Properties of secure software

- Confidentiality
- Integrity
- Availability
- Non-repudiation
- Authentication

Confidentiality

- The ability of a system to ensure that an asset is viewed only by authorized parties
- Sensitive information is not leaked to unauthorized parties
- Privacy for individuals, confidentiality for data

- **Examples**
  - Confidentiality: exam assignments should not be published (at least not until the exam is over)
  - Privacy: grades should only be visible to instructor and student involved

Integrity

- The ability of a system to ensure that an asset is modified only by authorized parties
- Sensitive information is not damaged by unauthorized parties

- **Examples**
  - Submissions are not edited by anyone other than student
  - Submissions are not edited by anyone after the deadline
  - Grades are determined only by instructor or auto-grader
  - Assignments, submissions, grades are not removed / only by instructor

Availability

- The ability of a system to ensure that an asset can be used by any authorized parties
- A system is responsive to requests

- **Example**
  - Should execute program tests fast (in reasonable time)
  - Should be responsive when editing text/code

Nonrepudiation (accountability)

- The ability of a system to confirm that a sender cannot convincingly deny having sent something
- Note: opposite of privacy/anonymity; requires a balance

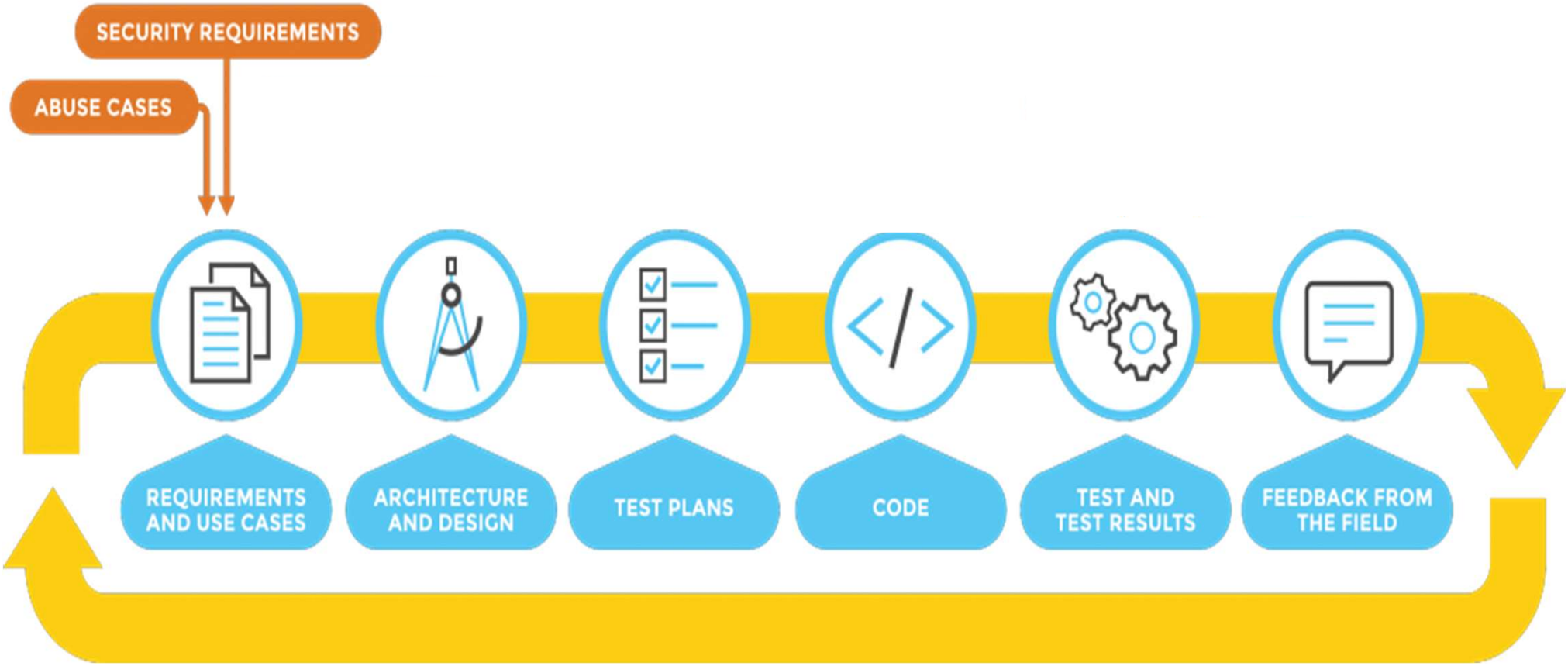
- **Examples**
  - Student cannot deny to have edited submission after the deadline

Authentication

- The ability to verify the identity of an individual or entity.
- Verify the identity of a person (or other external agent) making a request of a computer system

- **Example**
  - Should be able to determine if the user is indeed a student or an instructor

Secure Software Development Life Cycle (SSDLC)



Core principles

- **Identification:** “is how a user tells a system who he or she is (for example, by using a username or User ID);

- **Authentication:** “is the process of verifying a user's claimed identity (for example, by comparing an entered password to the password stored on a system for a given username).”;

- **Authorization:** “defines a user's rights and permissions on a system. After a user (or process) is authenticated, authorization determines what that user can do on the system.”;

- **Auditing:** “an evaluation of an organization, system, process, project or product”.

Excuses from practitioners

- *"No one will do that!"*
- *"Why would anyone do that?"*
- *"We've never been attacked."*
- *"We're secure, we use cryptography."*
- *"We're secure, we use ACLs."*
- *"We're secure, we use a firewall."*

More excuses

- “We've reviewed the code, and there are no security bugs.”
- “We know it's the default, but the administrator can turn it off.”
- “If we don't run as administrator, stuff breaks.”
- “But we'll slip the schedule.”
- “It's not exploitable.”
- “But that's the way we've always done it.”
- “If only we had better tools….”

Good and Bad Practices

Good practices: Input validation

- **Fields length** and **buffers** bound checking
- **Validate** input not only on client-side but on **server-side** environment too;
- Use "**preparedStatement**()" in **Java** and similar functions in other languages to avoid ***SQL Injection*** attacks;
- Possibly use **high level virtualized languages such Java, C#**;
- Low level languages like C and C++ are more exposed to buffer overflow exploits;

Good practices: Confidentiality

- Use **Public Key Cryptography** to do effective encryption;
- Encrypt and sign passwords with **PGP**, **GnuPG**, **RSA** or other encryption tools; store them in a secure place;
- **Zero memory stored passwords** after the use;
- Use a **well known encryption algorithm**: security is granted by the key and the well-known algorithm;
- Use **well known secure protocols** to implement channel encryption;
- Create **secure temporary files**;

Good practices: Integrity

- Use str**ong passwords but not too complex**: every password must be at least eight characters length (upper and lower case, number and special characters); passwords haven't to be too complex to avoid user writing down passwords everywhere!
- **Identification & Authentication** have to be done over **encrypted channels**;
- Adopt **well known access control policy**: DAC, MAC or RBAC;
- Do **not use applets or ActiveX** in Web application: user could be constrained to activate ActiveX or Applet execution in the Web Browser exposing the browser to malicious components.

Good practices: Activities

- **Documenting** security policies adopted by your software;
- Plan periodic **independent reviews**;
- Use **Checklists** to do security **tests**;
- **Comment your code**, this can help the security reviewer and tester;

Bad practices

- Write **passwords everywhere** or **say** them to everyone:
  - Social Engineering is very diffuse; memorize your passwords or encrypt them;
- Create administration **backdoor** in your applications:
  - create an “administrator” user with high privileges instead;
- “**Security through obscurity**”:
  - use well known security algorithm and secure keys;
- “**Retrofit**” security:
  - secure your software with SSDLC;

More bad practices

- **Think that software security is network security**!
  - Many security problems become from OS C/C++ programming buffer overflow problems:
- Think that **third party software is secure**:
  - it isn't true, check them;
- Think that **random functions are true Random**!:
  - Random is only in nature; in a computer world all functions are pseudorandom;
- **"Hard-code" password** in your software:
  - use asymmetric cryptography;
- Don't check "**cut & paste**" **code**:
  - analyse the code first!
- Think that **attackers come from the outside**:
  - Most attack activities are inside in the enterprise;