اونيۏرسيتي مليسيا فهڠ السلطان عبدالله
UNIVERSITI MALAYSIA PAHANG
AL-SULTAN ABDULLAH

Data & Network Security

Chapter 7 - Principles of Secure Design

Outline

7.1 Introduction
7.2 Weakest link
7.3 Defence in depth
7.4 Fail-safe defaults
7.5 Translation and execution
7.6 Open design
7.7 Least privilege and isolation
7.8 Economy of mechanism
7.9 Security composability
7.10 Prevention, detection, and deterrence

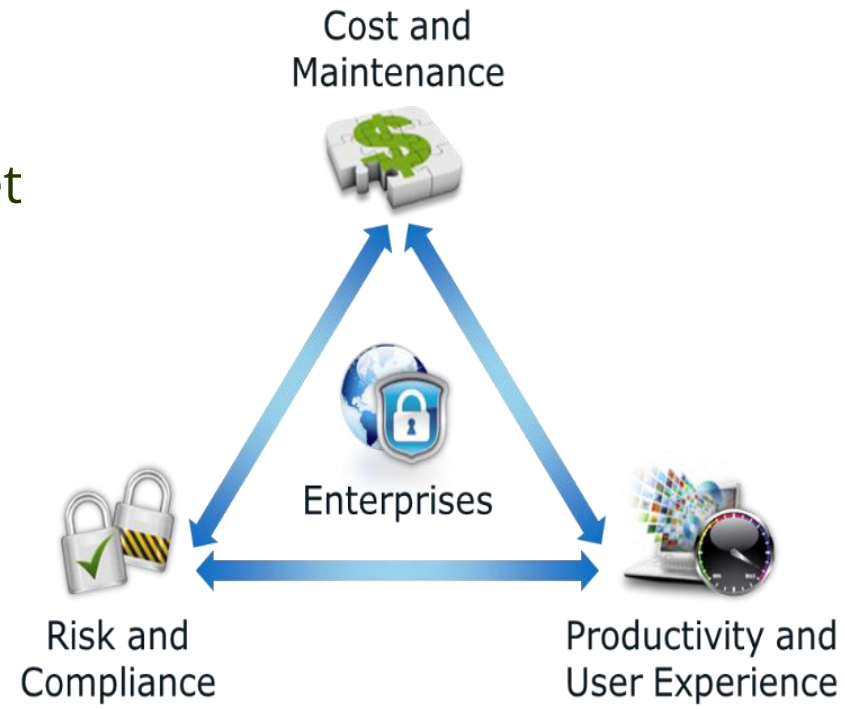
Learning Outcome

At the end of this chapter students are able to:

- Understand the principles of secure design.
- Explain defence in depth, fail-safe, security design, and least privilege and isolation.
- Explain the open design.
- Understand the concept of economy of mechanism.

7.1 Introduction

- Security is about
  - Trade-offs
  - Conflicting engineering criteria
  - Conflicting requirements
  - Human, technology and market
- Simplicity
  - Less to go wrong, less to check
  - Fewer possible inconsistencies
  - Easy to understand
- Restriction
  - Minimize access/interactions
  - Inhibit communication



7.2 Weakest link

- Attackers go after the weakest point in a system, and the weakest point is rarely a security feature or function.
- Secure design - consider the weakest link in your system and ensure it is secure enough.

- Humans are often considered the “weakest link” in the security chain.
- They ignore security policies and stick passwords to their computer monitors.
- They can be coerced, blackmailed, or “socially engineered” (=tricked).
- Even security experts are vulnerable to phishing.
- But “the user is not the enemy”.

7.3 Defence in depth

- The idea behind the defence in depth approach is to defend a system against any attack using several independent methods.

- It is a layering tactic, conceived by the US National Security Agency (NSA) as a comprehensive approach to information and electronic security.

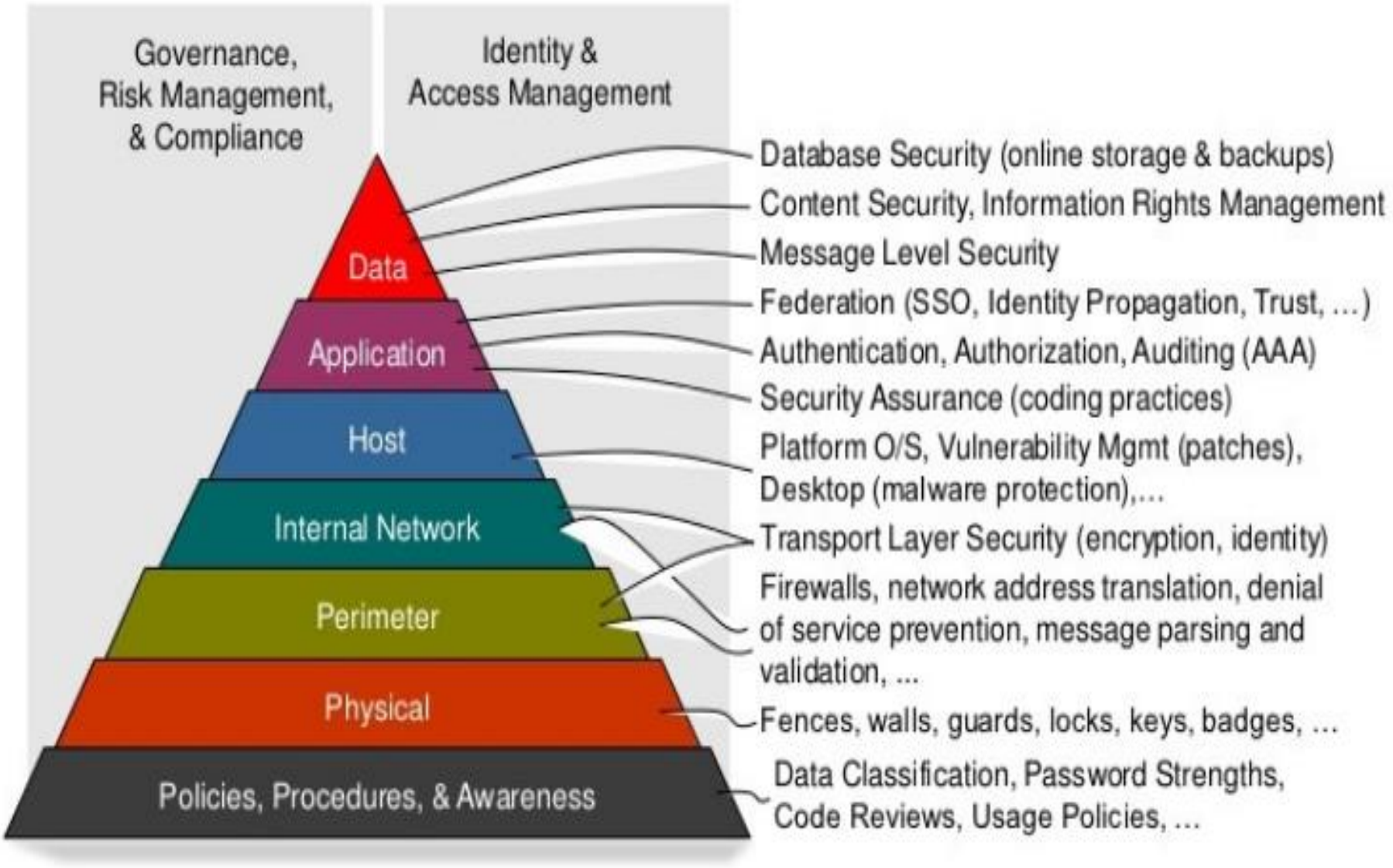
- Also called the "belt and braces" approach.

7.3 Defence in depth: Redundancy and layering

- Redundancy and layering are usually a good thing in security. Don't count on a firewall to block all malicious traffic; use an intrusion detection system as well. If designing an application, prevent single points of failure with security redundancies and layers of defence.

- The idea behind defence in depth is to manage risk with diverse defensive strategies so that if one layer of defence turns out to be inadequate, another layer of defence will hopefully prevent a full breach.

7.3 Defence in depth



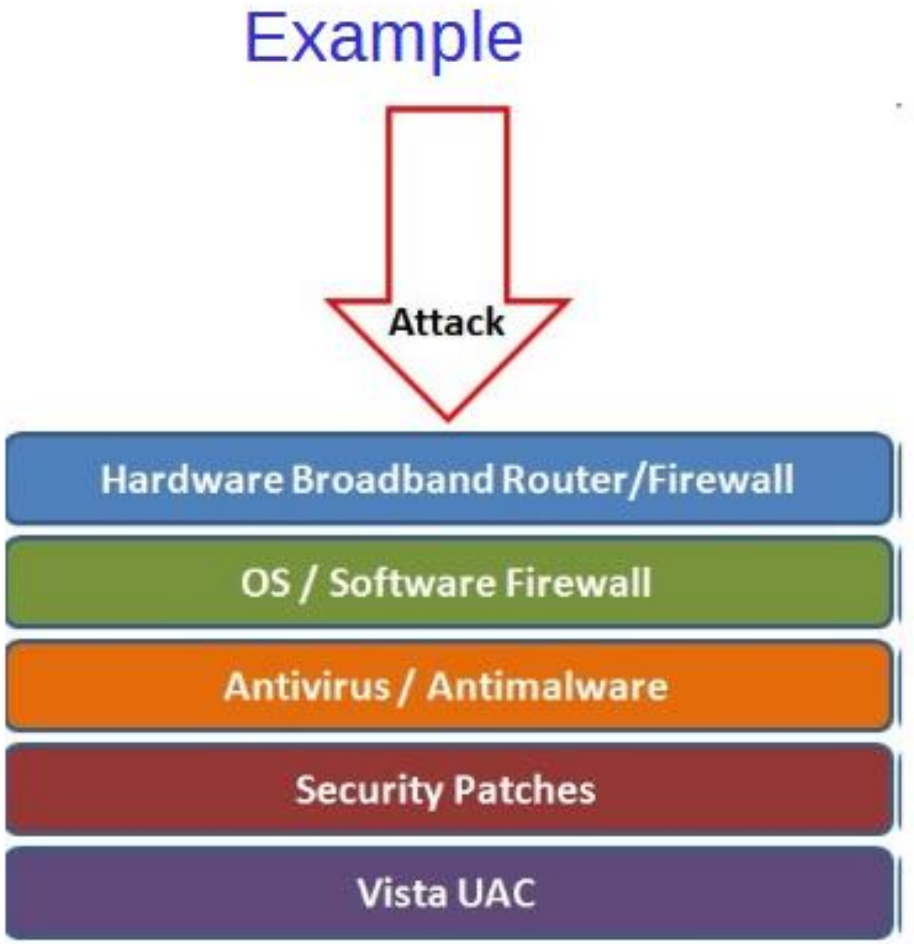
7.3 Defence in depth: Example

- Anti-virus software
- Authentication and password security
- Biometrics
- Demilitarized zones (DMZ)
- Encryption
- Firewalls (hardware or software)
- Hashing passwords
- Intrusion detection systems
- (IDS)
- Logging and auditing
- Multi-factor authentication
- Vulnerability scanners
- Physical security (e.g. deadbolt locks)
- Timed access control
- Internet Security Awareness
- Training
- Virtual private network (VPN)
- Sandboxing
- Intrusion Protection System
- (IPS)

Military: Defence in Depth

Layers

- Computer systems have multiple layers,
- e.g.
  - HW components
  - Chipset/MB
  - OS
  - TCP/IP stack
  - HTTP application
  - Secure http layer
  - Java script
  - User/smart card interface



Brainstorming defence in depth

- Helps to think about these three aspects:
  - People
  - Technology
  - Operations

- Example concerns
  - The attacker may get my users' passwords in-class exercise.
  - The attacker may cause his code to run on the servers.

Example: The attacker may get my users' passwords

- P: Education against social engineering and phishing.
- P: Education about choosing good passwords.

- T: Defend password file using a firewall.
- T: Hash/encrypt/... password file.
- T: Put additional security if not from a familiar IP address.
- T: Use one-time passwords.

- O: Log guessing attempts.
- O: Rate-limit guesses.

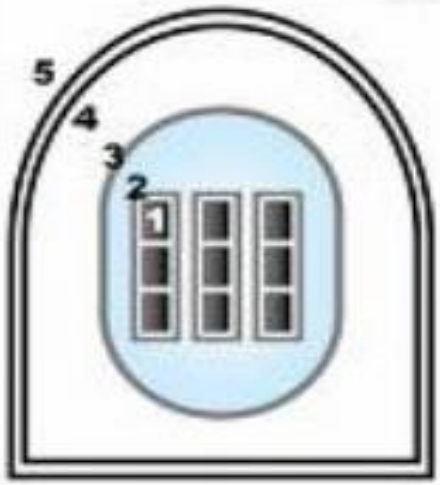
Exercise: Against virus attack

Identify how to overcome the virus attack by considering the three aspects (P,T,O).

Security can be like a chain:

or, <u>better</u>

Security can be layered



7.4 Fail defence - Fail-safe defaults

- Fail defence can be implemented in two ways; Fail-safe and fail-secure.
- Default to denied access.
- Fail-secure and fail-safe may suggest different outcomes.
  - For example: Physical, if a building catches fire, fail-safe systems would unlock doors to ensure quick escape and allow firefighters inside, while fail-secure would lock doors to prevent unauthorized access to the building.
- Programmers should check return values for exceptions/failures.
- Base access decisions on permission rather than exclusion. This principle means that the default situation is a lack of access, and the protection scheme identifies conditions under which access is permitted.
  - The alternative, in which mechanisms attempt to identify conditions under which access should be refused, presents the wrong psychological base for secure system design.

Fail-secure

- A fail-secure system is one that, in the event of a specific type of failure, responds in a way such that access, or data are denied.
- Default is to deny access.
- Data is secure, but availability is not preserved.
- In this case when the system is not secure, automatically the system will shut down to ensure the data can be protected.

Fail-safe

- In the event of failure, that causes no harm, or at least a minimum of harm, to other systems or personnel, the system should give access.
- Default is to permit access.
- Availability is preserved, but data may not be secured.
- Even there is an error occurs, the system may work as usual and maybe the data is not secured.

Fail-insecure designs

- Interrupted boot of OS drops the user into a root shell.
- Browser unable to validate HTTPS certificate.
  - Tells user, allows user to click-through to proceed anyway.
- Point-of-sale terminal unable to contact the card-issuing bank.
  - Allows the transaction if less than a certain threshold.
  - Risk management overrides security principles.

Why we often have fail-insecure

- Fail-insecure is often introduced by a desire to support legacy (insecure) versions.
  - E.g. TLS allows old clients to use weak crypto keys.

- Sometimes introduced because it may be easier to work with a short “blacklist” than with a long “whitelist”.
  - List of known phishing websites.
  - List of known malicious users.

The language design problem

- Language design is difficult, and success is hard to predict:
  - Pascal a success, Modula-2 (successor to Pascal – fixed syntactic ambiguity and added modules) a failure
  - Algol60 a success, Algol68 a failure
  - FORTRAN a success, PL/I a failure
- Conflicting advice

Efficiency

- The “first” goal (FORTRAN): execution efficiency.
- Still an important goal in some settings (C++, C).
- Many other criteria can be interpreted from the point of view of a type of efficiency:
  - programming efficiency: writability or expressiveness (ability to express complex processes and structures)
  - maintenance efficiency: readability.

Other kinds of efficiency

- efficiency of execution (optimizable)
- efficiency of translation. Are there features which are extremely difficult to check at compile time (or even run time)? e.g. Algol68 – solved dangling else problem (case/esac if/fi, do/od closing)
- Implementability (cost of writing translator)

Language features that aid efficiency of execution

- Static data types allow efficient allocation and access.
- Manual memory management avoids overhead of “garbage collection”.
- Simple semantics allow for simple structure of running programs (class variables, run time inheritance type checking - expensive).
- Expensive operations – array size increase

Reliability and Programming Language?

Hurts

- bad syntax source of bugs
  (if a=b when mean if a==b)
- complex syntax distracts the programmer (errors are more likely)
- Name hiding - confusion
- machine dependencies (length of Integer could cause different results when ported)

Helps

- rigorous type checking
- eliminate undefined variables (always init)
- runtime checks (array bounds, types)
- simple syntax
- less implicit computation (force explicit casts)
- compiler helps (check for declared but not used)
- correct return value types

Language Design Features

- Efficiency
  -execution
  -translation
  -implementation
- Reliability
- Readability
- Writability
- Expressiveness
- Regularity
  -generality,
  -orthogonality
  -uniformity
- Simplicity
- Extensibility
- Security
- Preciseness

Open design

- Security of a mechanism should not depend upon secrecy of its design or implementation (why not?)
  - Secrecy != security
  - Complexity != security
  - “Security through obscurity”
  - Cryptography and openness

Least privilege and isolation

- The principle of least privilege
  - also known as the principle of minimal privilege or the principle of least authority
- requires that in a particular abstraction layer of a computing environment, every module (such as a process, a user, or a program, depending on the subject) must be able to access only the information and resources that are necessary for its legitimate purpose.

Least privilege: rationale

- When code is limited in the system-wide actions it may perform, vulnerabilities in one application cannot be used to exploit the rest of the system.
  - For example, Microsoft states “Running in standard user mode gives customers increased protection against inadvertent system-level damage caused by malware, such as root kits, spyware, and undetected viruses”

Least privilege: example

- A user account gets only those privileges which are essential to that user's work.
  - A backup user does not need to install software: hence, the backup user has rights only to run backup and backup-related applications. Any other privileges, such as installing new software, are blocked.

Least privilege: example

- In UNIX systems, root privileges are necessary to bind a program to a port number less than 1024.
  - For example, to run a mail server on port 25, the traditional SMTP port, a program needs the privileges of the root user.
- Once set up, the program should relinquish its root privileges, not continue running as root.
- A large problem with many e-mail and other servers is that they don't give up their root permissions once they grab the mail port (Sendmail is a classic example).

Why fail to assign least privilege

- It's an effort to figure out what the least privilege needed actually is.
- The lazy (or overworked) designer or programmer will often assign the max privilege, because that's easy.
- "If we don't run as admin, stuff breaks"
- It's very hard to design systems that don't have some root/sysadmin that has all the privileges
  - E.g., Google employees getting fired for reading users gmail

Separation of Privileges

- Design your system with many different privileges
  - “file access” is not one privilege, but many“
  - network access”: same thing
  - Reading vs writing
- Split system into pieces, each with limited privileges.
- **Implementation in software engineering:**
- Have computer program fork into two processes.
  - The main program drops privileges (e.g. dropping root under Unix).
  - The smaller program keeps privileges in order to perform a certain task.
  - The two halves then communicate via a socket pair.
- **Benefits:**
  - A successful attack against the larger program will gain minimal access.
    - even though the pair of programs will perform privileged operations.

Related: Segregation of Duties

Achieved by (closely related):

- Principle of Functional separation:
- Several people should cooperate. Examples:
  - one developer should not work alone on a critical application,
  - the tester should not be the same person as the developer
  - If two or more steps are required to perform a critical function, at least two different people should perform them, etc.

  This principle makes it very hard for one person to compromise the security, on purpose of inadvertently.

- Principle of Dual Control:
  - Example 1: in the SWIFT banking data management system there are two security officers: left security officer and right security officer. Both must cooperate to allow certain operations.
  - Example 2: nuclear devices command.
  - Example 3: cryptographic secret sharin

Economy of mechanism

- Complexity is the enemy of security

- It’s just too easy to screw things up in a complicated system, both from a design perspective and from an implementation perspective.

- Keep it simple, stupid.
  - A design principle noted by the U.S. Navy in 1960.
- Minimalistic design.
  - Developers may create user interfaces made to be as simple as possible by eliminating buttons and dialog boxes that may potentially confuse the user.
- Worse is better.
  - Quality does not necessarily increase with functionality. Software that is limited, but simple to use, may be more appealing to the user/market.
- "You ain't gonna need it" (YAGNI)
  - A principle of extreme programming, says that a programmer should add functionality only when actually needed, rather than when you just foresee that you need them.

Why things get complex

- Desire for features; mission-creep
- New technologies, new possibilities
  - Stretches the designs we already have
  - Causes interfaces to get used in ways not intended
  - Causes new stuff to be layered on top of old stuff
- Desire for legacy compatibility
  - Desire to keep things interoperable

Why complexity leads to insecurity

- Cognitive overload
  - Designer can't keep all the possibilities in her head at once.
  - Security analyst can't reason about all the access possibilities

Security composability

- Security composability can be look as
  - Authenticate requests
  - Control access
  - Assume secrets not safe

Authenticate requests

- Be reluctant to trust
  - Assume that the environment where your system operates is hostile. Don’t let just anyone call your API, and certainly don’t let just anyone gain access to your secrets!
  - If you rely on a cloud component, put in some checks to make sure that it has not been spoofed or otherwise compromised.
  - Anticipate attacks such as command-injection, cross-site scripting, and so on.
  - Eg: Least Privilege

Authenticate requests: Trust

- A trusted system [paradoxical definition]: one that can break the security policy.
  - Trustworthy system: one that won't fail us.
- What is trusted and not trustworthy?
- What is not trusted, but is trustworthy?
- Definitions:
  - A trusted system [paradoxical definition]: one that can break the security policy.
  - Trustworthy system: one that won't fail us.

- Eg: An employee who is selling secrets
  is trusted and NOT trustworthy at the same time.

- Eg: Suppose Dropbox is trustworthy, and yet Alice encrypts her files
  before putting them there. Then, for her, Dropbox is trustworthy but not trusted.

Authenticate requests: Be Trustworthy

- Public scrutiny promotes trust.
- Commitment to security **is** visible in the long run.
  - even if the customers will not switch they will refuse to give you a rise (buy a more expensive version of the product).

Control access

- Mediate completely --
  - Every access and every object should be checked, every time. Make sure your access control system is thorough and designed to work in the multi-threaded world we all inhabit today.
  - Make sure that if permissions change on the fly in your system, that access is systematically rechecked. Don't cache results that grant authority or wield authority.
  - In a world where massively distributed systems are pervasive and machines with multiple processors are the norm, this principle can be tricky to implement.

Assume secrets not safe

- Assume your secrets are not safe --
  - Security is not obscurity, especially when it comes to secrets stored in your code. Assume that an attacker will find out about as much about your system as a power user, and more.
  - The attacker's toolkit includes decompilers, disassemblers, and any number of analysis tools. Expect them to be aimed at your system.
  - Finding a crypto key in binary code is easy. An entropy sweep can make it stick out like a sore thumb.

Prevention, detection, and deterrence

- Prevention, detection and deterrence are the action should be taken to protect, solve the security in organization.
- Can be do as below:
  - Make security usable
  - Promote privacy
  - Audit and monitor
  - Proportionality principle

Make security usable

- If your security mechanisms are too annoying and painful, your users will go to great length to circumvent or avoid them. Make sure that your security system is as secure as it needs to be, but no more.
- If you affect usability too deeply, nobody will use your stuff, no matter how secure it is. Then it will be very secure, and very near useless.

Make security usable: Be Even More Friendly

- Don’t annoy people.
  - Minimize the number of clicks
  - Minimize the number of things to remember
  - Make security easy to understand and self-explanatory
  - Security should NOT impact users that obey the rules.
- Established defaults should be reasonable.
  - People should not feel trapped.
- It should be easy to
  - Restrict access
  - Give access
  - Personalize settings

Promote privacy

- Collect only the user *personally identifiable information* (PII) you need for your specific purpose
- Store it securely, limit access
- Delete it once your purpose is done

Audit and monitor

- Record what actions took place and who performed them.
  - This contributes to both disaster recovery (business continuity) and accountability.

Proportionality Principle

- Reminder: security isn't the only thing we want.

Maximize security???

-vs-

Maximize “utility” (the benefits)

while limiting risk

to an acceptable level

within reasonable cost

commensurate with attacker's resources

Summary

- There are no specific action to be implemented.
- We need to consider about security means
  - Security is about trade-offs.
  - Conflicting engineering criteria....
  - Conflicting requirements...
- The goal is to have enough security to thwart the attackers we care about.

References

https://www.cs.bham.ac.uk/~mdr/teaching/dss16/02-principles.pdf

digital.cs.usu.edu/~allanv/cs4700/Topic3.pptx