithm Identifier)
- Issuer
- Validity
- Subject (the name of the entity)
- Subject Public Key Info
- Issuer Unique ID
- Subject Unique ID
- Extensions

- Occasionally certificates go bad and the CA has periodically issue a Certificate Revocation List.

These lists can be fetched using the Online Certificate status protocol (OCSP).

Some resons to revoke a certificate include, Key Compromise, Key Loss, Change of function ect.

- To check a certificate we start with a set of locally trusted CAs. We progress down the certification path checking if the signature is Ok and the certificate hasn't been revoked.

Fully outlined in RFC 5280

- There are many PKI protocols:

Registration + Renewal: ACME.
Certificate Retrival: LDAP, DAP, HTTP

Certificate Checking: OCSP, DVP
Certificate Transparency: Used by CAs to log every certificate they issue

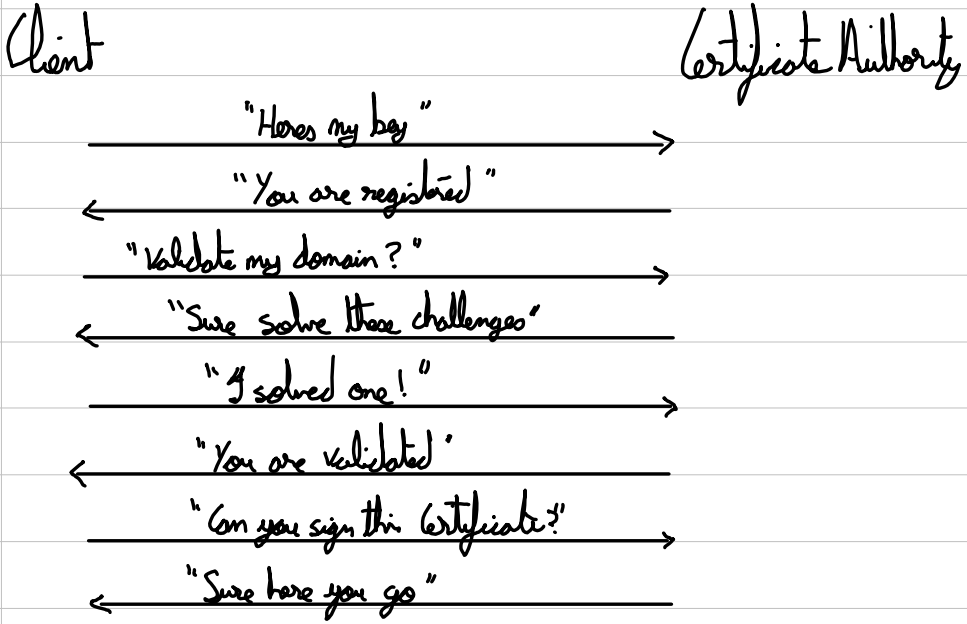
- Applications using the PKI need to have a local set of trusted root CAs. Browsers and OSs have those. Often, inclusion/exclusion from these lists can be politically motivated.

- CAs usually operate a for-profit model however "lets encrypt" has been operating as a free CA since December 2015.

- Certificates issued usually only last for 90 days which forces people to automate the process. This can be done using the ACME protocol. It is JSON based and outlined in RFC 8555

- The Automated Certificate Management Environment (ACME) is a HTTP-based protocol for managing certificates.

Typically this is set up as a chron job. The protocol is as follows:



- The ACME challenges are there to prove you own the domain. It might ask you to add some random content at a page on your domain.

- There are cases where CAs have been hacked and issue shetchy certificates so now all CAs are required to log all certificates they issue. This allows people to monitor the activity of CAs. This is called Certificate Transparency.

- New Post Quantum algos will still need Public keys in certificates. Some people are proposing ways to embed 2 keys into one certificate to support the use of hybrid protocols.

DNS

- The DNS is structured as a tree with a single root, below which we have .com, .ie ect and below

those we have example.com, ted.ie and below that we may have more pages eg. down.ted.ie ect.

- The main purpose of the DNS is to map human memorable names to IP addresses

- IPv4 uses the A resource record and IPv6 uses the AAAA resource record type

- The DNS can also be used for the right hand side of the email to the mail server name MX records.

The DNS can have block lists for spam sources. It can also be used for state/enterprise level censorship

The DNS can also have "passive taps" where we monitor the DNS queries for security purposes.

- ted.ie is a domain name. www.ted.ie is defined as a Fully Qualified Domain Name.

ted.ie is the parent of cs.ted.ie while cs.ted.ie is the child of ted.ie.

- We assume all the children of ted.ie will be managed by TCD. This is refered to as managing domain in their "Zone". Sometimes it is managed by a "Zone file".

The zone file will have keys and signatures for DNSSEC.

- The DNS starts at the root.

Then we have Top level Domains:
Country-Code: .ie, .uk ect.
Generic : .com, .net ect.

Usually have to comply with some restrictions

Below that is 2nd Level Domains or effective Top Level Domains:

eg. amazon.com, ted.ie

The .com zone has 160M names, .ie has ~330k names, .org has ~10M names.

The 3rd Level Domains are managed by the 2LD
eg. down.dsg.cs.ted.ie ect.

- Some TLD don't have 2LDs directly below the TLD ie .co.uk, .com.au ect.

These can cause issues for browsers when deciding to re transmit cookies for example.

The solution was to maintain a file called the "Public Suffix List" that contains a list of 2LDs that have the issues mentioned above eg. oxcon.co.uk.

- The PSL would ideally be maintained via information in the DNS and any attempts to do this have failed to do so (IETF DBOUND wg)

- The TLDs are operated by Registries

  IEDR is the Registry for the .ie TLD
  Public Intrest Registry operates the registry for .org.

- Registrars are accredited by registries and deal with the registration of names.

- The Registrant is the entity that wants a name to be registered.

So a Registrant goes to a Registrar to register a domain name. The Registrar checks with the relavent Registry to see if the name is valid. If it is valid then the Registrant pays money to the Registrar to

register the domain. The Registrar gives a portion of that money to the Registry.

- Registries are the entities that deal with name conflicts via some dispute resolution process.

- ICANN occasionally auctions new Generic TLDs (gTLDs) but it costs a few million to buy these new TLDs.

- There are protocols that allow Registars to talk to Registries eg. EPP or RDAP.

whois is a legacy protocol where a registry publishes some registrant data.

- A logical Zone file for a domain is served by an Authoratative DNS Server.

Typically we have 2 or 3 Authoratative DNS servers for redundancy

- The DNS protocol has Zone transfer commands eg AXFR that help to sync multiple name servers.

- Recursive DNS servers query the Authoratative to resolve names.

- The protocol spoken between clients, Recursives and Authoratative servers is the DNS protocol

- The root Zone "." is special, its contents is carefully managed by IANA and is handed over to root server operators

  Most root Zones instances are accessed using anycast IP routing. There are about a 1000 different instances worldwide managed by 12 organisations.

- Every Recursive needs at least one root server IP to start. So these root zone instances need to be stable.

- A recursive DNS Resolver with an empty cache will first ask a root server for the .ie TLD, then it will ask the TLD for the 2LD of example.com. Then it will ask the 2LD for www.example.com.

- DNS queries are sent by default via UDP to port 53.

The query contains QNAME, type, class, 16-bit ID, source port.

The answer contains QNAME, type, class, id, RR set additional data

Since its using UDP technically anyone can answer.

This is called DNS Poisoning where an attacker can flood a server with DNS answers that have random IDs with an IP pointing to a bad site in the hope that one of the IDs match with a query and the server will serve the bad IP.

- DNSSEC was the solution to prevent DNS poisoning among other attacks.

DNSSEC

- The goal of DNSSEC aims to have data integrity and origin authentication that is built upon the existing DNS Infrastructure

- To implement DNSSEC we need to create new Resource Records (RRs) and we want the DNS answers to have a digitally signed chunk of the Zone file.

  We also need to setup a "secure Zone" file which contains digital signatures and public keys.

- Vulnerable points in the DNS can be inbetween the authoritative server and the recursive resolver.

- DNSSEC protects man in the middle attacks via TSIG.

- DNSSEC provides message authentication and integrity verification through signatures. It does not provide authorisation or confidentiality. It also does not protect against DDOS attacks.

  Authorisation was avoided because it could allow for easier censorship

- DNSSEC makes use of PKC over the DNS.

  Private Keys are stored locally.

  Public Keys are published in the DNS as a DNSKEY resource record

  Signatures are published in the DNS as a RRSIG resource record

- Signing is done per Zone. Each Zone will have one or more key pairs for signing. Typically 2-3 key pairs.

If we have the public keys from the Zone, we can validate signatures made with the corresponding private key.

However signing a complete Zone does not scale well

- RRs are usually grouped by the same name, class and Type. This creates an RRset. Usually we sign the RRset rather than the individual RR.

RRsets are the atomic data unit in the DNS

- The DNSKEY RData is structured as follows:
  - 16 bit flags
  - 8 bit protocol number
  - 8 bits to indicate algorithm
  - N*32 bits of public key

- The RRSIG RData is as follows:
  - 16 bits - type covered eg A record, AAAA ect
  - 8 bit - algorithm
  - 8 bits - nr labels covered
  - 32 bit - original TTL
  - 32 bit - signature expiration (about 1 month)
  - 32 bit - signature inception
  - 16 bit - Key tag
  - Signer's name
  - Signature bits

- Each 2LD signs their zones and give their public key to their parents (TLDs), the TLD will publish a hash of the 2LD's public key. The TLD will then sign its zone and will give their public key to the root. The root will then publish a hash of the TLD public key.

This is the delegation of signing authority and this is how we build the chain of trust.

- A new DNS record is created "DS", which is a hash of the public key of the child.

- The chain of trust is built by using one trusted key to verify the authenticity of other keys. We build this chain from the root down. For this to work Parents need to sign the keys of their children.

- The "Delegation Signer" record (DS) indicates that the delegated zone is digitally signed and that the indicated key is used for the delegated zone

The parent is authoritative for the DS of the child zone. The DS should not be in the child's zone

- The DS RData is as follows:
  - 16 bit - key tag
  - 8 bits - algorithm
  - 8 bits - digest type
  - 20 bytes - digest (usually SHA 256)

- Interaction with the parent for administration can be expensive. The solution is to use 2 keys.
  - Key Signing Key (KSK)
  - Zone Signing Key (ZSK)

The key contained in the DS signs a DNSKEY RRset. A validated signature transfers trust to all the keys listed in the DNSKEY RRset. These keys then sign the Zone.

The actual KSK is in a DNSKEY RR.

The DNSKEY RRset contains the Zone Signing Key (ZSK) while the DS contains the hash of the KSK

- The ZSK needs daily or permenent use while the KSK is used less frequently. ZSK change needs no involvement with 3rd parties. Changing the KSK might.

- CoT verification:
  - Data in the zone can be trusted if they are signed with a ZSK.
  - ZSKs can be trusted if they are signed with a KSK.
  - KSKs can be trusted if pointed to by a trusted DS record.
  - DS records can be trusted if it is signed by the parent's ZSK or if those records are exchanged out-of-band and stored locally.

  This propigates all the way to the root.

- One issue with DNSSEC is that if a user asks for a non-existant domain, how can we provide proof that the answer doesn't exist? This is called Authenticated Denial of Existence.

- One solution to this is the use of the NSEC resource record.

This points to the next domain name in the zone and also lists what are all the existing RRs for "name".

NSEC RR provides proofs of non existence. It also allows for zone enumeration where we can find all the names under a domain.

- NSEC3 solves this by creating a linked list of the hashed names.

A Non existence proof on the list of hashed names proves the non-existence of the original.

- Dependency on the parent for the DS record means that DNSEC is hard to deploy.

The registrant is the one with the key pairs. But they deal with registrars.

If the burden of key publishing falls on the registrar, how does the zone get signed and how does the KSK get from the registrant to the registrar.

If instead the registrant publishes their keys to the registry, how does the registry know it's dealing with the right party?

- There was also a large delay in getting the root zone signed.

- DNSSEC adds new things to manage (the RRSIG). If the RRSIG expires, then the zone effectively gets erased because it will never get resolved by servers that do DNSSEC

- CDNSKEY provides a way for zone maintainers to publish a new DS record or a new KSK to their parents. The parents occasionally poll the children for a new DS record.

- KeyTrap was an attack developed for DNSSEC. The basic idea is to publish an "attack" zone with colliding Key IDs and many bad signatures and keys.

The resolver will spend hours at 100% CPU trying to find a good signature.

To execute, your submit a query to the target resolver, the authoritative forces query to TCP and encodes hundreds of sigs/keys in one 64K DNS message.

Most DNS servers have issued mitigations.

DNS Privacy

- Since all data published to the DNS is public there really wasn't any need for privacy.

- But people soon realised that accessing the DNS could reveal information that you want to keep private eg. accessing an Alcoholics Anonymous site

- RFC 9076 documents DNS Privacy Considerations

- Some Mitigations include:
  - Using a Tor Browser
  - QNAME Minimisation (RFC 9156)
  - Define ways to provide confidentiality for DNS traffic (DoT/DoH/ADoX)
  - Don't always send EDNS(0) client subnet

- DoT is DNS over TLS and was defined in RFC7858. DoT is usable between the stub and recursive.

We replace the system stub resolver (systemd, dnsmasq) with something that can do DoT (stubby, unbound)

There are public recursives who offer this DNS privacy service eg. 1.1.1.1, 9.9.9.9 ect.

- DNS messages could still leak some name information via their lengths. This is where DoT with padding comes in.

The Responders must pad if the requesters pad.

RFC 8467 describes ways in which one might use padding. It recommends:
  - Padding queries to lengths of N x 128 octets
  - Pad responses to length of N x 468 octets.

- We would still like to secure the path between the recursive resolver and the authoritative. This is where Authoritative DNS over TLS comes in

There are some performance issues that comes with setting up and teaming down TLS sessions between an authoritative and a recursive. It also isn't clear how to authenticate the authoritative servers.

Opportunistic ADoT can be deployed (RFC 9539). Essentially they can experiment with ADoT but if it fails it can seamlessly revert to normal DNS.

- Browsers can't really tell if DoT is being used. So DNS over HTTPS (DoH) was developed and explains how to encapsulate DNS traffic in HTTPS. This is described in RFC 8484.

- DoH moves from a system/os stub to an in-browser stub and causes many changes. Essentially network admins can't block outbound HTTPS connections, so users can then bypass the local recursive resolvers

- Once we get DNS Privacy then we can try to tackle SNI (Server Name Indication) encryption as part of the TLS Handshake.

  The idea is to publish a new DH public share in the DNS and use that to encrypt the SNI in the TLS ClientHello.

- Browsers like things to be fast. But DNS and TLS add RRTs to the process. So the HTTPS Resource Record (RFC 9460) was introduced so that we can bundle more stuff for TLS into one RR to reduce on RTTs.

  This can also solve the "CNAME at apex" issue for CDNs.

SVCB is a generalisation of HTTPS. HTTPS and SVCB RR definitions are extensible allowing new keys and pair values to be defined.

- There is discussion currently to see if we would benefit from having a SVCB-like RR higher up the tree.

This could possibly make it easier to handle Name Servers, DS RRs, ADoT and other things that need a "Zone-cut".

There is an idea to make DNS operators into "first-class" citizens like registrants, registrars and registries since more people outsource DNS operations.

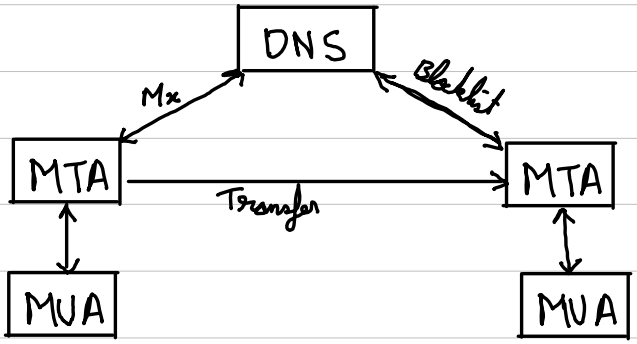
This would be handled through a new RR called the DELEG RR.

Email

- The email architecture is laid out in RFC 5598.

Typically we have a Mail User Agent eg. Thunderbird, Gmail client ect. It is used to view, compose and send emails

We then have a mail server which stores and receives and sends emails to other mail servers. One server is where we send email to and another is where we receive from. Often these are the same. This is also called a Mail Transfer Agent and it sends the mail to the receiving MTA.



The sender MTA queries the DNS for the IP of the receiving MTA. This is stored in an MX RR.

The mail is sent using the SMTP protocol. The receiving MUA uses the IMAP protocol to read mail stored on their MTA's message store.

Because of spam the receiving MTA can query the DNS for a blacklist to see if the sending IP is a known scammer. It can then decide to drop the email if it is a scammer.

- Since very little of the DNS is digitally signed it is very easy to spoof the Mx records. So an attacker could give a bogus Mx record and see all the mail for the intended recipient.

- The MTA will also check if it has the recipient in its domain. If not, it usually bounces the message. If the recipient is in the domain, the mail is stored in a message store.

- The message store usually saves all the user's email, this allows one user to have multiple MUAs. But now the message store becomes an attack point for attackers.

- The MTAs can also scan outbound and inbound email attachments for malware. It can also check if the email is spam.

- When performing SMTP and IMAP, we want to do them over TLS. This is called Mail Transport Security.

However is isn't really end-to-end, we only do TLS between the hops and the message. Securing the hop between MTAs is a lot harder than securing the transport between MTAs and MUAs.

- Between MTAs we perform opportunistic SMTP over TLS.

Here the sending MTA looks up the Mx of the recipients DNS domain.

The Mx name might not be the same as the recipients domain name. Eg. @tcd.ie emails is managed by msft/outlook.

The sending MTA is the TLS client and the receiving MTA is the TLS server

The issue is that many receiving domains (eg ted.ie) outsource their MX to someone big and good at anti-spam (eg. Google or Microsoft) meaning the sending MTA has to deal with the MX names and can't insist that the MX be able to serve the TLS certificate of the recipient domain.

Essentially we get strong encryption because of TLS but not strong authentication since we can't really say who we are sending to.

- There isn't really any form of Web PKI for mail. So most Mail Servers (MX) provide a self signed X509 cert as part of the TLS hand shake. Each MX essentially behaves as their own root CA.

- To improve on opportunistic SMTP over TLS, MTA-Secure Transport Security (RFC 8461) was introduced.

- MTA-STS was modeled on the web version of HSTS. MTA-STS defines a way to publish some information on a web server that allows sending MTAs to know what to expect when creating a connection with the receiving MTA.

The sending MTA can look this up and look for this information before starting to connect.

- This protocol has a "testing" phase which defines how sending MTAs can report errors and fallback to the previous protocol if MTA-STS fails.

This is useful since the whole philosophy of email is deliverability over all else.

Email Spam

- Spam is bad mainly because of resource consumption, malware and can contain phishing attempts.

- Spams can be injected into mailing lists or it can be sent via an open mail relay. (eg. tead.com)

- Spam emails use many tricks such as replacing an "l" with "1" in some fonts these look identical. Some spam replaces ascii letters with similar looking cyrillic letters.

Some spam uses BGP Spoofing or DKIM replay. They can also play around with 0 font size or 0 width characters

- Anti Spam is very hard to do unless you are at scale eg Google / Microsoft. They would be more likely to notice spam campaigns

- Spam is a very hard problem to solve since the prime directive of email is deliverability and the ability for anyone to send an email to anyone.

- One Anti-Spam technique is Grey listing. This is where a receiving MTA will bounce a mail as if it had a temporary failure when it receives some mail from a sending MTA it hasn't seen in a while.

  Most spammer won't resend an email to deal with the bounce.

- Some other techniques include Sender Policy Framework (SPF), Domain Keys Identified Mail (DKIM) and Domain-based Message

Authentication, Reporting and Conformance (DMARC).

- In SPF, the sender MTA will publish a set of IP addresses in the DNS that it will send mail from. A receiving MTA can query the DNS and check the source IP of incoming mail. This has become less useful as MTAs became centralised.

- In DKIM the sending MTA digitally signs outbound emails with private keys associated to the sending domain. The receiving MTA looks up the public key in the DNS and checks the signature. A missing signature or a failed check means we treat the email as an unsigned email.

→ Public Key is stored in the DKIM RR.

We never reject an email due to a bad signature we only increase its "spam score". If this reaches a threshold (~5) it is treated as spam.

- In DMARC the sending MTA publishes a "policy" in the DNS stating how it would like receiving MTAs to treat messages from them eg. requiring SPF or requiring the DKIM check to "pass" before accepting the email.

The DMARC policy can even tell receiving MTAs to reject an email if the checks fail.

DMARC is great for transactional mail (eg receipts) but is bad for mailing lists since mailing lists will almost always alter the email in a way that breaks the DKIM signature.

- There is a solution for DMARC's downside known as ARC (Authenticated Received Chain)

This allows intermediate processing MTAs to sign the original messages' validation results. Receiving MTAs can then verify this ARC chain. It tends to fail after a lot of forwarding

E2E Email Security

- We want to encrypt emails end-to-end, ie from the sending MUA to the receiving MUA.

  This will cause a lot of antispam checks that happen at the MTA to break.

- PGP (Pretty Good Privacy) was a way to perform E2E.

- S/MIME is a solution that tends to be deployed in enterprise networks since it is easier to deploy.

  The Cryptographic Message Syntax (CMS) is the basis for S/MIME and other various crypto applications using ASN.1

- S/MIME consists of CMS, a Message Specification and a certification specification.

- CMS defines how to MAC, sign and or encrypt application data in an ASN.1 oriented way.

  The algorithms and options are the same as X509. But to encrypt we need some "glue" between the arithmetic (for RSA) and ASN.1 Bit Strings. We often use the PKCS#1 v1.5 format

- With RSA we need to pad our data, in PKCS#1 v1.5 we use random non-zero padding octets.

  The first two bytes show whether we are signing "0001" or whether we are encrypting "0002"

- The Message Specification of S/MIME tells you how to:
  - Start with a MIME message
  - Treat that like plaintext the CMS way
  - Then take the resulting bytes and make them into a Mime message

This essentially tells you how to convert an email made of multiple MIME messages into an Octet string, then give it to CMS to encrypt and or signing, then converting the output back into a set of MIME messages to be sent as an email.

It also defines the reverse process (how to go from an encrypted email to plaintext)

- The Certificate specification tells you how to put an email address in an X509 certificate.

S/MIME + PGP only encrypt message body, not email headers.
We also need to unify S/MIME and PGP otherwise we will lose interop.
It also needs to be transparent to users if they will use it.

- PGP is an alternate scheme that can do everything S/MIME does.

PGP's basic format is in RFC4880.

- PGP can operate on a web of trust model where a bunch of people can sign that an email belongs to a person. This doesn't scale well.

PGP can be used for package signing

PGP key servers used to exist but their usability was horrible. Key management could also be based on the web of trust model.

- Some barriers to E2E need all major mail box providers (yahoo, gmail ect) to work together.

S/MIME and PGP also assume that each account only has one MUA. But now people

Encryption also breaks searching for emails in the MTA. We could store them in clear in the MTA, but now MTAs become an attack target

In S/MIME, the private key is usually stored in a specific MUA rather than the MTA, so users can only have one MUA, otherwise they have to deal with moving the keys.

can have one MUA on their phone, another on their laptop ect. So we also need a way to manage the private keys.

We also need a scalable way to retrieve the public key of the recipient ect.

- If we can solve the issue of E2E in instant messaging between different apps we can also solve it for email

Solutions are to store priv \ pub keys on gateway servers but this only works for large businesses.

S/MIME + PGP break malware + spam checking. Offloading this to users.

Transport Layer Security [TLS]

- The Secure Sockets Layer (SSL) was proposed by Netscape in 1994. This works with any application that uses sockets.

- This was then standardised as TLS in 1999.

- The latest version is TLS 1.3 written in RFC 8446. [You need to know and read this RFC throughly for the exam].

TLS 1.2 is standarised in RFC 5246

SSL and TLS was brought in for the advent of internet based shopping.

- SSL offered these services:
  - Server Authentication: The "buyer" can believe they are dealing with a bona fide "merchant" It can also support X509 but it is rarely used. SSL doesn't really get used for mutal authentication on the web.

  - Message Encryption, Message Integrity and Replay-Detection

  - It was relatively transparent to the user

and the app developer only had to change some small things.

- One benefit of SSL being at the transport layer is that now browsers and apps that use sockets can know if the TLS handshake failed.

- TLS is now used everywhere on the web. TLS/TCP on port 443 is now a defacto baseline for Internet-scale interop.

- TLS is broken into 4 interrelated sub-protocols
  - Handshake
  - Application Data protocol } ⇒ 2 most important ones
  - Alerts
  - Change Cipher

Extensions such as Server Name Identification allowed you to talk to a specific server. Max fragment length could be negotiated with extensions.

Quantum keys can be distinguished in extensions but this breaks some TLSv1.3 servers who don't expect large Client Hello messages. Demonstrated when Chrome experimentally turned on Kyber in 2024 April

- TLS v1.2 has a handshake Protocol

1) The client begins with a "Client Hello" that contains all the compression methods and ciphers it can use. It will also send a random number and a session ID. Compression is applied before encryption.

2) The server returns a Server Hello with a random number for key derivation and a chosen compression and cipher method.

3) The server will then send its X509 certificate. The client should verify this certificate

4) The server sends a Server Hello Done.

5) The client will generate a Pre Master Secret key and uses the RSA key transport protocol to send the key to the server in a Client Key Exchange message

→ Encrypted using Public Key from X509 cert. DH key exchange can also be performed to agree on Pre Master Secret.

The DH values can be ephemeral or can be contained in the server certificate

Post Quantum key exchange could be done using SIKE or Kyber. Keys generated could be used with AES or SNOW 3G

→ SIKE was broken in 2022

↳ This message is encrypted

6) The client sends a ChangeCipherSpec message that tells the server to turn on encryption for all subsequent messages.

7) The client then sends a Finished message. This will contain a hash of the transcript recorded on the client side. The transcript is a record of all the packets sent and received by the client. This gets encrypted and sent to the server.

The server decrypts this message and computes the hash of it's own transcript and checks that these two hashes match. If they don't, then someone has interfered with the packets.

8) The server then sends back a Change Cipher Spec message.

9) It will send an encrypted Finished message (similar to step 7). The client then verifies this like in step 7.

10) Finally we can encrypt all Application data

- To Compute Keys from the Pre Master Secret, we first compute the Master Secret: (48 bytes)

Master Secret = f(Pre Master Secret, Client random num, Server random num)

- The Master Secret is used to prime the key generator:

Key Block = f(Master Secret, Client Random number, Server Random number)

The Key block is split into 6 keys:

Client Mac Secret
Server Mac Secret
Client Key
Server Key
Client Stream Key
Server Stream Key

- To apply the keys to the Application Data, we divide the stream into blocks ($\sim 2^{14}$ bytes).

We then compress this fragment, compute the MAC, then we pad it. We can then encrypt it using a block cipher (with padding) or a stream cipher.

MAC is provided by HMAC using the SHA-256 hash function

The PRF is used for key generation + expansion

The TLS PRF = P_hash (secret, label + seed)

P_hash = Σ HMAC (secret, A(i) + seed)

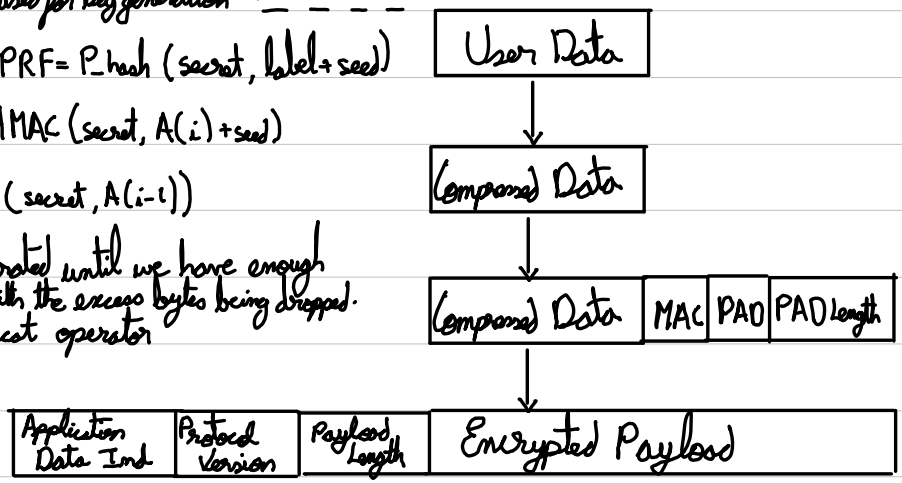
A(0) = seed

A(i) = HMAC (secret, A(i-1))

P_hash is iterated until we have enough output data with the excess bytes being dropped.

Σ is the concat operator

Encryption is performed using CBC or AEAD ciphers (AES-GCM)



- TLSv1.2 supports various cryptographic options such as digest algorithms, key transport ect. It was decided to represent all configurations with a 16 bit number eg. TLS_RSA_WITH_3DES_EDE_CBC_SHA

- The padding in the TLS protocol lead to the Bleichenbacher attack being developed.

The PKCS#1 v1.5 padding adds some formatting to the data

PKCS#1 adds padding to the message before RSA encryption is applied! The message "m" becomes:

$$x = 0x00 \| 0x02 \| \text{padding} \| 0x00 \| m$$

the RSA ciphertext is then $c = x^e \bmod N$

The attack works on the basis that if we have an "oracle" that will attempt decryption and tell me when the "recovered plaintext" has PKCS#1 padding, then I gain some knowledge. Turns out TLS can provide this information either with an explicit message that the padding was correct or a longer time between messages indicating decryption.

↑ This is done through an error (don't think Bleichenbacher sent decryption has failed)

- The attacker will generate a value "s" and sends as a Client Key Exchange $c' = cs^e \bmod N$. Most of the time the result of the decryption is not PKCS#1 compliant. Eventually there will be a value "s" such that $[cs^e]^d \bmod N$ is PKCS#1 compliant.

If c' is PKCS#1 then the attacker knows

$$2(256)^{(k-2)} - 1 < ms < 3(256)^{(k-2)}$$

After a few million connections the attacker will learn enough to pinpoint the exact "m" yielding the Pre Master Secret.

- RFC 7457 documents all known attacks on TLS and Datagram TLS. RC4 attacks only need 2^{26} session for attack. RC4 should not be used

- Some issues with TLS:
  - Weak key Generation
  - CA screw ups
  - Bleichenbacher
  - CRIME
  - Lucky-13
  - DROWN
  - Marvin

- Many issues have been found over the years due to its widespread use. Still it is far easier to use TLS properly and get security than to reinvent TLS.

- NetScape when generating keys messed up bad meaning the TLS PreMaster Secret could be guessed. The 128 bit key could be guessed in about 2^{40} tries

- CAs also screwed up when VeriSign issued "code-signing" certs to someone that was not microsoft.com. Allowing the attacker to sign malicious windows updates as microsoft. VeriSign had to revoke the cert.

- The Bleichenbacher attack was another attack used to gain the PreMasterSecret. Bleichenbacher still keeps returning due to the fact the PKCS#1 padding is implimented in a lot of hardware

- TLS also had a Re-Negotiation Bug. This was a protocol error rather than an implementation error.

This allows users to renegotiate TLS sessions using a session ticket from the previous session. The issue was that the new session is not cryptographically bound to the previous session

The fix is in RFC 5746 but was a breaking incompatible change with previous versions.

- Diginotar was another CA on the web and was in the browser root stores.

Diginotar got hacked and the attacker issued a bunch of certificates for google and microsoft.

This was found out and Diginotar was kicked from the browser root store.

The fix was to create the CAA resource record in the DNS where a company can say which CAs they will use to create their certificates.

CA transparency lists was also introduced as a response to this.

- Comodogate was another CA screw up. A registration authority under Comodo got hacked and the attacker issued a bunch of certificates for google and microsoft and it was approved by an automated workflow in Comodo

- A company in 2012 built a database (10^{16}) of public keys on the internet. They found that some keys had common factors which is bad for RSA keys. They think that it is down to embedded systems which generate key pairs on boot when they do not have enough entropy.

Essentially the very 1st key pair generated would likely have a common factor with another key. But subsequent keys would have more entropy.

- Stuxnet and Flame were attacks perpetrated by America and Israel.

- CRIME is an attacks that demonstrates that compression in TLSv1.2 was bad.

Assume we have a browser that connects to bank.com. The TLS session has compression enabled and then we get a cookie associated with that session.

This cookie gets sent with every HTTPS session.

An attacker can guess octets of the cookie and can see which packets get compressed more. This indicates the guess shares an octet with the cookie. The attacker can repeat this to guess the cookie.

The fix was to turn off compression in TLS v1.3. Also to not mix sensitive data such as cookies and attacker controlled data in the same compression context.

- Lucky-13 was a timing attack on the CBC method of doing CBC Residues. The fixed was to use AEAD ciphers.

- Starcom was a CA that was sold on the quiet to WoSign which was another CA that had some shady practices. WoSign couldn't justify these practices to the browsers so it was dropped from browsers store. (Dropped in 2016)

Certificate Transparency was deployed at this time which helped to kick WoSign out of the browser stores.

- Bleichenbacher returned in 2017 in the form of ROBOT. This demonstrated real issues in deployment. The fix was to get rid of RSA key transport completely in TLSv1.3

- Marvin is the latest (2023) version of Bleichenbacher. This time it was based on more accurate timing measurements and better statistical techniques. Marvin demonstrated that this could be done remotely.

- Vendors sell a lot of TLS man in the middle products. Browsers can sometimes detect this but won't tell the users. This is mainly used in corporations as it allows them to perform inbound malware scanning

- TLS is imperfect in many ways:
  - There are web PKI and other imperfections
  - It supports to many ciphersuites (often they have weaknesses)
  - Significant changes takes years to implement

TLS v1.3

- TLS v1.3 is layed out in RFC 8446. Pay attention to appendices C, D and E. It took about 4 years to get done.

- The change was motivated by all the attacks on TLS v1.2 and was a major change over its predecessor.

- Changes include:
  - Dropping less desireable algorithms and moving to AEAD everywhere.

- Changes how ciphersuites gets defined. Also included recommended suites.
- Added a "0-RTT" mode
- RSA key transport was removed and all key changes provide forward secrecy.
- More encryption of handshake including some extensions.
- ECC is now built-in.
- No more compression or custom DH groups
- Pre-Shared keying, tickets and session handling all done in one way.
- PKCS#1 v1.5 is used for certificates. But RSA PSS are used for protocol signatures.

→ Elliptic P256, ED25519 curve25519

- TLSv1.3 allows for a "1-RTT" handshake where we can set up a TLS using only 1 RTT. This works because the client can guess which ECC curve the server will use.

If the client guesses the ECC curve wrong the server sends a Hello Retry Request which informs the client of the correct curve. Everything else can then continue as normal.

- The "0-RTT" mode allows a browser to send early data to a server it has already talked to before. This allows the browser to avoid an RTT. This is the only reason that browsers implemented TLSv1.3.

- In all the above cases we still do Ephemeral ECDH even though we have a pre-shared secret with the server.

- "0-RTT" is a dangerous feature. The issue is that servers are vulnerable to a "replay attack".

An attacker can record a 0-RTT message that includes the early data.
They can then replay this message against another instance of a load-balanced server eg. in another data centre where load-balanced instances can't easily share an anti-replay cache.

The early data is not authenticated until after the server validates the client finished message. It has no way of knowing if the early data is legitimate or a replayed message.

The fix is to not act on early data until after the Finished message is checked. We can also limit the scope of the reuse of session tickets

- Ciphersuites in TLSv1.3 has been refactored. The TLSv1.3 only reflects the record layer encryption (bulk cipher and key derivation function and hash function) and not the key exchange since we know we are going to do ECDH.

- There is a sound key derivation process in TLSv1.3 backed by some security proofs.

- Middle boxes break things with TLSv1.3 so it pretends to be TLSv1.2 in many ways. There is a "supported_versions" extension will tell the end user that it is a TLSv1.3 packet while the packet version number will say TLSv1.2. Only a TLSv1.3 server will look at this extension while the middleboxes will look at the version number in the Client_Hello. Same thing for the Server_Hello.

The "change_cipher_spec" message from v1.2 is not needed since after we finish the DH key

exchange, we begin to encrypt things. But we send a dummy message to make the traffic look more like v1.2.

The HelloRetryRequest in v1.3 pretends to be the ServerHello message from v1.2. Only some magic values distinguishing it to be a HRR.

All the Record Layer messages (Application data) pretends to be TLSv1.2.

- There are many extensions in v1.3:

  cookie - helps with DDoS and DTLS

  post_handshake_auth - is how TLS client Auth is handled in v1.3.

  psk_key_exchange_modes and pre_shared_keys - when using psk.

encrypted_extension - used from server to client

- The Record layer in v1.3 makes use of AEAD and differently derived keys but has the same max record size (2^{14} octets) and same external headers.

- Forward Secrecy is not absolute. v1.3 attempts to provide FS wrt long term private keys but DH public value reuse for performance reasons can result in less than perfect FS.

- v1.3 attempts to confidentially protect identities which is new. Server identity protection cannot resist active attacks.

- Separation between key purposes is much more deliberate and for less ad-hoc than earlier TLS versions