Week 1 Introduction

Terminology of "Security engineering is about building systems to remain dependable in face of malice, error, or mischance."

System: It is subjective, it depends on which system has to be developed. Maybe a system to one person is not big enough to the other one.

Threat: A potential violation of security.

1. Disclosure: Unauthorized access to information.
2. Deception: Causing the system to accept false data, by pretending to be someone.
3. Disruption: Preventing services from occurring (like DoS).
4. Usurpation: Unauthorized control of some part of the system.

Attack: Action that can cause that violation to actually occur.

Thread model: Who are the attackers, what resources do they have and what is valuable in my system.

Security policy: Statement of what is allowed and what is not allowed in the system. Unambiguously partitions the system in to "secure" and "non-secure" states.

Security mechanisms: Methods and tools to enforce the security policy (like cryptography). It prevents the system from entering a non-secure state.

Security specifications: Specifies which security mechanisms are going to be used and how to use them.

Error: Human error (like not being properly trained).

Mischance: Something non-human (like natural disasters).

Dependable (trustworthy and reliable): If security policy and mechanisms are done correctly, then the system is dependable. It does what is supposed to do. No more; no less.

Goals of security: If you don't *prevent*, then you have to *detect*, and if you can't do that as well, you must *recover* (specially the confidence of customers and employees).

Design principles:

1. **Principle of Least Privilege**: Privileges are not assigned according to identity, but instead according to the job that needs to be done.
2. **Principle of Fail-Safe Defaults**: Your system needs to fail in a safe way. You need to be able to roll-back.
3. **Principle of Economy of Mechanism**: Use security mechanisms that are as simple as possible. "Complexity is the enemy of security".
4. **Principle of Complete Mediation**: Your system will mediate (check) every access. If access was given 10min before to someone, and if the someone request it again, then check it again.
5. **Principle of Open Design**: The security of a mechanism should not depend on the secrecy of its design/implementation. It is preferable to use open software. "Security through obscurity does not work".

6. **Principle of Separation of Privilege**: The system should not grant permission to critical resources based on a single condition. There most be more than one person to perform a sensitive action.
7. **Principle of Least Common Mechanism**: Sensitive and no-sensitive information shouldn't share the same resources (like the same channel).
8. **Principle of Psychological Acceptability**: The mechanisms that were put in place need to be usable by real people. Otherwise, people will bypass it.

Week 2 User Security

Passwords: Something you know that you reveal to another entity so that the other entity is sure it's talking to you.

Passwords problems:

1. **Naïve users**: People don't know that their password needs to be strong. They also may not change the default password. They may also share their password.
2. **We have too many or too few of them**: Hard to remember, but if you chose to have just one password for all the systems, that's also not a good idea.
3. **Design errors**: People use passwords that are easy to discover. Or maybe the system doesn't allow the password to be changed. Or the system doesn't force the user to change the password.

Passwords threads:

1. Targeted attack on a specific user.
2. Targeted attack on any account in a system.
3. Targeted attack on any system in a domain.
4. You may enter in a system, then enter the incorrect password of every user in the system three times, making them unable to access the service (DoS).

Measures to protect password:

1. **User training**: Train your users to use better passwords.
2. **Interface design**: Like hiding the pin-pad of an ATM machine to stay right in front of you.
3. **Protect password in transit**: The path that your password goes by must be secure.
4. **Trusted path**: Like the path of using control+alt+delete to enter your password.
5. **All-or-nothing error messages**: The system should say "wrong password or email" instead of saying wrong password as you type it in the keyboard.
6. **Protect log file**: The user may be tired and he types the password in the username field and the username field in the password. In the next attempt, he types it correctly. So, both the password and username are now in the clear in the log file.
7. **One-way function of passwords in password file**: The password must be hashed or encrypted before it is stored in a file.
8. **Hide password file**: Put the file location in a secret place.
9. **Exponential back-off or disconnection**: After a certain number of attempts, you are blocked, and the duration of the block is exponential. That way, it is harder to brute force the password.
10. **Password aging**: Forcing the user to change the password occasionally.
11. **Use graphical passwords**: People can easier memorize graphical passwords than letters and numbers.

12. **User salting**: Salt is a random sequence of characters. When the user creates the password, it will also create a salt. Then you will attach the salt to the password and hash it all. As the salt is unique to each user, if the bad guy knows the hash for a password, it will only compromise that password. Furthermore, as the salt is attached to the password, it takes more time to break through it.

Calculating suitable password size for an environment:

- Probability of guessing is P >= (GT)/N
- P = the probability of the hacker discovering your password over a time period;
- G = # of guesses per time unit;
- T = # of time units;
- $N = C^S$ = # of possible passwords;
- C = # of possible characters;
- S = password length.

Strategies to reduce G include exponential back-off (to counter an online attack), and salting (to counter an offline attack)

For example:

Password composed of **S** symbols each from an alphabet of 96 characters (26 upper/lower, 10 numbers, 34 special characters).

- Assume 10^4 guesses/sec.
- Want the probability of successful guess to be 0.5 over one year.

What is the suitable password length?

Size of password space = 96^S

P >= (GT)/N N >= (GT)/P N >= (10^4(365*24*60*60)) / 0.5 N >= 6.31 * 10^{11}

96^S >= 6.31 * 10^{11} log(96^S) >= log(6.31 * 10^{11}) S >= log(6.31 * 10^{11}) / log(96) = 5.94

Biometrics: Another way to identify a user to a system. Theory: it's easier than a password for the legitimate user, and it's harder than a password for a bad guy. Techniques:

- Fingerprints, voice recognition, iris scan, retinal scan, face recognition, hand geometry, key strokes (the way you type), gait (the way you walk), earshape.

Problems with biometrics:

- Biological or behavioral features can change over time.
- Reading is never identical to stored data.
- Bad guy can steal input in some cases (record & replay).

Identity: Is a computer's representation of an entity.

Authentication: How the computer knows that you belong to an identity.

What do we do when a user must be known to several systems? Globally-unique identifiers and identity mapping. It has to use hierarchical naming (divide and conquer). Therefore, the "name" of any entity is defined to be the full path of names starting from the root.

Five components of an authentication system:

1. Authentication information: ex. Password.
2. Complementary information: ex: Table to hash password.
3. Functions, ex: Is this bio similar to this one we have stored?
4. Selection function, ex: Password set, password reset…

Local versus remote authentication: Distinction is whether authentication information needs to travel over the network.

Week 3 Security Policies

Security policy: Statement of what is allowed and what is not allowed in the system. Unambiguously partitions the system in to "secure" and "non-secure" states. A secure system starts in an authorized state and cannot enter an unauthorized state. To have a secure system, you have to block the transitions that lead to unsecure states.

Security policies can be written with respect to confidentiality, integrity, or availability:

Confidentiality: Roughly equivalent to privacy. Designed to prevent sensitive information from reaching the wrong people, while making sure that the right people can in fact get it. What gives confidentiality: encryption and access control.

Integrity: It's about what changes can be made to the data and who is allowed to make those changes. These measures include file permissions and user access controls.

Availability: The services must be available. It can be ensured by system upgrades, redundancy, disaster recovery, safeguards, backups…

Confidentiality policies:

- **Bell-LaPadula**: Read down, write up (star property). Also has the discretionary security property which allows a subject that has a certain type of access on an object, to transfer rights to other subject of their choice. Focus on confidentiality. Useful to military.
- **Biba**: Read up, write down. Subject may not corrupt data in level ranked higher than the subject, or be corrupted by data from a lower level than the subject. Focus on integrity. Useful to banks.
- **Clark-Wilson**: It has well-formed transaction, which is a series of operations that transition a system from one consistent state to another consistent state. The Integrity Verification Procedures audit the well-formed transactions. Focus on integrity. Useful to business.
- **Hybrid policies:**
    - **Originator-Controlled Access Control**
    - **Policy** (ORCON): Originators of documents retain control over them even after those documents are disseminated. Watermarking can help.
    - **Role-Based Access Control** (RBAC): The ability, or need, to access information may depend on job function.

Week 4 Fundamental Security Tools

Cryptography:

Encryption: Both achieve equivalent levels of security.

- Asymmetric keys: Guaranties confidentiality, authenticity and integrity. Key distribution advantage.
- Symmetric keys. Guaranties only confidentiality. Speed advantage

Digital signatures: Authenticity and integrity. Uses asymmetric keys.

Hash: Maps data of arbitrary size to fixed-size. One-way function. Guarantees only integrity. Arbitrary input size, fixed output size. Collision resistance, it's a property of cryptographic hash functions where it's hard to find two inputs that hash to the same output.

MAC: Uses a private key, therefore guarantees both integrity and authentication.

Cryptographic algorithms:

- Don't "roll your own".
- Don't purchase from vender with proprietary algorithm.
- Use standardized open algorithms, like NIST.
- Crypto agility: Quickly change your crypto algorithm in case it has been broken.

Crypto keys:

- Don't use built in Pseudo Random Number Generator (PRNGs).
  - Not random enough.
  - Not unpredictable.
- Key hygiene: Use different keys for different purposes.
  - Asymmetric key: One for encryption and decryption and another for digital signatures.
  - Symmetric key: One for encryption and decryption and another for MAC.

Key space: n is the length of the bit string, key space is the number of bits the key has, 2^n.

- **Symmetric keys:** The minimum key space in generating key is 2^{80} the preferred is 2^{128}. Typical algorithms include 3DES and AES.
- **Asymmetric keys:** The minimum key space in generating key is 2^{1024} the preferred is 2^{3072}. Typical algorithm is the RSA.

Key Distribution Center (KDC):

- Part of a cryptosystem intended to reduce the risks inherent in exchanging keys.
- Used in symmetric key environment.
- KDC shares a key with each of all the other parties.
- KDC produces a ticked based on a server key.
- Kerberos is a system that includes KDCs, but in Kerberos the KDC has its functionality partitioned between two different agents: the AS (Authentication Server) and the TGS (Ticket Granting Service).

Certificate Authority (CA):

- Trusted entity that manages and issues security certificates and public keys that are used for secure communication in a public network.
- Used in asymmetric key environment.

- The CA is part of the public key infrastructure (PKI) along with the registration authority (RA) which verifies the information provided by a requester of a digital certificate. If the information is verified as correct, the certificate authority can then issue a certificate.

Public Key Infrastructure (PKI):

- Set of roles, policies, hardware, software and procedures needed to create, manage, distribute, use, store and revoke CAs and manage public-key encryption.
- The purpose of a PKI is to facilitate the secure electronic transfer of information for a range of network activities such as e-commerce, internet banking and confidential email.
- It is required for activities where simple passwords are inadequate authentication method and more rigorous proof is required to confirm the identity of the parties involved in the communication and to validate the information being transferred.
- The PKI role that assured valid and correct registration is called a registration authority (RA).

Trust Anchor:

- In PKI, the trust comes from CA, the third-party. The trust anchor is the CA's public key. It is used to verify certificates of other users. If Alice CA trusts Bobs CA the CAs create a certification for each other.

Assumptions for PKI to work:

- Alice and Bob know the public key of the CA, it's a trust anchor.
- Both Alice's and Bob's certificates are stored in a public place.
- The names (unique, because there must be more than one Bob) in certificates correspond to actual entities that which to communicate.
- Certificates have not been revoked (it's not simple to be done, it's a PKI disadvantage).
- Private keys have not been compromised.

Online KDC:

- Holds all keys in the system.
- If compromised, attacker can read all data.
- Need to protect for confidentiality and integrity.
- Better performance.
- Doesn't need a physical protected space.

Offline CA:

- Holds only its own private key.
- If compromised, attacker can read all data, but *much* harder.
- Online repository of certificates, needs to protect only for integrity.

Week 5 Access Control

The goal of access control is to control which principals have access to which resource in the system.

The access control matrix (Lampson, 1971) is simple, formal and complete. But it is large, sparse and inefficient.

Access control list (ACL):

- Stores columns only, it's stored with resources: File1: { (read,[user1, user3]), (write, [user2, user3]) }.
- Easier to create and understand.
- Not suited to change the status of a user (not good for large populations, specially when constantly changing), not suited for delegation, not suited to revocation, inefficient.
- The creator/originator is the one that can change the ACL. That is Dissemination & Extraction of Information Controlled by Originator (ORCON).
- ACL supports groups and wildcards. For example Unix: user ID : groupID : permission, that can be Alice - * - r, which is Alice, independently to which group she belongs, she can read everything.
- Can better answer: Given an object, which subjects have access to it. Can trivially check one ACL or hardly (impossibly?) check every CL in the system

Capability list (CL):

- Stores rows only, it's stored with users: User1: ((read, {file1,file2}), (write, {})).
- More suited to delegation and revocation.
- More complex to create and understand. Not suited to change the status of an object.
- Can better answer: Given a subject, what object can it access. Can easily check one CL, or hardly check every ACL in the system.

People may lean more towards ACL because it can better answer its question. However, it turns out, that the most asked question is the one that CL can answer better.

Authorization table:

- Each row has a user, a permission and an object.
- It can be manipulated to generate a CL or an ACL. Therefore, it has the benefits and drawbacks of both CL and ACL.

An alternative to ACL and CL is a "cell": attribute certificate

- Function (Pk certificate, permission).
- Easier to create and understand.
- Can do delegation without revealing too much info.
- Easy to revoke specific permission, or all permissions.
- Dynamic population handled through AC or Pkc creation/revocation.
- Inefficient.

Week 6 Operating System

Top four measures to prevent attacks at OS:

- Patch operating systems and application using auto-update.
- Patch third-party applications.
- Restrict admin privileges to OS and applications based on user duties.
- White-list approved applications.

Initial Setup and Patching:

- Systems security begins with the installation of the OS.

- Ideally new systems should be constructed on a protected network.
- Full installation and hardening process should occur before the system is deployed to its intended location.
- Initial installation should install the minimum necessary for the desired system.
- Overall boot process must also be secured.
- The integrity and source of any additional device driver code must be carefully validated.
- Critical that the system be kept up to date, with all critical security related patches installed.
- Should stage and validate all patches on the test systems before deploying them in production.

Virtualization:

- A technology that provides an abstraction of the resources used by some software which runs a simulated environment called a virtual machine (VM).
- Benefits include better efficiency in the use of the physical system resources.
- Provides support for multiple distinct OS and associated applications on one physical system.
- Raises additional security concerns.

Application virtualization: Allows applications written for one environment to execute on some other OS.

Full virtualization: Multiple full OS instances execute in parallel.

Virtual machine monitor (VMM): Hypervisor. Coordinates access between each of the guests and the actual physical hardware resources.

Native Virtualization Security Layer: Physical Hardware -> Hypervisor -> Guest O/S Kernel 1 -> User Apps.

Hosted Virtualization Security Layer: Physical Hardware -> Host OS Kernel -> Other User Apps OR -> Hypervisor -> Guest O/S Kernel 1 -> User Apps.

Week 8 Malware

Attack kits: It's a programming tool that allows people with no experience writing software to create, customize and distribute malware.

Attack sources: The attackers themselves. The current attackers are professionals, they have money, resources and time. It's an organized crime. You can hire attackers. It's harder to prosecute them.

History:

- 1960: Trojan horse was used by students in order to get more access to a (old) main frame computer.
- 1980: First virus.
- 1988: It was able to guess password and could hide itself. Due to a code bug, it shut down the internet for 2 days.

Malware types:

- Trojans are becoming more common than viruses. Viruses spread themselves from infected machines, consequently drawing more attention and getting the machine to clean it more quickly. Whereas trojans are sent directly from the source and they don't spread themselves, giving the herder more control.
- **Trojan horse**: A program that does something malicious by misleading users of its true intent. E.g. it may take control of your PC; steal information; install other malicious programs (like keyloggers). It masquerades as performing a benign action but also does something malicious. It spreads when users transfer Trojan files to other computers. It may infect a file and needs the user interaction to be spread.
- **Worm**: Replicates itself in order to spread to other computers (like via e-mails or USBs). It exploits a vulnerability in an application or OS. It spreads by using the network. It doesn't infect a file and doesn't need the user interaction to be spread.
- **Virus**: A worm which replicates itself when a program or file is executed. It inserts malicious code into a program or data file. It spreads when the user transfers infected files to other devices. It infects files and needs the user interaction to be spread.
    - **Viruses and worms' components:**
        - **Replication mechanism:** The commonest way for a virus to replicate was to append itself to an executable file and patch itself in, so that the execution path jumps to the virus code and then back to the original program.
        - **Payload:** Usually activated by a trigger, such as a date. After activated it may do the following (mostly) bad things:
            - Make changes to the machine's protection state.
            - Make changes to user data, e.g. encrypting your data and asking for a ransom to give you the key to decrypt it.
            - Lock the network (DoS), e.g. rapidly replicating itself.
            - Steal resources, e.g. stealing clock cycles.
            - Steal and/or publish user data, e.g. personal information.
            - Install spyware or adware in your machine, e.g. to tell marketers what you do online, or to get your bank password.
            - Create a backdoor, e.g. to be able to log as the legitimate user.
            - Install a rootkit, a software that hides (*stealth* software) itself and takes over your computer e.g. to act as a bot in a botnet.
            - Perform software upgrades (*benevolent software*), e.g. if you need to do software updates in many machines, you may spread a virus that does it for you.
            - Malicious data (therefore, it's not always about malicious code), e.g. decompressing a file that turns out to be huge, thus, taking down the server.
    - **Types of viruses:**
        1. **Polymorphic virus:** The virus *change itself each time it replicates*, making it harder to be detected by *scanners* (programs that search executable files for a string of bytes known to be from an identified

virus). E.g. re-encrypting itself with different keys; or using dummy variables.

2. **Metamorphic virus:** Instead of changing its instructions or encryption, *it changes its behaviour*. The new code generated bears no resemblance to its original syntax but is functionally the same.
3. **Stealth virus:** The virus watches out for the OS calls used by *checksummers* (program that keep a list of all authorised executables on the system) and hides itself whenever a check is being done.
4. **Boot sector infector:** Acts during the boot. They spread usually by infected floppy disks. In the past, these were usually bootable disks, but this is no longer the case.
5. **Executable infector:** It's the most common (like the .COM). They infect executable programs.
6. **Terminate and Stay Resident (TSR):** Stays in the non-volatile memory. Therefore, it stays alive even after the PC is shutdown.
7. **Encrypted virus:** Encrypts its payload with the intention of making it harder to be detected.
8. **Macro virus (or scripting viruses):** It's written in a scripting language that gets interpreted (not compiled), like PDF or Excel documents. *It's not bound by the machine architecture*. It's limited by the scripting language. E.g. Melissa virus (1994) which was executed during the opening script of the old Word MS.
9. **Rabbit (bacterium):** Typically, doesn't have a payload. Its only task is to replicate itself as fast as possible to overload the network (causing DoS).
10. **Logic bomb:** Activated by a specific trigger. E.g. a data, or e.g. by continuously searching for a name in a list and when the name is not in the list anymore, it gets executed.
11. **Stuxnet virus:** Makes Spreads slowly in order to make it more difficult to be detected. It can be spread through pen drives, network…. The first virus to attack physical system thus breaking it. E.g. it could change the spin of a fan in order to break the fan.

- **Key loggers:** Reads everything you type (and/or clicks). It doesn't send all the information to the attacker; it has a filter. A way to protect is to use graphical passwords, e.g. points in an image.
- **Phishing:** Usually comes in a spam e-mail that has a link to a (very similar) but fake website. It's used to perform identity theft, which may result in selling the information or using it to get a loan impersonating someone else.
    - **Spear phishing:** The spam e-mail is crafted personally to you or to a small group. E.g. sending an e-mail pretending to be the dem of uOttawa to the uOttawa staff.
- **Drive-by downloads:** Exploits your browser's vulnerability and (if vulnerable) instantly downloads malicious software when entering a malicious website.
- **Bots:** A PC that a remote attacker has taken over. It's used to attack a third victim. It's useful to hide the attacker's identity. When the attacker infects many (even hundreds of thousands) then he has a botnet. Differently of viruses, bots have a *remote-control facility*, which gives orders to them. As they have a channel connecting them to the remote-control facility, it's easier to detect the channels. How attacker uses their bots:

    - Mainly for Distributed Denial of Service (DDoS). E.g. the DNS dyn attack, where the DNS server was attacked by IoT zombies.
    - Spamming e-mails.
    - Sniff network traffic.
    - Key logging.
    - Spreading other malwares.
    - Financial gain, as setting up a fake website with an advertisement and using the bots to click on it.
    - Manipulating the voting (pull) process.
- **Rootkits**: Focused in being *stealth*. A root kit could operate in user level (like an API), kernel level or VM level (installing itself as a corrupted VM and installing the OS on top of it). Every mechanism that would find the root kit is not able to find it. The following methods should be able to return information about the root kit, but they are not able to find it. E.g. using the command netstat (gets the status of the network), ls (lists the directories), ifconfig (configures the kernel-resident network interfaces).
- **Ransomware:** Encrypts your data and asks for a ransom to give you the key to decrypt it. E.g. to decrypt your photos.
    - PC Cyborg: 1989 it didn't encrypt the files, instead, it moved the files around and encrypted their names. They used symmetric keys.
    - Young & Yong 1996: Suggested using asymmetric keys.
    - WannaCry 2017: A ransomware worm that infected 230.000 computers in 150 countries. It explored an NSA (National Security Agency) vulnerability. WannaCry requested access to a website that didn't exist and, in case it really didn't existed, WannaCry knew that it wasn't in a VM, thus, continued executing. Otherwise, if the website existed, WannaCry thought that it was in a VM (being explored by researchers) and stopped working. Someone created the website, and WannaCry started thinking it was always in a VM, consequently stopping working worldwide.
- **Phone worms:** The same as computer worms, its main objective is to reproduce itself and spread to other devices. It may contain harmful and misleading instructions. Mobile worms can be transmitted via SMS and typically do not require user interaction. Worms can be used to make calls to expensive numbers in order to promote profit to the attacker. Worms can also stick to the SIM card.
- **Phone trojans:** The same as computer trojans, its main object is not to spread, but to cause damage to your system. It appears like a friendly program, but it's actually not. The first one appeared in 2004. A famous one was the Droid Dream, giving the attacker full access to your phone. In 2011 Google deleted many apps in their store because they had the Droid Dream trojan. Regarding Apple, it has more protection against malwares, as they check the apps before putting them into their store.

Defenses against malwares:

- **First steps:** Use current software patches (update your system). Set AC properly to the applications in your system, so that the attacker has more difficulties in accessing it.
- **Antivirus software:**
    - **Host-based scanners** (installed in the computer itself).
        1. **First generation (simple scanners)**: Search for a string of bytes known to be from an identified virus. May detect viruses which have essentially the same structure and bit patterns in *all* copies, therefore, a polymorphic virus

is a counterattack to scanners. It's limited to the detection of known viruses. A zero-day attack is an attack that is not yet recognized; thus, you can't protect against.

2. **Second generation (heuristic scanners)**: Relies on heuristic rules for *probable* virus infection. It looks for fragments of code that are often associated with viruses, that is, it looks for a specific type of behavior. For example, it may look for encryption loop and a key in polymorphic viruses.
3. **Third generation (activity detection)**: Identifies a virus by its *actions*. It doesn't necessarily develop signature and heuristics for various classes of viruses. It is necessary to identify the small set of indicative actions. For example, system calls and accesses to files that ordinary people wouldn't usually do.
4. **Forth generation (full-featured protection):** It's a combination of the previous three. It also has:
    - **Generic decryption (GD):** Polymorphic viruses use encryption to hide malicious code. However, to execute such a code it has to be decrypted. GD tools are used to detect (fragments of) viruses at the stage they are decrypted and ready to be executed.
    - **CPU simulator:** It's essentially a VM. Using the CPU simulator, instructions in an executable file are interpreted by the emulator not effecting the underlying processor.
    - **Signature scanner:** Same as the first generation.
    - **Emulation control model:** Controls the execution of the target code switching between simulation and scanning modes.

    Drawbacks:
    - **Performance**: Every new piece of software will be emulated and scanned until you decide it's a trustworthy software.
    - **Behaviour-Blocking software:** Integrates with the OS of the host computer and monitors program behaviour in real-time for malicious actions. Blocks potentially malicious actions before they affect the system, such as: Attempts to open, view, delete, modify files; attempts to format disk drivers; modification of system settings; initiation of network communication.

- **Network-based scanners:** It's installed in a network. For example, it may be installed in the company's firewall. Its intended to lock down the perimeter before the attacker gets into your network.
    - Differently to the host-based scanner, it doesn't look at the behaviour of the malware, but on its *content*.
    - Differently to the host-based scanner, it monitors ingress and egress flow. It helps to detect botnets, as a channel has to be stablished between the attacker and every bot. Egress filtering is the practice of monitoring and potentially restricting the flow of information outbound from one network to another. Whereas ingress filtering is used to ensure that incoming packets are actually from the networks from which they claim to originate.
- **Distributed intelligence gathering approaches:** Combination of Host and Network-based scanners. Has a central agency. The "Leaf" machines communicate with the Administrative machine. The Administrative machine

sends requests to the Malware Analysis machine. The Malware Analysis machine returns an answer to the Administrative machine, which in turn returns the answer to the "Leaf" machines.

- **Rootkit detection:**
  - Rootkit Revealer 2005). If you make a call to the OS to get a file's size, and the OS returns a result and the Rootkit Revealer returns another, that probably means that you have a rootkit in your system.
  - GMER: The replacer for Rootkit Revealer.
  - If you have a rootkit in your kernel-level, the only option is to reinstall the OS.
- **Checksummers:** Keep a list of all the authorised executables on the system, together with checksums of the original versions, typically computed using a hash function (doesn't use symmetric key) or a MAC (uses symmetric key).
- **Type 'data':** Treat your new files as data (not executables). Then, it certifies the data, if the data is OK, it then becomes an executable. The certification process must be competent and not corrupted.
- **Flow distance:** Every time something is created, it is defined to have a flow distance of 0. Each time it is somehow shared the flow distance increases. Therefore, you may decide to work with files within a certain flow distance limit.
- **Reduce user rights:** Principle of least privilege (week 1). Give people only the privileges that they need to get their job done. Sandboxes and VM are part of this as well.
- **Proof-carrying code:** Software mechanism that uses a mathematical proof to verify properties about an application. Then, it compares the conclusions of the proof to its own security policy. Drawback: "its own" security policy may differ from your company security policy.
- **Look for unusual language characteristics in program code:** For example, if you know that three developers developed a code, but you notice that there are four patterns of comments in the code, you may suspect that is was altered by an attacker.
- **Isolation and evaluation:** If you have sensitive data, store it in a different machine that is not connected to the internet.
- **Education and training:** Such as not clicking in suspicious links and not downloading suspicious files.

Defenses against network attacks:

1. **Updates and AC**.
2. **Filtering**: Usage of firewalls to block malwares or malicious into your network.
3. **Intrusion detection:** Having programs monitoring your network and looking for malicious behaviors.
4. **Encryption:** Protocols such as TLS (Transport Layer Security) and SSH (Secure Shell) that enable you to protect specific parts of the network against particular attacks.

Week 9 Denial of Service

Denial of Service (DoS) attack: Action that prevents or impairs the use of a service by exhausting its resources.

- **Denial of Network Service:** It's about quantity of packets. Attacker has a higher capacity to send traffic than the service to receive it. For example, the attacker sends 10.000 packets and the service victim can only receive 1.000 packets. Therefore, the system will

be overloaded and will have to drop some packets that may have been sent by legitimate users.
    - Example: Sending a large volume of ping commands to a network with lower capacity. As the ping command echos back the request, the attacker uses source address spoofing which changes the address of the echo to the whole internet instead of going to the attacker.
- **Denial of System Service:** It's about type of packets. The attacker sends packets to consume the service's resources; or the attacker send a package that causes a bug to the system.
    - **Example:** TCP SYN flood. (→ indicates sending of a packet)
      Normal scenario:
      Client → Server: The client sends a SYN packet to the server. The server uses a table to manage it. It initiates a connection with the client.
      Server ← Client: The server sends back a SYN/ACK packet. Then, the server, starts waiting for an ACK from the client or a timeout.
      Client → Server: The client sends an ACK back to the server.

      The attacker can take advantage of that by not responding to an ACK and waiting for the timeout to send another SYN. Therefore, the server's table keeps full by keeping sending SYNs and not receiving ACKs back.

      Approaches to deal with this attack:
        1. **Vary timeout period** (less used)**:** As the table gets more and more full, the timeout period gets reduced.
        2. **Make client keep the state** (more used)**:** Also known as SYN cookie. All the information on the table is now client's responsibility, the service doesn't have a table.
- **Denial of Application Service:** Valid requests to an application that is computationally or resource expensive, causing the application to get busy and stop supporting requests of legitimate users.
    - **Example:** Denial of service in the application layer, such as SIP (Section Initial Protocol) flood (using VOIP [Voice Over IP]); HTTP flood.

DDoS (Distributed Denial of Service): A attacker uses a botnet to attack a user. A control hierarchy is often used. Possible hierarchy: 1 attacker -> 10 handler zombies -> 100 agent zombies (to each handler zombie) -> 1 target.

Reflector and Amplifier Attacks:

- **Reflector:** The attacker spoofs the victim's IP address and sends request via UDP or DNS or SNMP or… to servers known to respond to that type of request. The server responds to the victim IP.
- **Amplifier**: Similar to reflector, but multiple responses are generated for each package sent. For example, a broadcast service. The attacker sends one request that gets broadcasted to many answers, which all go to the victim IP.

DoS Defenses:

- **Limit/remove ability to send packets with spoofed source address:** Within your own organization, you can rely on your ISP to block IPs that don't belong to your organization. The ISP may come up with excuses to refuse to do it.
- **Limit rate at which certain packets can be sent:** The floods before mentioned can be solved by limiting the packages sent rate.
- **Use SYN cookies:** Make the user save the table, as mentioned before.
- **Block the use of IP-directed broadcasts:** To prevent amplification attacks. The downside to end up preventing it from your legitimate administrator tasks.
- **Modify applications:** Good against denial of application service. Usage of CAPTCHAs or graphical passwords.
- **Keep systems up-to-date and properly patched:** Normal good practice for many types of attacks.
- **Mirror and replicate your servers over multiple sites:** Besides providing reliability and fault tolerance, if one of your servers are taken, you can still use another of your server.

DoS attack response:

- **Must have a good incident response plan**: You must plan ahead of time. You must have a plan. You can work together with your ISP to decide what each of them are going to do.
- **Must have automated network monitoring and intrusion detection systems running:** So that you know as soon as possible when the attack is running, so that you can defend against it.
- **Capture packets and analyze:** To identify which type of DoS attack is taking place.
- **Consider tracing flow of packets back to source:** To know who is attacking you. However, that may not help, because usually it's a spoofed attack, and you'll end up tracing to innocent people.
- **May need to switch to alternate servers:** So that you can continue serving your legitimate users in case of a DoS attack.
- **Update response plan when attack is over:** You need to think about what you'd do differently next time. So that the next time you're attacked, you know what to do.

Intrusion detection

Software trespass: Viruses, worms, trojans, malwares in general.

User trespass:

- **Masquerader:** An outsider coming in. He doesn't have authorization to a computer, but penetrates the system to exploit a legitimate user's account.
- **Insider**: An insider. He's a legitimate user that accesses unauthorized data, programs and resources. It's the hardest to protect against, because they know the system and may be mad at the company.
- **Clandestine** Either a masquerader or insider, but is trying to hide as much as possible their actions.
- **Defense:**
    - Enforce the idea of least privilege, give access to the users based on the tasks that they have to perform.
    - Log all the accesses and commands.
    - For greater accountability, use two factor authentication.

- When firing someone:
  - Turn his access off.
  - Do it in a remote location.
  - Make a copy of his HD.

Intrusion detection system (IDS): Software that detects bad things happening.

- **Underlying premise:** The actions of users and process of your system can form a statistical pattern. If something is out of the statistics, there is something suspicious going on.
- **Requirements:**
  - Fully automated.
  - Restart from crashes.
  - Able to monitor itself.
  - Impose minimal overhead, so that the admin doesn't need to shut it down.
  - Configurable to conform with your security policy.
  - Dynamically reconfigurable, so that you don't have to stop it to change it.
  - Adaptable, as there may be promotions or change of salaries.
  - Scalable, as the organization grows over time.
  - Degrade graceful, so that if something stops working, the rest keeps operational.
  - Detects a wide variety of intrusions.
  - Fast when detecting intrusions.
  - Good GUI, so that when an intrusion is detected, the person responsible for dealing with it would easily understand what is going on.
  - Needs to be accurate.

- **Models:** A combination of those models are used nowadays.
  - **Anomaly detection model:**
    - Doesn't know about the attacker's behavior.
    - Looks for anomalous *user* behavior based on previous statistics. That is, it's a model of a good system.
    - It has to run for some time until it has enough statistics.
    - Hopes to detect attacks that weren't previously recognized.
    - Uses AI techniques.
    - Example: It'd be suspicious if the commands being used are far off the current statistics.
    - Drawbacks:
      - To come up with the statistics, how can we be sure that the system is free of attacks?
      - How long we have to run the model until we reach a good model?
  - **Misuse detection model:**
    - Knows about the *attacker's* behavior.
    - Looks for types of behavior that the *attacker* usually behaves.
    - You need to create *rules* about the attacker's behavior.
    - Example: An antivirus scanner looking for a byte string that corresponds to a known virus.

- Specification model: Newer. Not associated with user, but with softwares. If the software does anything different from its exactly functionality, then the program is suspicious. No Machine Learning is used, you know the exact specification of the programs.
    - It needs a set of rules of every program behavior.
    - You can catch any type of attack because you know the programs behavior and know what the program shouldn't do. But it's not a guarantee.

IDS Architecture: Three components:

- **Agent(s):** The collect statistics to send to the central manager.
    - Host agent: Located (somewhere). Looks at the activity at the system
    - Network: Located in the firewall
    - You may put an agent outside of the firewall, so that you know the attacks that are coming to your agency.
    - You may put it inside in the firewall to know what attacks are coming through.
    - You may put it in a subnet or smaller network.
- **Director (central manager):** The director can ask to agent to send more information or to send less information. Makes decision about what the response should be. Is something that we need to change, filter…?
    - Normally run on a separate system (an isolated system). Due to performance but also because it's proprietary
- **Notifier:** Receives a message from the director saying to notify the users, the employes or smoneone else.

Reposnses to IDS:

- **Planning:** What will I do when my system gets attacked. What do I do first, next, who do I contact…
- **Identification of the attack:** What is happening, what kind of attack is it.
- **Continment:** Keep the attack in a special place where it can't do much harm, like putting it in a honeypot. Honeypot: fake data that looks legitimate. You use it to understand the attacker's behavior. There are entire Honey netowrks, that has a whole fake infrastructure.
- **Erredication:** To delete the attacker code.
- **Recovery:** Return your network to a secure state.
- **Follow-up:** Updating your response plan. May include taking action against the attacker (but it's usually spoofed address). They may be underage or in another country. Don't do it by yourself, use the police.

A table from Week9, where has (a) hacker, (b) enterprise, (c) internal threat.

Firewalls: An additional layer of defense. Your organization's network is in one side, and the internet in the other side.

- **Goals:**
    - Single choice point.

    - Only authorized traffic can pass.
    - Immune to penetration.
- **Provide:**
    - Service control.
    - Direction control.
    - User control.
    - Behavior control.
- **Capabilities:**
    - Single admin point.
    - Monitor security related events.
    - Platform for other (non-security related) functions.
    - Platform for IPsec: Allowing you to set a virtual private network. The packets are divided into header and body.
        - Transport mode: Protects the body of the packet using encryption.
        - Tunnel mode: Creates another header and makes the old body and header protected by encryption. Replaces the sender by your organization's firewall and the receiver by the other organization's firewall.
- **Limitations:**
    - Can't protect against attacks that don't use the firewall.
    - Can't fully protect against internal attacks.
    - Can't protect against naïve users. As getting a virus (using your notebook) from StarBucks and brining it to your company.
- **Types of filters:**
    - Positive filter: only allows packets with specific types.
    - Negative filter: only *blocks* packets with specific types.
    - Examine only headers.
    - Examine header and payload (door packets inspection [DPI]).
    - Examine the pattern generated by a sequence of packets. Looking for patterns that may indicate possible DoS attacks.
- **Types of firewalls:**
    - Packet filtering firewall: Applies any of those previous types of filters. That is, applies a set of rules for every packet that comes in. Discard some of the packets or saving them for more examination. Image Table 9.1 Packet Filter Rules, from the book.
    - Stateful inspection firewall: It does what a packet filtering firewall does and also monitors TCP connections.
    - Application-level gateway (application proxy): Any connection that you want to make will be break into two connections. The first to the firewall and then to the firewall to the destination. It allows filtering, blacklist, whitelist. But creates overhead because everything is being inspected by this firewall. Users may need to login to do that. Application layer.
    - Circuit-level gateway (circuit-level proxy): Same as above, but in TCP layer. It doesn't look at the content, but at the connection itself.
    - You can mix firewalls: Example, using application proxy for receiving packets and circuit-level proxy for sending packets. In the case that your employees are trustworthy.
- **Firewall installation:**

    - Host-based firewall. More complex.
    - Personal firewall: Firewall in the user machine. Simpler.
- **Firewall location:**
    - Most common corporate network architecture: Internet -> Outer firewall -> (DNS server, mail server, web server, log server) [Demilitarized zone {DMZ}, zone between the outer and inner firewall]-> Inner firewall -> (Development subnet, customer data subnet, corporate data subnet, internal servers, internal mail server).
    - The inner firewall can be split into more internal firewalls.
- **Intrusion Protection System (IPS):**
    - Combination of IDS (Intrusion Detection System) and a firewall.
- **Unified threat management**: Unified threat management products (image). Used to simplify the management of threats. Simplifies the security admin job. On one hand, you have just one thing to manage and set up, on the other hand, it slows down the performance of your system a lot.

Buffer Overflow: It counts as a big percentage of threats found nowadays.

- Problem exists because it is so easy to write beyond the bounds of arrays. For example: char x[12]; x[12] := 5; True to C++, but not for Java, Python, C#.... So the array is x[12], so it goes from x[0] to x[11]. Legacy code problem.
- System's loader puts data in 3 areas: Programs and processes (image book).
    - Data segment (initialized data). Buffer overflow usually doesn't occur here.
    - Stack (function calls and variables local to function).
    - Heap (dynamically-allocated storage).
- Example of buffer overflow attack: (IP = Instruction Counter).

Memory address

1009

1010 MOV AB

1011 SUB A I

1012 JEQ SUBR (2059) # Jump to a subroutine at address 2059

1013 ROTR B

1014 MUL B 5

1015

…

Subroutine variables: char: name[10], int: age.

2059 ADR

2060 MOV

2061 MUL

...

2185 RETURN

...

Stack:

10(x)… Name # 10 Bytes for the name

4(x)… Age # 4 Bytes for the age

00

00

03

F5

Attacker installs malicious code at 10950 (by using a virus, a trojan…)

10950 DO

10951 BAD

10952 STUFF

10953 NOW

…

The program asks for your input and the attacker input:

Attacker inputs “41 44 41 40 53 5C 5C 5C 5C 5C 5C 00 00 00 00 00 00 2A C6” (Adams, his age, and puts something else at the end, the 00 00 2A C6, which correspond to 10950.)

The attack is done with the same privileges as the program that is has affected.

Defenses against buffer overflow:

- Compile-time:
    1. Choice of programming language: Choose modern program languages.
    2. Safe coding techniques.
    3. Improved programming environment: Hardening is done in programming environments;
        - OpenBSD project: It has gone through extensive tests in thirteen and two bugs were found.
        - Windows: It had many bugs, but not anymore.
        - Augmented compilers: Compilers that automatically checks for problem causing actions.
        - Safe standard libraries: Use standard over others, they are usually more protected
            - LibSafe: With this library you wouldn’t need to replace and recompile your code.
    4. Stack protection mechanisms: Changes the way that the stack operates.
        - For example: Stack guard (canary), a warning sign that means not to trust a return. Your need to recompile the program.

        - For example: Stack shield, return address defender. You don't need to recompile the program.
- Run-time:
    1. Executable address space protection: You don't allow executable code in the stack. However, there are legitimate programs that would need to use the stack.
    2. Address space randomization: You randomize the address that the stack and heap is located. If the attacker wants to overcome it, it would need to run that in his computer. Therefore, it would limit worms.
    3. Guard pages: The buffer overflow, It can't pass the stack to go to another place.
    4. Deep pocket inspection: Look for basic attempts to cause buffer overflow.

Buffer overflow still happens a lot. Your best defenses are to write better code in modern programming languages. Legacy code and poorly written code are the biggest problem. That why we need some of the other techniques.

Starting of a new big topic:

Software (program) security: Writing better code. Related to software quality and reliability, but different.

- Now, in **software security**, this is an attacker trying to make your system work differently that it's supposed to work. Like the program expecting integer and the attacker entering an executable code. It has to consider where is going to be executed and which data it will deal with. You should always follow safe programming practices, even for small programs.
- Handling programming input: Input from network connection, mouse, keyboard, sensors, cameras, microphones. It has to consider:
    1. Input size (related to buffer overflow).
    2. Input content:
        - Interpretation of program input. The input has to conform with what you are expecting.
            - Injection attacks: Where the input is not what is expected and intended to do damage.
                - Command injection: Against OS.
                - SQL injection: Related to databases.
                - Code injection: Will execute on its own
            - Cross site scripting (XIS): Goes to another user?
        - Validate syntax: "C14N" = "canolicalization.
    3. Input testing:
        - Input fuzzing: Trying random inputs, to check if you program can handle them properly, and if it fails, it fails gracefully.
- Writing safe code:
    1. Ensure correct algorithm implementation: Make sure that the algorithms that you are using are actually correct. Like if you make calls to a library, you can't be sure that the library was implemented correctly.
        - For example: Does the program uses a poorly seeded random number generator?
        - For example: Predictable sequence number.

        - For example: Debug code, the debug code can be used by the attacker for him to gain some information.
    2. Ensure machine language corresponds to source code: In very highly sensitive classified security environments, you don't trust your compiler, you check your machine code. Or you use a certified and very trustworthy compiler.
    3. Ensure correct interpretation and use of data values: In C is very easy to change an integer by a pointer, making it vulnerable to overflow or corruption.
    4. Ensure correct use dynamic memory: It gets allocated in the heap. You need to make sure you are allocating and deallocating properly. Otherwise, it's a easy target for DoS attacks.
    5. Ensure that race conditions are preserved: Race condition is when two process are trying to get the resources are the same time. You can use methods like synchronization to get rid of that.
    6. Ensure that interactions are protected: Like interactions between your program and the OS.
        - For example: PATH environment variable. The attacker can add a path to his folder and use the same names of the files that you are looking for.
        - For example: Always use least privileges. Functions only have the privileges that they need to complete their task.
        - For example: Ensure safe temporary file use.
        - For example: Consider the size, confidentiality, integrity and interpretation of inter-program data.

New topic:

Containers:

- Packages code and all its dependencies. Advantages:
    - Lightweight.
    - Reliable across environments. Because everything your application needs is in the package.
- "Container image" -> "Container". The thing that you ship is the container image; the container is what you execute.
- Linus and Windows: They are in Linux and Windows, but probably not for Mack.
- Docker: Is *the* container company. Everyone uses docker containers.
- Image repositories: When you create a container image you can store it somewhere. A container repository is like a sore where you can pick containers' images.
    1. Docker Hub is a docker repository.
- Orchestrator: A tool that automatically monitors and manages your containers. Examples:
    1. Docker Swarm.
    2. Kubernetes.
- Containers are conceptually similar to VMs.
    1. VMs:
    2. Server -> Host OS -> Hypervisor -> VM [(Guest OS -> Binaries and Libraries that App A needs -> App A) (Guest OS -> Binaries and Libraries that App A' needs -> App A')]
    3. Containers:

4. Server -> Host OS -> Docker Engine -> Container [(Binaries and Libraries) -> (App A) (App A')]
5. More applications can run, easier for developers, simple shipment (?).

- Can make achieving security (much) more complex. Motives:
  1. Shorter development cycles: months -> days or hours. Because it's easy to go from development, testing, shipping (because it's all together). This is good because it's fast to develop. But there's no time for security then.
  2. Complex interactions between containers. Checking security policy/rules become more difficult.
  3. Shared resources. By definition containers share resources (at least the OS kernel is shared).
  4. Public repositories are public. That makes easy to spread malwares.
  5. Container defaults can introduce vulnerabilities, for example:
     - EXPOSE 80, allows *all* traffic through the 80's port.
     - You should always specify a tag in the FROM statement. Because, by default, you would get the latest.
- How we achieve security in a container environment. You need to look into the entire container process.
  1. Make sure you have secure images.
     - Pull only the images that you need, not more. Only include needed tools library.
     - Don't start with a base image from public repository.
       - Start from scratch if possible.
     - If you must start from a base image, only use a base that has been signed and that you trust.
       - Scan for vulnerabilities and compliance (to your security policy) before you take that image. Do it frequently.
     - Give your app/container only the privileges it actually needs. Principle of least privilege and principle of separation of something.
  2. Secure the repositories (the storage of images).
     - Continuously monitor your repository, like checking to see who has access to it.
     - If possible, use a private repository. Because:
       - You have access control (obviously, as it is not public).
       - You can add some useful security features. Like:
         - Image metadata.
         - Tagging so that you can filter/sort images.
         - Automated policy checking.
  3. Secure deployment.
     - Embrace the concept of immutability. That means that once an image has been deployed, you don't change it anymore. Instead, you create a new one from scratch. As updating the old one could cause problems.
       - Patching containers is never as good as rebuilding then.
  4. Secure runtime.
     - Establish baseline behavior for containers in a normal, secure state.
     - Networked micro services: attack surface is large and complex.
       - Only allow connectivity between containers that actually need it.

        - Restrict open ports and who can access them. Force SSL/TLS. So that you can be sure that you are talking to the right person.
    5. Secure orchestration.
        - Multi-tenancy (having software from many different people): Prevent risks from over-privileged accounts, attacks over the network, unwanted lateral movement (container to container movement).
        - Configure orchestrator to use proper access control for each of the containers. Least privilege for each container. White-listing (this is allowed, this is not allowed).
    6. Secure the host OS.
        - Scan for vulnerabilities.
        - Harden according to relevant guidelines/benchmarks.
        - Ensure container isolation. to anytime it doesn't need to interact with other containers.
    7. Continuously monitor for security (this actually applies everywhere).
        - Log every access to (everything) containers, apps, services, systems…
        - Perform regular audits.
        - Monitor for anomalies.
        - Stay on top of the current research, because they discover the vulnerabilities and publish them.

New topic:

Information flow

- Flow of info through a system.
- Confidentiality.
- Integrity.
- Two ways: through programs and through channels. These are the ways to protect information flow.
    - Through programs:
        - Compiler-based mechanisms to check if everything is fine.
        - Execution-based mechanisms.
    - Through channels:
        - System mechanisms.
        - Protocols

Protecting information flow through programs:

- Let x and y be variables in a program.
- Command sequence C.
    - If we check y and we can notice that it came from x, then, there was an information flow from x to y.
- Explicit:
    - y := x; (not related to code below)
    - temp := x; y := temp;
- Implicit:
    - If x = 1 then y := 0;
    - Else y := 1; (if you enter here, you know something about x)

- If x is a variable, then $\underline{x}$ (x underscore) is the "information flow class" of x. It could be a security label, integrity label, category… So, for every variable in your code you'd assign it a variable class.
- Info can flow from x to y if $\underline{x}$ (information class of x) is less or equal $\underline{y}$ (information class of y). Confidentiality.
- Info can flow from x to y if $\underline{x}$ (information class of x) is more or equal $\underline{y}$ (information class of y). Integrity.
- If there are several classes (eg. $\underline{A}$, $\underline{B}$, $\underline{C}$).
    - That is, least upper bound {$\underline{A}$, $\underline{B}$, $\underline{C}$} is less or equal $\underline{y}$. Like 10 from {5, 3, 10}.
- Compiler-based mechanisms check that info flows throughout a program are authorized.
- A set of program statements is certified with respect to an information flow policy if the info flows in these statements do not violate policy. For example, consider the statement:
    - If x = 1 then y := a
    - Else y := b;
    - Information flows from x and a into y or from x and b into y.
    - If $\underline{a}$ <= $\underline{y}$, $\underline{b}$ <= $\underline{y}$ and $\underline{x}$ <= $\underline{y}$ then the information flow is now secure.
- Statements:
    - Assignment (:=)
    - Composed.
    - Conditional (if).
    - Iterative (while, for)
    - Go to.
    - Procedure calls.
    - Function calls.
    - I/O statements.
    - Eg:
        - 1) y >= f(x1, x2, … xn)
        - As all the information is flowing from those x's to the y, we would need the lowest upper bound {$\underline{x1}$, $\underline{x2}$, … $\underline{xn}$} <= $\underline{y}$.
        - 2) if (x1, … xn) then
        - S1;
        - Else
        - S2;
        - Check if information flow within S1 and S2 are secure: lub {$\underline{x1}$, … $\underline{xn}$} <= glb ($\underline{y}$ | y is the target of assignment in S1 or S2).
        - 3) Infinite loop:
        - procedure SurpriseFlow (Boolean x; Boolean y)
        - begin
        - y := 0;
        - while x = 0 do
        - (nothing)
        - y := 1;
        - end;
        - Meaning that there is a flow of information from x to the behavior of the program.
        - 4) begin

        - y := 0;
        - while x = do
        - (stuff not involving y)
        - y := 1;
        - (do stuff not including x or y)
        - end;
        - If you insert a break point at the program, you can know x based on y.

Lecture 17:

Infoflow through programs:

- Compiler-based mechanisms
- Execution-based mechanisms
    - Hard to detect flows when they are implicit
    - 1974: Fenton: Data Mark machine
        - Turn implicit into explicit
        - PC, PC
        - Eg: (300) if x = 0 then goto n;
        - (301) Else x:= x-1;
        - (302) temp := 10;
        - (5000 “goto”) y := 2; (if you look at y and you see that it’s 2, then you know that the program counter is 5000, that’s an information flow, and you also know that there is a implicit from from x to y, because you know something about x based on the y value).
        - That would be equivalent to:
        - If x=0 then {push(PC,PC), PC = lub(PC,x), PC:=n}
        - Else: {if PC <= x then {x:=x-1;} else skip}
        - Then, with that, you can check the information flow validity and skip if it’s not valid.
        - Explicit flow from PC -> y
        - .:. need to verify that PC <= y
        - But there is an implicit flow from x -> y
        - (x, PR) -> y .:. neeed lub {x,PC} <= y
        - .:. set PC = lub(x,PC)
        - And check PC <= y
        - Turns implicit flow (by using extra statements) into explicit flow so that they can be handled by usual mechanisms (can be easily detected).

Information flow through channels.

- System-based mechanisms (to protect information flow): Putting something between a communication to check for integrity (and confidentiality in the second case).
    - For example: Security Pipeline Interface (SPI).
        - Host -><- SPI -><- Main Disk
        - SPI -><- Second Disk
        - The SPI checks for integrity of the communication by using cryptography.
    - For example: Secure Network Server Mail Guard.
        - Work Station (Secret Network) -><- Data [queue] -><- Filter [queue] -><- Mail Guard -><- Work Station (Unclassified Network).

        - Checks for integrity and confidentiality.
- Protocol-based mechanisms (to protect information flow):
    - For example: Passwords.
        - Use a protocol that does not rely on reusable passwords.
    - One-time password (OTP):
        - For example: (Lamport 1981, S/Key)
            - Let h be a one-way hash function.
            - Alice picks an initial seed k_0 and computes $h(k_0) = k_1$, $h(k_1) = k_2$, … $k(k_{n-1}) = k_n$
        - Seed k_0 is shared with system (at initial time).
        - System store (n, k_n).
        - Alice logs in: enters "Alice", system challenges her with i-1. Alice responds with k_{i-1}, system checks that $h(k_{i-1} = k_i)$.
        - If successful, system sore (i-1, k_{i-1}).
        - Instead of the system asking for a password, it asks for a challenge, which is a hash that the user has to compute, and the user can use only once that challenge, that's why *n* challenges are created.
    - Public key based challenge response authentication:
        - Alice sends "Alice" to System.
        - System sends a challenge *n* to Alice.
        - Alice sings that challenge with her private key and sends that signature to the System.
        - System verifies her signature by using her public key.
        - Advantage over OTP: In OTP the system has to keep track of *n*, and, when it runs out, Alice has to ask for more by physically visiting the system. But here (in this method), *n* is a random generated number, there is no need to keep track of *n* and you have 2^{128} possible *n*s (infinite)
        - Nonce (number used once).

Note: "covert channels": Path of communication that you didn't realize that existed in your system. You should discover it to protect against it.

- For example: Alice uses a file (that's not a covert channel). Bob may ask Alice to check if the file exists, that's a covert channel, you have learned something from it.
- For example: Using sounds not heard by humans to transfer data.
- Ways to protect against covert channels. First you look for all covert channels in your system. Then (both of them will affect performance, therefore, it should only be used by exceptionally secure systems):
    - Give every process the same amount of resources.
    - Injection of randomness. That is, for example, creating and deleting random files. That creates a noisy covert channel.

Containment/Isolation:

- Total isolation: Any process that has total isolation doesn't flow information to anything else (not so useful to us).

- Partial isolation: More useful to us. **Virtual machines (simulates a machine), docker containers (abstracts the OS of the machine), sandboxes are ways to achieve partial isolations**.
    - Sandboxes: Puts constraint at what the thing inside a sandbox can do.
        - You get two types of protection, you can protect the system from the application that is inside the sand box, and vice-versa.
        - Different types of sandbox:
            - Minimal sandbox: Program has access to the CPU, a slice of memory, keyboard, mouse.
            - Default sandbox: Adds to that, access to the webserver from where it was downloaded.
            - Broad sandbox: Adds to that, a set of specific resource that is in the machine.
            - Open sandbox: Adds to that, has access to anything that the host machine has access to.
        - Java security model has:
            - Bytecode verifier. So that you don't go beyond what Java can do.
            - Class loader. Sets permissions to classes.
            - Access controller. Gives access to the OS itself based on policies.
            - Security manager. Gives access to all the rest of the system resources.

Why is software security a bigger problem now than in the past? Connectivity (many machine), extensibility (plugins and extensions that may compromise security) and complexity (many code bugs and security bugs, can't build a bug free big software). Defect rate is the square of the code size.

Lecture 18:

Flag -> Record -> Flag -> Record …

1B -> 100B ->… ->

1. 1. i := 0;
2. 2. while (not EOF)
    a. {
    b. 3. x := readbyte(file[i]);
    c. 4. if (x = FLAG2) then /*ie.sensitive*/
        i. {
        ii. 5. Port:=FLAG2;
        iii. 6. for j:= 1 to 100
            1. {
            2. 7. x:= readbyte[filei+j]);
            3. 8. y:=protect(x);
            4. 9. port:=y;
            5. }
        iv. 10. i:=i+j;
        v. }
    d. 11. else port := x;

    e. 12. i++;
    f. }
3. 1000. Protect (char x)
    a. {
    b. 1001. Tmp:=ENC(x); /*encrypt*/
    c. 1002. Return(tmp);
    d. }

Flag 1 = Sensitive data

Flag 2 = No-sensitive data

Assume there are 2 information flow classes (“1”, and “2”)

Variables:

- i,j: class “1”
- File: class “2”
- Port: class “1”
- x: class “2”
- y: class “1”
- tmp: class “1”
- IP: class “2”

Lines:

1. – (no information flow)
2. –
3. File[i] -> X OK
4. – (just because FLAG 2 is a constant, it’s not sensitive)
5. –
6. –
7. File[i+j] -> X OK
   100: (IP, ENC(x)) -> tmp
    a. i.e. lub (IP,ENC(X)) <= tmp NOT OK
    b. So IP it’s better to change “1”
8. Tmp -> y OK (cuz class 1 to class 1)
9. Y -> port OK (cuz class 1 to class 1)
10. J -> I OK
11. X -> port NOT OK (cuz one of them is class 2)
    a. So change code to “port := unclass(x);”
12. – (no information flow)

New topic

Writing better (more secure) software.

McGraw’s book, these are his pillars to write better code:

Pillar 1: Applied Risk Management (most important) IMPORTANT

- Risk Management Framework: A decision support tool for the business leaders in your organization. Identify, rank and prioritize and the risks in your software, which is based on business goals and requirements.
- Risk: Probability * Impact
    - Quantify impact in terms of business requirements (market share) and mission requirements (deliverables, tasks). If you can't do it, you'll never convince the CEOs, CFOs…. Thus never giving you the money to fix those things.
    - Five activity steps:
        1. Understand the business context (goals, priorities, circumstances, more money, increase market share).
            - Meet SLAs (service level agreements), like performance, quality, down-time. Like we having a server and saying that it will not fall frequently and having a contract.
            - Reduce development cost.
            - Generate high ROI (return of investment).
        2. Identify the business ([a risk that can damage our business goals] financial loss, damage to brand, violation of regulatory constraints, increasing development costs…) and technical risks ([technical are identified in the context of business goals], like a crash of a service [related to financial loss to the business], like unauthorized data access [related to privacy problems to the business]).
        3. Prioritize risks:
            - We want to answer the question "what do we do first"? That's why prioritization is important.
            - What is the best allocation of resources for risk mitigation? -> Buy a tool (like buying a firewall) ? Train my people? Hire more people?...
        4. Define risk mitigation strategy:
            - Cost (time, dollar, people…).
            - Likelihood of success. (if you use a month of developer's work, what is the likelihood of success for them to develop what need to be developed).
            - Likelihood of completeness.
            - Impact over larger body of risks. Solving a risk may solve others. Like using two-factor authentication., that solves the internal and external attacks.
        5. Carry out fixes and validate. That is, after implementation, did the mitigation actually solved the problem? Has to be repeatable and measurable (how mush I solved the problem). So that you can show the CEO: here is what we did, and here is what we solved (in order to convince them). So, based on that, you can receive more money to do more things.
- Eg.:
    - Business goal: time to market.
        1. Maybe you want to take 25% of the market before the competitors do it.
            - Rank: High priority.

- Business risk: software is not ready.
    1. Maybe you need your software to be ready in 6 months (before Apple releases theirs)
        - Rank: High priority.
- Business impact (this is what you show to the CEO): revenue loss of 10 million dollars; market share loss: 15%; reputation loss: negligible (as we are a start-up).

Pillar 2: Software security "touch points"

- Focus on software artifacts (something that is left behind, that includes many things), like:
    1. Requirements and use cases document.
    2. Architecture and design document.
    3. Test plans.
    4. Code (software)
    5. Test results
    6. Feedback from the field.
- Touch points in order of effectiveness. Rank of most effective touch points>
    1. The most effective touch point is code review (analysis of "code" artifact).
        - Find bugs in code (security bugs).
        - Static analysis tools can be very helpful. But, needs humans' eyes as well (other perspectives are important).
    2. Architecture risk analysis ("requirements and use cases", "architecture and design", "test results").
        - Find flaws in overall design.
        - Must document assumptions (like having a commentary on the code that assumes that the user has already been authenticated).
    3. Penetration testing ("test results", "feedback from the field").
        - Probe system in its final operating environment (in the field)
    4. Security testing (looks at the "test plans" document artifact) (testing in your lab).
        - Need to think like an attacker.
        - Need external experts.
    5. Abuse cases (looks at the "requirement and use cases" artifact).
        - For example: Ordinary users shouldn't be able to modify their own salary.
    6. Examining security requirements ("requirements and use cases").
        - Explicit description of data protection needs.
    7. Security operations ("feedback from the field" artifact).
        - Understand what leads (led) to a successful attack.

    .:. most (all) touch points are best done by people not involved in the design and environment (it's better to bring external people).

Pillar 3: Knowledge

- Principles, like slogans, like we focus more on people.
- Rules of the company.
- Guidelines, like comment your code.

- Vulnerabilities, exploits, attack patterns, historical risks.
- Coding errors: tools to help with errors.
    1. Input validation, it's important to validate the input that is coming into your program.
    2. API abuse (unchecked return values).
    3. Security features: Insecure randomness, bad access control.
    4. Time and state like dead lock, TOCTTOU – time of check to time of use [URL changing to change price]).
    5. Error handling.
    6. Code quality, like memory leaks
    7. Encapsulation
    8. Environment

0WASP Top 10 2017, CWE 2019 and TOP 10 IT Security Canada was mentioned.

End of Pillars subject, now this is chapter 20 of our textbook

Vulnerability analysis (security testing) logging and auditing

Vulnerability analysis:

- That is not only just software problems, it also includes management, procedures… It includes all your organization as a whole.
    - Failure of controls: "vulnerability", "security flaw".
    - Using that failure to gain some advantage: "exploiting the vulnerability".
    - Who uses it: "attacker".
- Formal verification:
    - {preconditions} program {post recommendations}
    - But we need
    - {system state} system {compromised state}
        - Penetration testing (the (hired) people: tiger team attack, red team attack do the penetration study): It's a testing technique, not a proof technique. It's generally scattered testing (doesn't test everything).
            - Try to achieve one specific attack.
            - Try to find all vulnerabilities in a specific period of time.
            - Study conducted from attacker's point of view.
            - Flaw hypothesis methodology. That's when the hired people have access to your system and try to find vulnerabilities (not the attacker's point of view).
                - For example a flaw that a attacker can get in, then you find that someone who gets in can change the authorization of other people, they the attacker can allow others to get it.
                - Testing -> Hypothesis -> Generalization -> Eliminatio

Logging and auditing: A posterior technique, that is, after the fact, after the breach.

- **Logging:** Logs sequence of events leading to your system being in an insecure state.
    - Need to log info that will facilitate useful analysis.
        - Need to understand what is needed to violate the security policy.
    - Logs can also (besides helping auditing):

        - Deter the attacks (scare them away, because they know that everything that they do will be recorded).
        - Understand expected patterns of usage/behavior. So that you can detect anomalies.
            - Architecture of an auditing system.
                - Logger.
                - Analyzer.
                - Notifies.
        - How to log:
            - Many system log ambiguously, that is, there is some contextual information missing.
            - Need to include stuff context (may be a trial and error process).
            - May use a formal grammar-based approach, so that it's a form that is useful for hymans.
        - Log sanit (sanitation) portion.
            - Logging system -> log -> sanitizer -> users. OR
            - Logging system -> sanitizer -> log -> users.
- **Auditing**: Analysis of the logging that gives you actionable information about what you should do.
    - Auditing mechanisms must be integrated with system design and implementation. That is, it must be a good interface, it must be configurable.

System Assurance and System Evaluation

System Assurance: Convincing yourself that you did the right thing. This is hard.

System Evaluation: Convincing others that you did the right thing (like your boss, customer, judge, suppliers). This is harder.

System Assurance:

- Our estimate of the likelihood that a system will not fail in a particular way.
- May be based on:
    - Process that was used to build the system, like Garry Magraws three pillars. It seems to be a good way to build a system, so it's probably wise to use.
    - Identity of designer/developer. People may do really good work, so if they've designed it, I could probably trust it.
    - Experience of designer/developer. People that previous successfully developed a system, could do it again.
    - Use of formal methods.
    - Introduction to bugs. The developers deliberately introduce bugs so that you can test your test team and check if it's a good team.
    - Complexity (simplicity) of the system. "Complexity is the enemy of security".
    - May want assurance with respect to:
        1. Functionality, does the system do what it's supposed to do?
        2. Strength of mechanisms, like the strength of the cryptographic algorithms and access control.

3. Implementation, are you confident enough that there won't be buffer overflow in the system?
4. Usability, can the system cope with unexpected user inputs? Can the developers easily test the system, and they don't forget to remove the code used to test?
5. Note: May be very different expectations from different set of users and audiences. For example, the functionality that your mother needs to use the system is different than yours.

- Techniques to prove to yourself that the system is secure:
  1. Security techniques, like white box and black box testing.
     - Look for obvious flaws, common flaws (like buffer overflows) and uncommon flaws.
     - Show tat a set of objectives was assured by at least one protection mechanism.
     - Established lists of how you do your testing, there are some standards and guidelines out there.
  2. Formal methods.
     - BAN logic, proposed to analyzing and reasoning about beliefs.

       Example of the notation:
       - A |(and three horizontal bars) x -> A believes X.
       - A |~ X -> A once said X.
       - A | => X -> A is an authority on X.
       - A (triangle) X -> A sees X.
       - A <-k-> B -> A and B share the key k.
       - #X -> X is fresh (eg. contains current timestamp),
       - A |(and three horizontal bars) A <-k-> B, A(trinangle) {x}k all that means that A|(and three horizontal bards) B |~ X.
       - (not copied 2 other statements).
       - If you continue that, you could build a logical path from A trusting CA, to A trusting a certificate assumed by CA.
       - May help to root out the assumptions
  3. Analyze bug discovery.
     - Fault injection, deliberately introduce bugs. If you introduced deliberately 100 bugs and your team caught 70, you can know that when you didn't deliberately introduced bugs, you may have 30 unknown bugs out there.
     - Look at bugs found and estimate, 10 bugs in 1000 lines of code, but product contains 1000000 lines.
     - Look at rate of bug discovery.
     - Make developers responsible for fixing their own bugs, no matter when the bug is found. Therefore, if the bug was found 10 years later, the same guy who wrote the bug would need to fix it. This makes people more careful.

- Hire good people, train them well, encourage good habits -> You will get better quality code.
- ISO 9001 certification.
    - Extensive documentation of everything that happens in your project development. People should actually read it thereafter.
- Do prolonged testing. MTBF (mean time between failure), that's the difference between security testing and normal testing.
    - Eg.: Assume a large complex product:
        - 1 million bugs, each within MTBF of a billion hours.
        - Bob is the bad guy.
        - Alice is the angel.
        - Bob: 1000 hours of testing/year.
        - Alice: 10 million hours of testing/year.
        - After a year, Bob finds a bug and Alice finds 10000 bugs.
        - But the probability that Alice has found Bub's bug is 1%. (she found 10000 out of a million).
        - Alice increases to 100 million hours testing/year by hiring people and buying stuff.
        - After 10 years, she'll find Bob's bug.
        - We shouldn't look only at the number of bugs (and get depressed), but also by the type of the bugs. That is, we need to consider the types of bugs, as well as their number. If we find that buffer overflow is common, if we change the compiler, we get rid of many bugs at once.

System Evaluation:

- Assembly evidence.
- You want to reassure principals (customers, boss…) who will rely on your system.
- Problems:
    1. Sometimes the evaluation or the results of the evaluation get ignored. That is, your principals ignore it. For example, buying a product that was poorly evaluated.
    2. The evaluation may be too narrow. It may apply only to the technical parts of the system, and completely forget about laws or usability for example.
- 2 broad classes for evaluation:
    1. Formal evaluations:
        - Evaluation done by a Relying party (the party that will use your system).
            - Eg: Trusted computer systems evaluation criteria (TCSEC) "Orange book". This was in US. From 1983-1999 and was aimed on OS. That is, it was an evaluation criteria for OS.

- Information technology security evaluation criteria (ITSEC). This took place in Europe from 1991-2001 and it was an evaluation criteria for any IT product.
- Canadian Trusted Computer Product Evaluation Criteria (CTCPEC). This took place in Canada from 1993-early 2000.
- Process: The process is similar to each of these evaluation criteria.
    - It starts with government user wanting some product to be evaluated.
    - People in the government allocated to do the evaluation.
    - Evaluation is completed within (2-3 years).
    - If the evaluation is successful, the product would go on "evaluation products" list, which is a list specifying the products that can be bought.
    - Taxpayers would pay the bill for the evaluation.
    - Evaluation products were 1-2 generations behind, and an order of magnitude more expensive. Therefore, it was a system that didn't work.
    - A bigger problem: the evaluations were geographically specific. So if it was passed in US, you wouldn't be able to sell it to Canada and Europe for example.

- Evaluation done by a third party.
    - To overcome these process' problems. It was created a **Common Criteria** specification (qualification, evaluation).
        - The ISO 15408 (1996-present) was created. Not geographically specific anymore.
        - This (expensive) evaluation stablishes one of seven levels of certification, EAL1, EAL2… EAL7.
        - Why to do it: It's a requirement if you want to sell for the USA government. And may be a requirement if you're trying to sell to others.
    - Elimination of re-evaluations.
    - CLEF (CCEF) is the lab where the evaluation is done.
    - Vendor pays (not the government, when compared to the "relying party" above).
    - Can be helpful even without evaluation. Because you can check the documents that would be tested, and test it yourself, so that you know that your system is safe.
    - It takes a long time, it's expensive, and the labs can make mistakes when evaluating.
- Evaluation done by relying party and third party
    - FIPS 140:
        - 140-1: 1994-2001
        - 140-2: 2001-present

- Evaluation tamper-resistance of a cryptographic model. That is, the previous two evaluation looked at the product, this one looks at the cryptographic model.
- It's a "relying party and third party" because it used to be required by the USA government. But it is also currently used to impress normal companies.

2. Informal evaluations (public)
    - Semi-open design
    - It's open source.
        - Advantages:
            - Anybody can look at the code and tell you about it.
        - Disadvantages:
            - When it gets complex, people stop looking at the code and telling you about it.
    - CERT, bug track, etc… Where bugs get posted public and you can fix your code.
        - Comprise: Instead of publicly posting the bug, the person can actually send an e-mail to the company, so that the company can react in time to the bug. When the bug is fixed, the person that discovered the bug is recognized. If the company doesn't give attention to it, the person who discovered the bug can post it anyway.
    - "Bug bounty" programs: The company may pay to people for them to find bugs.

AI/ML/DP: Cyber (in)security:

- AI: computer programs that have intelligent, human-like abilities.
    - Eg. speech recognition:
        - "Smart" home devices.
        - Travel assistants.
- ML: subset of AI that "learns" from data.
    - Supervised/unsupervised.
    - Eg. predicting the next song you'd like.
    - Eg. predict maintenance needs, optimizations to companies about how can they improve.
- DL: Subset of ML that can correct itself.
    - Has multiple layers/levels.
    - Eg. self-driving cars.
    - Eg. medical diagnostic tools.

AI in the hands of the "good guys":

- Help organizations analyze the threats and respond to attacks.
- Automate menial tasks.
- Examples:
    - Identify and remove malware from infected machine.
        - Quickly observe anomalies (something look strange here), predict threats, react.

        - In particular, in devices that have not been patch against that specific threat.
    - Identify and defend against threats to mobile devices.
        - Eg: Malware.
        - Eg: BYOD (bring your own device). That is, bringing your own device to the company.
    - Enhance threat intelligence (within you organization).
        - Detecting potential attacks, assessing vulnerabilities, analyzing the network, protecting end point, etc. All in an automated way of course.
    - Enhance human analysis with reduced human effort.
        - Eg. Triaging alerts. That is, filtering alerts (finding the needle in the haystack).
        - Eg. Triaging logins, phishing, etc…
    - Reduce zero-day vulnerabilities.
        - Eg. Especially in IoT devices, which are highly unprotected.
        - Eg. Use ML to monitor traffic on dark web.
    - Improve authentication.
        - Eg. AI-based face recognition.
    - Improve vulnerability analysis (from external information).
    - Security policy creation.
    - Perform behavioral analytics.
        - Eg. Your typing patters, your scrolling patters, your IP address that you log in from.
        - May be useful for (continuous) authentication. That is, it may continuously authenticate you based on your expected patterns.
        - May be useful for keeping permissions current.
        - Useful for feeding into IDS (intrusion detection systems).
    - Improve security testing.
        - As it scans your network and learns about it, it can help to improve security testing.

AI in the hands of the “bad guys”. help attackers to become more effective:

- Weaponize AI:
    - Eg. automate attacks on a massive scale (many systems). Both launching attacks and coordinate them.
    - Eg. Exploit vulnerabilities. So, you better use your IA in your company first, before the attacker does it.
    - Eg. Disguise attacks. So, you better train better your IDS.
- Attack image recognition systems.
    - Eg. Using some tapes in a stop signal so that the IA recognizes it as a 45km/h signal.
- Using AI to defeat IA: bypass spam filters.
- Using AI to defeat IA: bypass malware detection tools.
- Using AI to defeat IA: bypass facial recognition systems.
- Using AI to defeat IA: forging voice commands. Like adding something that you can’t even hear, and the system will now receive a command that you couldn’t even notice.
- Using AI to defeat IA: fool sentiment analysis system. For example making slight changing on posts to make something negative been understood as something positive.

- Modify functionality of AI system.

Course objectives:

Goals:

1. Better understand about how and where you place your trust. Understand the technologies and specially their limitations.
2. You build systems that are better than we have today.

Diagrams:

1)

User -> System -> Defenses -> Management System.

User -> Passwords, identity (globally unique identifiers), identity mapping, biometrics, authentication.

System -> Security policies, cryptography (sym, assym, hash, key generation, key management), access control, database security, OS security (hardening, virtualization).

Attacks, defenses -> Malwares (viruses, worms, defenses), denial of service (and types, defenses), intrusion detection (goals, requirements), architecture in an IDS system and response, firewalls (goals, limitations, types, installation, location, configuration), buffer overflow (defenses, compile time and run time), software security (handling program input size content, writing safe code, McGralls three pillars methodology, risk management, touch points, knowloged), OS top 10 (CSI top 10).

Management issues -> Security assurance, security evaluation.

2)

Threat <–> Security Policy <–> Security mechanisms and security specifications <–> Design <–> Implementation. They are <-> connected to operation and maintenance. On the side, management issues.

Threat -> Who is the attacker, what resources they have, what is valuable in our system. Definition of a system.

Security Policies -> Confidentiality, hybrid policies…

Security Mechanisms -> Cryptographic, biometrics, passwords...

Design -> Information flow within our system and between or system. Confein the information.

Implementation -> Attack, vulnerabilities, protections. Trojan horses, viruses, worms… tools that we have IDS, firewall, network architecture (configuration, redundancy), building secure software.

Management issues -> Proper security testing and the difference between other tests. Logging and auditing. Hiring the right people.

Operation and Maintenance -> Constant vigilance, updates the threat model, security policy…

3)

This course is about a way of thinking. Be able to think about the possible problems and potential solutions. So that everything you see, you ask yourself "how is it going to help me". If there are two possibilities, which one would you use (giving the reasons). For that you need to understand the limitations, including the effort and price. You need to think about what the bad guy will do and what you will do, repeatedly. You also need to think about the end user (usability) and the manager and customer (cost).

About the final:

It's worth 55%. Closed books. Not calculator, no anything. Most of the questions have multiple parts. Short answer, fill in the blanks, long answers, multiple choices. Topics: Malware, 2 or 3 questions. 1 or more question on these: Intrusion detection. Assurance and evaluation. Denial of Service. Firewalls. Passwords biometrics and identity. OASP and software security. Buffer overflow. OS security. Access control. Database security. Cryptography. Logging and auditing. Information flow.

Topics:

1. Malware.
2. Intrusion detection.
3. Assurance and evaluation.
4. Denial of service.
5. Firewalls.
6. Passwords biometrics and identity.
7. OWASP and software security.
8. Buffer overflow.
9. OS security.
10. Access control.
11. Database security.
12. Cryptography.
13. Logging and auditing.
14. Information flow.

Midterm "actual" questions and its "actual" answers:

1) **What's a confidentiality policy and what's an integrity policy?**

Confidentiality policy hides data from people that are not supposed to see it. And people that are supposed to see it, can actually see it.

Integrity policy what changes can be made to the data and who is allowed to make those changes.

2) **What kind of policy is RBAC (Role-Based Access Control)?**

It's a hybrid policy. The main advantage of RBAC is the amount of data that is reduced. The administrator has to do much less work. Because, for example, compared to an access control matrix, there are much fewer rows, because instead of each user having a row, in RBAC, each group has a row.

3) **Advantages and disadvantages for ACL (Access Control List) CL (Capability List) and Authorization Table. Explain one implementation consideration of an ACL, CL and Authorization Table.**

ACL cab easy check all the people that have access to a resource, can be easily created and maintained… CL is the opposite of an ACL, it's easy to see what resources a specific person has access to, better suited for delegation… An authorization table can be sorted to act as an ACL and can also be sorted to act as a CL. However, it's stored centrally (may become a bottleneck), therefore it has less performance than locally stored in each computer.

Implementation consideration is done after you made the choice of using an ACL, CL or Authorization Table.

Implementation consideration for ACL: Which subjects can modify the ACL? ACL applies to privileged users? Does it support groups? Does it support wild cards? Does the ACL modify default permissions? How to deal with delegations?

Implementation consideration of CL: To protect if from the users: hardware solution (architecture), software solution (protected pages in memory), cryptographic solution (a CL is signed by a private key known only by the OS). If you have a resource which is accessed by a lot of people and you want to turn off some of those permission, what can I do? We can use indirection where everyone points to a name and we somehow revoke that name.

4) **Biometric techniques.**

Instead of "eyes", we should have said "iris scan" or "retinal scan". Instead of "face", we should have said "face recognition", "facial geometry", "facial thermo patterns". Instead of "hand", we should have said "hand print", "hand geometry".

Advantages: You can forget it, you can't lose it, hard for the attacker to guess.

Disadvantages: It can be stolen, like your fingerprint and your face which is quite public. If it gets stolen, you can't replace it. Reading is never the same twice, which leads to type 1 and type 2 errors. More costly.

5) **Compare KDC (Key Distribution Center) and CA (Certification Authority) by the data that they hold, the data that they create and the implications for security and privacy. What's the trust anchor?**

KDC holds a symmetric key for every user in the system. KDC creates a symmetric key (session key) for every pair of people that want to talk to each other. CA holds only its own private key. CA creates a certificate for every user.

KDC implications. Implication for privacy: If an attacker breaking into a KDC he ceases the data that is held and generated, and can read all traffic (previous, present and future). Implication for security: the attacker could impersonate every user in the system.

CA implications. Implication for privacy: If the attacker gets CA's private key, they can generate certificates that will be trusted. You can create a new key pair for Bob, you can put the public

key in a certificate and sign it with the CA's private key. If someone uses that public key to encrypt some data for Bob the attacker will be able to read it. Bob wouldn't be able to decrypt it because he doesn't have the private key. Then Bob would immediately recognize that something has gone wrong. The attacker can't read previous data. The attacker can sign as he was Bob.

The trust anchor is the CA's public key. The user holds and is sure that he has the CA's public key, she knows it by a fact because she could go to the CA's office physically and get it installed in her machine; or she can get it from the browser, which comes with certificates in it (risky). That way, Alice can be sure that the public key, that came with the certificate, is from Bob. Therefore, the CA's public key allows the users to get to any other public key in the system.

If Alice and Bob have different trust anchors, their CAs have to cross certify each other.

6) **What's the underlying assumption that is the basis for the password formula?**

The assumption is all about the password space. The assumption is 64 possible choices times the number of characters in the password. As people choose from a much smaller space (like only small case letters). So, what can we do about it? We put password rules in place (like it has to have at least one special character).

7) **Techniques to counter online and offline guessing password attacks.**

Online: Exponential back-off, limit attempts, two factor authentications, all or nothing error messages.

Offline: Hashing, salting, password aging.

8) **Operating system security (multiple choice).**

The two forms of virtualization: Native and hosted.

Backup: Something that you do very frequently (usually daily). It's used in case you lose some data or the system crashes. Using the backup, you can restore it. You put Microsoft Office here.

Archive: Something that you store for decades. Used for important data, typical for legal and operational reasons. You put your logs here.

Midterm Questions

1) **What is the definition of a security engineering? What are the steps to build a secure system? What is the definition of a system?**

Security engineering is building systems that are dependable in face of malice, error, or mischance. Dependable (trustworthy and reliable), meaning that it does what it's supposed to do, no more, no less. Malice meaning the corrupt intentions of dangerous users. Error meaning human error, like not being properly trained. Mischance meaning something that is non-human, like natural disasters.

To build a secure system you start by defining your thread model: who are the attackers, what resources do they have, and what is valuable in my system. Then, based on your thread model, you define your security policy, which defines unambiguously the secure and unsecure states of your system. After that, you define the security mechanisms, which are responsible to enforce

your security policy. Following that, you define your security specifications, that specifies which security mechanisms are going to be used and how to use them. Then, you design, implement and maintain the system. Building a secure system is an iterative process.

The definition of a system is subjective, its size and complexity depends on the subject.

2) **What is equation for calculating the probability of guessing a password?**

The formula is P = GT/N. Where:

- P = the probability of the hacker discovering your password over a time period;
- G = # of guesses per time unit;
- T = # of time units;
- $N = C^S$ = # of possible passwords;
- C = # of possible characters;
- S = password length.

3) **Differences of Access Control List, Capability List and Authentication Table? Which one should be deployed? Who will you need to talk to? What will you ask?**

Access Control List (ACL): It's stored with the resources, e.g.: Resource 1: { (read: [user 1, user 2]), (write: [user 2, user 3]) }. Not suited to change the status of a user, for large populations (specially when constantly changing), and for delegation and revocation. Suited to change the status of an object. The originator can change it. Easier to understand.

Capability List (CL): It's stored with the users. It's the opposite of an ACL.

Authentication Table: Each row has a user, a permission and an object.

Deployment depends on: How dynamic is the user population; how rapidly is the company growing; how prevalent is delegation and revocation in the environment; which hardware do we have at our disposal; the mechanisms that are already in place.

Who we need to talk to: Ask to the IT staff how people access the resources. Talk to the security administrator, talk to the manager, talk to the HR staff. Figure out who needs what permissions. Find organizations needs.

4) **What biometrics should be used for a central server (Face image, retinal scan, hand scan)? What is the initial intuition of which to use?**

Biometrics advantages:

- Hard to fake.
- Convenient, as you don't need to remember and change it frequently.
- Stable and enduring.
- Strong accountability, as someone cannot later renounce having taken an action.
- Enables two-factor authentication.
- Easy to use.

Biometrics disadvantages:

- Difficulties in correctly capturing the biometric.
- Privacy, as once hacked you cannot change it.
- Errors, as false negative and false positive.

- High cost getting the system up and running and also the storage.
- Integration into the security program is more complex compared to passwords.
- Some people may consider invasive.
- Not in favor for physically challenged people.

Face image scan: Doesn't work properly for people that are constantly changing their appearance, as changes in beard, hair cuts and kids. Higher error rates, as twins exist.

Retinal scan: More invasive. Less susceptible to errors. Expensive. More easily affected by eye related diseases. May lead to eye problems.

Fingerprint scan: Doesn't work properly for older people, as their fingerprint is not so evident.

5) Deciding between a PKI and Kerberos? What is a KDC and a CA? What is a trust anchor?

A Key Distribution Center (KDC) is part of a cryptosystem intended to reduce risks inherent in exchanging keys. It is used in symmetric key environments. It shares a key with each of all the other parties thus, it can read all the data. Therefore, if KDC is compromised, the data will be readable by the attackers. It has a good performance. Need to protect for confidentiality and integrity. The KDC system is part of the Kerberos system. But, in Kerberos, the KDC is divided between two different agents: the AS (Authentication Server) and the TGS (Ticket Granting Service).

A Certificate Authority (CA) is a trusted entity that manages and issues security certificates and public keys. It is used in asymmetric key environment. The CA is part of the Public Key Infrastructure (PKI) along with the Registration Authority (RA) who verifies the information provided by a requester of a digital certificate. If the information is verified as correct, the CA can then issue a certificate.

In PKI, the trust anchor comes from the CA (a third-party). The trust anchor is a public key for the CA. It's used to verify certificates of other users. If Alice CA trusts Bobs CA then the CAs create a certification for each other.

6) What are the different security models and their innovation? What is a security policy?

The Bell La Padula model focus on confidentiality. It allows to write-up (star property) and read-down. It allows a subject to transfer rights to other subject of their choice. It's useful for military.

The Biba model focus on integrity. It allows to read-up and write-down. Subject may not corrupt data in level ranked higher than the subject, or be corrupted by data from a lower level than the subject. It's useful for the banks.

The Clark-Wilson focus on integrity. It has well-formed transaction, which is a series of operations that transition a system from one consistent state to another consistent states. The Integrity Verification Procedures audit the well-formed transactions. Useful for business.

A security policy defines the secure and non-secure states in the system. It states what is allowed and what is not allowed in the system. It is ensured by the security mechanisms (with the help of the security specifications). It takes into account the thread model.

7) What are the top four measure to prevent attacks at OS? How is the initial setup and patching of the OS done? What is virtualization? What are the virtualization alternatives?

The four top measures to prevent attacks at OS are 1) Patch OS and application using auto-update; 2) Patch third-parties applications; 3) Restrict admin privileges to OS and applications based on user duties; 4) White-list approved applications.

The initial setup begins with the installation of the OS in a protected environment. The source of any additional device driver code must be carefully validated. The initial installation should install only the minimum necessary for the desired system, default configurations has to be avoided (hardening). The system must be kept up to date, with all critical security related patches installed. All patches must be validated before deploying the system for production.

Virtualization is the process of using special software on a physical machine in order to create virtual machines. The special software is called a hypervisor. A hypervisor is a process that separates a computer's OS and applications from the underlying physical hardware.

The virtualization alternatives include:

- Application virtualization: allows applications written for one environment to execute on some other OS.
- Full virtualization: multiple full OS instances execute in parallel.
- Virtual machine monitor (VMM): Hypervisor, which coordinates access between each of the guests and the actual physical hardware resources.