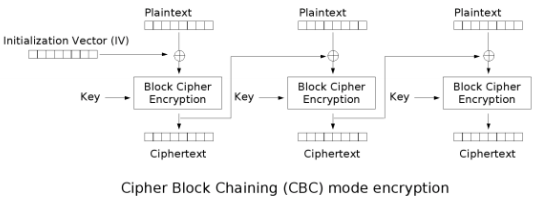
# Introduction to Computer Security – Revision Guide

## Introduction

* What does Computer Security protect?
  + **Confidentiality** – Attacker can’t read my data.
  + **Integrity** – The data I receive is genuine.
  + **Availability** – I can get my data when I need it.

## Cryptography

* **One Time Pads**
  + These provide **perfect encryption**…
    - …but we need a key as long as the message.
    - We add the plaintext and the key to produce the cipher text.
      * Plaintext: HELLOALICE
      * Key: SGFKPQYEIJ
      * Ciphertext: ALRWERKNLO
* **Block Ciphers**
  + They are made up of a series of ***permutations*** and ***substitutions***repeated on each block.
    - **Shuffle, substitute and XOR.**
    - The **key** controls the exact nature of the permutations and substitutions.
  + **AES** – Advanced Encryption Standard
    - A state-of-the-art block cipher that works on blocks of 128-bits.
      * The backbone of internet security.
    - It generates **10 round keys** from a **single** **128-bit key**.
  + **DES** – Data Encryption Standard
    - This was the previous block cipher standard. It was designed by IBM in the 70s.
  + **3-DES**
    - Triple DES was a stop gap until AES.
    - 3-DES takes **three keys**: K1, K2 and K3.
      * Setting K1 = K2 = K3 gives you DES!
    - It is still used in bank cards and RFID chips.
  + **Padding**
    - Block ciphers only work on **fixed size blocks**.
      * If the message isn’t the correct size we need to add **padding** to it.
      * But the receiver needs to be able to tell the difference between the padding and the message.
    - PKCS 5/7
      * If there are 3 bytes of space write 030303
        + And so on…
      * If the message goes to the end of the block add a new block of 16161616161616161616161616161616.
        + This is because **the message *needs* padding**, because otherwise we would not be able to tell if there *was padding or not*.
      * PKCS 5: 8 byte block / PKCS 7: 16 byte block
  + Block Cipher Modes
    - **ECB – Electronic Codebook Mode**
      * Each block is **encrypted individually**.
      * Ciphertext blocks are assembled in **the same order** as the plaintext blocks.
      * **Vulnerability:** If blocks are repeated in the plaintext, this is revealed by the ciphertext. An attacked could use **frequency analysis**.
    - **CBC – Cipher Block Chaining Mode**
      * Each block is **XOR’d with the previous block**.
      * Start with a random **Initialisation Vector** (**IV**).



* + - * **Probabilistic encryption** schemes use **random elements** to make *every encryption different* – removes threat of **frequency analysis**.
        + CBC with a **random IV** is a good way to make encryption probabilistic.

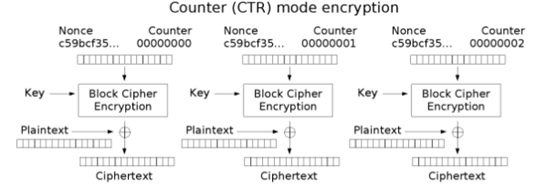
This lets me encrypt the same message, with the same key, without an attacker realising.

* + - **CTR – Counter Mode**
      * Plaintext = B1, B2, …, Bn
        + IV = random number (sent in the clear)
        + Encrypt

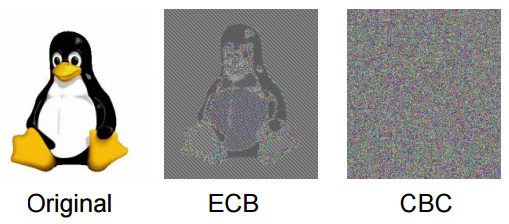
C1 = B1 xor encryptK (IV)

C2 = B2 xor encryptK (IV+1)

… Ci = Bi xor encryptK (IV+i-1)



* + - Which block chaining mode should you use?
      * **ECB** when the length of the data encrypted is the same as the key (i.e. you are encrypting one block of data).
      * **CBC** / **CTR** when encoding multiple blocks, so we can use **probabilistic encryption**.

****

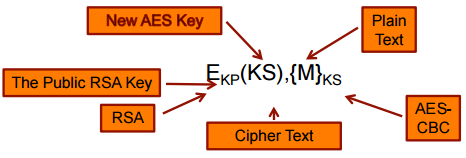
* **Public Key Cryptography**
  + The previous types of crypto we discussed required a singular shared key.
    - But this does not always work… we do not always want to have to send a completely secure key every time we wish to buy something from Amazon!
  + The idea of **public key encryption** is that you have two keys:
    - A **public key** for encryption
    - A **private key** for decryption
  + **Diffie-Hellman** is a widely used **key agreement protocol**.
    - It relies on some *modular arithmetic* (a mod b = n).
    - It also uses two *public parameters*:
      * A **generator** “g” (often 160 bits long)
      * A **prime** “p” (often 1024 bits long)
    - How Diffie-Hellman works:
      * Alice and Bob pick random numbers (rA) or (rB).
      * They calculate (tA = grA mod p) or (tB = grB mod p).
      * The protocol then simply exchanges these numbers:
        + A 🡪 B : tA
        + B 🡪 A : tB
      * Alice calculates (K = tBrA mod p) and Bob calculates (K = tArB mod p)
      * This gives them each the key:
        + K = grArB mod p
    - An observer cannot work out rA and rB from tA and tB.
      * Therefore, they cannot calculate the key!
      * We did not need to share keys at the start – this is a powerful protocol.
    - So we have a “**Good Key**”.
      * But we do not have **authentication** of *who* we are communicating with.
      * In practice, we use DH to set up a secure channel and then use something else to **authenticate** the person at the other end.
  + **Elgamal**
    - This is Diffie-Hellman being used *as* a public key crypto scheme.
      * **Private key:** rA
      * **Public key:** tA
    - To encrypt message M, Bob picks rB and finds and sends...
      * … (grB mod p, M.(tArB mod p) )
  + **RSA**
    - This is a public key crypto scheme invented by Rivset, Shamir & Adleman.
      * It is more efficient than Elgamal and can be used for **signing**.
    - How RSA works (in brief):
      * The scheme generates e, d and n.
        + Public key is (e,n)
        + Private key is (d,n)
      * Because of how e, d and n are generated:
        + To encrypt message “m” as ciphertext “c”:

c = me mod n

* + - * + To decrypt ciphertext “c” as message “m”:

D = cd mod n

* + Using Public Key Crypto
    - Public key crypto is **much slower** than symmetric key!
    - So instead of just using public key crypto, we:
      * Make a new **symmetric key** (session key)
      * **Encrypt that** with the public key
      * **Encrypt the message** with the symmetric key

****

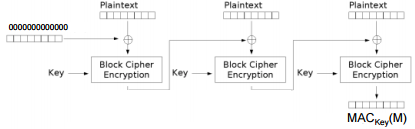
* + **Signatures**
    - Using RSA: Epub(Dpriv(M)) = M
      * This is because the encrypt and decrypt operations in RSA cancel each other out – they can be performed in any order.
    - This can be used to **sign messages**.
      * We sign a message with the private key and this can be verified with the public key.
      * No real crypto suite will use the same key for encryption and signing, as this can be used to trick people in to decrypting.
  + Saving a key
    - **KeyStores** provide protected storage for keys.
      * *Password protected* private keys
      * Public keys are stored as *certificates*
    - Certificates and Key Servers
      * A public key certificate **binds a public key to an identity**.
        + As well as the public key it contains a name, email address, etc.
        + It is **signed with the private key** and anyone else that trusts it.
      * **Key servers** store public key certificates.
        + Many clients can automatically search a key server for unknown email addresses.
        + But beware, there is not guarantee that the key is not a fake.
* **Hashes** and **MACs**
  + Hashes
    - A **hash** of any message is a *short string* generated from that message.
      * The hash of a message is always the same.
      * Any small change makes the hash totally different.
      * It is very easy to hash a message.
      * It is very difficult to go *from* the hash *to* the message.
      * It is very unlikely that any two different messages have the same hash.
    - Uses of Hashing:
      * **Signatures**
        + We can use hashes to extend how signatures work.
        + Alice wants to sign message M with her private key.

Instead of signing the message, we sign the hash.

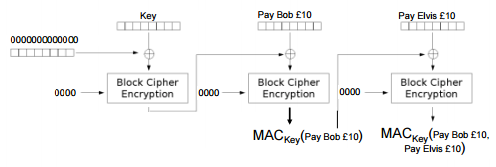
This is more efficient!

She sends: M,Dpriv(#(M))

* + - * **Message verification**
        + Including the order of the message, as it is hashed as a whole.
      * **Password protection**
        + We store the hash in the database, not the password.
    - **Attacks** on Hashes
      * **Preimage Attack:** Find a message for a given hash – **very difficult!**
      * **Collision Attack:** Find two “random” messages with the same hash.
        + **Prefix Collision Attack:** A collision attack where the attacker can pick a prefix for the message.
  + **MACs** – Message Authentication Codes
    - MACs are hashes with a key.
      * Written MACKey(M).
      * You can only make or check the hash if you know the key.
      * This stops guessing attacks.
    - They are used for **authentication**.
      * Alice and Bank share keyA.
      * Alice sends to Bank: “Pay Bob £10”,MACkeyA(“Pay Bob £10”).
    - Possible **attack** on MAC
      * **Length extension attack:** Add data to a MAC without knowing the key.
    - Making a CBC-MAC – we must have:
      * A **key**.
      * The **hash** of M.
      * The **totally different hash** of M *with a single bit-change*.



* + The **SHA** Family of Hashes
    - The most common (and best) hashes are the **SHA hashes**.
    - SHA-1
      * In 2005, a 263 attack was found.
    - SHA-2
      * An improved version of SHA-1 that uses a **longer hash**.
        + 256 or 512 bits – also called SHA256 or SHA512.
      * But is based on SHA1, so it has some of the same weaknesses.
    - The SHA-3 Competition
      * A competition was held from 2008-2012 to develop the next SHA.
      * This is expected to soon be in the standard APIs.
  + The **MD** (Merkle-Damgard) Family of Hashes
    - **MD hashes** are also popular. MD4 and MD5 are used, but weak.
      * They are only useful when we only care about preimage attacks or integrity.
  + Broken Hash to MAC
    - If we had a hash, we could try to make a MAC by:
      * MACKey(M) = H(Key,M)
    - But this allows for a **length extension attack**:

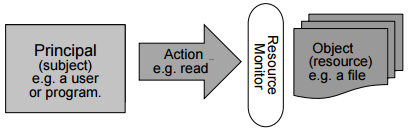


* + - To stop this, we use **HMACs**:
      * HMACKey(M) = H( (K xor opad),   
         H( (K xor ipad), M) )
        + opad and ipad are constants (e.g.: 0x5c36..90)

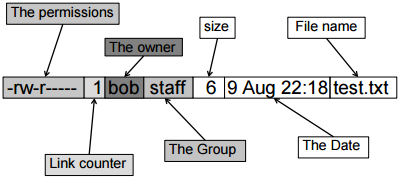
* Authenticated Encryption
  + **Encryption != Authentication**
  + Effect of ciphertext alterations in different block modes:
    - CBC – Any change affects all of the rest of the message.
    - ECB – Any change affects only the block.
    - CTR – Any change affects only the corresponding bits.
  + This means that **if I know the plaintext I can change CTR encrypted messages**:
    - If I know Enc(M1) and I know M1, I can make a ciphertext that decrypts to any message I want, M2.
      * Dec( Enc(M1) xor (M1 xor M2) ) = M2
  + **Authenticated Encryption Modes**
    - With **authenticated encryption**, you can only form a valid ciphertext if you know the key.
    - The most common way to do this is to **add a MAC to the ciphertext**.
    - **CCM Encryption** (RFC 3610)
      1. Calculate an AES CBC-MAC on the data.
      2. Encrypt the message followed by the MAC using *the same key* but in CTR mode.
         * If I change a bit in the MAC, the MAC check fails.
         * If I change a bit in the message, the MAC check fails.
         * If I change a bit in both the MAC and the message, the likelihood of the MAC check succeeding is astronomically small.
    - This is **authenticated, unbreakable encryption**.

## Access Control

* Model of Access Control
  + **Access control** is all about **permissions**.



* + **Access Control Lists** – ACL
    - Unix/Linux/Mac use ACLs with groups.
      * uid set when you log on.
        + root owns everything (“get root” = control the system).
      * Linux Kernel then dynamically enforces the ACLs.
      * “ls –l” displays files with their ACL.



* + - * + UNIX File Permissions:

“- --- --- ---” at the start of the “ls –l” entry.

First bit is the **file type**.

- : file

D : directory

b/c : device file

1st set of three bits are the **owner permissions**.

2nd set of three bits are the **group permissions**.

3rd set of three bits are the **default permissions**.

* + - * + Access Control for **Directories**

“r” is read-only for directory contents.

Without this I can’t list the files in the dir.

“x” is the permission to traverse (switch to, run files).

Without this I can’t run commands in the dir.

* + - * + Access Control for **Processes**

“x” is the permission to run the process.

“s” permission indicates that the process runs with the *permissions of its owner*.

* + - The Confused Deputy Problem
      * Users can run “s” programs with more privileges.
        + If there was a mistake in an “s” program that was owned by root, we could use it to perform root-only actions.
      * The **Confused Deputy Problem** is when a low-level attacker gets to a high-level process by misusing its authority.
        + *Make sure processes have as low a level as possible.*

## Protocols

* The Internet
  + TCP/IP
    - TCP is a protocol that runs on top of IP.
      * If an IP packet gets lost, it requests that it is resent.
  + Netcat & Nmap
    - Netcat is a tool to make Internet conections.
      * Listen on port 1337: nc –l 1337
      * Connect to localhost on port 1337: nc 127.0.0.1 1337
    - Nmap checks if the 1000 most common ports are open: nmap 127.0.0.1
      * It also sends messages to ports to find out what the service is.
  + **Packet Sniffing**
    - Packets can be intercepted in transit and their data read.
      * This is why we need to use **protocols** to ensure secure communication.
    - Wireshark is a network protocol analyser that **records all network traffic so that it can be viewed and analysed**.
  + ***“The attacker owns the network.”***
    - The Internet was not designed with security in mind.
    - We know that traffic may be monitored or altered.
    - Therefore, all good security products assume that the attacker has complete control over the network (but can’t break encryption).
* Protocolsin **Alice and Bob Notation**
  + Alice and Bob Notation
    1. A -> B : “I’m Alice.”
    - This **unencrypted** message can be read by the attacker.
    - The attacker can pretend to be anyone.
      1. E(A) -> B : “I’m Alice.”
  + A Simple Protocol
    - If Alice and Bob share a key “Kab”, then Alice can encrypt her message.
      * **{**…**}Kxy** means symmetric key encryption.
      1. A -> B : {“I’m Alice”}Kab
    - But the attacker can intercept and replay messages!
      1. A -> E(B) : {“I’m Alice”}Kab
      2. E(A) -> B : {“I’m Alice”}Kab
    - This is why we always assume the attacker “owns” the network.
  + Using a **Nonce challenge** (random number) gives **verification**
    1. A -> B : A
    2. B -> A : {Na}Kab
    3. A -> B : {Na+1}Kab , {Pay Elvis £5}Kab
    - We have to send back Na+1, because it requires us to manipulate the data.
      * This means simply replaying the message is not enough.
    - The weakness here is that the two parts of message three are not tied together.
      1. A -> B : A
      2. B -> A : {Na}Kab
      3. A -> B : {Na+1}Kab , *{Pay Elvis £5}Kab*
         * E is listening to this message…
      4. A -> B : A
      5. B -> A : {Na2}Kab

A -> E(B) : *{Na2+1}Kab* , {Pay Bob £5}Kab

1. E(A) -> B : {Na2+1}Kab , {Pay Elvis £5}Kab
   * A Better Protocol
2. A -> B : A
3. B -> A : {Na}Kab
4. A -> B : {Na, Pay Elvis £5}Kab
   * + We do not have to send back Na+1, because we are adding to our message.

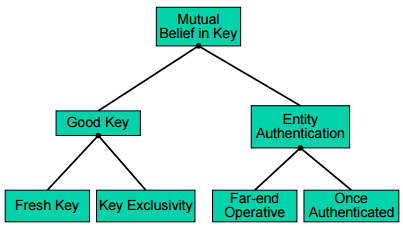
* **Key Establishment Protocols**
  + That last protocol was possible because A and B shared a key.
    - Often the principals need to set up a session key using a “**Key Establishment Protocol**”.
    - They must either **know each other’s public keys** or use a Trusted Third Party
  + **The Needham-Schroeder Key Establishment Protocol**
    - **The Needham-Schroeder Public Key Protocol**
      * Assume Alice and Bob know each other’s public keys.
      * They can set up a symmetric key.
    - **Forward Secrecy**
      * A protocol has “**forward secrecy**” if it keeps the message secret from an attacker who has **a recording of the protocol run** *and* **the long term keys** of the principals.
      * **Station-to-Station Protocol**
        + This adds in the Diffie-Hellman protocol!
        + **Certificates**

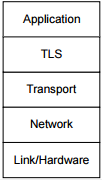
What if Alice and Bob don’t know each other’s public keys at the start?

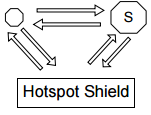
They could meet face-to-face and set up keys…

...or get a Trusted Third Party (TTP) to sign their identity and public key.

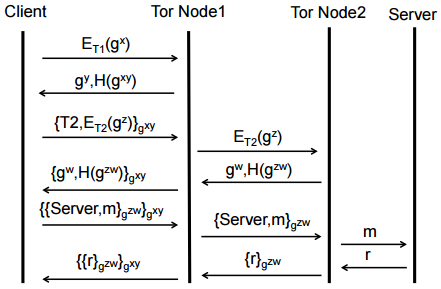
* + - * + The “full” STS protocol adds certificates for A and B.



* 3 State-of-the-Art Security Protocols:
  + **Kerberos** – Key establishment using TTP
  + **TLS** – Secure web traffic
  + **Tor Onion Routing** – Best anonymity
* **Kerberos**
  + This is a protocol for **key establishment using a TTP** *with* **authentication**.
  + It has corporate use, where many computers are owned by the same business.
* **TLS**
  + This protocol is most prominently used to **secure web traffic**.
    - It is possibly the most common security protocol out there.
  + It provides **encrypted socket communication** *and* **authentication**, based on public keys.
  + It may use a range of ciphers (RSA, DES, DH) that are **negotiated at the start of the run**.
  + At the heart of this protocol are **certificates**.
    - **X.509 Standard Certificates** contain a subject, their public key, issuer name, etc.
    - The **trusted issuer** signs the hash of all the data.
      * To check a certificate I hash all the data and check the issuer’s public key
      * If I have the issuer’s public key, and I trust the issuer, I can then be sure of the subject’s public key.
  + The TLS Layer runs between the Application and Transport layers of the IP Stack.
    - The encryption is **transparent** to the application layer.
    - Normal TCP/IP protocols etc. can still be used at the low layers.
  1. C -> S : Nc
  2. S -> C : Ns, CertS
  3. C -> S : ES(K\_seed), {Hash1}Kcs
  4. S -> C : {Hash2}Kcs
     + Hash1 = #(Nc, Ns, Es(K\_seed))
     + Hash2 = #(Nc, Ns, Es(K\_seed), {Hash1}Kcs)
     + Kcs is a session key based on Nc, Ns and the K\_seed.
  + All data is then **encrypted with Kcs and hashed for integrity**.
  + The **server is verified to the client** – but not vice versa!
    - There exists an alternative version of TLS wherein the client also sends a certificate – but this is not realistic.
* **Tor’s Onion Routing**
  + This protocol provides the **best anonymity** you can get on the Internet.
  + “You have zero privacy anyway, get over it!”
    - But we can **protect our privacy**.
  + **VPNs** securely connect you to another network.
    - For example, you can connect to printers via the school’s VPN.
    - This is secured with certificates and encryption, e.g.: VPN.
  + **Proxies** – VPNs for **Anonymity**
    - An Internet connection reveals your IP number.
      * This promises “anonymity”…
    - To get some anonymity, you can **route all your traffic** via a VPN.
      * The server thinks you are the VPN provider.
      * The ISP only sees the connection to the VPN.
    - *But*, there is no anonymity to the VPN.
      * A global observer could probably link your connections.



* + **Onion Routing**
    - You get the **best anonymity** by routing your traffic via *many different* proxies.
      * Protocol ensures your traffic *really is* routed via the proxies you want.
      * The **TOR network** uses this protocol.
    - Each proxy only learns the IP of the proxy before and after it.
      * Source IP is visible to the first proxy.
      * Destination IP is visible to the final proxy.
    - The public key of each proxy is known.
    - User picks three proxies, and is anonymous as they **aren’t *all* corrupt**.
      * Only in danger in the unlikely chance that *all three* are corrupt.



* + - The above example only shows two nodes, but can easily be extended to more.
    - Tor’s Hidden Service Protocol **hides the server** *as well as* the client.
    - **Anyonmity != Security** – The webservers can still be attacked.
      * For example, the FBI planted malware on criminal sites running on Tor.

## Web Security

* **SQL Injection Attacks**
  + …cart.php/?product=X
    - The web server may look up “X” in an SQL database using:
      * SELECT \* FROM cart WHERE (id=’$\_GET[product]’);
    - What else could we use instead of X?
      * X = ‘); DROP TABLE cart; --
        + … WHERE (id=’’); DROP TABLE cart; --
  + Secret Item: “dh2aeb£”
    - Perhaps the server gets the secret item details by performing:
      * SELECT \* FROM user WHERE (item=’dh2aeb£’);
    - But what if…
      * X = ‘ OR ‘1’=’1
        + … WHERE (item=’’ OR ‘1’=’1’);
    - Obviously 1=1, so we get the secret item!
  + **Countermeasure**
    - You can stop SQL attacks by checking the input:
      * $pass = **mysqli\_real\_escape\_string**($con,$pass);
    - This code **sanitises** (escapes all SQL characters in) the input.
  + Remember – any source of data can be used for an SQL attack:



* **XSS** (Cross Site Scripting) **Attacks**
  + Web browsers are dumb: they will execute anything the server sends them.
    - Can an attacker **force a website to send** you something?
  + **XSS** is an **input validation vulnerability**.
    - It allows an attacker to **inject** client-side code (JavaScript) into web pages.
    - This is then served by a vulnerable web application to other users.
  + **XSS Methods**
    - Drive-by-Download
      * The attacker **injects malicious code** into the vulnerable web server.
      * The victim visits the vulnerable web server.
      * Malicious code is **served to the victim** by the web server.
      * **Malicious code executes** on the victim with web server’s privileges.



* + - **Reflected XSS**
      * The injected code is **reflected** off the web server:
        + An error message
        + Search result
        + Response includes some/all of the input sent to the server as part of the request
      * Only the user issuing the malicious request is affected.
      * For example…
        + Server code:

out.println(“Results for ” + request.getParameter(“searchQuery”));

* + - * + User inputs the following as the search query:

<script>alert(“pwnd”);</script>

* + - **Stored XSS**
      * The injected code is **stored on the website** and served to its visitors on **all page views**.
        + **All users** are affected
      * User **messages and profiles** are good methods of Stored XSS.
      * For example…
        + Server code:

out.println(“Forename: ” + request.getParameter(“fname”));

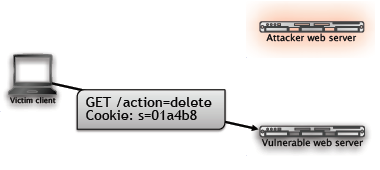
* + - * + User inputs the following as their forename:

<script>alert(“pwnd”);</script>

* + **Possible XSS Attacks**
    - Steal Cookie
      * JavaScript can **access cookies** and make remote connections.
      * An XSS attack can be used to **steal the cookie** of anyone who looks at a page, and send the cookie to the attacker.
        + Proof of concept: window.alert(document.cookie);
      * The attacker can then **use this cookie** to login as the victim.
    - Phishing
      * The attacker injects script that **reproduces look-and-feel** of an “interesting” site (for example, the PayPal login page).
      * The fake page asks for the **user’s credentials** (or other sensitive information).
      * This data is **sent to the attacker**’s site.
    - Redirect
      * The attacker injects script that auto-redirects victims to their site.

<script> document.location = “http://www.evil.com/”; </script>

* + - Run Exploits
      * The attacker injects a script that **launches various exploits** against the user’s browser or its plugins.
      * If the exploits are successful, **malware is installed** on the victim’s machine without any user intervention.
      * Often, the victim’s machine becomes part of a botnet.
  + **Solution –** Sainitisation
    - Sanitising *all* user inputs is difficult.
      * Sanitisation is context-dependent: JavaScript, CSS, URLs, etc.
      * Should we blacklist or whitelist? Reuse code or make our own?
    - Twitter blocked “<script>” but not quotes:
      * <a href=”” onmouseover=”pwnd”/>
    - The PHP command Htmlspecialchars() removes characters that might cause problems in HTML: %, <, >, ‘ and “.
* **CSRF** (Cross Site Request Forgery) **Attacks**
  + **Method**
    - Victim is logged into vulnerable website
    - Victim visits malicious page on attacker website
    - Malicious content is delivered to victim
    - Victim involuntarily sends a request to the vulnerable website

****

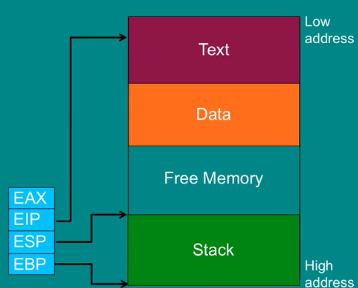
* + **Solutions**
    - Check the value of the Referer header:
      * Attacker cannot spoof the value of the Referer header in the user’s browser (but the user can).
      * Legitimate requests may be stripped of their Referer header:
        + E.g. Proxies, Web application firewalls
    - Every time a form is served, add an additional parameter with a secret value (**token**) and check that it is valid upon submission.

<form>…<input name=”csrf” type=”hidden” value=”aed123f”/>…</form>

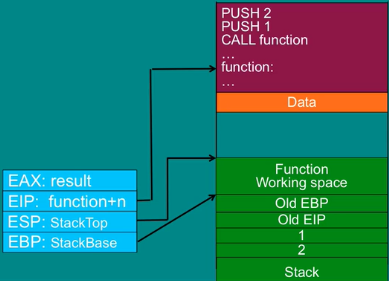
* + - * If the attacker can guess the token, there is no protection.
      * If the token is not regenerated each time the form is served, the application may be vulnerable to replay attacks.
* **The OWASP Top Ten**
  + **A1: Injection**
  + **A2: Broken Authentication**
  + **A3: XXS**
  + **A4: Insecure Direct Object Reference**
  + **A5: Security Misconfiguration**
  + **A6: Sensitive Data Exposure**
  + **A7: Missing Function Level Access Control**
  + **A8: CSRF**
  + **A9: Using Components with Known Vulnerabilities**
  + **A10: Invalidated Redirects and Forwards**
* To conclude…
  + To secure a website, you need to know how it works:
    - How clients request resources
    - How clients are authenticated
    - How HTTP and webservers work
  + Errors are often down to bad app logic.
  + **Always sanitise everything!!!**

## Buffer Overflow Attacks / Reverse Engineering

* **Buffer Overflow Attacks**
  + Introduction
    - In languages like C, you have to tell the compiler how to manage the **memory**.
    - If you get this wrong, an attacker can **exploit** this bug to make your application run ***their code***.
    - This is responsible for a huge amount of sophisticated attacks.
  + **x86 Architecture**
    - Memory:
      * Low address
        + The program code (**Text**)
        + Static variables, strings, etc. (**Data**)
        + **Free Memory**
        + Data in use (**Stack**)
      * High address
    - General Registers:
      * **EAX** – The **Accumulator** (used for calculations)
      * **EIP** – (Next) **Instruction Pointer**
      * **ESP** – **Top of Stack Pointer**
      * **EBP** – **Bottom of Stack Pointer**



* + **Function Calls**
    - When we **call functions** we need to…
      * ...**PUSH arguments**.
      * …change the **address of the EIP** to point to the new function.
      * …**PUSH the old EIP** so we can return later.
    - function(1,2);
      * PUSH <2>
      * PUSH <1>
      * CALL <function>
        + PUSH <EIP>
        + EIP = \*function
    - The function then requires a new section of the stack, so that it does not interfere with the previous one.
      * PUSH <Old EBP>
      * EBP = ESP
      * ESP = [New ESP]



* + The **EIP** controls which code executes…
    - …the EIP is stored on the stack…
    - **...I can write to the stack. >:)**
  + **The Attack**

function(input);  
function(char \*str) {  
 char buffer[16];  
 strcpy(str,buffer);  
}

* + - But what if the input is more than the 16-bytes of buffer[16]?
      * The **buffer overflows**!
      * The remaining bytes go down the stack, in to the EBP **and EIP**.
    - **Malicious version:**
      * The attacker sends a very long message, ending with the **memory address** of some code that gives him a shell.
        + This address needs to be **raw bytes**, not ASCII.
        + This code could also be part of the message.

e.g. **Injection:** {Code to open a shell}, {Padding},  
{Address of Start of Injection}

* + - * The attacker’s value **overwrites the EIP**.
      * When the function returns, the **attacker’s code is run**.



* + To conclude:
    - **Buffer overflows** are the result of **poor memory management** in languages such as C.
    - They exploit these bugs to **overwrite memory values**.
    - This often lets an attacker **execute arbitrary code**.
* **Buffer Overflow Defences**
  + **Good Programming**
    - **Vulnerable:** gets(buffer)
      * **Safe:** fgets(buffer,**n**,stdin)
    - **Vulnerable:** strpy(dest,src)
      * **Safe:** strlcpy(dest,src,**length**)
    - **Vulnerable:** sprintf(string,format)
      * **Safe:** snprintf(string,**length**,format)
    - This pretty much solves the problem… but *people* don’t use it (like **sanitisation**).
      * Instead we develop secure *systems*…
  + **NX-bit**
    - In a normal program, code should be in the **text** area of memory – not the **stack**!
      * The **NX-bit** provides a hardware distinction between the text and stack.
        + When enabled, the program will **crash if the EIP every points to the stack**.
    - This is **on by default** in gcc.
      * It is sometimes turned off to allow dynamic code construction (e.g. when code is decrypted at runtime, so it is on the stack).
      * But usually the protection is on – so **must be overcome by attackers**.
    - **Overcome** – *Reuse their code*
      * Jump to another function in the program.
      * Jump to a function from the standard C library (**Return-to-libc**).
        + libc is often packaged with executables to provide a runtime environment.
        + It includes lots of useful calls like “system”, which runs any command.
        + It links to executable memory, therefore bypassing NX-bit protections!
      * String together pieces of existing code (**Return-Oriented Programming**)
        + How **returning from multiple function calls** works:

Reset **stack** frame (EBP), go to **return address** (EIP)

Reset **stack** frame (EBP), go to **return address** (EIP)

…

* + - * + We can **construct multiple return addresses** by injecting multiple EBPs and EIPs (simulating multiple functions).
        + This means that we can force the program to run many different lines of *its own**code* in the *order* we want!

With a big enough program, we can get almost any functionality we want using the **ROP** method.

We can use this in addition to libc for maximum functionality.

* + **ASLR** (Address Space Layout Randomisation)
    - **ASLR** simply adds a **random offset** to the *stack and codes’ base* each time the program runs.
      * Jumps in the program are automatically altered to point to the right line.
    - The idea is that it is **difficult** for an attacker to **guess the address** of where they inject code, or the addresses of particular functions.
    - **On by default** in all OS.
    - **Overcome** – *NOP slide*
      * In x86 the opcode assembly instruction 0x90 does nothing.
      * If the stack is 2MB, I could inject nearly 2MB of 0x90 followed by my attack code (after the return pointer).
      * I then **guess a return address** and hope it is somewhere in the 2MB of NOPs.
        + If it is, the program ‘slides’ down the NOPs to my attack code.
  + **Stack Canaries**
    - These place a **random value** onto the stack at the **start** of every function.
      * This value is **checked before the program returns**.
      * If the attacker has overwritten the return pointer, they will have also overwritten the canary – the values will not match and the program will crash.
    - **Overcome** – *Guessing/learning the Canary Value*
      * We can **learn the canary value** by, for example, performing a **format string attack**:
        + printf(“string %s, hex %x”, a, b)

This prints a as a string and b as a hex.

* + - * + printf(“hex %x, hex %x, hex %x”, a)

This print a as a hex, and **the next two items on the stack** as a hex.

* + - * + This lets us see what is on the **stack**.

This includes the **address of stack variables and functions** – including the **stack canary**.

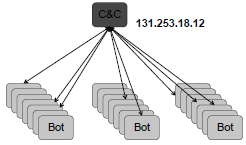
Additionally, **%n lets us write to these variables**.

This is yet another sanitisation problem -- %x, %s, %n must be escaped in any string passed to printf.

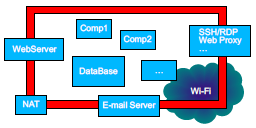
* Additional Buffer Overflow Information
  + **x64 Assembler**
    - x64 stores the **return address in a register**.
      * But when a second function is called, the previous return address is put on the stack and we *can* attack this.
    - x64 has a much **larger address space**, so **ASLR** is much more effective and **brute-force becomes impossible**.
  + **Overwriting other values**
    - Sometimes it is possible to overwrite other parts of memory, such as a key variable.
    - If you can overwrite a buffer that is placed above an unknown password, you can overwrite the password with one you know.
    - If the program has a pointer to the next function, you can overwrite this and make it execute the function you want.
  + These attacks are difficult to implement.
    - With all the protections it can take a team of experts six months to “weaponise” a buffer overflow.
    - As soon as the bug is found by the developer, they will patch it.
  + **Metasploit** is a framework for testing/executing **known buffer overflow attacks**.
    - It catalogues these known bugs as modules.
    - If an application is unpatched, it can probably be taken over with it.
    - It also includes a library of shell code which can be injected.
* **Reverse Engineering**
  + Introduction
    - **Compiled code** is just data.
      * Data can be **edited** and **inspected**.
      * By **examining low-level code**, protections can be removed and the **function of programs altered**.
      * Good protection tends to slow down this process, not stop it.
    - Reasons for **Reverse Engineering**:
      * Analyse Malware
      * Debug Memory Errors
      * Analyse Legacy Code
      * Security Auditing
  + **Java Byte Code**
    - A Java program (.java) is compiled to **Java Byte Code** (.class).
      * javap –c <ClassName> allows you to see a class’ byte code.
      * E.g. “iconst\_0”, “goto”, “iadd”…
    - This is then ran on the OS’ version of the JVM as **assembly**.
  + **Decompilation**
    - It is much easier to work with the **source code** that the Java Byte Code.
    - The program **JD-GUI** is a Java **decompiler** that transforms Java Byte Code in to Java Code.
      * Although this is not perfect.
    - **Bypassing the Password Check**
      * Decompilation makes it easier to understand what a program is doing.
      * It also makes code easy to alter and recompile.
      * All code that is used to protect the code can be removed.
  + **Binaries**
    - Binaries are written in **assembly**.
    - This is a much lower-level than Java Byte Code.
    - Assembly compiled for one type of machine won’t run on another.
      * But the same techniques apply.
    - The program **IDA** is an **I**nteractive **D**is**A**ssembler.
      * It helps a human understand binaries.
      * This is the best tool for malware binary analysis, security analysis of firmware… and **reverse engineering**.
  + **Common Reverse Engineering Techniques**
    - Look for strings
    - Identify key tests and check the values in the register using a debugger
    - Swap JE and JNE
    - Jump over the instructions that perform checks.
  + **Defences** against Reverse Engineering
    - **Dynamically** construct keys/passwords
      * But the attacker can run the code
    - **Encrypt** the binary
      * But the key must be included somewhere in plaintext, so can be found
    - **Obfuscate** the code (i.e. make it complex, mix data with code, etc.)
      * This does not stop attacks but can slow down significantly.
    - Require online activation.
      * Although activation can be completely disabled, and users don’t like it
    - Require online content
    - Hardware-based protection (store part of code in tamper-resistant hardware)
  + **Summary**
    - **Machine code** can be **inspected** and **edited**
    - Many **tools** exist to **inspect**, **debug** and **decompile code**
    - Most **software protection can be removed**
    - But **slowing this down** by months or years can save a business

## Common Threats

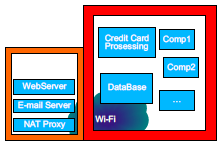
* Common **Internet Threats**
  + **Botnets**
    - Most attackers are in it to **make money**
      * A single credit card number/spam e-mail isn’t worth much
        + Networks of hacked computers (**bots**) are organised in to large networks (**botnets**)
        + They are controlled by a **C&C** (Command & Control)

****

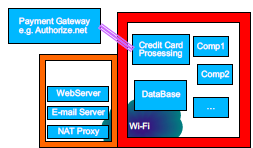
* + - **Denial-of-Service Attack**
      * With this many computers, it’s easy to **overload** some website
      * Easiest type of attack uses (rents) a **botnet to perform a DDoS**
      * Often used to **blackmail companies**, or for **political reasons**
    - **Fast Flux**
      * Instead of using an IP address, bots look for a URL
      * New IP addresses are registered for the C&C every few minutes
      * This makes it impossible to go after the hosts
    - **Domain Flux**
      * Bots continuously generate new URLs
        + Based on a hash of the date and a secret value
      * Bot-masters know and register the URL in advance
      * Even if all C&Cs are shut down, bots will switch to a new URL soon
      * We can try to block all future URLS; but this is difficult
    - **Conflicker** 
      * This is a computer worm that installs a botnet.
      * It has largely been contained by security researchers who have blocked tens of thousands of domain names.
    - **P2P**
      * More recent malware sets itself up as a P2P network
      * Malware connects to a C&C and other bots
      * If the main C&C goes down, bot-masters can connect to any bot and update them all with a new C&C
* Common **System Exploits**
  + Attacks on Networks
    - Common Attack Vectors on Networks:
      * Webpage / Incoming e-mail / All servers / Local Wi-Fi / Insider Attacks



* + - Are any services using the **default password**?
      * Attackers can ssh in on the remote login port (22)
    - **Known Memory Exploits**
      * Memory based attacks on services (e.g. using metasploit)
      * New, unknown exploits are very rare!
        + You are much more likely to be attacked via a known one
    - **Phishing Sites**
      * Phishers set up **fake sites** to look like real ones.
        + Send spam asking user to connect
      * Redirection by other malware
      * **Spear Phishing**
        + Random phishing e-mails don’t stand much chance
        + Spear phishing refers to very carefully crafted, targeted phishing e-mails
    - **Viruses** & **Worms** are **self-replicating** programs
      * Viruses **require interaction** to spread; worms can **spread on their own**
    - **Trojans**
      * These are malware that needs a user to download and run them.
      * The malware is often **masquerading** as something else, e.g. a keygen
    - **Web Attacks against the Server**
      * SQL Injection the most common)
        + Many other kinds of injection…
      * Most usually leads to data leakage from DB, DoS, defaced website…
    - **Web Attacks against the Client**
      * Drive-by-download is one of the biggest sources of malware
      * Viewing a page with a vulnerable browser and OS will get you infected
      * Often JavaScript based, via XSS
      * Can give the attacker complete control of your computer
    - **Insider Attacks**
      * Businesses need to consider attacks *from* their employees
        + E.g. Edward Snowden
      * Separation of powers, background checks, keep staff happy, fire them quickly…
  + Defences
    - **Firewalls**
      * Firewalls **block Internet traffic**
      * May be on the computer (**host**) or built into a router (**network**)
      * Firewalls can be ‘stateless’ or ‘statefull’
        + **Stateless firewalls** could block all traffic not on port 80
        + **Statefull firewalls** record the traffic and use it to make future decisions



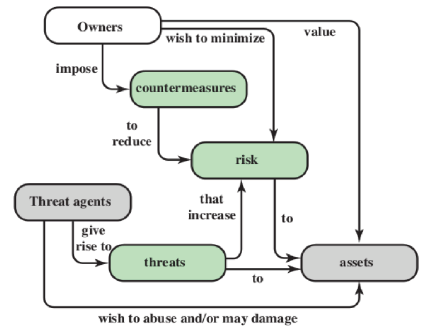
* + - **Fast Patches / Patching**
      * The **most important** defence
      * Make sure all **security patches** are installed immediately
      * There is almost always a patch to **stop any known exploit**
    - **Anti-Virus**
      * These products **scan the computer** for **known malware**
      * They can also scan e-mail and network traffic
      * They are **only as good as the last update**
      * Can be **disabled by an attacker** with admin access
  + **Cyber Security Policy**
    - NIST defines “Policy” as **documentation** of **computer security decisions**.
      * Usually the needs of the business come first, the security comes second.
    - **PCI-DSS** (Payment Card Industry Data Security Standard)
      * All organisations that handle **credit card data** *should* comply with the **PCI-DSS standard**.
      * Card payments could be refused for non-compliant organisations.
        + In practice, most of the time, non-compliant organisations will only get into trouble if there is a problem or an audit.



* + - **ISO 27001**
      * **ISMS** (Information Security Management System)
        + An ISMS must be **continually monitored**.

**Reports** of new faults, IDS monitoring, **patch policy**…

* + - * + If an organisation’s activities shift, the **ISMS needs an update**.
      * **ISO 27001** is the **international standard** on how to do an ISMS.



* + - * + Establish the ISMS

Define the organisation and what it does

The scope of ISMS (what’s in it and what’s not)

**Assets**

* + - * + Identify the **risks**

Identify the **assets** within the scope of the ISMS and their owners

Identify the **threats** to each of those known assets

Identify the **vulnerabilities** that might be exploited

Identify the **impact of loss** of each asset

Is it confidentiality, integrity and/or availability?

Identify the **likelihood** of each threat

**Risk = Impact x Likelihood**

* + - * + For example…

**Asset:** Purchase History

**Threats:** Loss, Corruption, Out of Date, Stolen

**Vulnerabilities:** Bugs, SQL injection, Insider attack, Fire

**Impact of Loss:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Loss** | **Corruption** | **Out of Date** | **Stolen** |
| 2 | 3 | 1 | 5 |

**Likelihood:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Bugs** | **SQL Injection** | **Insider Attack** | **Fire** |
| 2 | 3 | 5 | 1 |

**Risk:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Loss** | **Corruption** | **Out of Date** | **Stolen** |
| **Bugs** | 4 | 6 | 2 | - |
| **SQL** | 6 | 9 | 3 | 15 |
| **Insider Attack** | 10 | 15 | - | 25 |
| **Fire** | 2 | - | - | - |

* + - * + Treating the Risk

**Avoid** – Take steps to stop it happening

**Mitigate** – Take steps to make the impact less serious

**Transfer** – Make someone else responsible for it

**Accept** – Decide to live with it

* + - * + For example…

Loss of data: **Avoid** by not collecting data

Stolen data: **Mitigate** this by encrypting stored data

Data destroyed by fire: **Transfer** using insurance

Main and backup disks fail at same time: **Accept** it

* + - * + Final steps:

Specify the **controls** (mitigation and avoidance techniques)

Obtain **management approval**

Prepare a **statement of applicability** (overview of ISMS)

* + - * ISOs give *some* **assurance** to other organisations that your organisation is secure.
* **Usability & Security**
  + The **easiest way into any system** is usually to **exploit human factors**.
    - Breaking SSL is as easy as having a team member click through on the “Unverified Certificate” warning.
  + There are many ways **information can leak out of a system**.
    - Computer security experts often overlook the **non-technical attacks**:
      * “Shoulder Surfing”
      * “Dumpster Diving”
      * **Social Engineering**
        + In general, people want to be helpful.
        + Why not just phone them up and ask for passwords, credit card numbers, etc.?
        + Attackers might say they are from the phone company and walk right in to the server room.
  + This does not mean “people are stupid, they are the problem”!
    - Users are doing what they need to get their job done.
    - **Bad security design/policy** is often the problem.
      * When is a new security measure good?
        + When it **saves more than it costs**!
  + A good **security policy** can help, but a bad one can make things worse.
    - Users want to get work done, and will find a way around a policy if it stops them.
    - Users are sometimes “right” to ignore security advice.