Mestrado em Engenharia Informática (MEI)
Mestrado Integrado em Engenharia Informática (MiEI)

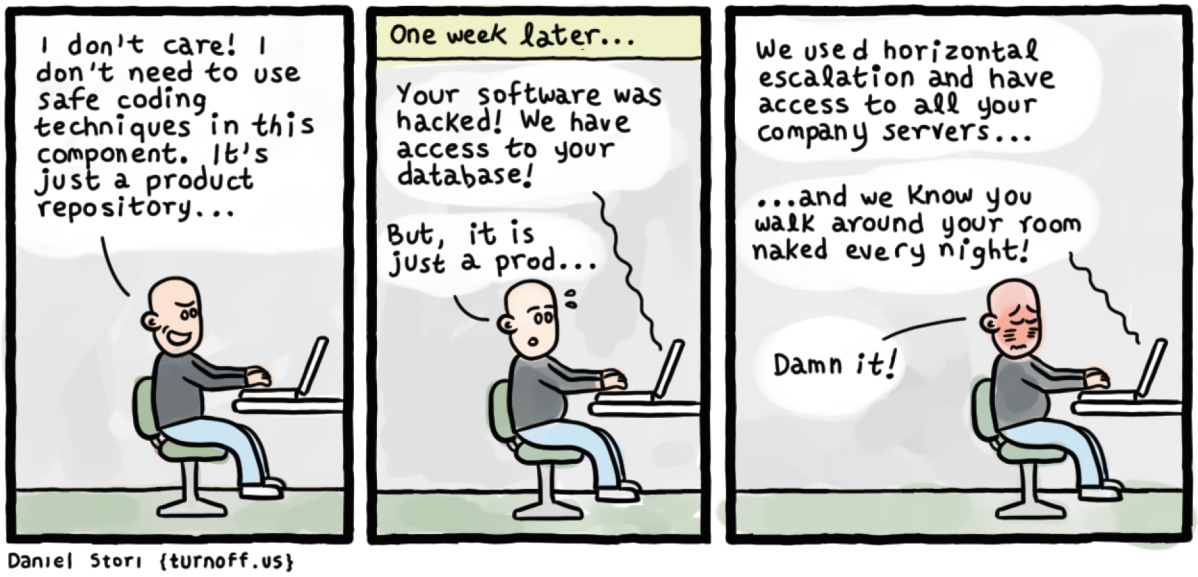
Perfil de Especialização **CSI** : Criptografia e Segurança da Informação

Engenharia de Segurança

jose.miranda@devisefutures.com

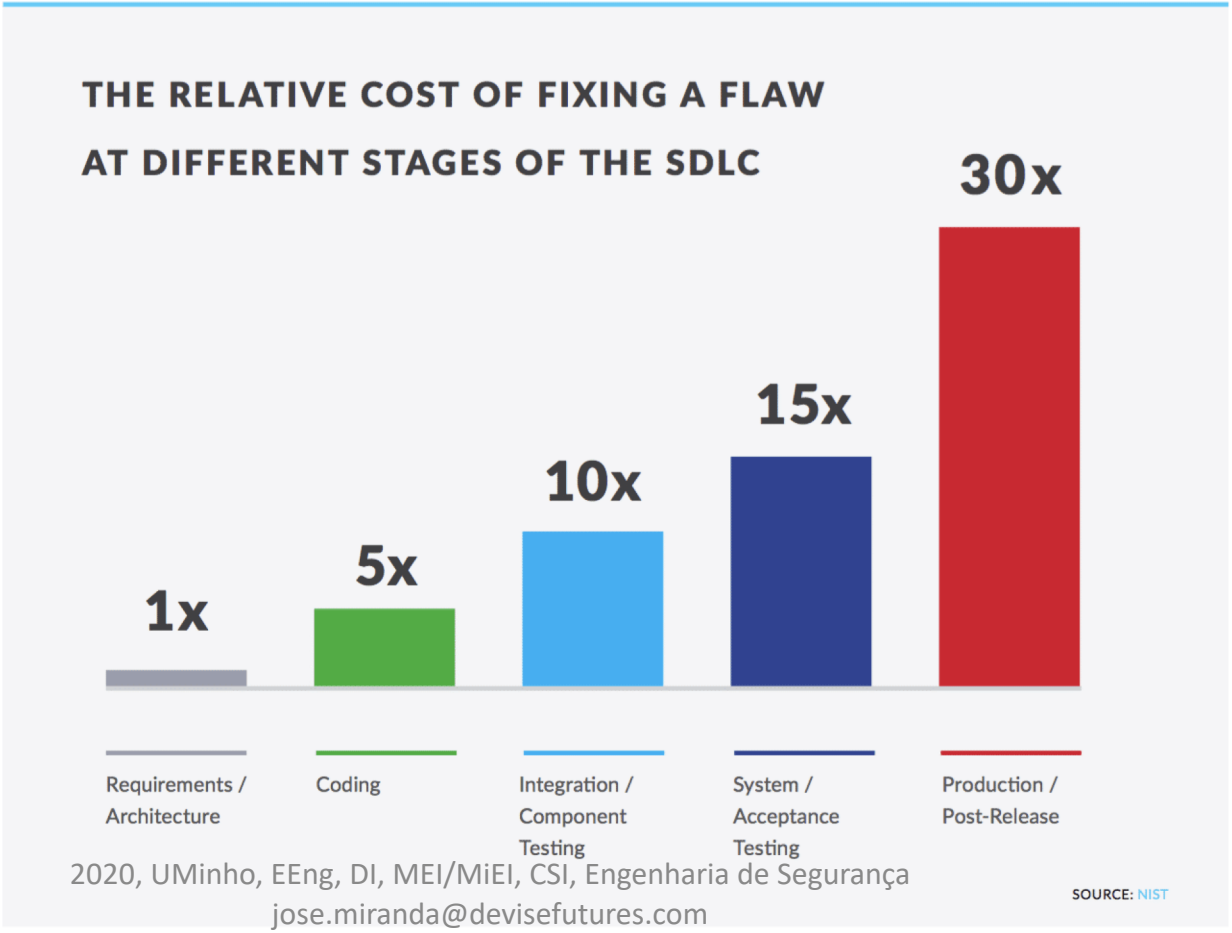
Topics

- Principles of software design
- Avoid Predominant Project Vulnerabilities



Project Vulnerabilities

- As seen in previous class, **project vulnerabilities** are introduced during the software design phase (requirements and design).



Principles of Software Design

- Principles of software design to follow in order to avoid the most common errors:
  - **Frugality** - keep the project as simple as possible, since complexity causes vulnerabilities. In the scope of security it applies, for example, to access control or input validation mechanisms;
  - **Decisão segura por omissão** – Utilizar escolhas por omissão à prova de falhas. Decisão por omissão é o "não acesso", sendo o acesso permitido apenas por decisão explícita. No âmbito da segurança, as aplicações deste princípio são evidentes, assim como os exemplos de violação deste princípio (sistemas de acesso sem senhas definidas por omissão, ou com senhas conhecidas; acesso a objetos/dados definidos por omissão e não para quem necessita de aceder);
  - **Secure decision by default** - Use fail-safe defaults. Decision by default is "no access", being access allowed only by explicit decision. In the context of security, the applications of this principle are evident, as are examples of violation of this principle (system access passwords defined by default or with known passwords; access to objects/data defined by default and not for those who need access);
  - **Complete mediation** - the authorization of all accesses to any object must be verified, i.e. it should not be possible to circumvent the access control;
  - **Open design** - security should not depend on design or design secrecy, since "security by obscurity" generates only an illusion of security;
  - **Separation of Privileges** - argues that a mechanism that has to be unlocked by two distinct keys is more robust and flexible than another that requires only one key. In security, it applies, for example in the case of authentication (multifactor authentication - something you know, something you have, something you are);

Principles of Software Design

- Principles of software design to follow in order to avoid the most common errors:
  - **Minimal privileges** (Trust with Reluctance) - Each process and each user must have the minimum "need to know" permissions to do what they need, in order to limit the damage caused by an accidental error or intrusion. In security, it is used, for example, to run Web/mail/... Servers with low privileges, instead of root/administrator;
  - **Minimal common mechanisms** - the dependence of processes and users on resources shared between them should be minimized, since anything shared (e.g., files or memory) is a path by which data (private or confidential) can be disclosed;
  - **Psychological Adequacy** - argues that the person-machine interface should be designed to be simple to use. This is a security usability theme, whose objective is for users to be able to correctly apply protection mechanisms, routinely and automatically;
  - **Secure by default** - software must be designed taking into account the possibility of intrusion. For example, unnecessary services must be turned off, and default settings should be conservative;
  - **Think security since the beginning of the software project**;

Principles of Software Design

- Principles of software design to follow in order to avoid the most common errors:
  - **Secure installation** - software should contain:
    - Tools and documentation that allow to be configured correctly, from a security point of view;
    - Security management and analysis tools that enable the administrator to define and configure the desired security level;
    - Tools for installation of updates and patches;
  - **Communication** - processes to respond to vulnerabilities must be defined, as well as the involvement of programmers with the user community to respond to security and vulnerability doubts;
  - **Defend in depth** - multiple levels or protection mechanisms (security by diversity), so that the software is not compromised even if it is misconfigured, is vulnerable or one of the protection levels / mechanisms fails;
  - **Protect the weakest link** - attackers tend to attack the weakest link of the system, which should therefore be the first concern to take into account. It should be noted that the weakest link of many systems are users (poorly aware to security issues);
  - **Fail securely** - when a flaw occurs, the system must stay in a safe state. That is, the error conditions must be handled very carefully.

Avoid Predominant Project Vulnerabilities

- How to Avoid Predominant Project Vulnerabilities[1]:
  1. **Never assume or trust without validating**
     - Designs that place authorization, access control, enforcement of security policy, or embedded sensitive data in client software thinking that it won't be discovered, modified, or exposed by clever users or malicious attackers are inherently weak. Such designs will often lead to compromises.
     - A rule of thumb in building secure software is to never trust on inputs. When untrusted clients send data to your system or perform a computation on its behalf, the data sent must be assumed to be compromised until proven otherwise. So, make sure all data received from an untrusted client are properly validated before processing.
     - If IP (Intellectual Property) or sensitive material must be stored or sent to the client, the system should be designed to be able to cope with potential compromise.
  2. **Use an authentication mechanism that cannot be bypassed or tampered with**
     - One goal of a secure design is to prevent an entity (user, attacker, or in general a "principal") from gaining access to a system or service without first authenticating. Once a user has been authenticated, a securely designed system should also prevent that user from changing identity without re-authentication. Authentication techniques require one or more factors such as: something you know (e.g., a password), something you are (e.g., biometrics such as fingerprints), or something you have (e.g., a smartphone).
     - Authentication encompasses more than just human-computer interaction; often, in large distributed systems, machines (and/or programs running on those machines) authenticate themselves to other machines.
     - The act of authentication results in the creation of a token representing a principal that is used throughout the system or service. If such tokens (or credentials) are deterministically derived from easy to-obtain information, such as a user name, then it becomes possible to forge identities, allowing users to impersonate other users. If an authentication system does not limit the lifetime of an authentication interaction, then it may inadvertently grant access to a user to whom it should not.

[1] Relatório IEEE – https://www.computer.org/cms/CYBSI/docs/Top-10-Flaws.pdf

Avoid Predominant Project Vulnerabilities

- How to Avoid Predominant Project Vulnerabilities[1]:

3. **Authorize after you authenticate**
   - Authorizing after authenticating is also important, since not all authenticated users must have access to all system operations.
   - Authorization should be conducted as an explicit check, and as necessary even after an initial authentication has been completed. Authorization depends not only on the privileges associated with an authenticated user, but also on the context of the request. The time of the request and the location of the requesting user may both need to be taken into account.
4. **Strictly separate data and control instructions, and never process control instructions received from untristed sources**
   - Co-mingling data and control instructions in a single entity, especially a string, can lead to injection vulnerabilities. Lack of strict separation between data and code often leads to untrusted data controlling the execution flow of a software system.
5. **Explicitly validate all data**
   - Software systems and components commonly make assumptions about data they operate on. It is important to explicitly ensure that such assumptions hold: vulnerabilities frequently arise from implicit assumptions about data, which can be exploited if an attacker can subvert and invalidate these assumptions.

[1] Relatório IEEE – https://www.computer.org/cms/CYBSI/docs/Top-10-Flaws.pdf

Avoid Predominant Project Vulnerabilities

- How to Avoid Predominant Project Vulnerabilities[1]:

6. Use criptography correctly

- Cryptography is one of the most important tools for building secure systems. Through the proper use of cryptography, one can ensure the confidentiality of data, protect data from unauthorized modification, and authenticate the source of data.
- Common pitfalls to avoid:
  - Rolling your own cryptographic algorithms or implementations, since designing a cryptographic algorithm (including protocols and modes) requires significant and rare mathematical skills and training, and even trained mathematicians sometimes produce algorithms that have subtle problems;
  - Misuse of libraries and algorithms, since for exemple, it may be necessary to choose an initialization vector (IV) with certain properties for the algorithm to work securely;
  - Poor key management, include hard-coding keys into software (often observed in embedded devices and application software), failure to allow for the revocation and/or rotation of keys, use of cryptographic keys that are weak (such as keys that are too short or that are predictable), and weak key distribution mechanisms;
  - Randomness that is not random, since cryptographic operations require random numbers that have strong security properties. In addition to obtaining numbers with strong cryptographic randomness properties, care must be taken not to re-use the random numbers.

7. Identify sensitive data and how they should be handled

- Data sensitivity is context-sensitive. It depends on many factors, including regulation (which is often mandatory), company policy, contractual obligations, and user expectation.
- Data processing should be the subject of a data protection policy that allows to define a unified approach that complies with all applicable regulations;
- Policy requirements and data sensitivity can change over time as the business climate evolves, as regulatory regimes change, as systems become increasingly interconnected, and as new data sources are incorporated into a system. Regularly revisiting and revising data protection policies and their design implications is essential.

[1] Relatório IEEE – https://www.computer.org/cms/CYBSI/docs/Top-10-Flaws.pdf

Avoid Predominant Project Vulnerabilities

- How to Avoid Predominant Project Vulnerabilities[1]:

8. Always consider the users

- The security stance of a software system is inextricably linked to what its users do with it. It is therefore very important that all security-related mechanisms are designed in a manner that makes it easy to deploy, configure, use, and update the system securely.
- Software designers and architects must consider how the physical abilities, cultural biases, habits, and idiosyncrasies of the **intended users** of the system will impact its overall security stance.
- The challenge to designers and architects lies in creating designs that:
  - Facilitate secure configuration and use by those interested in doing so,
  - Motivate and incentivize secure use among those not particularly interested in software security, and
  - Prevent or mitigate abuse from those who intend to weaken or compromise the system.

[1] Relatório IEEE – https://www.computer.org/cms/CYBSI/docs/Top-10-Flaws.pdf

Avoid Predominant Project Vulnerabilities

- How to Avoid Predominant Project Vulnerabilities[1]:

9. Understand how integrating external componentes changes your attack surface

- The decision to use-rather-than-build components to provide some or all of the functionality of the sowftare means that the software as a whole inherits the security weaknesses, security limitations, maintenance responsibility, and the threat model of whatever components you are including (binaries, libraries, source code, or APIs).
- This inheritance can amount to a deficit of security, which must be solved, mitigated, or accounted for when the system is finished. At a minimum, consider the following:
  - Isolate external components as much as your required functionality permits; use containers, sandboxes, and drop privileges before entering uncontrolled code.
  - When possible, configure external components to enable only the functionality you intend to use.
  - If you cannot configure the security properties of the component to align with your security goals, find another library, or document that you are accepting the risk and inform relevant stakeholders of your decision.
  - Validate the provenance and integrity of the external component, and consider maintaining a local mirror of the library's source.
  - Maintain an up-to-date list of consumed external components and at a pre-established cadence verify that it matches the versions included in your product, as well as that those are the latest known-secure versions available for each external component.
  - Whenever possible, authenticate the dataflow between your system and external components.
  - Be watchful to extra (undocumented) functionality/communications of external components.

[1] Relatório IEEE – https://www.computer.org/cms/CYBSI/docs/Top-10-Flaws.pdf

Avoid Predominant Project Vulnerabilities

- How to Avoid Predominant Project Vulnerabilities[1]:
  10. **Be flexible when considering future changes**
      - Software security must be designed for change, rather than being fragile, brittle, and static. Many changes occur over time, with possible impact on security: changes in the software itself, in the environments where the software runs, as well as changes in threats and software attacks.
      - Secure software design should be flexible to minimize the impact of future changes by adopting the following principles:
        - Design for secure updates – Have the system being upgraded verify the integrity and provenance of upgrade packages; make use of code signing and signed manifests to ensure that the system only consumes patches and updates of trusted origin.
        - Design with the ability to isolate or toggle functionality – It should be possible to turn off compromised parts of the system, or to turn on performance-affecting mitigations, should the need arise. Not every vulnerability identified can be readily mitigated within a safe time period, and mission-critical systems cannot simply be taken offline until their vulnerabilities are addressed.
        - Design for changes to objects intended to be kept secret – Keeping secrets safe is a hard problem, and one should be prepared to have secrets replaced at any time and at all levels of the system.
        - Design for changes in the security properties of components beyond your control – Good design allows for intermediate layers of abstraction between code and imported external APIs, so that developers can change components providing needed functionality without changing much of the code.
        - Design for changes to entitlements – Systems are sometimes designed in which support staffers have privileged access to certain parts of the system in order to perform their job. However, the support staff's access to various system components likely changes over time. The system must have a way to revoke access to areas when a user no longer has a need to access them. This revocation of access should be part of an existing auditing mechanism in which access to critical system components is regularly reviewed to confirm that those individuals with access still require that level of access.

[1] Relatório IEEE – https://www.computer.org/cms/CYBSI/docs/Top-10-Flaws.pdf