13 Security and Trust Part 1

13.1 Goal of this section

The goal of this section is for readers to understand the five goals that network security attempts to achieve and to understand some basic ways that we go about achieving those. Readers should also come away from this section with an uncomfortable feeling that network security is complicated, easy-to-screw-up, and needs to be approached with care – but also that it is understandable and definitely not magic.

13.2 The golden rule of security and the rule of thumb.

Security is hard to get right, and in particular it is a holistic problem with a human component – if we don't make security automatic or trivially easy to use, humans make the trade-off for usability over security every time. So we need to design it in from the start.

However, when building things, humans want to solve problems and make things work, and while we know we should do things properly™, we usually whack something together that works first, and worry about edge cases and bad actors later. That was kind of acceptable back when the internet was being invented. We know better now.

Golden Rule…

Always build security in from the ground up. It cannot be successfully tacked on later.

…Rule of Thumb

Security is never built in from the start. It is always added later. And it's usually a bit rubbish.

13.3 The 5 Goals

When we talk about network security, there are 5 different things that we want to achieve for our network and communication within it. We'll cover each in its own section below.

Confidentiality

A sender can send a message to a receiver in a manner that malicious third parties cannot understand.

Authentication

A receiver can verify that a message that claims to be from a sender is indeed from them and not a malicious third party.

Non-repudiation

A receiver can prove publicly that a message was sent by a particular sender.

Integrity

The message that the sender sent is exactly the same as the message that the receiver received.

Availability

The network is available to be used by senders and receivers at all times, even when other malicious actors attempt to prevent this.

13.4 Confidentiality – Cryptography

Cryptography is the science of encoding messages that you want to send with some secret information, such that it is not possible to understand without the secret information. If you can do that, then you can broadcast the encoded message to *everybody*, but only your intended recipient with the secret can decode it and understand your message.

How do we achieve this?

Mathematically, we're looking for an encryption function E(x) and a decryption function D(x) E(x) and D(x) that will operate on a **plaintext** (unencrypted) message x to produce **ciphertext** (the encrypted message) E(x) and decode it again. We're looking for functions with the properties that

- For all messages x, D(E(x)) = x
- For any given message x, it is hard to calculate x from E(x).
- For any given set messages x paired with their cyphertext E(x), it is hard to calculate D(x) or E(x).
- For any x, it is cheap to calculate E(x), and for any E(x) it is cheap to calculate D(E(x)).

13.4.1 Symmetric Keys

Symmetric (also called secret) key cryptography has encryption and decryption functions that work off a shared secret key. The main advantage of symmetric encryption is that there are a bunch of computationally cheap and algorithms for en/decrypting data. The disadvantage is that you need to somehow share the secret between both endpoints.

13.4.2 Asymmetric Keys

Asymmetric key cryptography has encryption and decryption functions that work off a pair of keys. In most cases, the encryption and decryption functions are the same function/algorithm, F that involves a key, and the pair of keys (d, e) are chosen carefully such that $F_d(F_e(x)) = x$. This solves the problem that you need a publicly shared secret, but Asymmetric key cryptography is usually much more computationally expensive.

Example: Diffie-Hellman

Diffie Hellman Key Exchange (named after the inventors Whitfield Diffie and Martin Hellman) is an elegant *Asymmetric* way for two parties to generate a shared secret that they can then use with a symmetric key cipher.

Both actors, A and B, agree in advance on a public (not secret!) generating element g for a finite cyclic group. Typically this is multiplication modulo p for some large prime p, and then g is some p-1th root of that group (such that $g^{p-1} = 1$).

Both A and B chose and publish p and g. For example: "We're doing arithmetic modulo 37 and using 2 as our generator."

A and B each then *secretly* choose a power a and b, and calculate g^a mod p and g^b mod p. They then publish those. In our example, A chooses 3 and B chooses 5. A: "My special number is 8" , B: "My special number is 32".

A and B then secretly calculate the value that is the other person's published number raised to their *secret* power. These will be the same (because $(g^a)^b == g^{ab} == g^{ba} == (g^b)^a$) and become their shared secret. In our example: A: 32^3 mod 37 = 23. B: 8^5 mod 37 = 23. This secret remains secret, because while modulo multiplication is relatively cheap (going from (g, a, p) to g^a mod p), deconstructing a from g^a and p is expensive – effectively requiring a brute force of every possible value.

Having obtained a shared secret, A and B can use this as a key in the symmetric key algorithm of their choice to encrypt data between them. Should they wish they can also use Diffie-Hellman to generate a new key.

Example: Rivest-Shamir-Adleman Encryption

RSA encryption (named after the inventors Ron Rivest, Adi Shamir and Leonard Adleman, though equivalently secretly invented by Clifford Cocks for GCHQ 5 years earlier) again works using modular arithmetic and is based on the fact that factoring large numbers is provably hard™. It has the useful property that having generated a pair of keys, you can make one public such that

- anyone can encrypt messages that only you can decode
- anyone can decode messages that only you can encode (which is helpful for authentication – see below).

Generating an RSA key pair:

- Choose 2 large primes, p and q, and calculate n = pq and z= (p-1)(q-1).
- Choose e coprime to z, (if we choose a prime for e, we only need to check it is not a factor of z).
- Calculate d such that e*d mod z = 1.
    - (Calculating d from e is easy *if* you know z. We won't cover how to do this in this course).
- Publish (e, n). This is your public key.
- Keep (d) secret. This is your private key. Also keep p and q (and thus z) secret as this is the information that you need to generate d from e.

Using an RSA key pair:

To encrypt a message x, we calculate x^e mod n. We can decrypt by applying (x^e mod n)^d mod n, which returns x, because for all x, x^{ed} mod n = x.

Note: When generating the key pairs, we chose e and d such that *ed mod **z** = 1*. I'm now asserting that for any x, *x^{ed} mod n = x* but I have not proved that. That proof is not part of this networks course – though it is not particularly difficult and mathematically minded folks are welcome to look it up.

13.5 Authentication – Certification

We want to know that the person we're communicating with really is who they say that they are. This might seem self-evidently important, but I want to call out quite how insidious and hard to detect this can be.

Consider the concept of **middle-person attack** (also called a **man-in-the-middle attack**). This is an actor that inserts itself into the network between A and B, but relays on messages between them. Neither A nor B can detect it as different from any switch in the network. Now A and B want to share a secret, using Diffie-Hellman key exchange. The bad actor intercepts the messages and performs Diffie-Hellman with each to generate shared secrets. It can then sit in the middle, decrypting and re-encrypting each message as it comes through.

Fortunately, we already have a solution to individual authentication – from RSA encryption above. I can digitally sign a message as mine by encrypting with my private key. You can decrypt the messages with my public key *and know that I must have encrypted them.* The only problem now, is who or what guarantees the link between my private key and me?

13.5.1 Public Certificates and certificate chains

I can assert my Public key belongs to me if you approach me personally and I give you my public key. That doesn't scale well, so we introduce the concept of a **certificate authority** (CA). I prove to the CA once that a particular key is mine, and the CA issues a public certificate asserting this fact. In future, people wanting to check if a key is mine can check with the CA.

How does someone know they are communicating with the CA (and not a malicious actor *pretending* to be the CA)? When the CA sends you the certificate, they sign it by encrypting with their own private key to show that they really are them.

But… How do we know that the CA's key really belongs to the CA.? Well they in turn can get a wider, larger and more public CA to provide them with a certificate. This eventually builds up a chain of trust until you get to some really big central certificate root authority that is so large and well-known that they just assert their key and everyone believes them. (At this point everyone only has to seriously check one central key, and everything else runs from there.)

Supposing you let slip your private key, and it is no longer solely yours. You can create a new key, but how do you revoke the certificate for your old key? CAs can explicitly revoke the certificate, though that then requires everyone in the chain to update their databases. Certificates are also time-limited, so will simply age out.

Real World CAs

In the real world, certificates time out after a year or so to reduce the faff of getting new ones. Certificate revoking isn't really a thing that people are aware of or do. And the checks that a CA does to verify the owner of a key are minimal – and rely on people not being bad.

As an example of being bad beyond the checks done by CAs, I could register some homomorph domain names such as below in order to send people links to my nefarious site. I can then register my own keys with them and get security certificates so that people could securely connect to them.

- www.gooogle.com (Extra "o". Or other common typos)
- www.goog1e.com ("1" not "l". Common letter similarities)
- www.google.com ("o", not "o" – Extended ascii set character that is pixel identical on screen)

The CA chain of trust would be happy. As far as they are concerned I genuinely own those names and people clicking on those links are reassured that they are genuinely connecting (to my nefarious site!)

13.6 Non-repudiation

By signing a message with a private key, you are both providing authentication that the message is yours *and also non-repudiation.* Because your public key is public, any third party can also verify that the message sent was sent by you.

13.7 Integrity – hashing etc.

Even if encryption guarantees a message has not been read and understood by a third party, it doesn't *necessarily* guarantee a message has not been modified between sending and receipt. For example, here's someone encrypting a simple promise of payment, a block at a time.

```
Message Encrypted and Sent...
I Promise T   ==>  23478293498
o Pay You £   ==>  84939238482
10            ==>  28393482081
```

A malicious person in the network doesn't have the ability to decrypt but does know that the last block in bank messages is usually a number. They replace with a random number (in this case 18493028583), cross their fingers, and hope it decrypts into a bigger value…

```
...Received and Decrypted
23478293498 ==> I Promise T
84939238482 ==> o Pay You £
18493028583 ==> 2348284
```

That's also a problem with non-repudiation. If the original message was signed, the original receiver could have made that similar swap, and third parties can now all see and all agree that the original sender promised to pay £ 2,348,284.

In addition, the encrypted message doesn't necessarily guarantee that the sender sent it recently – a middle person could intercept the £10 message above and send a copy of it again the next day.

To protect against these, we use **authenticators** and **nonces**.

An **authenticator** is redundant information (like for example a checksum) added to a transmitted message such that it is possible to detect modification of the original – in particular such that modified encrypted text doesn't decrypt into some modified plaintext. The simplest version of this is a **cryptographic hash**, which is an algorithmic digest calculated over the entire message.

Cryptographic hashes need to be cheap to calculate over a message, large enough to prevent easy discovery of collisions (multiple messages that hash to the same value) and sufficiently random that it is hard to force a particular message to have a particular hash.

A **nonce** is a one-time use word, used within an encrypted message, which prevents someone reusing the message. Nonces typically include a timestamp, to make it obvious *when* a message was encrypted/sent independently of the message.

This course won't go into detail on calculating hashes and nonces.

13.8 Example protocols including security

SSH

SSH (Secure Shell) is actually a suite of protocols, including SSH-TRANS which is a protocol establishing an encrypted transport channel.

SSH is included on most operating systems and functions as encrypted Telnet (already covered).

With SSH you establish an encrypted channel between a client and a server as follows:

- The client authenticates the server (using the server's RSA public key).
- The client and server agree a session key for the session, and use this to encrypt the data on the session.

How does the Client get hold of the server's public key? The server announces its public key when the client connects. How does the client know it has the correct public key? If your computer doesn't already know the server's key, you'll get a pop-up like the following:

```
The authenticity of host '...' can't be established.

RSA key fingerprint is d9:42:46:6e:b5:44:b5:8d:4d:a8:b0:73:82:ad:26:93.

Are you sure you want to continue connecting (yes/no)?
```

You choosing "yes" is you being the out of band Certificate Authority, and your computer will then store the key (in ~/.shh/known_hosts on Linux) and check it here on subsequent attempts.

At this point the *server* is authenticated, but the client has not provided authentication. How does the client authenticate? The client has the server's public key *and has authenticated the server* so can safely provide user and password credentials over the encrypted link. Alternatively, in a closed system, an administrator may have provided/authenticated various clients' public keys to the server already.

Try it

You've already played with Telnet. You can set up SSH access and then SSH connect and capture the trace in Wireshark in the same way. Can you still see the contents of the traffic?

Lots of protocols have secure versions that work in a similar way…

SFTP

SFTP (Secure FTP) is File Transfer Protocol (already covered) similarly encrypted.

SSL / TLS

SSL (Secure Sockets Layer) and later TLS (Transport Layer Security) provide a secure transport shim on top of TCP, and these are then used by other protocols further up the stack. For example HTTP**S** is HTTP on top of this.

13.9 Availability

All of the above is about providing a secure network. But a secure network is no good if it isn't usable. Bad actors can attempt to prevent use of your network without necessarily breaking your encryption. There are lots of examples of these, but they mostly boil down to attempting to either subvert some critical resource, or simply use up some critical resource in the network.

13.9.1 Poisoning attacks

Poisoning attacks subvert the network infrastructure. We've seen an example of this already in the section on BGP, where it's possible to advertise bad routes into the network to divert traffic.

Two other examples:

- DNS poisoning (DNS spoofing). If someone manages to add incorrect DNS information cached in a DNS server, then traffic will be misdirected to where they choose – for example their own fake version of a well-known website login page…
- ARP Poisoning (ARP spoofing). If someone can get an endpoint on a LAN, then they can send gratuitous ARP messages, advertising their own MAC address as the resolution for other legitimate IP addresses. While the last ARP response received by other host or routers on the network was from the bad actor, they'll get the IP traffic.

As you can see from these, to be truly safe you need security and authenticate at every level in your network and on every piece of infrastructure.

13.9.2 Denial of service attacks

Denial of Service (DoS) attacks simply attempt to overload some network resource. Three examples:

- Network flooding. If an actor has access to more bandwidth and packet-producing power than their target, they can send more packets into the target network than it can handle. It doesn't matter if those packets are dropped by the destination, they're still being delivered, and the target network simply runs out of bandwidth to send legitimate traffic.
- TCP SYN attacks. From the previous section on TCP, you'll see that setting up and maintaining the TCP connection costs each end about the same amount of resources. However, when a server receives an initial TCP SYN, it will typically book resources (e.g. a port), and respond with its own SYN/ACK. If the server doesn't receive anything further, the connection will time out (after a minute or two), but a bad actor can simply send SYNs as fast as it can (doing no other work), and the server will book up all its resources, meaning legitimate attempts to connect cannot be accepted.
- SIP Invite attack. Many other protocols besides TCP operate with one endpoint initiating a connection and servers on route and the receiving endpoint booking resources and responding. For example: Session Initiation Protocol (SIP) is used for setting up Voice over IP

phone calls. In this case a bad actor spamming the initial message (in SIP an "Invite") can book up all the resources, preventing the server from being able to respond successfully to other valid requests.

We'll cover some general ways that we try to protect against attacks like these in the next section.