Secure Product Lifecycle

Requirements Management & Secure Design

Yagoba GmbH, Austria, www.yagoba.com, info@yagoba.com

Today‘s Agenda

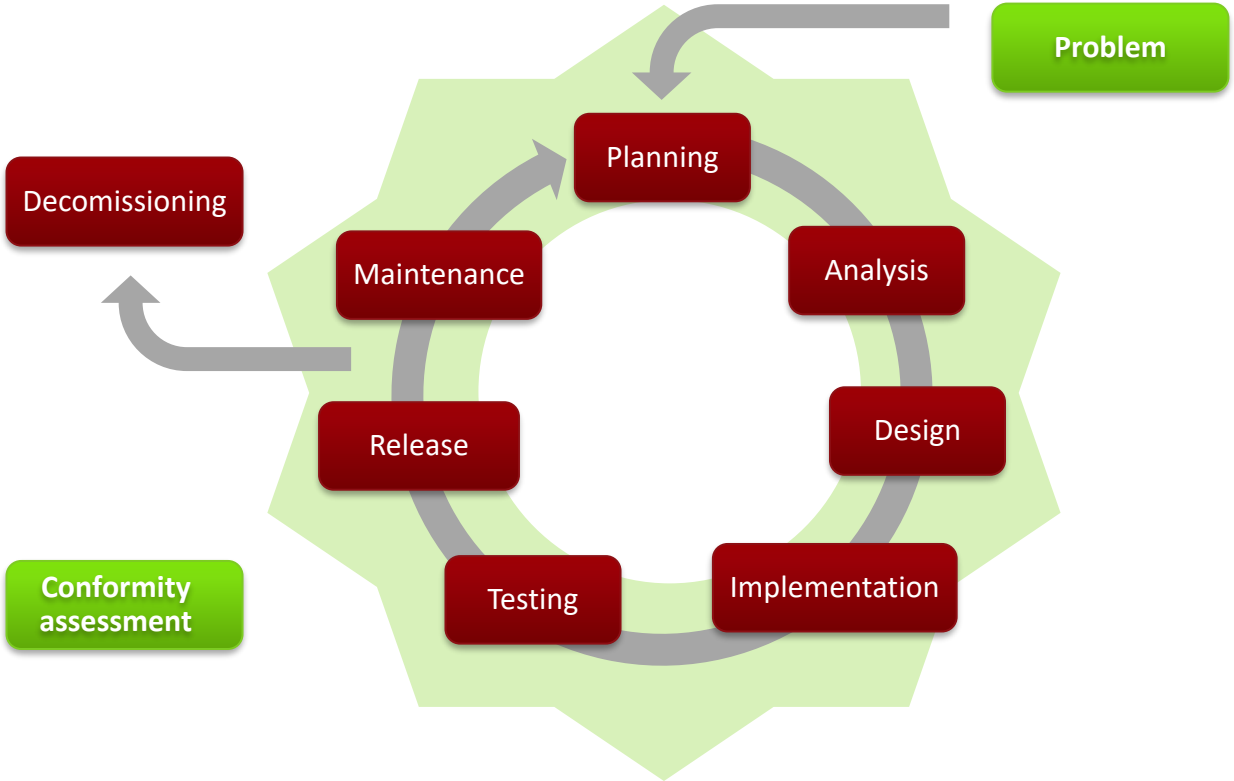
- Context within SPLC – Recap
- Planning
- Requirements Management
- Security by Design
- Security Mechanisms

Communication

- Karin Maier k.maier@yagoba.com
- Christoph Herbst christoph@yagoba.com

- Discord: #spl

Secure **Product** Lifecycle



PLANNING

Planning: Best Practices

- Misconceptions:
  - Doing the work is more important than planning
  - Plans are for managers
  - Plans are hypothetical, it gets changed anyway, save the effort…
- Least questions you should be asking:
  - Who is doing what and by when?
  - How do we handle change?
- Planning is about being “less” wrong and having a map how to navigate all the iterations of the secure product lifecycle

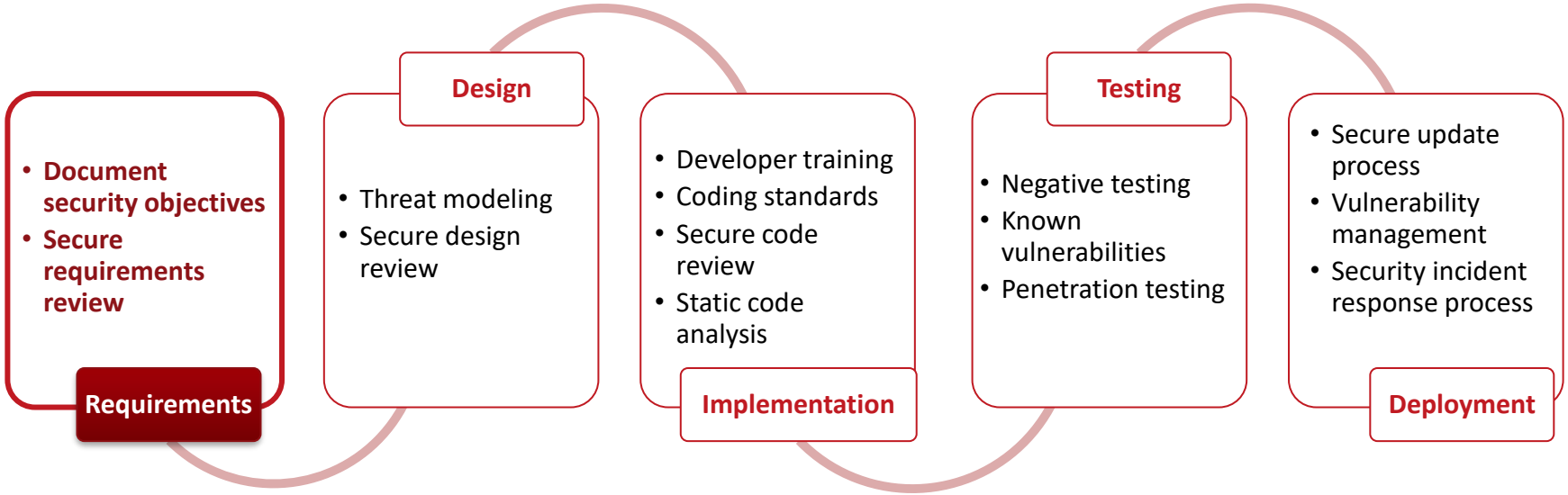
Recap

- Security should be considered from the ground up
- Include security in the software requirements
  - When defining what a system must do, also consider what a system must **not** do
- Include regulatory requirements
- Risk and threat analysis to understand
  - Business risks of successful exploits
  - Costs of liability, redevelopment, and damage to brand image and market share

Recap

- Document key security objectives
- Separate security requirements from functional requirements so explicit review and testing is possible
- For every use case, write misuse case (intentional misuse)
- Write requirements for industry standards & regulatory rule

Secure Development Lifecycle



Glossary

- Use cases
  - The use case has been an effective form of representing user requirements visually
  - use case = scenario (user story) + actors (who interacts with the system)
- Misuse cases
  - can help to represent security requirements visually from attackers' point of view
- Abuse cases
  - represent security requirements from a much stronger destruction aspect of the system

REQUIREMENT MANAGEMENT

Yagoba GmbH, Austria, www.yagoba.com, info@yagoba.com

It’s not a bug, it’s feature…no wait, it’s a bug

- The hardest part of building software is not coding …

… it's requirements

Types of software requirements

- Functional requirements (FR)
  - Describe specific system functions
- Non-Functional Requirements (NFR)
  - Define aspects like performance, security, usability, reliability, and scalability
- Further groupings/subtypes include e.g. domain specific requirements (e.g. given by governments, laws)

Examples

- *… Should calculate the sum.*
- *For every .. a valid keypair has to be derived.*
- *… at most 10 ms to authenticate…*
- *… for any given input return all available from the database.*
- *Only verified user is allowed to do …*
- *… should not exceed more than … kB on the secure element.*
- *… It should be easy to retrieve ..*
- *10.000 consecutive operations without error*

Examples

- *… Should calculate the sum.* FR
- *For every .. a valid keypair has to be derived.* FR
- *… at most 10 ms to authenticate…* NFR
- *… for any given input return all available from the database.* FR
- *Only verified user is allowed to do …* FR
- *… should not exceed more than … kB on the secure element.* NFR
- *… It should be easy to retrieve ..* NFR
- *10.000 consecutive operations without error* NFR

Requirements matter

- Requirement problems are the primary reason that projects
  - are significantly over budget and past schedule
  - have significantly reduced scope
  - deliver poor-quality applications that are little used once delivered
- One source of these problems is poorly expressed or analyzed quality requirements, such as security and privacy
- Difficult and expensive to significantly improve an application after it is in its operational environment

Security requirements

- Requirements are the starting point, responsible for any system, legal and contractual issues, governance, and provide full functional perspective of the system being developed
- Among all non-functional requirements security requirements are the most important ones
- Process for analyzing security requirements and then applying security techniques should be a systematic and an intuitive way

Security Quality Requirements Engineering (SQUARE)

- SQUARE is a nine-step process that helps build security, including privacy, into the early stages of the production lifecycle.

https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=484579

SQUARE –
Elicitation and Analysis Process

6	Elicit security requirements	Artifacts, risk assessment results, selected techniques	Joint Application Development (JAD), interviews, surveys, model-based analysis, checklists, lists of reusable requirements types, document reviews	Stakeholders facilitated by requirements engineer	Initial cut at security requirements
7	Categorize requirements as to level (system, software, etc.) and whether they are requirements or other kinds of constraints	Initial requirements, architecture	Work session using a standard set of categories	Requirements engineer, other specialists as needed	Categorized requirements
8	Prioritize requirements	Categorized requirements and risk assessment results	Prioritization methods such as Triage, Win-Win	Stakeholders facilitated by requirements engineer	Prioritized requirements
9	Requirements inspection	Prioritized requirements, candidate formal inspection technique	Inspection methods such as Fagan, peer reviews	Inspection team	Initial selected requirements, documentation of decision-making process and rationale

Requirement Engineering Tasks

- Requirements Engineering is a discipline on its own, which provides process, tools, techniques, modelling, cost estimation, project planning, and contractual agreements Extracting and defining requirements
- Eliciting and extracting requirements
- Prioritizing security requirements
- Risk assessment for security requirements

Requirement Management Tasks

- Managing security requirements throughout the life cycle
- Risk assessment for security requirements

Followed by

- Design and implement security requirements
- Trace security requirements

Requirements Traceability Management

- We need to identify, analyze, and incorporate security requirements as part of the functional requirements process
- Requirements traceability can be defined as the ability to describe and follow the lifespan of a requirement in both a forwards and backwards direction

Security Requirements Traceability Matrix (SRTM)

- SRTM is a grid that supplies documentation and a straightforward presentation of the required elements for security of a system
- SRTM assures accountability for all processes and completion of all work
- SRTM not only for technological but also for conformity assessment reasons
- You get more than one use out of it (e.g. defect analysis)

SRTM Example

Req ID	Req Desc	Type/Stakeholder	Prio (1 High – 5 Low)	…	TestCase ID/Name
R1	The calculation must be faster than 5 ms.	NFR (Quality Department)	3		T203; T204;
R2	The sum of is caluclated correctly	FR	1		T40
R3	…	…	…	…	…

SECURE DESIGN

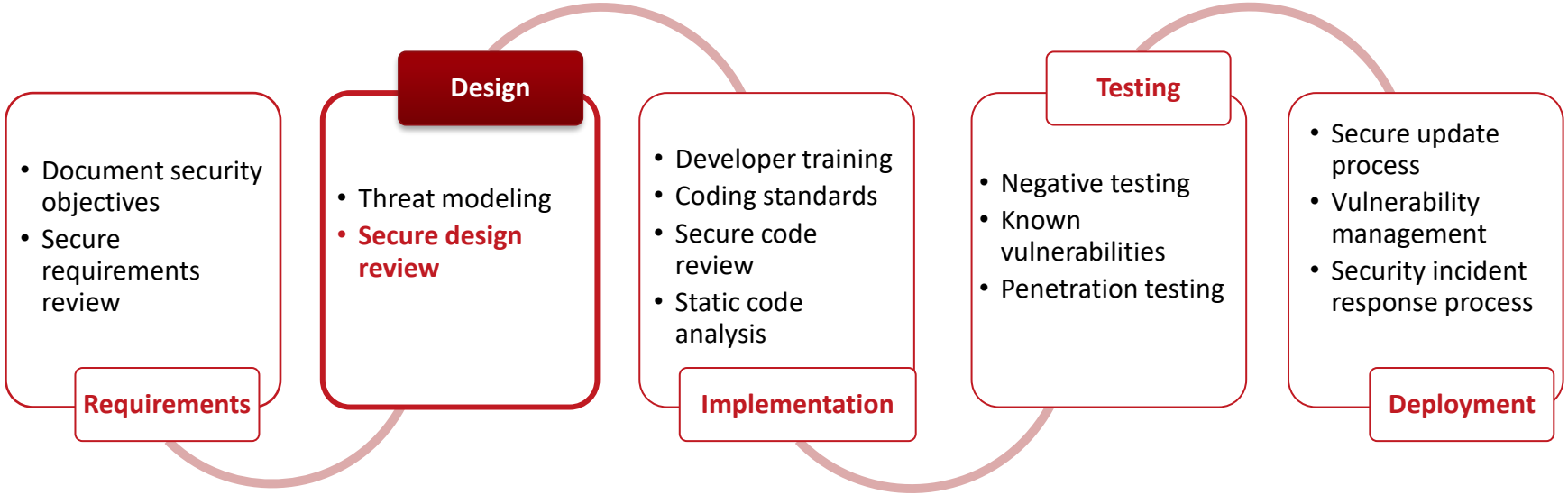
Glossary

- Software Design aims to help transform requirements into implementation
- Difference between Software Design and Software Architecture
  - Software Design focus is more on a module/component/class level
  - Software Architecture focus is more on the design of the entire system

- Architecture is "what" we're building; design is "how" we're building
- Architecture is strategic, while design is tactical

- Therefore for the SDLC leads us to Secure Design and Security Architecture

Secure Development Lifecycle



Meaning of Security by Design

- In a broad sense it is a systematically organized framework for whole product life cycle
- **Make security an integral part of design phase of product development**
- **Design your product with the fact in mind that it will be attacked**
- Integrate security controls from the beginning
- Make security tests a critical part of development process

Goals

- Define threats to a system in a detail that allows developers to understand and code against
- Provide system architecture mitigating as many threats as possible
- Design techniques that force developers to consider security with every line of code
- Enforce necessary authentication, authorization, confidentiality, data integrity, privacy, accountability, availability, safety & non-repudiation requirements
- Design a robust security architecture
- Preserve (implemented) architecture during software evolution
- **Take malicious practices for granted**

Steps

- Perform threat analysis
- Identify design techniques that mitigate risks
- Identify components essential to security
- Build security test plan
- Plan for incident response

Minimize Attack Surface

- SW attack surfaces: weaknesses associated with code
  - Missing coding guidelines
  - Unnecessary code, untested code
  - 3rd party SW
- Network attack surfaces: weaknesses associated w/ networking components
  - Ports, protocols, channels, devices & their interfaces
  - Cloud servers, data, systems, and processes
- Human attack surfaces: exploit directed at humans

Attack Surfaces

Examples

- Open ports on outward-facing web servers
- Services available inside firewall perimeter
- Code that processes incoming data, email, XML & office documents
- An employee with access to sensitive information is socially engineered

Minimize Attack Surface

- Reduce area & exposure of attack surface
  - Apply principles of least privilege & least functionality
  - Deprecate unsafe functions
  - Eliminate Application Programming Interfaces (APIs) vulnerable to cyberattacks
- Reduce accessibility of attack surface
  - Limit amount of time adversaries have (i.e., the window of opportunity) to initiate & complete cyberattacks

Attack Surfaces

Principle of least privilege

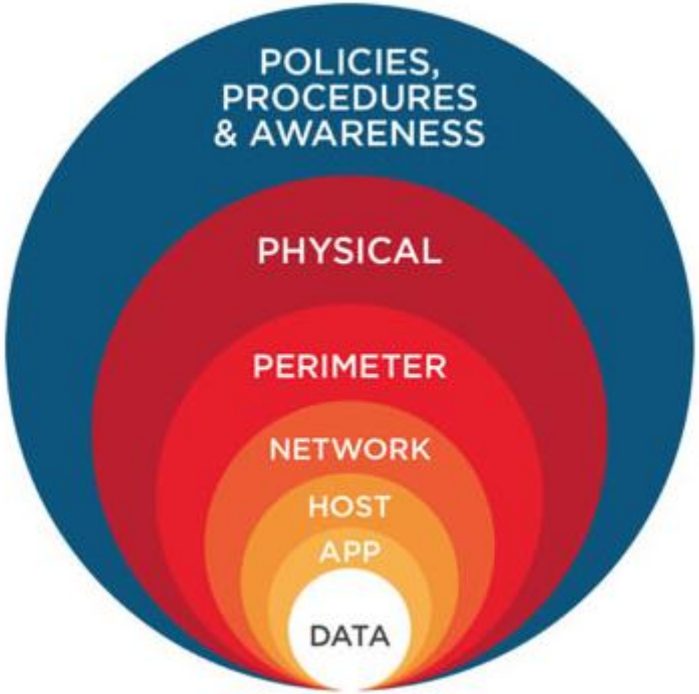
- SW processes & their authorized users shall be granted only those privileges required for them to carry out their specified function(s)

Principle of least functionality

- Disabling/uninstalling unused/unnecessary functionality, protocols, ports & services
- Limiting installable SW & functionality of that SW

Defense in Depth

- Multiple layers of security controls in place
- If one mechanism fails, another will already be in place



Ensure Confidentiality

- Encrypt sensitive data
- Use standardized algorithms

Ensure Integrity

- Use secure boot
- Design secure update process
- Verify integrity

Ensure authenticity & non-repudiation

- Use standardized protocols & algorithms

Secure Communication

- Never communicate over insecure channels
- Verify authenticity of data
- Use standardized protocols

Programming Language

- Languages running inside VMs (e.g. Java, C#) reduce risk of buffer overflow vulnerabilities
- Modern languages with security in mind
  - Go
  - Rust

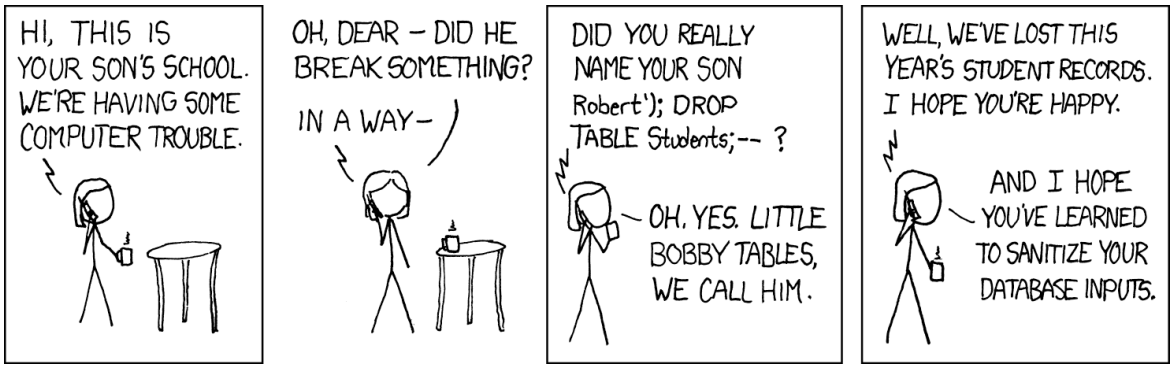
Total reported open source vulnerabilities per language:

1. **C (46.9%)**
2. **PHP (16.7%)**
3. **Java (11.4%)**
4. **JavaScript (10.2%)**
5. **Python (5.45%)**
6. **C++ (5.23%)**
7. **Ruby (4.25%)**

https://www.whitesourcesoftware.com/most-secure-programming-languages/

Input Validation

- Still one of the most common issues
- Proper testing of any input supplied by a user/application



https://xkcd.com/327/

Secure error handling

- Detect errors & handle them appropriately
- Log errors
- Investigate errors

Don‘t invent your own…

- **… crypto**
- **… protocol**

Instead, use

- standardized algorithms & protocols
- existing & proven implementations/libraries
- existing (security) frameworks

Further considerations

- Design considerations include both architectural issues at system level & at individual component level
  - System level: techniques to reduce attack surface
  - Component level: how best to implement each module
- Define roles & authorization concept
  - Which roles are required, e.g. admin, user, other devices, etc.
  - What assets should each role have access to?
- Define audit concept
- Define secure update concept (signature) & roll-back option
- Define secure storage
- Design deletion concept for decommission phase

Defensive Programming

Defend against the impossible, because the impossible will happen!

Defensive Programming

Defensive programming is a form of defensive design intended to ensure the continuing function of a piece of software under unforeseen circumstances. Defensive programming practices are often used where high availability, safety or security is needed.

From: https://en.wikipedia.org/wiki/Defensive_programming

Defensive Programming

- Source code should be readable & understandable
- Make software behave predictable even in case of unexpected inputs / actions
- Assume that software will be attacked
  - Use safe functions

Key Principles

- **DRY** (don’t repeat yourself)
  - Duplication in logic should be eliminated via abstraction
  - Duplication in process should be eliminated via automation
- **SOLID** (object oriented)
  - S – [each class has a] Single-responsiblity principle
  - O – Open [for extension] – closed [for modification] principle
  - L - Liskov [sub-type] substitution principle (child class should be usable in place of parent class)
  - I - Interface segregation principle [no extraneous methods] (specific interfaces instead of general purpose interfaces)
  - D - Dependency Inversion Principle [i.e. repository pattern] (high level modules should not depend on low-level modules)
- **CQRS** (Command Query Responsibility Segregation)
  - Segregate operations reading data from operations updating data (managing security permissions can be easier)

Defensive Programming

Never trust user input

- Always assume you're going to receive something you don't expect
- Do whitelists not blacklists, e.g. when validating input
- Don't check for invalid types but check for valid types
- Be strict

Defensive Programming

- Use Database abstraction
  - Use existing abstraction tools & libraries
- Don’t reinvent the wheel
  - Only reason why you should build it on your own is that you need something that doesn’t exist/exists but doesn’t fit within your needs
- Don’t trust developers
  - Developers shouldn’t trust others developers’ code
  - We should neither trust our own code
- Write tests

SECURITY MECHANISMS

Overview

- Cryptography
- Protocols
- Privacy techniques
- Hardware security
- Authentication and authorization
- Key Management

Cryptography

- Confidentiality
- Integrity
- Authentication
- Anonymity
- Non-repudiation
- Time stamping

Cryptography

- Symmetric cryptography
- Asymmetric cryptography
- Cryptographic hash functions
- Random Number Generators

Cryptographic Standards

- National Institute of Technology (NIST)
  - U.S. Government Federal Information Processing Standards (FIPS)
- Internet Engineering Task Force (IETF)
- International Organization for Standardization (ISO)
  - https://webstore.ansi.org/industry/software/encryption-cryptography
- National Standards
- European Union Agency for Network and Information Security (ENISA)
  - https://www.enisa.europa.eu/topics/data-protection/security-of-personal-data/cryptographic-protocols-and-tools

Guidelines

Examples

- https://csrc.nist.gov/Projects/Cryptographic-Standards-and-Guidelines
- https://www.owasp.org/index.php/Guide_to_Cryptography
- https://www.enisa.europa.eu/publications/algorithms-key-sizes-and-parameters-report/at_download/fullReport

Communication Protocols

- Transport Layer Security (TLS)
- IPsec
- Wireguard
- …

Hardware Security

- Trusted Platform Module (TPM)
- Hardware Security Modules (HSM)
- Secure Elements
- Trusted Execution Environment (TEE)
- Root of Trust
- …

Authentication

- Password-based authentication
- Multi-factor authentication
- Certificate-based authentication
- Biometric authentication
- Token-based authentication
- …

Authorization

- Attribute based
- Claims based Approach
- Group/Role based Approach
- Rule based Approach
- …

Key Management

- Asymmetric vs symmetric
- Key generation
- Key distribution
- Key storage
- Key destruction

SUMMARY

First Stages of a SPLC

- Plan and plan for change
- Utilize your threat model and risk assessment in the later phases
- Know your types of requirements
- Security requirements need to be traceable and testable
- Apply design principles to achieve Security by Design
- Do not reinvent the wheel and use well established security mechanisms