**`Security Observations from the Past Week**

* Side-channel attack
  + Attack that is based on information gained from the physical implementation of a cryptosystem, rather than brute force or theoretical weaknesses in the algorithms.
  + E.g. Timing, power consumption, electromagnetic leaks or even sound can provide an extra source of information which can be exploited.
  + Example: I had to vote for a politician – either Hilary or Trump  
    You have to write either Trump or Hilary on a piece of paper  
    The side-channel attack / timing attack is watching someone write Hilary’s three-word-name vs. Trump’s easier two-word name. You can easily see who they are voting for.
* AES: Advanced Encryption Standard
  + A beautiful algorithm but can be cracked by side-channel attacks.

**Bugs, Vulnerabilities and Exploits**

* Vulnerability
  + A bug that’s not perfect about the design of the system and potentially would let someone attack / compromise the system in a way that you don’t intend it to behave.
* Bits / Bytes
  + 0 / 1
  + 8 bits in 1 byte
  + Computer memory is divided into blocks of bytes
  + Each byte is given a memory address 🡪 computer when writing / reading from memory needs to know the address so it can read / write to that particular location
* Hexadecimal
  + 0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 …
* **Memory map**
  + All memory of a computer drawn out.
  + Typically they are represented as a rectangle, but realistically memory is in many forms
  + Memory can be 4 or 8 bytes wide, depending on 32bit / 64bit computer.
* Memory map: PROGRAMS
  + Programs are usually stored up high in the memory map (towards 0x000000 address)
  + Programs have variables which need to be stored, usually in the lower areas in a memory map (0xFFFFFFF)
* Memory corruption is dangerous
* **Stack Frames**
  + When a program is doing something / carrying out a function, every function has an area of memory called its FRAME, which it uses as temporary storage.
  + Frames of functions are stored in the STACK at the lower / bigger memory address (towards 0xFFFFFF).
  + E.g. A function may have its frame live between addresses 0xFFFE00 🡪 0xFFFEFF.
  + We set variable x = 7, compiler will set area aside in the stack for x and will store 7 inside it.
  + We create an array of many variables, the stack frame becomes bigger / extended while the function is running.
* **Return Address** (inside a Stack Frame)
  + A return address is something inside a function’s stack frame which is stored near the bottom.
  + It tells a function where to go next after the function finishes executing + start executing whatever program is at the return address.
* **Stack Overflow**
  + Since the return address sits at the bottom of a stack frame, if you add more data than expected in an array, you can eventually override the return address with your own data.
  + Once the function finishes, the computer will look at the return address to see where to go next. It can go to a location specified by your own data which has overridden the original return address.
  + If you have hidden a nasty program in certain location in memory, then you can tell the computer to jump to the nasty program and execute it.
  + Making use of a vulnerability like this is called **EXPLOITING**.
* Exploit examples
  + Your exploit can **POP A SHELL** (get the comp to execute a Shell program + let you control the Shell), the attacker can type in commands as though he’s sitting at your terminal.
  + Your exploit can **RUN MALWARE**, where the comp executes Malware on your computer and installs a **Rootkit**, **hide itself** or **give you multiple access points** or **ways of getting back in if you get locked out** or any malicious software that you want.
  + Often attackers will try and trick the program to ping information about itself, they can get hints about certain locations in memory addresses. There is often a trial and error process.
  + Sometimes it is difficult to load Malware into memory, so you can store the Malware program in the array and then overflow the array with the return address, then the return address will jump back into the array and execute it.
* **NOP Sled (NOP = No Operation, an instruction that does nothing)**
  + Old day technique and wasteful of space.
  + It helps circumvent stack randomisation in a buffer overflow attack.
  + **Stack Randomisation** may make the address where the program will jump to impossible to predict, so the attacker places a NOP sled in a large range of memory. If the program jumps anywhere into the sled it will run all through all the NOPs / doing nothing until it reaches the **payload code**.
  + A NOP Sled basically makes the target address bigger, so that the code can jump anywhere in the sled and go towards the payload.
* **Data and Control**
  + You can sometimes use data to affect control.
  + If a user has control over data (e.g. ability to input a string into a textbox), then you have given the ability for them to type something in, overflow an array and affect control.
  + Arrays where we temporarily store data until it is processed = **BUFFER**
  + Hence **Buffer Overflow** attack. Caused by not enforcing / checking length of user input against array.
* **Heap**
  + When you MALLOC things (storing arbitrary bits of memory using Malloc), it is stored in roughly the middle area of memory.
  + As things are added on to the Heap, it grows downwards (towards 0xFFFFFF).
  + As things are added on to the Stack, it grows upwards (towards 0x000000).
  + Eventually, the two may collide and you will get an “out of memory” error
  + To overflow in the Heap, you can overflow possibly critical data below the Heap instead of the return address in a Stack Frame.
* **Overflowing SIGNED integer**
  + If you overflow a signed integer, it will roll back to the smallest negative number and go back upwards.
* **Overflowing UNSIGNED** **integer**
  + If you overflow an unsigned integer / number, it will wrap around.

**Printf Vulnerabilities / Format String Attacks**

* Enter your name:
  + “ %x%x%x%x%x%x%x ”
  + This will print out the next 7 numbers in Hexadecimal
  + 7 integers of size 4 bytes / 32 bits = It will print out the next 28 bytes of memory immediately under the current parameter in the Stack Frame, so you know the rest of the values in the Stack Frame.
* %n is a really bad format specifier to use
  + EXAMPLE: n\_chars would = 12

int n\_chars = 0;  
printf(“Hello, World%n”, &n\_chars);

* + It stores the # of bytes written to the output so far in an integer variable.
  + This is bad because it allows an attacker to write to memory through printf.

**Bugs, Vulnerabilities and Exploits CONTINUED**

* **Shellcode**
  + The actual code that you want the exploit to run that achieves the affect you want.
  + The term literally refers to written code that starts a command shell.
  + Shellcode is typically the payload of an exploit.
* CVE – Common Vulnerabilities and Exposures
  + A database of vulns with CVE ID numbers attached to each vulnerability.
  + As vulnerabilities get found and reported, they are given ID numbers and certain companies / naming authorities such as Google, Microsoft Research can hand out these numbers and they are all put into this centralised database.
  + **Zero Days** are vulnerabilities that haven’t been reported yet, so they wont be in the CVE database.
* **Responsible Disclosure**
  + If you find a bug / vulnerability, you can either:
    - Exploit it and get arrested / get rich
    - Sell the bug
    - Announce it to the media and cause lots of embarrassment
  + Responsible disclosure process is:
    - Disclose it to the software vendor (person who owns the software + give them a period of time to do something about it)
    - If they don’t do anything about it, you report it to someone.
    - If you don’t report to the media, you report it to a CERT (Computer Emergency Response Team) or to a responsible person / company to try get ae CVE number.
* Web bugs
  + OWASP: Open Web Application Security Project
  + They put out top-ten current vulns with web software. Whenever you write web-software, you can take a look at the list of OWASP top web vulnerabilities and make sure you don’t violate any of them.

**Assets**

* As Security Engineers, we are going to be protecting things, but what do we actually want to protect?
* People usually skip thinking of what they ACTUALLY want to protect because they think they know, but when the obvious asset is actually lost, they will kick themselves for losing it.
* Strategies for identifying assets:
  + **Get lots of eyes looking at the asset, including people who own the assets. See what they all value about it.**
  + **Develop a sensible plan – well designed to tease this information out of them**
    - Humans are generally poor at regurgitating everything that they know.
    - However, they are very good critics
  + **Periodically revise current list of assets**
    - Don’t set and forget.
    - Values and assets of an organisation can drift
* A very common hidden asset is REPUTATION, since its not a physical thing
* Valuing assets / defining what is important
  + **Tangible Assets**: Those that are easily given a value
    - A gold chain valued at some static amount
    - Jewellery inside a jewellery store
  + **Intangible Assets**: Those that cannot be easily / objectively valued
    - Employee morale and security
    - Customer information
    - Company secrets
    - Availability of services
* Strategies for assigning values to assets
  + **Survey what many people think**
    - No single person should be solely evaluating the assets
    - “How much money would you lose if this data centre was to go down for 24 hours?”
    - “How much will you lose if your company is disconnected to the internet for 3 hours?”

**Bits of Security / Hash Properties**

* **Bits of Security**
  + 10 bits of security = it takes 210 operations to break the security.
  + It is a measure of work to break some security
* **Entropy**
  + A measure of chaos
  + English has low Entropy because it relies on a lot of patterns and ordering
  + Because English has low Entropy, if you are decoding a message you will know when you have decoded the message. It is hard to decode a message into the wrong message.
* **Bits of Security EXAM EXAMPLE:**  
  If we are given a key, of length 7, comprised of alphabetical letters:
  + It would take **267 possible combinations** to brute force key.
  + Most likely only 1 of the 267 key combinations would look like plaintext English
  + 267 is the worst case amount of work it would take to solve the message.
  + **Convert 267 into bits = 267 🡪 327 🡪 25^7 🡪 235**
  + Hence 267 = 35 bits of work (roughly, as obviously there is a difference between 26/32)
  + We use 32 as 25 = 32 and it is the closest we can get to 26.
    - 24 = 18 and 26 = 64.
  + This is crackable, as we need at least 128bits of work to make it too difficult for an attacker.
* **Alternative Exam QN: How long would it take to crack 60 bits of work?**
  + 260
  + Need to make a lot of assumptions
    - How fast is the attacker’s computer?
    - How many computers do they have?
  + Assume they can do 250 per second 🡪 They need to do 210 seconds worth of work = 20 minutes of cracking.
* Hash Functions / What makes a good hash function
  + 3 properties that a cryptographic hash function should have:
    - Pre-image resistance
    - Second pre-image resistance
    - Collision resistance
* **Pre-image resistance**
  + For essentially all pre-specified outputs, it is computationally infeasible to find any input that hashes to that output.
  + I.e. given y, it is difficult to find an x such that h(x) = y.
  + You can’t reverse the hash / go backwards to find the input.
* **Second pre-image resistance**
  + It is computationally infeasible to find any second input which has the same output as that of a specified input
  + I.e. given X, it is difficult to find a second pre-image X’ != X such that h(X) = h(X’)
  + Given an input X with output Y, you can’t find a second input X’ that produces the same hash output Y.
* **Collision Resistance**
  + It is computationally infeasible to find any two distinct inputs X / X’ that hash to the same output  
    i.e. such that h(X) = h(X’)
  + Not given anything, you can’t find a two inputs that will collide in the same way.
* With each of these properties, as long as an attacker has to use BRUTE FORCE to solve the problem, then your hash function has high-resistance. If the attacker knows of a way to do it faster than brute force, then your hash function has low resistance.
* **One-way function**
  + Attack against this is called pre-image attack
* **Collision free**
  + Attack against this is called Birthday attack
* Random mapping is ideal
  + Given a message, you hash it into a smaller message. It should just look like a random collection of things in the output.
  + A small change in the input should ideally change on average half of the output / a diffuse effect
  + Change in 1 bit should result in 50% chance of each hash bit changing.
* **Passwords salts**
  + You have a password and want to authenticate it but don’t want to let the other party know your password, only the hash of your password. The hash of your PW will be stored rather than the true PW.
  + However, the PW which belongs specifically to you can be worked out if the other party hashes every password that they have on file until it matches your one = **Dictionary Attack** on a pw / hash.
  + You add extra “Salt” / data to your message / password before you hash it in. So the attacker does not have the advantage of **space/time trade-off** and can only attack 1 person but not everyone.
  + If they want to attack everyone, they would need to create a whole new dictionary for each person, since everyone’s salt value is different.  
    (because without the salt, if the attacker processes the hash of every password, they can store the result and do a parallel-attack by referring to what they’ve processed before)
* How many bits in a UNIX salt?
  + 4 bytes