Introduction to Computer Security

Year Two, Semester Two

Lecture One – An Introduction

Computer Security is a very broad topic, but can be simplified as the protection of:

1. **Confidentiality –** Preventing attackers from reading data.
2. **Integrity –** Ensuring incoming data is genuine.
3. **Availability –** Guaranteeing data is accessible when you need it.

Ways of safeguarding these traits successfully include:

- Correctness and efficiency,
- Knowing which assets to protect,
- Estimating the impact, likelihood, risks and mitigations of incoming attacks,
- Analysing systems to spot vulnerabilities and build protection.

Potential Attackers

It is very important to know *who* we are trying to keep our assets safe from. To build a strong security system, you must state your assumption about what the attacker might try to do.

This assumption is known as a *threat model* or an *attacker model.*

The omniscient super-hacker working in their basement that you see time and time again in films is highly inaccurate. Instead, the main groups you need to be thinking about are:

- **Script Kiddies –** Relatively low-skilled hackers running well-known attacks using pre-written scripts.
- **Professional Criminal Gangs –** Organisations dedicated to making money by hacking. Typical methods include spam-based phishing attacks, DoS attacks, and spreading ransomware via bugs in web-browsers.
- **Governments –** With the power of their state behind them, Governments have ridiculous computing power, allowing them to track and process copious amounts of data. They also use wiretaps and have the law on their side.
- **ISPs –** While your Internet Service Provider don't break any laws, they *do* see every piece of data you send or receive over the internet. Not only is this a privacy concern for many, but they may sell or lose your data.

Codes & Ciphers

Codes and Ciphers are *not* the same thing.

A code is any way of representing data, whether it be in a more readable, compact, or otherwise-efficient way. For example, Morse Code, ASCII, Hex, Binary and Base64 are *all* examples of codes.

Ciphers are a *type* of code, designed to be difficult to read. Perhaps the most famous and early example of a cipher is the Caesar Cipher, invented by Julius Caesar, where each letter in a text is replaced with the letter that's three places to the right in the alphabet.

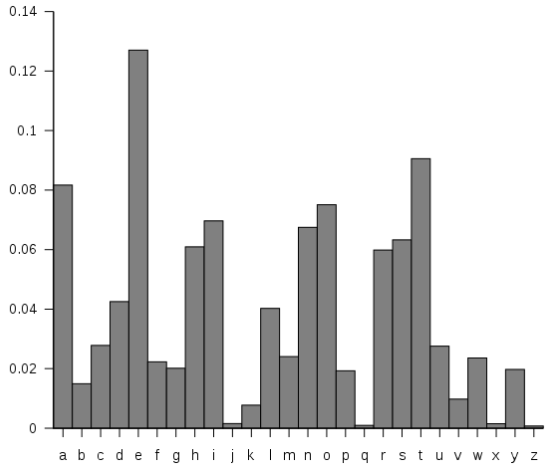
Compared to modern day standards, the Caesar Cipher is rubbish. This is because as soon as you understand it, you can break it. Modern day ciphers instead make use of *keys*.

Kerckhoff's Principle: *A cipher should be secure even if the attacker knows everything about it apart from the key.*

A simple way to convert the Caesar Cipher into a key-based approach would be to, instead of using 3 spaces to the right in the alphabet, we rotate by 'n', where n is the key. While this improves upon the original cipher, it's still terrible, as there are only 26 possible keys – meaning any attacker can just try all 26 keys in an insignificant amount of time (a method known a "brute forcing").

Perhaps a better way to do it would be to randomly replace each letter in the alphabet with another letter. Indeed, brute forcing $4x10^{26}$ keys would take a *very* long time, but there are other methods of cracking ciphers – such as Frequency Analysis.

Frequency analysis counts the number of times each symbol, pair of symbols, and so on occurs, and attempts to draw conclusions from this data – using the fact that we know which letters are far more common than others.



One Time Pads

The use of One Time Pads enables flawless encryption.

It uses a key that is the same length of the message to be encrypted, which is then XOR'd with the message to produce the cipher text. The cipher text can later be decrypted simply by XOR'ing it with the key. Very convenient!

It is easy to show how this method is unbreakable by using an example:

Message: `HELLOALICE`
Key: `SGFKPQYEIJ`
Cipher text: `ALRWERKNLO`

Given just the cipher text, frequency analysis will not work, so the key must be found through brute forcing. However, the wrong key may give a legible – but incorrect – message, and you've got no way of knowing whether what you've decrypted is the right message or not. For example:

Cipher text: `ALRWERKNLO`
Key: `TWCSCTFLWM`
Message: `GOODBYEBOB`

Unfortunately, the downfall of this method of encryption is that it requires a key the same length as the original text – and generating random numbers is a non-trivial computational problem. Also, they can only be used once, as multiple cipher texts produced by the same key allow an attacker to work out pieces of the key.

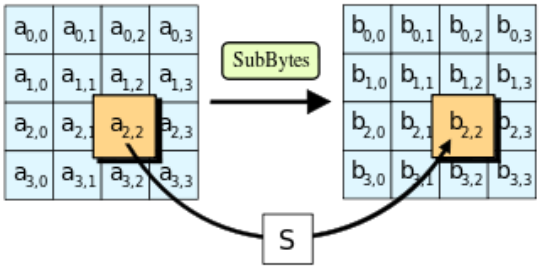
Modern day ciphers work on blocks of plain text instead of single symbols. They are made up of a series of permutations and substitutions repeated on each block, where the exact nature of the permutations and substitutions is controlled by the key.

Advanced Encryption Standard (AES)

AES is a state-of-the-art block cipher, capable of working on blocks of 128 bits. It encrypts data by generating 10 keys from a main 128-bit key, and then performing a repetitive sequence of mathematical steps. These steps are explained below.

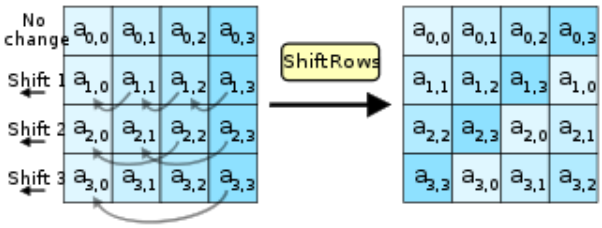
SubBytes

The "`SubBytes`" step provides the cipher's non-linearity. It uses an 8-bit substitution box known as the Rijndael S-Box to replace every byte in the state.



ShiftRows

The "`ShiftRows`" step ensures that the columns in the state do not end up linearly independent, as this characteristic would allow AES to be broken with much greater ease. It works by cyclically shifting the bytes in each row by an increasing offset. This is better shown by the diagram to the right.

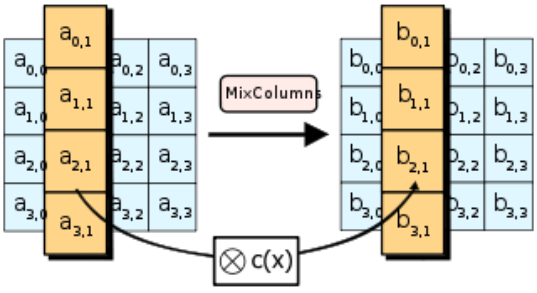


MixColumns

This step carries out a substitution of each column such that:

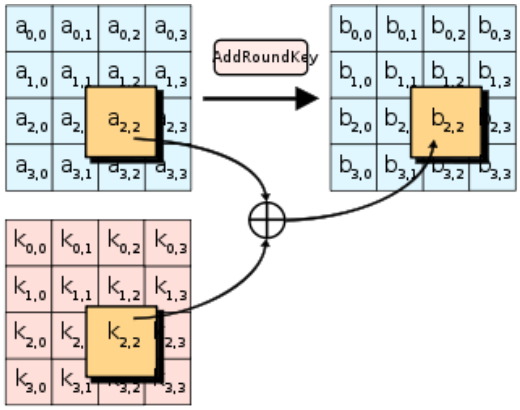
$$(a_0 \cdot x^3 + a_1 \cdot x^2 + a_2 \cdot x + a_3) \times (a_0 \cdot x^3 + a_1 \cdot x^2 + a_2 \cdot x + a_3) \bmod (x^4 + 1) = (b_0 \cdot x^3 + b_1 \cdot x^2 + b_2 \cdot x + b_3)$$

This step increases the cipher's diffusion and drastically increases the amount of time it takes to decrypt.



AddRoundKey

This step simply XOR's the block with the appropriate 128-bit round key, generated from the main key.



Full 128-Bit AES

The 128-bit version of AES works by performing 9 rounds of:

- `SubBytes`
- `ShiftRows`
- `MixColumns`
- `AddRoundKey`

followed by a final round of all but the `MixColumn` step.

LECTURE TWO – BLOCK ENCRYPTION

DES and 3DES

DES, while insecure, gives rise to ***3DES*** (pronounced "Triple DES") – which is the standard that came before AES. It was the result of an IBM competition, but was modified by the NSA just before release.

The NSA demanded the inclusion of S-Boxes in the standard, and that the key length would be fixed to 56 bits – which although was only half of the key length for AES, was exponentially weaker. Academics were suspicious of the inclusion of S-Boxes at first, but it later became apparent that the S-Boxes were exactly what made DES resistant to a new type of cryptographic attack.

This new type of attack was called *differential cryptanalysis*, and essentially consisted of comparing two cipher-texts produced using the same cipher, as a way to find out certain information about the key. The issue with this however was that the NSA added the S-Boxes to DES a whole *20 years* before differential cryptanalysis was discovered by academics, meaning they kept it a secret for their own use.

The NSA clearly didn't want any old hacker to be able to break the new golden standard – hence they insisted on the addition of S-Boxes – but the fixing of the key length to 56 bits meant that they were unique in their ability to brute-force DES, due to their abnormal volume of computing power.

At the time (1977) it was theorised that DES could be broken in 1 day, for $20M. In 1997, the EFF (an Electronic Rights group) broke DES in 56 hours, for just $250,000, as part of a sponsored security project. This was quite rightly deemed completely unacceptable for a cipher, so AES went into production, and 3DES was put into place as a temporary replacement in the meantime – simply DES, three times, using three keys.

3DES is reasonably secure, and would theoretically take around about 15 years to break. It's still used in bank cards and RFID chips.

Padding

Block ciphers only work on fixed sizes, so what do you do when you don't have a whole block? You simply pad it out.

However, it is *not* that simple. *The type of padding must be considered.* For example:

- Random Bytes – Will eventually give rise to proper ASCII values, so the recipient would have no idea where the actual message finished and the padding began.
- Zeroes or Zero Bytes – What if the message *also* ended with a zero or zero byte? These would be removed from the message during decryption.

These kinds of padding are clearly unacceptable. Therefore, the type of padding we use is in accordance with Public Key Cryptography Standards #5 and #7 (***PKCS5/7***):

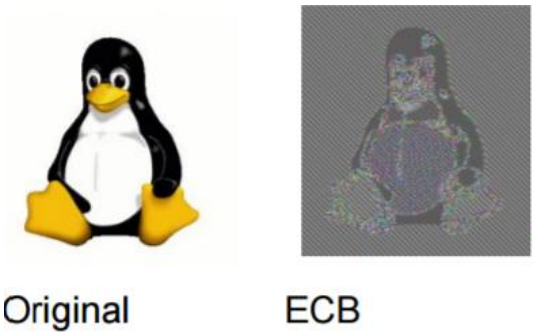
- If there is 1 byte of space, write "01".
- If there are 2 bytes of space, write "0202".
- If there are 3 bytes of space, write "030303".
- ... and so on.
- Importantly- if the message goes to the end of the block, add a new block of "16161616...". This stops genuine message text being stripped off during decryption when the message is exactly one or more blocks long.

Block Cipher Modes

Padding resolves the problem of messages being less than one block in length. But what about when a message consists of multiple blocks? The answer is – yet again – simple in principle but not in implementation.

It is possible to simply encrypt each block individually and independently, and then arrange them in the same order as the plaintext blocks were originally. This block cipher mode is called ***Electronic Codebook Mode* (ECB)**, and is very bad practice because it allows the cipher text to reveal repeating blocks of the plaintext.

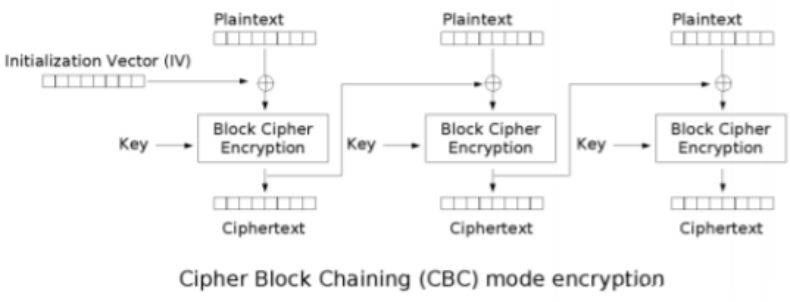
This is a textbook example of *insecure use of a secure cipher.*



Cipher Block Chaining (CBC)

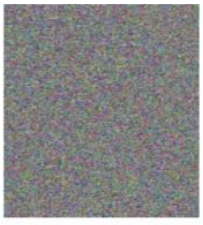
Cipher Block Chaining is a far more secure way of encrypting multiple blocks of plaintext. It works as follows:

1. Generate a random Initialisation Vector ("IV") and add this to the front of the encrypted file in plaintext.
2. XOR the first block of plaintext with the IV, and then encrypt the result to form the first block of cipher-text.
3. XOR the second block of plaintext with the first block of cipher-text, and then encrypt the result to form the *second* block of cipher-text.
4. ... and so on.



Cipher Block Chaining (CBC) mode encryption

This completely obscures repeating patterns, preventing frequency analysis.


CBC

[Side Note:** Java's* Random *class is not secure – it can be brute forced. Instead, always use* SecureRandom *when writing encryption programs in Java.**]

Using CBC with randomly generated IVs allows someone to send the same message with the same key over and over again, without the attacker knowing it's the same message, because the cipher-text will look completely different.

Plaintext when padded in accordance to PKCS5/7 and encrypted in CBC-mode AES is ***completely unbreakable***. That doesn't mean, however, that it can't be *used* incompetently.

Counter Mode (CTR)

$$C_i = B_i \text{ XOR } encrypt_k(IV + i\text{-}1)$$

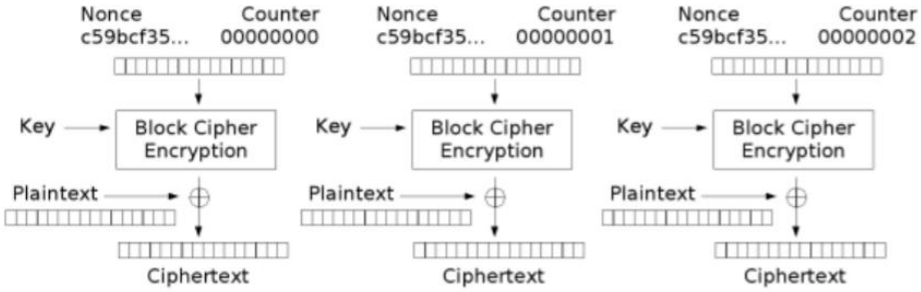
Sometimes it's is necessary to be able to encrypt/decrypt single blocks of data in a message, or on a hard drive, but CBC's sequential nature makes this impossible.

This is where another mode comes in handy – ***Counter Mode***.

Counter Mode still makes use of a random Initialisation Vector, but instead of chaining each block of cipher-text together, it chains it to a *version* of the IV. Here's the method:

1. Generate a random Initialisation Vector ("IV") and add this to the front of the encrypted file in plaintext.
2. Encrypt the IV, and then XOR it with the first block of plaintext to form the first block of cipher-text.
3. Encrypt (IV+1), and then XOR it with the second block of plaintext to form the second block of cipher-text.
4. Encrypt (IV+2), and then XOR it with the *third* block of plaintext to form the *third* block of cipher-text.
5. ... and so on.

This mode also enforces the correct order of the message – if an attacker was to intercept and re-order the message, decryption would fail because the counter would be wrong.



Lecture Three – Public Key Cryptography

The Key Problem

We've already established that AES is effectively unbreakable when used properly, but how can you use it to communicate with someone else? How do you give them the key?

Public Key Encryption helps with this problem. The idea is that there are two keys involved, instead of one:

- One for encryption (the public key)
- One for decryption (the private key)

It is not possible to derive the private key using the public key, which allows the user to post their public key anywhere they want on the internet for anyone to see while retaining the privacy of their private key.

Diffie-Hellman Protocol

The Diffie-Hellman is a widely used key agreement protocol, although it is not quite an example of public key cryptography.

To use Diffie-Hellman to communicate with another person, two numbers must be agreed on:

- The generator "g" (often 160 bits long)
- The prime "p" (often 1024 bits long)

These numbers tend to be plucked from RFCs on the internet, and can then be combined to form a key that is private to both participants.

If Alice and Bob wanted to communicate using Diffie-Hellman, here's how they'll do it:

1. Alice and Bob agree on values of **g** and **p**.
2. Alice and Bob both generate their own random numbers – let's call them $\mathbf{r_A}$ and $\mathbf{r_B}$.
3. They then both raise the **g** value they agreed on to their random numbers, and then mod's that result with the agreed **p** number – let's call those results $\mathbf{t_A}$ and $\mathbf{t_B}$.
4. Alice then sends Bob her $\mathbf{t_A}$ value and Bob sends Alice his $\mathbf{t_B}$ value.
5. Alice calculates $\mathbf{t_B^{rA}}$ **mod p**, and Bob calculates $\mathbf{t_A^{rB}}$ **mod p** – they will both get the *same value* ($\mathbf{g^{rArB}}$ **mod p)**. This can then be used as a key for Symmetric Key Encryption – such as AES.

This allows two people anywhere on the internet to establish a secure connection. An attacker could listen in and intercept $\mathbf{t_A}$ and $\mathbf{t_B}$, but it would be completely useless to them, as they still wouldn't know $\mathbf{r_A}$ and $\mathbf{r_B}$.

An issue with Diffie-Hellman however is that anyone could intercept the connection while it is being established, and respond to Alice's $\mathbf{t_A}$ with their own $\mathbf{t_B}$, and hi-jack

the connection without Alice knowing. This means that Diffie-Hellman cannot be used to *authenticate*.

Elgamal

Elgamal is a public key-scheme version of Diffie-Hellman, which uses a fixed **g** and **p**. If Alice was to use it, she would generate $\mathbf{r_A}$ and $\mathbf{t_A}$ as before, keeping $\mathbf{r_A}$ as her private key and using $\mathbf{t_B}$ as her public key.

If Bob wanted to send Alice a message, he would generate a one-use $\mathbf{r_B}$ and find $\mathbf{g^{r_B}}$ **mod p**, multiply his message with $\mathbf{t_A^{r_B}}$ **mod p**, and send both values to Alice as a pair. – ($\mathbf{g^{r_B}}$ **mod p**, M.($\mathbf{t_A^{r_B}}$ **mod p**)).

When Alice wants to decrypt Bob's message, she takes Bob's $\mathbf{t_B}$ (the first part of the pair) and raises it to her random number, $\mathbf{r_A}$. The result of this calculation is exactly what Bob's message is multiplied by, so all she needs to do now is divide the second part of the pair by that result and hey presto – the message has been decrypted.

A more efficient public key cryptography system that also enables signing is ***RSA***, and we will come across it many times in this course.

Using Public Key Crypto

Protocols such as RSA are incredibly slow and inefficient to use because of the index-based calculations that they are based on. Symmetric Key Cryptography is *far* faster. For this reason, we never use Public Key Crypto for encrypting large volumes of data such as movies and music albums.

Instead, we combine Public Key and Symmetric Key Cryptography. The most common method of this type of encryption is as follows:

1. Generate a new AES key and encrypt it using RSA with the public key.
2. Encrypt the plaintext using the newly generated AES key, in CBC mode.
3. Make the cipher text by first adding the encrypted AES key, and then the AES-encrypted plaintext.

Signatures

RSA has the very handy feature of allowing someone to run decryption on a plaintext message, and if encryption is then applied to the message using the same key combination, the message is restored:

$$\text{RSA: } E_{pub}(D_{priv}(M)) = M$$

It's this property that allows signature-based authentication in RSA. For example, if Alice wishes to sign a message and send it to Bob:

1. Alice runs the *decrypt* function on the message using her private key.
2. She then adds this cipher text (known as the ***signature***) to the end of her message and sends it to Bob.
3. Bob can run the *encrypt* function on the signature, using Alice's public key, and compare it with the message. If it matches, he knows for sure the message was sent by Alice.

It's good practice to use *different key pairs* for signing and encrypting.

Key Management

It is an ***awful*** idea to store your key in the source code of your encryption program – otherwise if an attacker somehow gets hold of your source code, they've also got your key.

Instead keys must be saved separately and password protected.

Java includes a key management tool (called **`keytool`**) as part of its standard install. To generate an RSA key-pair using **`keytool`**:

```
keytool -genkey -keyalg RSA -alias <keyName> -keyStore <keyStoreName>
```

This command generates a new key and saves it to the specified key store. It will ask for lots of personal details, and it is always good practice to fill these details out as it prevents attackers from muddling up your keys. You will also be able to choose a password, which will be used to encrypt the key within the key store.

To export the public key, use the command:

```
keytool -export -keyStore <keyStoreName> -alias <keyName> -rfc
```

This public key can then be published so that people can send encrypted messages to you.

Lecture Four – Hashes and MACs

Public Key Cryptography in Real Life

GPG comes in-built with most Linux distributions that provides encryption, signing and key management for personal key pairs, as well as keys of other people that you may wish to send encrypted messages to.

When asked to sign a message, GPG will compress the message and add a compressed version of your signature. This isn't particularly useful for sending emails as it isn't legible, so to send a human-readable version you can *clear-sign* instead.

Mozilla Thunderbird is an email client that handles PGP automatically, and is very useful for sending and receiving signed and encrypted messages.

Hashes

A hash of any message is a short string generated *from* that message. The hash of a message is *always* the same, and any small change makes the hash totally different.

It is very hard to discover the message by analysing the hash – even though it is very unlikely that any two different messages have the same hash.

While signatures are a great way of verifying that a message has come from a particular person, without hashing they provide no assurance that the bytes of the message haven't been tampered with by an attacker.

In practice, instead of signing the whole message (which could potentially be very long), the message gets hashed – and it's the hash that the signature comes from. This provides assurance that the message came from the person you thought it came from, and it definitely hasn't been tampered with.

Hashes are very useful for:

- Verifying downloads/messages.
- Tying parts of a message together (by hashing the whole message).
- Signing
- Protecting passwords (by storing the hash instead of the password).

Attacks on Hashes

There are a few different types of attacks on hashes – some more likely (but not necessarily more useful) than others. For example:

- Preimage Attack – Finding a message for a given hash. (Very difficult)
- Collision Attack – Finding a message that gives the same hash as a given message (Counter-intuitively likely, but not very useful).
- Prefix Collision Attack – A collision attack where the attacker can pick a prefix for the message.

Message Authentication Codes (MACs)

MACs are similar to hashes, but have keys – without which you cannot make or check the hash. This makes them very good at preventing brute-force guessing attacks.

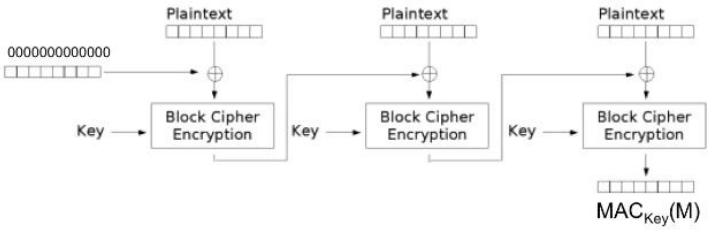
To authenticate a message, the MAC must have the following properties:

1. Uses a Key
2. Produces short string from long message,
3. Single bit change in message = totally different hash.
4. Difficult to go from hash to message.
5. Unlikely that two messages have the same hash.

The deduction from this list of properties may be that AES (in CBC mode) is a great place to start when coming up with a MAC function – the only missing characteristic is #2.

To turn AES-CBC into a MAC function, fix the IV as a constant and throw away everything apart from last block of cipher text - as it's this particular block that is affected by every block of cipher text prior to it. This provides a very cheap way of generating a MAC, and is known simply as a CBC MAC.

This, however, isn't the most efficient way of doing it, as the XOR-ing becomes redundant.



The SHA Family

The SHA hash family works in a very similar way to AES-CBC but improves upon efficiency by not using AES. They are considered the best hashes and are the most common as a result.

SHA-0 was developed in 1993 by the US National Institute of Standards and Technology (NIST), but just two years later the NSA "fixed" it, giving birth to SHA-1.

SHA-1 was a 160-bit hash, and it was found very quickly that a Collision (or Birthday) Attack on SHA-1 would only need 2^{80} hash tests – this is acceptable, but not great. However, in 2005 an attack was found that only needed 2^{63} hash tests. It still wasn't a particularly practical attack, but people lost trust in it very quickly.

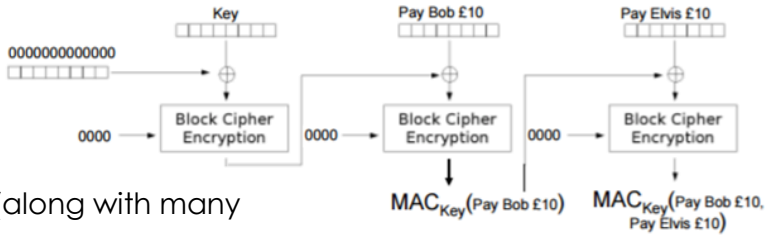
SHA-2 was developed and is the current standard. It's based on SHA-1 but with a longer hash – either 256 bits (known as SHA256) or 512 bits (known as SHA512). Because of its heavy basing on SHA-1, it inherited some of SHA-1's weaknesses, so although it considered secure *enough* for use, cryptographers worry that a way to break SHA-2 could be discovered very soon.

In 2008, a competition was opened up, allowing anyone to submit ideas for SHA-3. The winner was announced in 2012, and it is starting to appear in APIs, but it is not yet mainstream.

Broken Hash to MAC

Once you have a hash, how can you build a MAC from it?

It may be tempting to simply put the key in the hash alongside the message, but this method is not secure – it leaves the message vulnerable to a ***length extension*** attack. Because the key is only in the first portion of the transmitted message, an attacker can easily add a message to the end of it, without knowing the key.



HMAC

To prevent this type of attack (along with many others), we use HMACs.

Given a hash function **H**, we define the HMAC as follows:

$$\mathbf{HMAC_K(M) = H((K \text{ xor } opad), H((K \text{ xor } ipad), M))}$$

Or, in English:

1. XOR the key with some padding.
2. XOR the key with different padding, and hash it alongside the message.
3. Combine both results and hash *again*.

Most MACs you come across in the real world with either be a HMAC or a CBC MAC.

Known-Plaintext Attacks

When using AES-CTR, an attacker can tamper with your message if they know the plaintext.

[METHOD]

AES-CBC encrypted messages are also vulnerable to this kind of attack, but not as serverly. Because each block depends on all of the blocks prior to it, you cannot target pieces of the plaintext by flipping bits in the corresponding block of cipher-text. Instead, if one bit if flipped, the entire corresponding block will end up garbled.

It's possible to target bits of plaintext by flipping bits in the *previous* block of cipher-text (or in the case of the first block of plaintext, flipping bits in the IV) – but as the plaintext block corresponding to the altered cipher-text will be garbled, it may not be particularly useful to do this. There are very few useful applications for the Known-Plaintext attacks on CBC-encrypted messages, but two of note are:

1. Flipping bits in the IV to make changes to the *first block* (only) of the decrypted plaintext.
2. Flipping bits in website code – browsers will simply ignore garbled text, so it's possible to sneak a message into a website if losing the whole preceding block is acceptable.

Authenticated Encryption

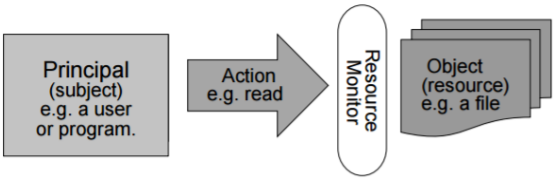
The way to prevent Known-Plaintext attacks is actually very easy – add a MAC to the end of the encrypted message.

1. Calculate the CBC MAC of the data.
2. Encrypt the message together with the CBC MAC, using the same key and CTR mode.

This exact method is known formally as RFC 3610, or CCM-mode. When using this mode, decryption will fail if the message has been tampered with. This makes AES-CCM the gold-standard for encryption.

Lecture Five – Access Control

Access Control can be summarised as the control of permissions between principals (such as users and programs) and resources (such as files and data).



In Unix-based systems, the base permissions are:

- **x** – execute
- **r** – read
- **w** – write

Permissions can be stored in a table known as an Access Control Matrix, such as the diagram on the right, but this is generally a bad idea because they get very big, very fast. Also if the one file containing the matrix becomes corrupted then all information pertaining to permissions on the system is lost.

	Operating System	Accounts Program	Accounting Data	Audit Trial
Alice (manager)	x	x	-	-
Bob (auditor)	rx	r	r	r
Accounts Program	x	r	rw	w
Sam (sys admin)	rwx	rw	-	-

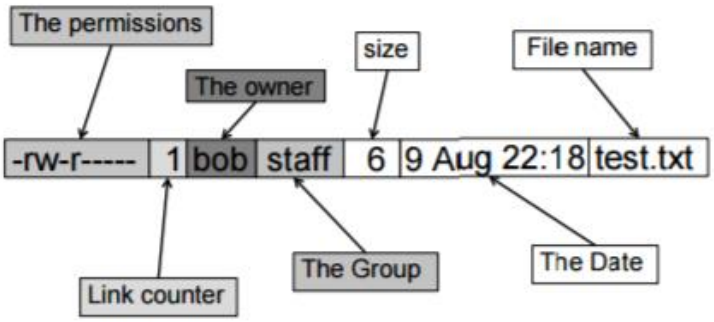
Working out which permissions to give to principals for certain resources can be a challenging task, as there is often more than one correct answer and it depends entirely on the context.

Access Control Lists (ACLs)

Instead of storing one huge matrix, in practice we store the column of the matrix with the object it refers to, as part of the metadata. The `ls -l` command can be used in Unix-based systems to display the permissions of files.

When a user logs on to a Unix-based system, their **`uid`** is set, and this is used to check permissions against. Users will also belong to a particular group.

When the `ls -l` command is used, a list of entries such as the following will be shown:



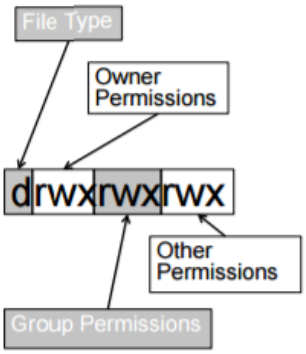
[Side Note:** The data portion of the file's metadata – though useful – is not secure.**]

The permissions section of the metadata, as you can see, is represented by a sequence of dashes and letters.

The first letter of the permissions code refers to the type of the file:

- **-** : file
- **d** : directory
- **b** / **c** : device file

Next comes the Owner's permissions. For example, **`-rwx------`** means that the owner can read, write and execute the file, and nobody else (apart from root) can either read, write or execute it. It may be useful to restrict the file's owner from writing to it, in order to prevent accidental (or malicious by third party) tampering.



The next set of permissions refer to the file's group. For example, **`-rwxr-x---`** means that any user in the group can read and execute the file, but only the owner can write it.

The last set of permissions refer to anyone on the system. For example, **`-rwxrwxrwx`** means that *any user* can read, write and execute the file.

Directory Permissions

The way permissions work when applied to directories is subtly different.

The **r** permission allows the user to list files within the directory, and the **x** permission allows the user to execute or read the contents of files in the directory.

[Side Note:** It is possible in systems such as the SoCS system to read other people's directories. Just because you can, this doesn't mean it is legal. In fact, to do this would be to break the Computer Misuse Act - so don't do it.**]

"Execute As" Permission

The **x** permission can be substituted for the **s** permission. If a file has an s-permission, when it is executed, it has the same permissions as whichever level of permission it is specified in.

For example, **`-rws--x--x`** means that anyone can execute the file, but when they do, the file will have the permissions of the owner. **`-rwx--s--x`** means that the file will have the permissions of a member of the group it belongs to when executed.

A confused deputy is a program that can be fooled by a principal with low authority into misusing its elevated authority, by virtue of its **s** permission.

Altering Permissions

The **`chmod`** command can be used to alter permissions of files and directories. There are many ways of using it, but the easiest is to think of each permission group as a binary number and then concatenate them together:

- **`----------`** → 000:000:000 → 000
- **`-rwxrwxrwx`** → 111:111:111 → 777
- **`-rwxr-x---`** → 111:010:000 → 720

So, for example, if you wanted everyone to have full access to your file, you could use the command **`chmod 777 myFile.txt`**.

Storing Passwords

In Unix-based systems, all user passwords are stored in the *shadow* file. However instead of being stored as plaintext, the hash of the password is stored – alongside its hash salt and hash type. The format is like this:

<username>:$<hash-type code>$<salt>$<hash>

To check a password, the plaintext password is hashed with the salt, and compared against the stored hash.

It is possible to use a brute force attack on a shadow file by using a program such as *John the Ripper* or *Hashcat*, alongside a good word-list.

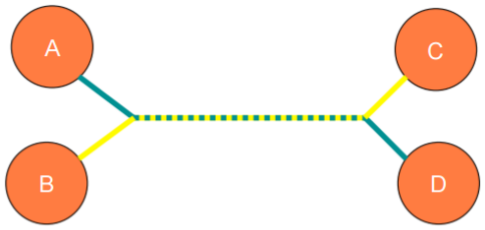
Lecture Six – The Internet

The Internet is very difficult to define. To get the best chance of understanding it, we need to start from the very beginning.

Before the internet, computers could be connected in the same way as phones – using "leased wires", where lines were buried under the ground on request.

At the start, in 1969, DARPA were giving research grants to universities so that they could buy computers. The Universities decided they'd like to link their computers, and they quickly began researching the best method for achieving this.

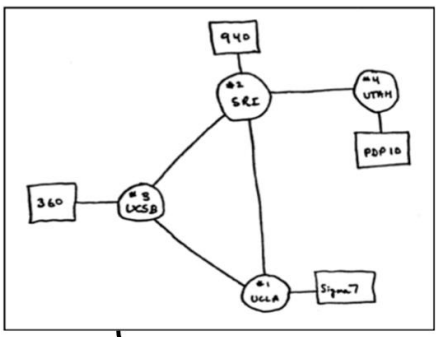
One of the first major steps was to use leased wires between universities but only sending small packets of data along the line instead of continuous connections, so that the line could be used by multiple parties at the same time.

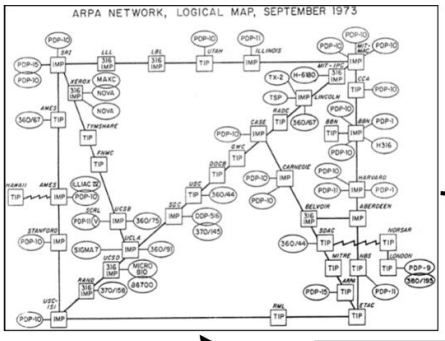


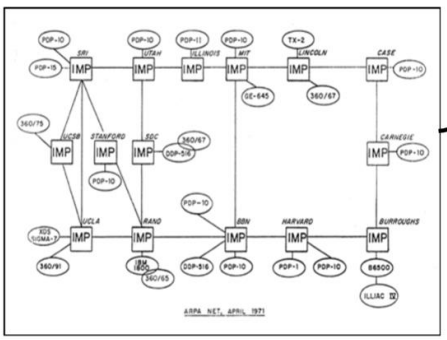
The packets had a strict structure and were called Internet Protocol (IP) packets. The destination address portion of the IP packets allows packets to be sent indirectly, which saves having to lay cables down between two nodes that already have a mutal node. Instead, the packet can be sent towards the mutual node, which will check the destination address and route it towards the intended destination if necessary. This became a very important selling point of this new type of networking, and more and more universities started joining the network.

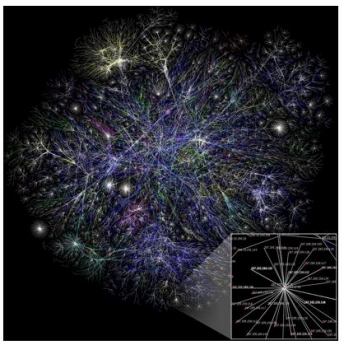
A satellite link was established with Hawaii by 1973, and connection was made with computers in London via existing undersea cables between the US and Norway, and Norway and the UK.

Eventually whole networks began to connect together using IP, and the Internet was born.









From A to B

Nowadays, a packet sent from work PC to a site in the US will pass through many nodes. It may first bounce around your department's network, before being passed to the company's IT services. When at this point the IT services deduce that they do not recognise the destination address as one in their system, the packet will be passed up to the ISP, and then possibly to a larger ISP that has access to transatlantic data connections. The process is then unravelled on the other side of the Atlantic.

Anyone along this node path has the ability to read the contents of that packet, and ISPs are even known to be able to relay the packets to the relevant intelligence organisation (GCHQ, NSA etc.), where contents can be read, analysed and perhaps even changed on the fly.

Transmission Control Protocol

Back to 1974. The daily traffic on the ARPA network is over 3 million packets per day, and many are getting lost. Transmission Control Protocol (TCP) is a protocol than runs on top of IP, that allows for the requesting for re-send of IP packets that didn't make it to their destination.

To allow multiple connections, TCP uses "ports". Different ports tend to be dedicated to a particular service. A few examples are:

- Port 80: HTTP
- Port 22: SSH
- Port 53: DNS
- Port 443: HTTPS

A TCP "socket" connection is defined as a destination IP address and port, and a source IP address and port.

[Side Note:** The IP address 127.0.0.1 is a reserved address that always means "myself".**]

DNS Servers

It is far too difficult for humans to remember whole IP addresses, so instead we associate the addresses with names. For example, if we want to see the news, instead of sending requests to 212.58.226.141, we send them to news.bbc.com.

There is a hierarchy of servers called the Domain Name System (DNS), whose servers are responsible for handling these types of requests.

Tools

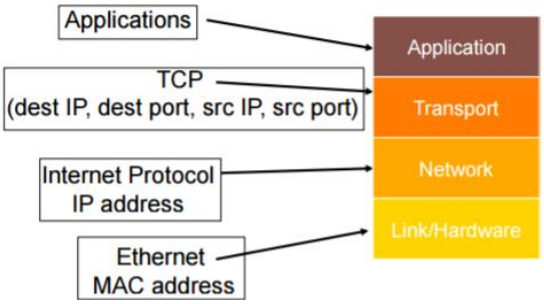
Netcat is a very useful tool for making Internet connections. Syntax varies between Operating Systems, so for examples, look online.

It is also possible scan for the 1000 most common ports on an IP address using "**`nmap`**", and wildcards (*'s) can be used to check for ports on varying ranges of addresses.

Wireshark is a network monitoring program that can be used for reading the contents of outgoing and incoming traffic

The Internet Protocol Stack

Internet Communication uses a stack of protocols, with each protocol using the protocol below it to send data. The stack (usually) looks something like this:

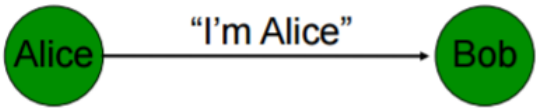


Attacks

The Internet was *not* designed with security in mind. It's incredibly important to always bear in mind that traffic can easily be monitored or altered. All good security products assume that the attacker has complete control over the network – but can't break encryption.

LECTURE SEVEN – ALICE AND BOB

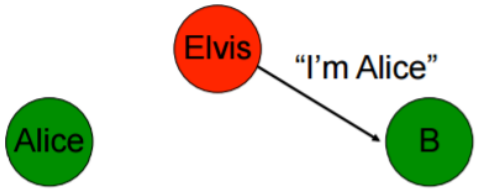
Discussing protocols using a programming language such as Java can get very tedious, very quickly. Instead we use a notation known as Alice and Bob notation – a far simpler way of illustrating network protocols. As an example, here's a very simple protocol:



This represents Alice sending the message "I'm Alice" to Bob, and can be written as:

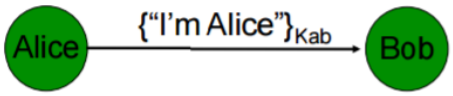
$$A \rightarrow B : \text{"I'm Alice"}$$

Interception and falsified identity can also be illustrated using this notation:



$$E(A) \rightarrow B : \text{"I'm Alice"}$$

As can symmetric key encryption:



$$A \rightarrow B : \{\text{"I'm Alice"}\}_{Kab}$$

Replay Attacks

One of the simplest attacks on protocols such as the previous one – where encryption is involved – is to intercept the encrypted message (perhaps using a tool such as Wireshark) and replaying it to the intended recipient. The recipient – Bob, in this case – has no idea that the attacker is not who they think they are – Alice.

The most basic replay attack looks like this:

1. $A \rightarrow B : \{\text{"I'm Alice"}\}_{Kab}$
2. $E(A) \rightarrow B : \{\text{"I'm Alice"}\}_{Kab}$

Nonces (random number) can be used to prevent replay attacks if used properly. For example, Bob can send Alice an encrypted nonce, and Alice must in future use the encrypted nonce *plus one* in her messages to Bob:

1. $A \rightarrow B : A$
2. $B \rightarrow A : \{N_a\}_{Kab}$
3. $A \rightarrow B : \{N_a + 1\}_{Kab}$, { "Pay Elvis £5" }$_{Kab}$

This known arithmetic adjustment to the nonce can not be made by an attacker who does not have the encryption key.

However, the it is possible for the attacker to intercept messages and split them into the relevent chunks they need to perform a more complex relay attack:

1. A → B : A
2. B → A : $\{N_a\}_{Kab}$
3. A → B : $\{N_a + 1\}_{Kab}$, $\{$ "Pay Elvis £5" $\}_{Kab}$

[... Time Passes ...]

4. A → B : A
5. B → A : $\{N_{a2}\}_{Kab}$
6. A → E(B) : $\{N_{a2} + 1\}_{Kab}$, $\{$ "Pay Bob £5"$\}_{Kab}$

6`. E(A) → B : $\{N_{a2} + 1\}_{Kab}$, $\{$ "Pay Elvis £5" $\}_{Kab}$

This type of replay attack can be avoided by encrypting both the nonce and message together:

1. A → B : A
2. B → A : $\{N_a\}_{Kab}$
3. A → B : $\{N_a + 1$, "Pay Elvis £5" $\}_{Kab}$

Protocol Efficiency

It is not always the case that "more encryption is better" – sometimes it's unnecessary, and can result in lots of redundant computation which costs money if you are using a service such as Amazon Web Services or Digital Ocean to host your server.

For example, looking at the previous protocol, the encryption on the original nonce is not necessary because as long as the attacker doesn't have the encryption key, they cannot encrypt the message as expected by Bob.

4. A → B : A
5. B → A : N_a
6. A → B : $\{N_a + 1$, "Pay Elvis £5" $\}_{Kab}$

Removing this encryption step could save Bob a lot of money if he's part of a multinational internet-based company with thousands of transactions happening per minute.

Key Establishment Protocol

Secure communication between Alice and Bob is very easy to achieve, as we've just shown – but only if they share a key. In most practical applications, the Alices and Bobs of the Internet do not share keys. Instead, they must set up a session key using a form of "Key Establishment Protocol", which relies on either Alice and Bob knowing each other's public keys, or a "Trusted Third Party".

Needham-Schroeder Protocol

A nice and elegant key establishment protocol is the Needham-Schroeder Protocol. It works by Alice sending her own identity and a nonce to Bob, encrypted with Bob's public key:

1. $A \rightarrow B : E_B(N_a, A)$

[Side Note: *E_B("Hello") means the message "Hello" has been encrypted using B's public key.**]***

When Bob then sends Alice back her nonce (encrypted with Alice's public key), Alice knows for sure that Bob is indeed Bob, because only Bob could have accessed her nonce. In order for Alice to authenticate herself too, Bob also includes a nonce of his own in the packet:

2. $B \rightarrow A : E_A(N_a, N_b)$

Alice then responds with Bob's encrypted nonce:

3. $A \rightarrow B : E_B(N_b)$

Now, Alice and Bob can both be sure that they are communicating with each other, and use their nonces to generate symmetric session key.

Unfortunately, about 10 years after its invention, it was discovered to be completely insecure.

Man in The Middle Attacks

If a situation arises where a potential victim (A) is running the Needham-Schroeder Protocol with a potential attacker (C), such as a malicious website, the attacker is able to pretend to be the victim as they initiate the same protocol with a secure service, such as a bank (B). It looks something like this:

1. $A \rightarrow C :$ $E_C(N_a, A)$ *// A establishes connection with C.*
   - 1`. $C(A) \rightarrow B :$ $E_B(N_a, A)$ *// C establishes connection with B, using A's nonce.*
   - 2`. $B \rightarrow C(A) :$ $E_A(N_a, N_b)$ *// B issues nonce challenge, encrypted with A's public key.*
2. $C \rightarrow A :$ $E_A(N_a, N_b)$ *// C relays the challenge to A.*
3. $A \rightarrow C :$ $E_C(N_b)$ *// A willingly completes the challenge.*
   - 3`. $C(A) \rightarrow B :$ $E_B(N_b)$ *// C relays the completed challenge to B.*

A serious issue with this attack is that it's difficult to detect -

A sees:

1. $A \rightarrow C :$ $E_C(N_a, A)$
2. $C \rightarrow A :$ $E_A(N_a, N_b)$
3. $A \rightarrow C :$ $E_C(N_b)$

B sees:

- 1`. $C(A) \rightarrow B :$ $E_B(N_a, A)$
- 2`. $B \rightarrow C(A) :$ $E_A(N_a, N_b)$
- 3`. $C(A) \rightarrow B :$ $E_B(N_b)$

To defend against Man in the Middle attacks, we can use a very similar version of the protocol, that also includes B's identity in the second packet:

1. $A \rightarrow B : E_B(N_a, A)$
2. $B \rightarrow A : E_A(N_a, N_b, B)$
3. $A \rightarrow B : E_B(N_b)$

This way the recipient knows for sure who the packet has come from.

Forward Secrecy

Forward secrecy is an optional protocol property that prevents compromised long-term keys from also compromising past session keys. This is particularly useful when the attacker is, say, your own Government, who can demand that you hand over your private keys. Without forward secrecy, your previous transmissions would be compromised.

An example of a KEP with forward secrecy is the Station-to-Station protocol, which is a heavily modified version of the Diffie-Hellman Protocol. While Diffie-Hellman can be used to generate a secure session key, it doesn't authenticate its two participants. S2S does, with B signing his g^y and including it alongside A's nonce. This way A knows that g^y definitely came from B, and that it's a fresh request. A then returns the favour:

1. $A \rightarrow B : g^x$
2. $B \rightarrow A : g^y, \{ S_B(g^y, g^x) \}_{g^{xy}}$
3. $A \rightarrow B : \{ S_A(g^x, g^y) \}_{g^{xy}}$
4. $B \rightarrow A : \{ M \}_{g^{xy}}$

[Side Note:** S_B("Hello") means the message "Hello" has been signed by B.**]

x, y, and g^{xy} are not stored beyond the protocol's run, and A & B's keys alone do not let the attacker read M – meaning S2S has forward secrecy.

Using Third Parties

As mentioned earlier, it is possible for A and B to use a trusted 3rd party (S) to establish their key. This works as follows:

1. $A \rightarrow S : A, B, N_a$
2. $S \rightarrow A : \{ N_a , B, K_{ab}, \{K_{ab} , A \}K_{bs} \}K_{as}$
3. $A \rightarrow B : \{ K_{ab} , A \}K_{bs}$
4. $B \rightarrow A : \{ N_b \}K_{ab}$
5. $A \rightarrow B : \{ N_b + 1 \}K_{ab}$

This protocol uses A and B's trust in the 3rd party to establish a secure session key. An issue with it is that A can use the same packet from step 3 to establish a connection with B at any point in the future.

This isn't a problem if A and B are willing to trust each other eternally from the first time they authenticate, but if the server needs the capability to blacklist compromised machines, this can be a security issue.

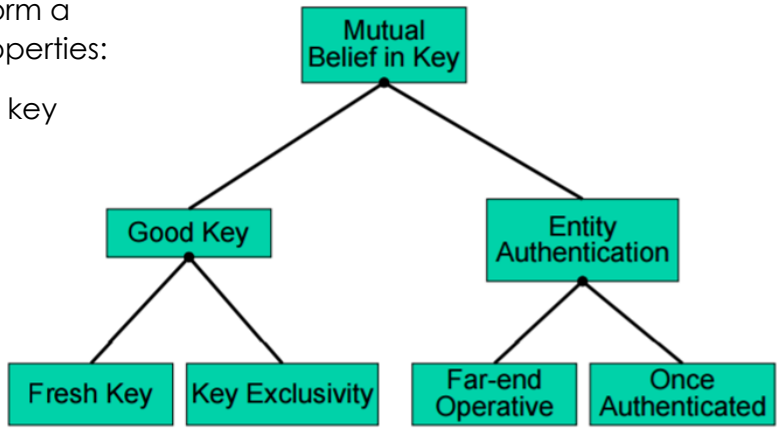
An easy way around this is for the Server to timestamp its packet to B in step 2, so that B can compare the timestamp with the current time and decide whether or not the authentication is fresh enough.

Key Establishment Goals

There are a few goals that a key may satisfy:

- Fresh – The established key is new, either from some trusted 3rd party of because it uses a new nonce.
- Exclusive – The key is only known by the principals in the protocol.
- Good – The key is both fresh and exclusive.
- Far-End Operative – A knows that "B" is currently active.
- Once Authenticated – A knows that B wishes to communicate with A.
- Entity Authenticated – A knows that B is currently active *and* wishes to communicate with A.
- Mutual belief – Key is both good and entity Authenticated.

These goals link together to form a hierarchy of desirable key properties:



A protocol that gives rise to a key with mutual belief is the gold standard, but not all key establishment goals are absolutely necessary, so there are many protocols out there that do not exhibit mutual belief in the keys they produce. What's truly important is *knowing* which key establishment goals your protocol needs, and ensuring that you satisfy them.

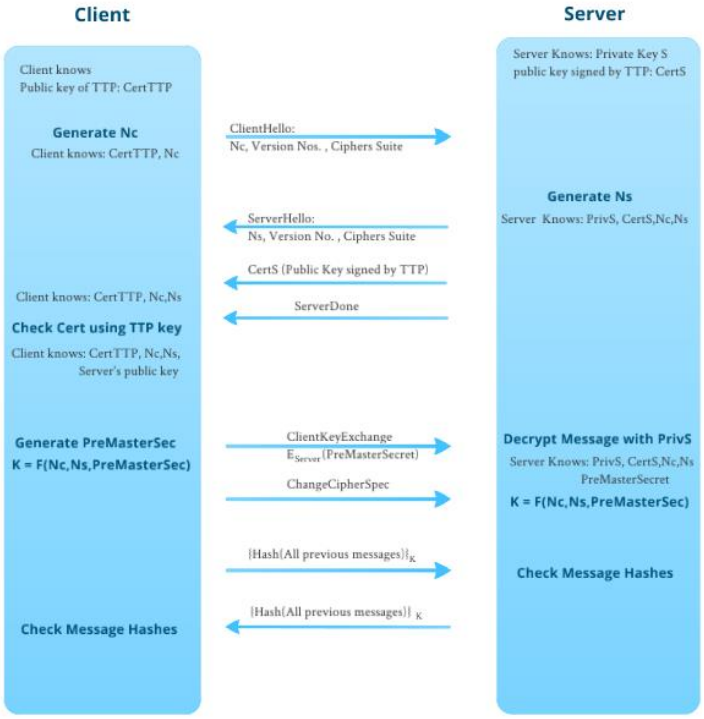
For example, S2S is safe to use in some applications but not others because it doesn't exhibit Once Authentication. Gaming is an example of where S2S cannot be used securely, as a client could (for example) forward "You just got killed" messages that it receives from the server to another client, so that they get killed instead.

LECTURE EIGHT – TLS AND TOR

The TLS Protocol

The Transport Layer Security (TLS) protocol – once named the Secure Sockets Layer (SSL) protocol – provides encrypted socket communication and authentication, based on public keys. It can use any one of a range of ciphers, where the appropriate cipher for a particular run is chosen at the beginning of the run.

TLS is the protocol responsible for the green padlock that appears in-browser during secure communication with secure sites. It provides an extra layer of security on the IP stack, between the Application and Transport layers. It allows the client to be sure that they are communicating with the intended server, but doesn't authenticate the client for the server.



Once the protocol has been completed, the client and server are able to use the session key to communicate.

Servers typically authenticate clients by asking for a password, but there is a version of TLS that exists which allows server to authenticate the client if the client has the certificate. This two-way TLS authentication deems password-based logins redundant, as the client and server both know for sure exactly who each other are – the only problem is that the typical internet user simply doesn't have a trusted certificate set up. So it looks like we'll still be using passwords for a while longer.

TLS, when used correctly, is very secure. However there are ways of attacking it, based on how it's applied.

Configuration Weaknesses

Cipher Downgrading Attack

As mentioned previously, TLS supports a range of ciphers, including some insecure ones. When TLS runs naturally, the chosen cipher will be the most secure cipher that both the client and server support.

However, a man in the middle attacker could remove the support for more secure ciphers on the client side, forcing the TLS into using an insecure cipher. This means that TLS in a particular instance is only as secure as the weakest cipher supported by both the client and the server.

Certificate Mismanagement

Many developers believe that maintaining a set of certificates for trusted certificate issuers is difficult, especially for small pieces of software such as mobile apps and Internet of Things (IoT) devices. Instead, it's easier to accept any incoming certificate without even checking it – allowing an attacker to simply create their own certificate, knowing there's a fair chance it won't get checked.

While this won't work against websites, it will work against a third of mobile phone applications, and two-thirds of IoT devices.

Implementation-Targeted Attacks

LogJam

It was believed for a long time that as long as the client only supported highly secure cipher suites, it didn't matter if a server supported a few low-security suites. However, the Snowdon leaks revealed that the NSA was regularly using Man in the Middle attacks on TLS-authenticated connections in spite of this.

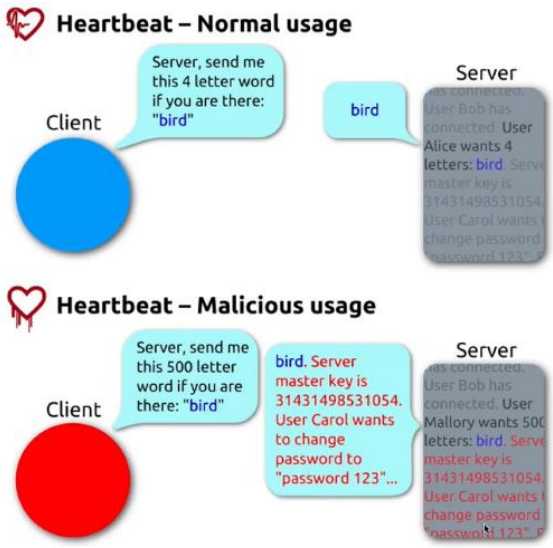
It was soon discovered that the NSA were able to do this by exploiting a feature of Diffie Hellman key-pairs – strong Diffie Hellman keys are compatible with weak ones. They would allow a secure client to believe they were using high-security Diffie Hellman (with a very large random number), but forced the Server on the other side to use a low-security export version (with a much smaller random number) of the exchange protocol. This meant the NSA was able to brute force the key and listen in to the connection.

HeartBleed

HeartBleed was the result of a programming error in OpenSSL. It exploited a feature which would allow the client to check if a connection with a server was still alive after long periods of no communication.

This check was known as a heartbeat under normal circumstances, and worked by sending a request from the client for the server to repeat back some arbitrary data, of a specified length.

This could be exploited in an almost trivial way. Simply by lying about the length of the arbitrary data contained in the heartbeat, an attacker could coerce the server into sending as much data from its memory as the attacker wanted. This data often (by co-incidence) contained the server's private key, which then allowed the attacker to impersonate the server.



TLS: A Summary

Despite the highly-secure nature of TLS, the vast majority of the general public simply click "ignore" when warned that their connection may have been hi-jacked. So what's the point, really?

The Tor Protocol

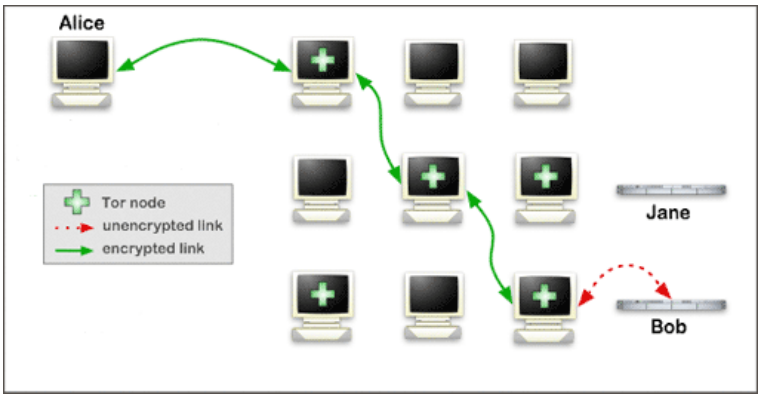
Under normal circumstances when you connect to a server, your IP address is revealed to the server – which in turn reveals your location (to a scary degree of accuracy). This may well be an issue for you. If it is, you could use a third-party proxy or VPN – services that will allow you to communicate with servers *through* them, so the servers will see the service-provider's IP address instead of yours.

It's notable, however, that this gives you no privacy from the proxy provider/VPN.

Onion Routing

A way to achieve better anonymity is by routing your traffic via a *number* of proxies. By far the most popular version of this type of protocol is Tor.

To establish a connection using the Tor network, the client selects three Tor proxies (usually at random). During connection, each proxy only knows the IP address of the nodes directly before and after itself. It operates on the assumption that even if two of the chosen tor nodes are corrupt, your anonymity is maintained – because no single node can see the whole picture.



To lose your anonymity, *all three* Tor nodes must not only be corrupt, but be conspiring. This is highly unlikely.

Packets sent using the Tor protocol are encrypted by several layers of encryption – one for each node. Connection establishment follows the following procedure:

1. Client downloads public keys for all Tor Nodes.
2. Client initiates Diffie-Hellman with Tor Node 1.
3. Tor Node 1 replies with its g^y-value and a hash of the key g^{xy}.
4. Client sends a request to Tor Node 1 (encrypted with key g^{xy}), containing a message asking for Tor Node 1 to connect the client with Tor Node 2, and a new Diffie-Hellman Initiation packet for doing so.
5. Tor Node 1 decrypts the request, interprets it and forwards the DH initialisation to Tor Node 2.
6. Tor Node 2 replies with its g^w value and a hash of the key g^{zw}. At this point, the client has now established keys for communicating with two Tor nodes.
7. Do similar stuff to establish key for communication with Tor Node 3.

Once connection is established, each time the client wishes to send a packet to a server they simply have to first encrypt the packet with each of the three keys – starting with the key that Tor Node 3 has access to and ending with Tor Node 1's. Return traffic reverses the process.

Lecture Nine – Introduction to Web Security

One of the more prevalent methods of internet navigation is by use of Uniform Resource Locators (URLs). These are split into three parts:

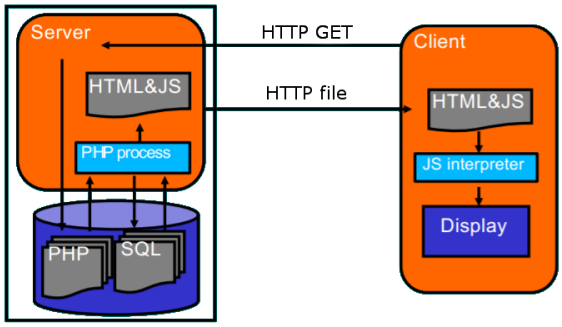
https://www.cs.bham.ac.uk/internal/index.php

1. The protocol through which the resource should be accessed,
2. The hostname of the server,
3. The file locator.

To display a webpage to you that've requested, your browser makes a request to the machine on the internet whose IP address resolves to the hostname, using the specified protocol, for the file described by the file locator. It then receives a HTML file from the server and renders it in human-readable way – usually interpreting embedded JavaScript in order to render more dynamic features.

PHP

For a web-page to be interactive, the client must be able to send data *back* to the server. This is achieved by using PHP – code that executes on the server. PHP code is never sent to the client, allowing the server to perform secure operations.



When, for example, a user fills some data into collection of textboxes, the data may appear in the URL, prepended by a question mark:

http://www.mysite.co.uk/example.php?numOne=2&numTwo=8

This is called a query string. In order for the server to process this data, the server must have an appropriate program stored. This program is given by the file locator.

The PHP method that puts this data in a query string is called "GET", but it has an obvious problem: Anything sent to the server using this command is visible to anyone who can read the requested URL. This is clearly not appropriate for confidential data.

Instead, when sending confidential or large volumes of data, the method "POST" is used. This command stores the data in the body of the request instead of the header.

When using PHP to process user input, it's important to validate data to avoid breaking the PHP – this can be done using JavaScript on the client-side before the request is even sent.

User Authentication After Log In

Once a user has logged into a site, the site needs to be able to keep track of who they are in order to show them correctly-customised content, and prevent other users from seeing that content.

There are a few candidates for this role:

- IP Address-Based:
  - ✗ Traffic routed through a proxy will all share the same origin IP.
  - ✗ Users may have multiple or varying IP addresses.
- Certificate-based:
  - ✓ Very secure.
  - ✗ Not practical – very few members of the general public have certificates and if they did, who would sign them?
- Cookie-based;
  - ✓ Accepted as standard.

A cookie is a string that a website has stored inside the client's browser that allows the website to check who the client is.

While this is a standard and pretty secure way to authenticate, it's possible to impersonate another person by stealing their cookie and replacing yours with it. This problem can be resolved if the website uses TLS, but a very common problem is that many websites only use TLS during security-pertinent activities (such as logging in and payment) – websites that don't use TLS at all times may as well not use TLS at all.

In aid of security, cookies have expiry dates – though many websites give their cookies expiry dates that are extremely far in the future, usually through negligence rather than malice.

SQL Injection Attacks

Any reasonably large website *probably* communicates with a database in some way, using SQL. For example, a simple product search URL request may look something like this:

http://www.myshop.co.uk/range.php?search=X

The search would then be processed on the server side using an SQL statement similar to this:

```
SELECT * FROM products WHERE (name='X')
```

Where SQL has been badly handled on the server, it's possible to exploit these statements by injecting our own SQL code into the request. For example, if we set:

X = '); DROP TABLE products; --

then the statement sent to the database would be this:

```
SELECT * FROM products WHERE (name='');
 DROP TABLE products; --')
```

[Side Note:** "--" in SQL is a comment, which in SQL injection servers to comment out the remainder of the intended SQL statement so that the database doesn't complain about syntax errors.**]

This would cause the database to delete the "products" table. There are more subtle injections that can be used, such as "`' OR '1'='1') --`" to show hidden items in a table or log in without the correct username/password.

Websites can prevent SQL injection from being used against them by *escaping* any user input that ends up in SQL queries, using a statement such as the following:

```
$user = mysqli_real_escape_string($con, $user);
```

This method places slashes in front of key SQL characters, such as apostrophes, turning the injected code into a harmless string:

```
\' OR \'1\'=\'1\') -
```

An alternative name for this process is sanitisation.

Lecture Ten – Top Threats to Web Security

SQL injections are far from being the only vulnerability for websites. Here we shall discuss the top ten web security threats, as listed by the 2013 edition of OWASPS Top Ten, which can be found here.

#1: Code Injection

While we looked at SQL injections in the last lecture, it's possible to inject many types of code in particular circumstances, such as PHP, XML processing commands, or even shell code. This type of attack is particularly serious as, if successful, it allows the attacker to run any code they want (language permitting) on the victim server.

A few nasty shell code injections are:

- **`nc -l -p 9999 -e /bin/bash`** - opens port 9999 and runs a shell on that port.
- **`useradd tpc -p rEK1ecacw.7.c`** - adds user tpc:password
- **`rm -f -r /`** - deletes *everything*, no questions asked.

Code Injection can be prevented by proper sanitisation (such as escaping).

#2: Broken Authentication

There is no set standard for post-password authentication, which forces web developers to come up their own authentication solutions – which are often vulnerable. Broken authentication may allow attackers to view other people's data, see files on the server they shouldn't be able to, or access admin-only functionality.

There's no set way to avoid these kinds of vulnerabilities, apart from using secure programming techniques.

#3: Cross-Site Scripting (XSS)

XSS is where an attacker is able to inject JavaScript into a HTML page being sent to other users, such as via HTML snippets. While many websites will block the <script> tag if input by a user, it's possible to sneak JavaScript onto the server by attaching it to events such as "onmouseover", which don't use the <script> tag.

This form of attack can be used for anything from causing annoying pop-ups and self-retweeting tweets (see here), to redirects, phishing scam initiation and drive-by downloads leading to browser-based buffer overflows.

XSS can be avoided, again, by proper validation and sanitisation of user input.

#4: Insecure Direct Object Reference

This threat arises when a server inexplicably trusts the client to only use the resources that it should, which may allow attackers to access other users' data by simply changing their name in the URL query string:

htttp://mySite.com/details.php?user=alice →
htttp://mySite.com/details.php?user=bob

Or to access files on the server by traversing paths via the file locator portion of the URL:

http://mySite.com/../../../etc/shadow

The best way to avoid path traversal is to make a user account for the webserver and configure access control settings properly so that the webserver can only access public files.

#5: Security Misconfiguration

Security misconfiguration is where things that should not be visible to the public, are, such as error messages, admin/debug panels, and directory listings. Bad security configuration may also allow confidential files to appear in the robots.txt file, which tells Google and other web-crawlers which areas of a website should not be processed or scanned.

#6: Sensitive Data Exposure

All sensitive data such should be protected at all times. For example, passwords should be hashed (with salt) for storage, credit card numbers should be encrypted in databases and decrypted by the PHP page when required, and TLS should be used everywhere.

#7: Missing Function-Level Access Control

Occasionally the type of PHP action to be run may sent within the URL, allowing an attacker to guess different commands and see what happens. To avoid this, never just rely on the URL request for authentication – use cookies.

#8: Cross-Site Request Forgery (CSRF)

CSRF is a serious but recently less common attack that allows an attacker to force another user to execute an unwanted action on a web service they're currently logged into. This is made possible by a unsuspecting user visiting the attackers website, which sends a request containing a malicious action to the legitimate website – either when the website loads or upon the triggering of some event, by attempting to load an image or similar with the reference set to the malicious URL request.

To defend against CSRF whenever forms are served, many websites now generate a random nonce, which is stored alongside the username in the database and included in the form. Every time a request is made, the nonce in the request will be checked against that user's nonce. The nonce on the attacker's page will not match the nonce for that user in the database, so the request will be rejected.

#9: Using Components with Known Vulnerabilities

Security patches for applications are released very frequently. Some patches may not be applied through neglect, while other times someone running a server may decide to not apply a patch because they'll have to take their site down to do so (which will cause missed business) or because it'll break the website. Either way, it leaves the server vulnerable to buffer overflow attacks.

#10: Invalidated Redirects and Forwards

If an attacker can forward a user to another page, then they can use it as a fake log in page for the base of a phishing attack, for ad fraud, or to launch browser-based exploits. This is not currently a particularly major threat.

Web Security Threats: A Summary

More information on each of the discussed threats can be found in the OWASP Top Ten 2013 edition (here). There are also websites (such as Google's Gruyere site: here) and virtual machines available for use for playing around with these attacks in order to get an understanding of how they work, but use of any of these attacks on a standard website is *very* illegal.

Lectures Eleven & Twelve – Buffer Overflow Attacks

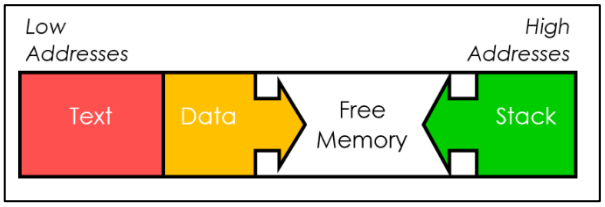
When programming in C, developers must manage memory themselves – this is notoriously difficult. If memory is managed incorrectly, an attacker can usually exploit this bug to make your application run arbitrary code. This is incredibly dangerous.

The Stack

To understand how buffer overflow attacks work, we need to understand how the stack and memory works. To do this, we'll look at the x86 architecture.

The diagram on the right is a simplified representation of memory during the execution of a program:

- Text – Stores program code.
- Data – Stores static variables.
- Stack – Contains data currently in use.



As data is added to the stack, it grows *downwards* – i.e. newer data has lower addresses than older data. Once the stack reaches the data segment, the program is out of memory.

Registers

Stored within the stack there is also a collection of fixed-purpose registers. The ones most important to use are:

- **The accumulator (EAX)** – Stores the result of a function.
- **Stack pointer (ESP)** – Points to the "top" of the stack.
- **Base pointer (EBP)** – Points to the "bottom" of the stack.
- ***Instruction pointer (EIP)*** – Points to the line of code to be executed next.

It's worth noting that the instruction pointer register *is in the stack* – the part of memory that the program is able to modify.

Function Calls

When a function is called, generally, the following happens:

1. The function's arguments are pushed onto the stack.
2. The EIP register's current value is also pushed onto the stack.
3. The function's address is stored in the EIP register.
4. The stack pointer is updated.
5. To create a new stack workspace, the base pointer is set to match the stack pointer, while the old base pointer is pushed onto the stack.
6. The function executes and the return value is stored in the EAX register.
7. The stashed old EIP value (called the **return address**) is popped back into the EIP register.

We can see from this that the instruction pointer controls which code is to be executed. So, if we're able to change the instruction pointer, we have control over the flow of the program.

Buffers

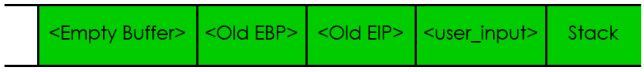
A buffer is an area of memory set aside to hold data, often while moving it from one section of a program to another. As an example of how a buffer may be used, let's take a look at a slice of a *very* simple program (written in C):

In the program on the right, function **`my_function`** is being called from some currently-running function. **`my_function`** takes some user input and copies it into a char array of size 16 bytes.

```
...
my_function(user_input);
...

my_function(char *str) {
        char buffer[16];
        strcpy(buffer, str);
}
```

At the point during execution immediately before **`strcpy`** is called, the stack will look something like this:



The nature of the input will determine what happens next.

Case One: Input Size ≤ 16 Bytes

When the size of the input is no greater than the size of the buffer, there won't be a problem. The input will be written into the buffer and have no impact on surrounding memory. If the input for our example program was "HelloWorld" (10 Bytes), the stack would look like this immediately after **`strcpy`** is executed:

Case Two: Input Size > Buffer Size, Accidental

When the size of the input is greater than the size of the buffer (and memory is managed poorly for this particular operation), the input will be written into the buffer and will continue to be written beyond the end of the buffer, overwriting adjacent data.

If the input for our example program was "YourBufferIsTooSmallXXXX", the memory would look something like this (immediately after **`strcpy`** is executed):

The old base pointer value and the return address have been corrupted, and are no longer valid memory addresses, so a segmentation fault will occur when **`my_function`** returns.

Case Three: Input Size > Buffer Size, Malicious

It's possible to engineer the input so that the return address is overwritten with a valid memory address. This new return address could either point to:

- **Program code** – useful for bypassing the program's checks/passwords, or accessing methods that shouldn't be accessed by a normal user; or

- **Arbitrary code** – if the attacker's input includes byte code, the return address can be overwritten to point to this code. This allows the attacker to take complete control of the computer, such as by granting the attacker a shell with root permissions.

If we wanted to run some of the program's code, starting from address \x29850408, we could input "1111222233334444555\x08\x04\x85\x29" (assuming we have configured the input terminal to interpret anything starting with "\x" as raw hexadecimal, instead of ASCII).

Note that we have reversed the bytes of the return address for the input. This is because x86 is a big-endian architecture, meaning each address in memory stores its bytes in order of decreasing significance.

Once **`strcpy`** is executed, the memory would look something like this:

The return address has successfully been overwritten with a valid memory address, so once **`my_function`** returns, the code at address \x29850408 will run.

Injecting Arbitrary Code

As mentioned briefly earlier, we are also able to configure user input for a vulnerable program to contain code, which we can then configure the return address to point towards.

Knowing what to inject is handy. Here's an example:

\x6a\x0b\x58\x99\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x52\x53\x89\xe1\xcd\x80

The code above will give the attacker a shell, which will have the same permissions as the program. For example, if the program is running as root, so will the shell. Bingo.

Defending Against Buffer Overflow Attacks

These days, there are several methods we can use to defend programs against these kinds of attack.

NX Bit

Code should always be in the text area of memory – not the stack. The NX bit provides a hardware distinction between the text and the stack, which – when enabled – causes the program to crash if the Instruction Pointer register ever points to the stack.

The standard attack against the NX Bit is to reuse code from the text area. This could be:

- Another function in the vulnerable program.
- A function from the standard C library (a "return-to-libc" attack).
- A sequence of chunks of existing code (return-oriented programming).

Return-to-libc attacks are particularly dangerous, the C standard library includes calls such as "system" which allow execution of shell code.

Address Space Layout Randomisation (ASLR)

ASLR randomly arranges the address space positions of the stack, heap and code's base each time the program runs. This makes it far more difficult for an attacker to guess where the buffer is in relation to the return address, and what the address of the desired code is.

In situations where little memory is currently available, it may be possible to brute force the addresses by attempting the attack thousands of times. However, there is a more elegant solution.

In the x86 instruction set, the assembly instruction 0x90 does *nothing*. This instruction is colloquially known as a NOP (pronounced "No-Op"). A way of improving an attack's chances of success is to prefix the payload with a large amount of NOPs - known as a NOP slide. The attacker can then guess where the return address is stored and if it's anywhere in the block of NOPs, the program will "slide" down to the attacker's shell code.

Metasploit

It takes a *lot* of time, skill and effort to find a buffer overflow vulnerability in a modern program. Instead of a lone hacker crafting a buffer overflow attack, it'll take a whole team of experts to crack one particular piece of software. When someone successfully conducts a buffer overflow attack against a piece of software for the first time, it spreads rather rapidly – usually through Metasploit.

Metasploit is a framework for testing and executing **known** buffer overflow attacks. If there is a well-known vulnerability in an application, there will probably be a patch for it, but there will also be a Metasploit module for it. If the vulnerable application is left unpatched on a target's system, it can probably be taken over using Metasploit.

Metasploit also includes a library of shell code, which can be injected.

Windows XP

It's worth noting that Microsoft no longer releases patches for Windows XP. This means that any vulnerabilities found since patches stopped being released will *very* likely work against a system still running XP.

Lecture Thirteen – Reverse Engineering

Many of the attacks we have seen so far involve tricking a program into accepting data that is, in actual fact, code. Now, we're going to look at doing the opposite.

Executable code and be written and edited, just like any other document. Ultimately an attacker/analyst can do *anything* they want with a program. It is impossible to give someone a program and keep how it works a secret.

Reverse engineering can be useful for:

- Analysing Malware
- Debugging memory errors
- Analysing Legacy code (executable code where the source code is no longer available.
- Security auditing.

Java Byte Code

Java, once compiled, can run on any platform. This is achieved by compiling to a language halfway between Java and assembly – Java byte code. You'll recognise Java byte code as the contents of a ".class" file.

Each platform has its own version of the Java Virtual Machine (JVM), which takes Java byte code and runs it as the appropriate assembly code for that platform.

The Java byte code contained within a file called "Program.class" can be viewed using the command:

```
javap -c Program.class
```

It may be possible to guess what quite a few pieces of byte code do, but Google can be used to determine the function of some of the less obvious commands.

An alternative to working with byte code is to decompile the byte code using JD-GUI. However, decompilation is not perfect. For example, **`1`**'s may get confused with **`true`**'s, or there may be larger logical errors. Despite these errors, the decompiled code should still give an analyst a far more than reasonable chance to understand what the program is doing.

Binaries

Binaries are written in assembly language, which is far lower level than Java byte code. Assembly code compiled for one type of machine architecture and one particular OS will not run on another – but the same reverse engineering techniques apply.

In this module, we are particularly interested in code compiled for Linux and x86 architectures. A few very common and useful x86 instructions are:

- **PUSH** – Add to the top of the stack
- **POP** – Read and remove from the top of the stack.
- **CALL** – Execute a function.
- **RET/RETN/RETF** – End a function and re-start the code that called it.

- **JMP** – Jump to some code (similar to writing to EIP register).
- **MOV** – Move value between registers.

IDA is an interactive disassembler and debugger that can be used to gain an insight into how a binary works. It's not crucial to understand the entirety of a program to be able to analyse, alter and exploit it. Instead, we can just look for patterns.

Defending Against Reverse Engineering

If you don't want your extremely valuable piece of software to have its purchase verification skipped by someone who illegally downloaded it and disassembled it, or want to prevent cheating on your online game, there are a few methods of defending against reverse engineering:

- **Dynamic Key Generation** – Useful for software that requires a verification key to be evaluated.
- **Binary Encryption** – It's possible to encrypt part of the binary code. However, the key must be stored somewhere in the program in plain text.
- **Code Obfuscation** – Strip all useful method and variable names and turn them into nonsense so that analysists can't tell what things do or store just by looking at them.

These defences can slow attacks down by months or even years, but can't stop them entirely. Remember that any attacker can run, analyse and edit your code, just as if it was any other file. ***There is no security through obscurity.***

Another good way to defend against reverse engineering is to involve the internet. Expensive software may now require online activation, and games can be made to require online content. These server connections may be used to decrypt encrypted binaries, or verify that code hasn't been tampered with.

Hardware based protection may also be employed, such as by storing part of the code in tamper resistant hardware. This technique is often employed by games consoles.

Most software protection *can* be removed with time and effort, but slowing these types of attacks down by months or years can be enough to save a business.

Lecture Fourteen – Common System Exploits

When managing an entire network, there are a great range of attack vectors to consider. Some of the most common are:

- Phishing attacks from incoming e-mail
- Attacks on Web Page
- Attacks on all servers
- Local attacks via wi-fi
- Insider attacks.

Phishing

Setting up a website that looks like another can be a pretty trivial task. Phishers do this in order to mimic services such as banks, email and social networking. A phisher could then send spam to a user, asking or otherwise convincing them to fill in details on this fake page, or spread malware that causes automatic redirection to the false pages.

For a very large company, a phisher could send a spam email with a link to every employee. It would only take one incompetent employee to fall for the scam for the phishing attack to be a success. Still, it takes a fairly inept or ignorant individual to fall for a generalised phishing attack.

Instead, spear phishing can be used to target one particular person. This technique involves carefully crafting a phishing email and sending it to the target. This can be done by searching for the employee on LinkedIn and working out what motivates them. Perhaps they are new, or seeking promotion, and thus intimidated or eager to impress their superiors.

Common Passwords

Never underestimate the number of people or services that will use very insecure credentials. Phishing is completely unnecessary if someone can SSH into a local machine using password "password123".

Known Memory Exploits

We've seen just how easy it is to exploit memory-based vulnerabilities using the Metasploit framework. By far the best way to defend your system against this kind of attack is to ensure all applications and software are completely up to date. Patches may be annoying and frequent, but attacks through known exploits are *far* more likely than attacks through newly discovered (**zero day**) exploits.

Windows Dumbfuckery

Windows-based systems will store the passwords of logged-in users in memory – all of which can be read by anyone with SYSTEM access. This is yet more incentive to make sure all patches are up to date, to prevent people gaining SYSTEM access in the first place.

Viruses & Worms

A virus is a self-replicating program that requires interaction to spread. These interactions can be as simple as opening a PDF that contains malware, or inserting a USB stick – which usually run *something* upon detection by the OS.

The Pentagon was once attacked by a few USB drives found in the car park and plugged in by staff for investigation – because autorun.inf ran malware. The Pentagon's response was to superglue all USB ports and ban USB drives – a ban only lifted in 2011.

Worms are self-replicating programs, requiring no interaction. Be careful not to confuse the two.

Trojans

Trojans are malware that need a user to download and run them, often pretending to be key-generators or anti-virus software. Most attacks against Apple have been in the form of Trojans.

Inside Attacks

Businesses must consider attacks by their employees.

For example, a system administrator for the city of San Francisco named Terry Childs refused to give supervisors the system passwords, saying they were "unqualified" – meaning Childs had full and unparalleled access to the City's system. He then locked system for twelve days.

The actions of malicious insiders can be mitigated by separation of powers, but it's safer to conduct background checks to keep baddies out, and keep staff happy to stop goodies *turning into* baddies. When someone in a large company is fired, they're often immediately escorted off the premises by security, just in case they try to do any damage.

Defences

There is a hierarchy of defences that can be employed against the aforementioned exploits:

1. **Firewalls** –
    - Used for blocking internet traffic.
    - Either installed on each computer, or built into the network.
    - Can be stateless or statefull.
2. **Fast Patches** –
    - The practice of making sure all security patches are installed immediately.
    - Almost always a patch to stop known exploits.
3. **Anti-Virus** –
    - Scan the computer for known malware.
    - Can scan e-mail and network traffic.
    - However, only as good as the most recent update; and
    - Can be disabled by an attacker with admin access.
4. **Intrusion Detection Systems** (**IDS**) –

    - Used for looking at all network packets and reporting suspicious behaviour.
    - Can catch **`nmap`** and Metasploit.
    - An example is Snort (here).
5. **Check File Hashes**
6. **Good password and user policies**

For the average home user running Linux or Mac OS, the first two steps should be sufficient. Windows users will require anti-virus as well. A system administrator should be employing all six.

Computer Security Policy

Policy is the documentation of computer security decisions. Computer Security Policy is very important in businesses, but often gets side-lined for the needs of the business instead.

An example of policy is the Payment Card Industry Data Security Standard (PCI-DSS). All organisations that handle credit card data *should* comply with this standard, but in practice, non-compliant organisations will usually only get into trouble if there is problem or an audit. Card Payments *could* be refused for these non-compliant organisations.

Information Security Management Systems

An ISMS is a set of policies for systematically managing an organisation's sensitive data. They must be continually monitored, with reporting of new faults and monitoring of the IDS.

If an organisation's activities shift, the ISMS will need to be updated. Even if the organisation's activities *don't* shift, the first ISMS could have missed something – so it needs to be regularly reviewed.

ISO 27001

ISO 27001 is the international standard on how to do an ISMS. It provides a guide for what companies need to do, and can be audited, allowing organisations to prove to others that it has an ISMS. ISOs give some assurance to other organisations that your organisation is secure.

The steps prescribed by ISO 27001 roughly follows:

1. **Establish the ISMS** – Involves defining the organisation in terms of what it does and what its assets are, as well as defining the scope of the ISMS.
2. **Analyse the Risks** – Involves identifying the assets within the scope of the ISMS and their owners, and then identifying the threats to each of those assets. Potentially exploitable vulnerabilities must be identified, as well as the impact of the loss of each asset, which could be:
    - Confidentiality
    - Integrity
    - Money
    - Availability

3. **Specify Threat controls** – Involves defining mitigation and avoidance techniques.
4. **Obtain Management approval** –
    a. Some risks may be deemed not worth the prevention/mitigation costs, and given the label of "accepted risk". This of course is not the responsibility of the system administrator – approval from Management should be sought.
    b. The overall ISMS must also be approved by Management.
5. **Prepare Statement of Applicability** – The final step; providing an overview of the ISMS.

Risk

Risk analysis is a very significant part of any ISMS, so let's take a look at it in more detail, starting off with an example.

We start by considering an asset – for example, Purchase History – and then considering the threats to it, such as:

- Loss
- Corruption
- Expiry
- Theft

We also need to consider the possible vulnerabilities. These could be anything we've covered so far in this module, plus other things. For example:

- Bugs in record systems
- SQL injection vulnerabilities
- Faulty access control
- Malicious/incompetent staff
- Fire/Flood/Other natural disaster.

Impact

With this done, the next stage is identifying the impact of the threats. We may do this in a table such as this:

	Lost	Corrupted	Out of Date	Stolen
Single Record	2	3	1	5
> 50%	4	5	2	5
50%-100%	5	6	4	6

It's very hard to know when threat impact figures are correct, so they must be continually reviewed.

Likelihood

Each vulnerability must also be rated in terms of their likelihood. There are a few ways of doing this:

- Probability
- Events per Year
- Arbitrary 1-10 scale.

Either way, likelihood must be assessed based on a combination of history and good guess work.

Risk

The risk of a particular threat/vulnerability is an amalgamation of its likelihood and impact. This means that risk assessment depends on the methodology employed for assessing impact and likelihood:

- For levels of 1 to 10, we can say that:

$$\text{Risk} = \text{Impact} \times \text{Likelihood}$$

- Another good option is the expected cost per year.

With this data, we can then form a table, such as the following:

	Out of Date	Lost	Corrupted	Stolen
Flood	-	5	-	-
Bugs	16	10	12	-
SQL Injection	12	15	18	18
Hackers	10	16	16	24
Fire	-	20	-	-
Insiders	20	25	30	30

Treatment

Ultimately there are four possible options when deciding how to deal with a risk. These are:

- **Avoid it** – Take steps to stop it from happening. For example, loss of data could be avoided by not collecting data in the first place.
- **Mitigate it** – Take steps to make the impact less serious. For example, the threat of data theft can be mitigated by encrypting the data.
- **Transfer it** – Make it someone else's responsibility. For example, the risk of data being destroyed by fire can be transferred using fire insurance.
- **Accept it** – Decide to live with it. For example, the risk of main and all backup disks failing at the same time is completely negligible, so this is acceptable.

Lecture Fifteen – Common Internet Threats

There are many ways an attacker can gain access to your machine, and unfortunately, there is a selection of things they can go once they have that access. For example:

- Steal passwords, credit card numbers and/or personal data.
- Spy on you, through the web-cam, microphone, screen or key-strokes.
- Send spam.
- Enslave the machine for use in DDoS (or other) attacks.

Man in the Browser Attacks

Malware can inject code into your browser, enabling all certificate authentication to be faked and anything that looks like a credit card number or bank credentials can be collected upon input. Man in the Browser attacks render TLS completely useless.

Botnets

Most attackers don't attack for fun – they attack for material gain, usually in the form of money.

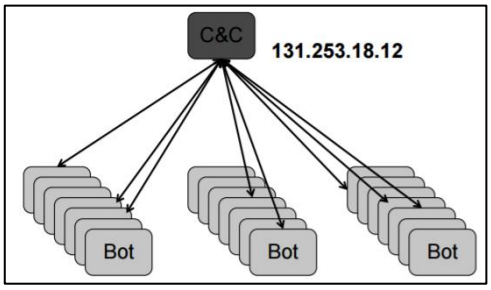
A single credit card number of spam e-mail isn't worth very much, so instead networks of hacked computers (bots) are organised into large networks – known as **botnets** – and used for greater purpose.

DoS Attacks

With access control over a botnet it's very easy to overload someone's website. The easiest type of attack uses (or rents) a botnet to perform a distributed denial-of-service attack. The attacker can then use this advantage to blackmail or make a political statement.

Command & Control

The most primitive command & control strategy for botnets is to configure the bots to connect to a specific IP address.



However the issue with this is that if that one IP address gets blocked or even seized, control over the bots is lost.

An alternative strategy is known as **Fast Flux**. This is where the bots are configured to look for a URL instead of an IP address. This stops the issue of IP addresses getting blocked, as new IP addresses can be registerd every few minutes to serve the URL.

An even cleverer way of maintaining connection is **Domain Flux.** Here, bots continuously generate new URLs, using a pre-defined algorithm. Because the algorithm is pre-defined, the botmasters are able to register the URL in advance. This means that even if all C&C servers are shut down, the bots will switch to a new URL in a few days – and finding out what the future URLs will be is very difficult.

The most recent malware sets the botnet up as a Peer-to-Peer (P2P) network. The malware then connects to C&C servers and other bots. This allows botmasters to connect to any bot and update them all with a new C&C if the main C&C goes down.

ZeuS

Zeus is one of the larger and most infamous botnets, using Fast Flux and many C&C servers. ZeuS spreads mainly via Trojans, and - as well as enslaving machines - will perform phishing and man-in-the-browser attacks, and send spam. It's possible to purchase code for operating ZeuS over the black markets. ZeuS is succeeded by Gameover ZeuS, which uses P2P encryption for communication between its bots and C&C servers.

Conflicker

Conflicker is a worm that installs a botnet. Since its detection in 2008 it has infected more than 10 million machines. It uses Domain Flux and P2P connection, and spreads by using NetBIOS buffer overflows and guessing admin passwords.

The first version was noted to not infect computers that had a Ukrainian keyboard layout. It's now largely contained by security researchers, who collectively have blocked tens of thousands of domain names.

Torpig

Torpig is another botnet that spreads through machines compromised by the Mebroot root kit – a particularly nasty root kit that writes itself into the Master Boot Record, allowing it to execute before the OS loads, which makes it very difficult to detect. Mebroot spreads via drive-by-downloads, and then downloads and installs other payloads.

In 2009, a team from the University of California managed to reverse engineer Torpig's domain flux algorithm, and noticed that some of the URLs Torpig was due to use in the future hadn't yet been registered – so they registered the addresses themselves.

When the addresses came into turn, it took ten days for Mebroot to replace Torpig with a new payload, meaning the team at UoC had complete control of the botnet and saw all of the data.

Data Type	Data Items (#)
Mailbox account	54,090
Email	1,258,862
Form data	11,966,532
HTTP account	411,039
FTP account	12,307
POP account	415,206
SMTP account	100,472
Windows password	1,235,122

Underground Marketplaces

Some parts of the internet are used for very shady transactions. Internet Relay Channel (IRC) and web forums are historically used for these transactions, but Tor Hidden services are by far the fasted growing vector for the black market.

There are a few terms that may be useful to understand when discussing the black market:

- Attacker – Steals data, via botnets or phishing, etc.
- Cashier – Takes credit cards and bank accounts and withdraws cash.

- Drop – Provides a place to send goods.
- Service Seller – Sells a service. For example, a bot master may rent their botnet for spam, DDoS or phishing.

Bitcoin

Bitcoin is an electronic currency that can be used for many things, including making anonymous transactions. It's based on partial SHA collisions – so if you can find a partial collision, you have minted a bitcoin.

They are passed from one person to another by signing an entry in a public database. Only the person with the signing key can pass it on.

Alternative payment methods include:

- "Webmoney" – an online payment system based in Russia.
- Western Union money transfer

Typical Transactions: 1

Let's take a look at how black market sales may go down, by looking at a few made-up scenarios. Here's the first:

1. Hacker steals 1000 "fullz" (full collection of credit card number, CVV, name, address, etc.)
2. Hacker sells them on forum for 10 bitcoins (approximately £4300).
3. The Buyer re-sells them in groups of 20 to cashiers, for £300 each, in Webmoney.
4. Cashiers meet drops in Internet chat rooms who agree to receive goods paid for with stolen bank credentials.
5. Cashiers order goods online and has them sent to the drops.
6. Drops sell goods and send half the money to the cashiers via Western Union.

Typical Transactions: 2

1. Bot master offers network for DDoS attack at £200 per day.
2. Attacker hires the botnet to attack a small company and bring down their site.
3. Attacker anonymously contacts the company and asks for £10,000 in bitcoins to stop.

Lecture Sixteen – Usability & Security

Awaiting Lecture Video.