Trust and assurance

Segurança Informática em Redes e Sistemas
2024/25

Ricardo Chaves, David Matos

Ack: Miguel Pardal,
Miguel P. Correia, Carlos Ribeiro

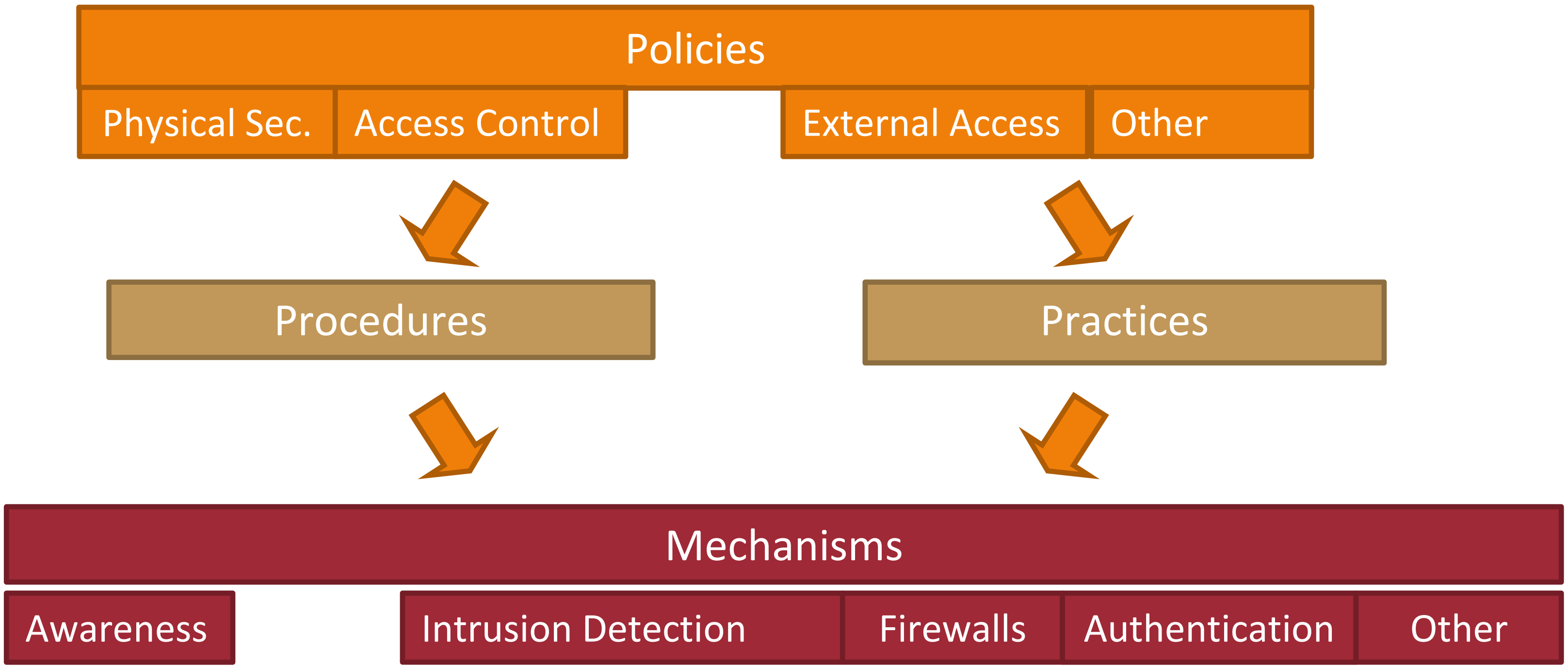
Roadmap

- Security Architecture
- Development cycle
- Recommendations
- Certification of applications and systems

Roadmap

- **Security Architecture**
- Development cycle
- Recommendations
- Certification of applications and systems

Security Architecture



Security Specifications

- **Policies**
  - Define acceptable behavior
- **Procedures**
  - Plans on how to do/enforce **policies**
- **Practices**
  - Rules to facilitate communication

- Analogy with legal world:
  - policies are the constitution,
  - procedures are the laws, and
  - practices are the regulations

Mechanisms

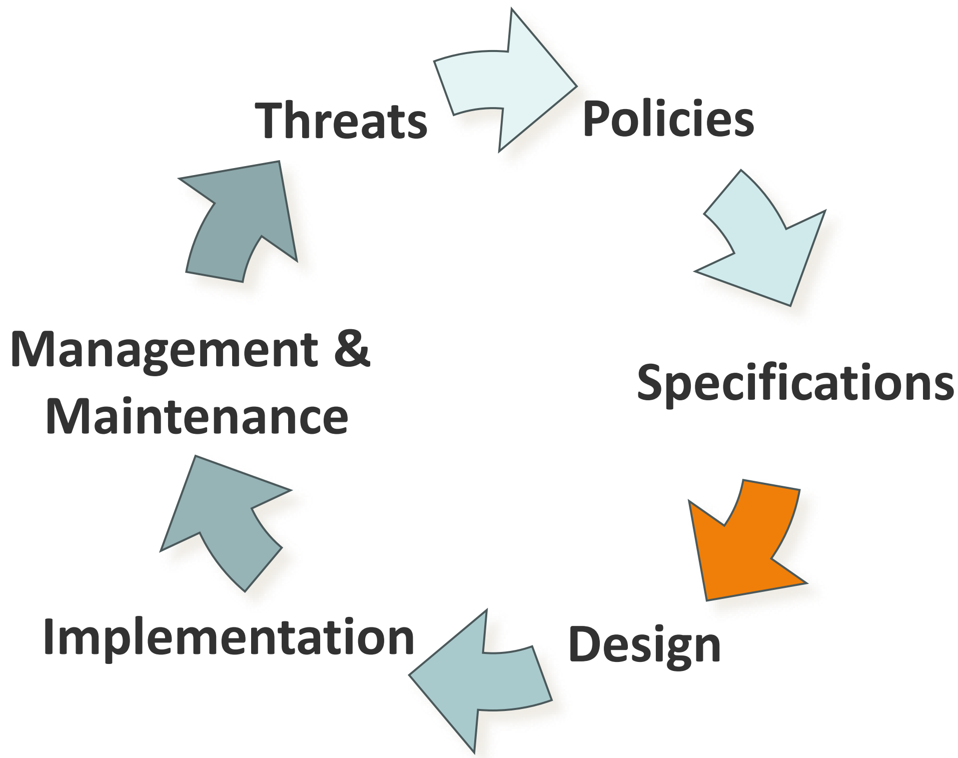
- **Mechanisms** implement the specifications made in **policies**, **procedures**, **practices**
- Example generic security **mechanisms**:
  - Confinement (e.g., sandboxing)
  - Privileged execution
  - Authentication
  - Access Control
  - Filtering
  - Auditing
  - Cryptographic algorithms and protocols

Roadmap

- Security Architecture
- **Development cycle**
- Recommendations
- Certification of applications and systems

Development of secure applications

Security life cycle

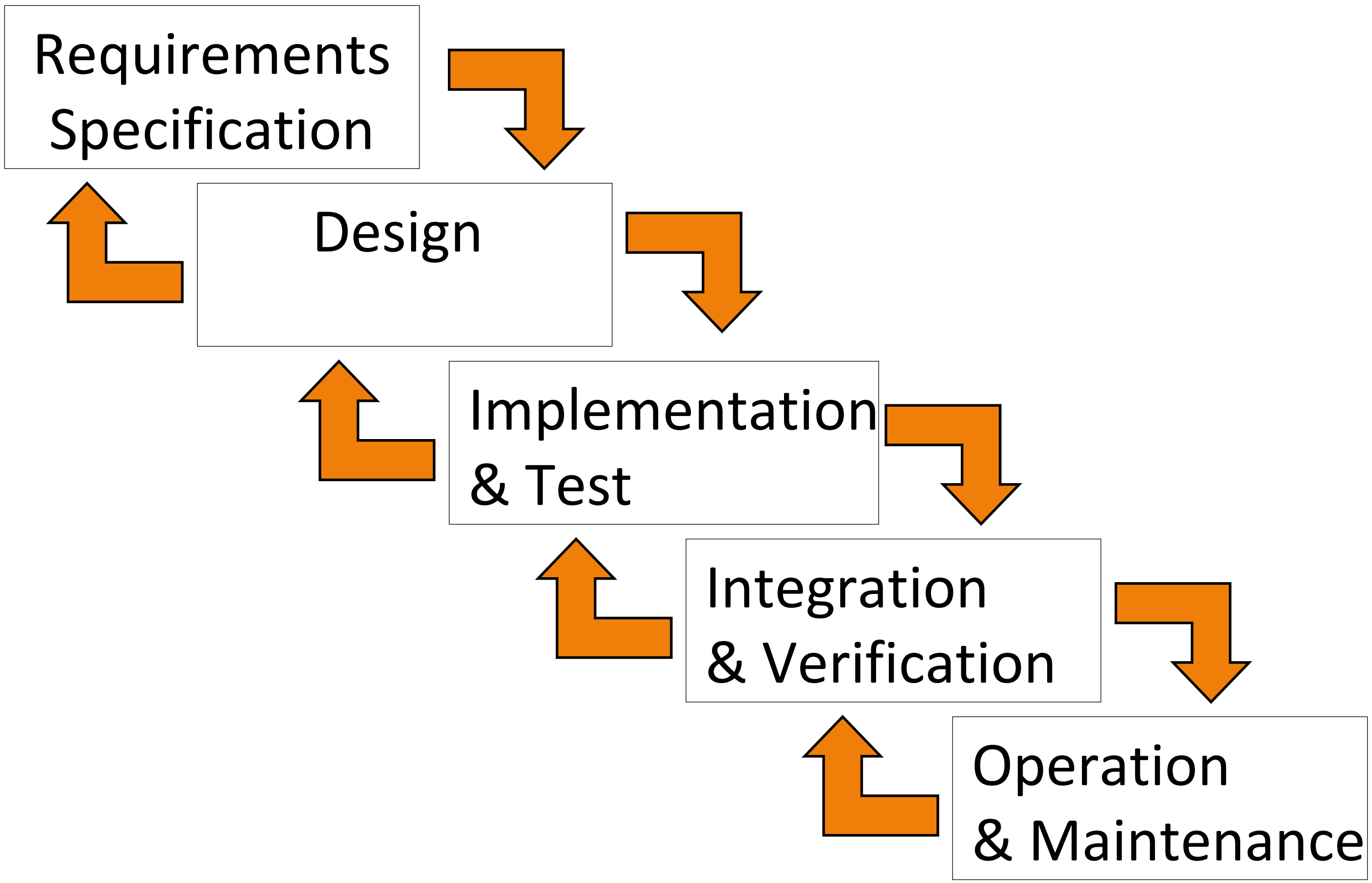


Development of secure applications

- Secure applications do not exist!
- Trust in the application:
  - A system is said to be trustworthy if there is enough evidence that it satisfies the security requirements
- Trust is obtained through assurance techniques:
  - Development methodologies
  - Formal methods
- Certification is the acceptance by assurance experts and the assignment of an assurance level

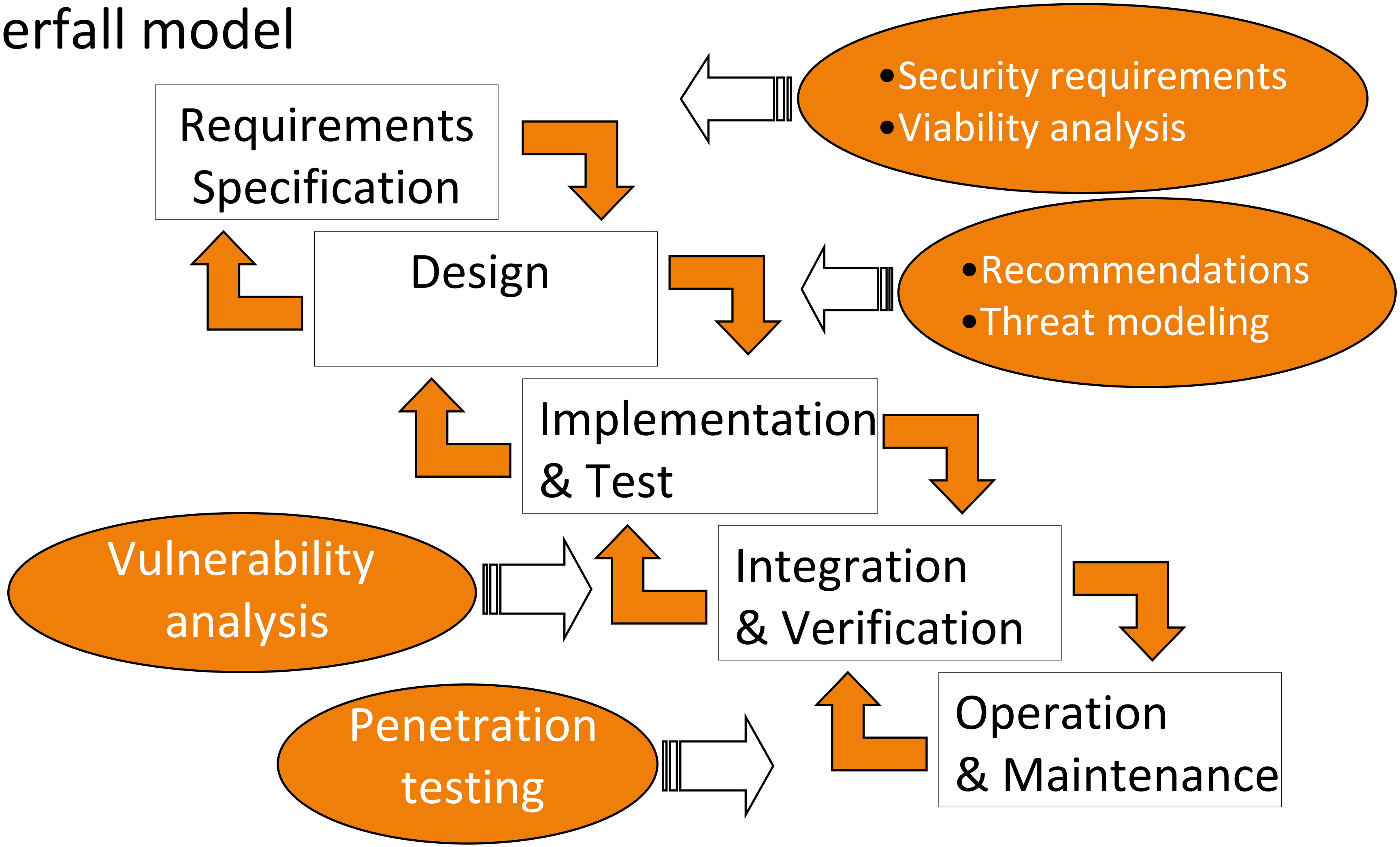
Development cycle

Waterfall model

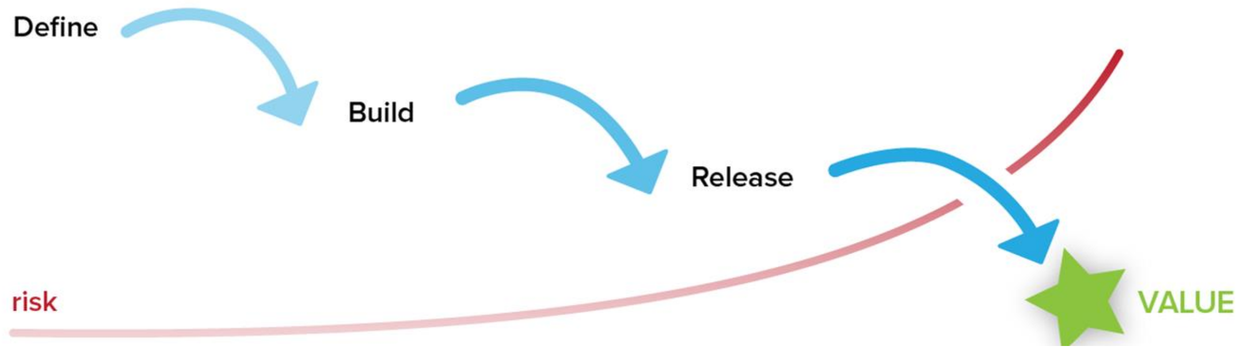


Security in the development cycle

Waterfall model

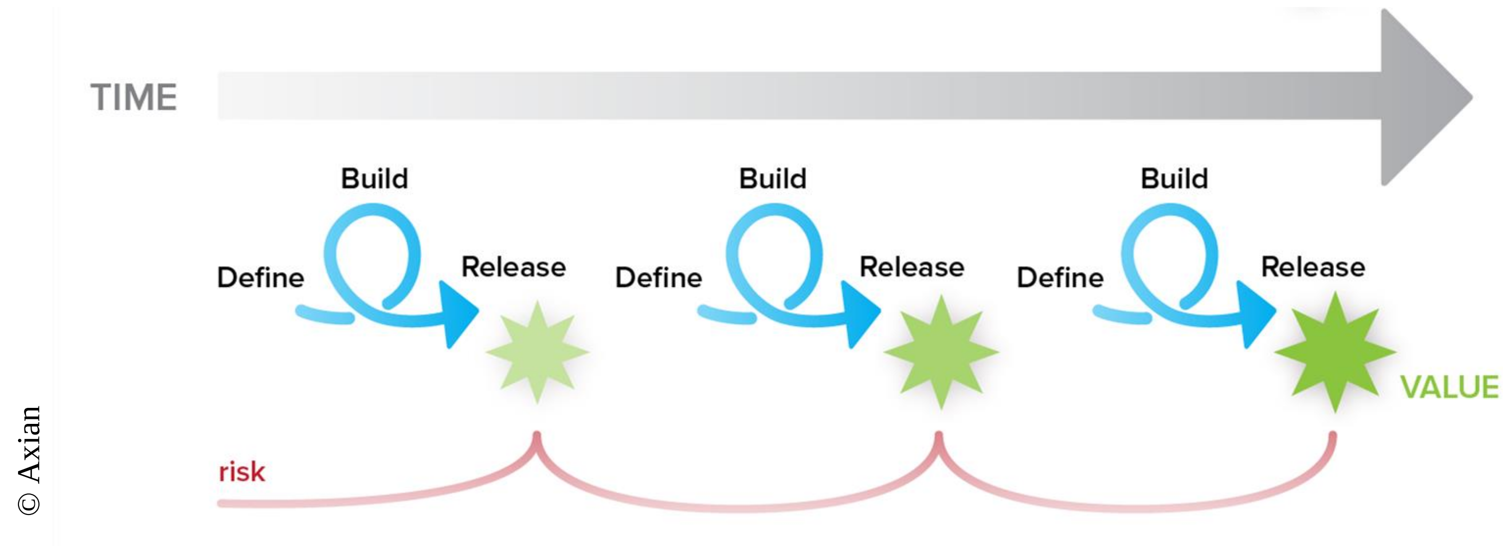


Classic development model: the "waterfall"



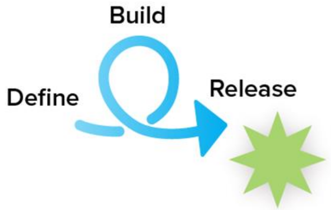
© Axian

Agile development: "sprints" that add value and lower risk



Agile Security Practices

- **Inception** practices: at the start of the Agile project
  - Risk Assessment
  - Requirements Definition
  - Incident Response
- **Iteration** practices: should be performed in every release
  - Threat Assessment
  - Code Review
  - Design Review

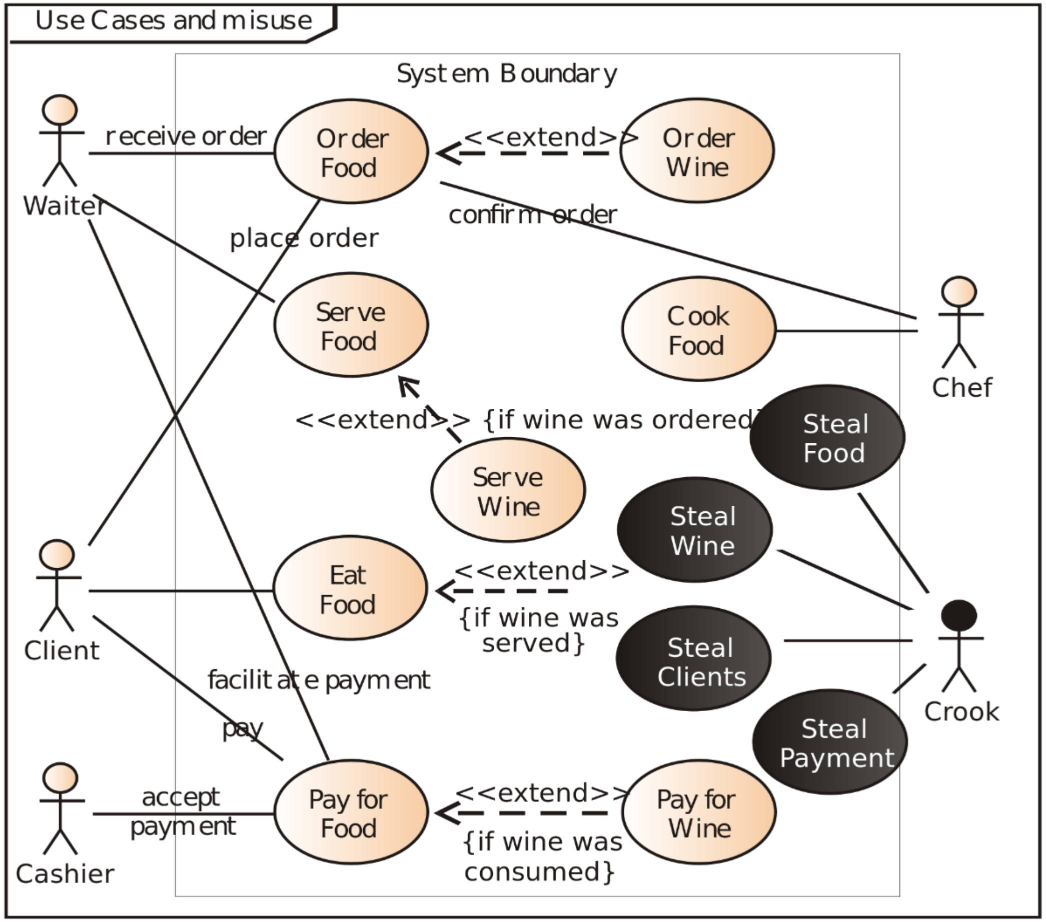


- **Regular** practices: on multiple sprints during the project
  - Dynamic Security Testing
  - Fuzz Testing (misuse)

Adapted from M. Bartoldus (gdssecurity)

Misuse case

(extension of UML use cases with explicit attackers)



Attacker model

- Attacker model is a formalization of the attacker capabilities
  - Allows us to justify the existence of defense mechanisms
  - What is each mechanism protecting against?

Attacker model example

- We consider the following capabilities to model different types of attackers:
  - A1 - Record any number of IP packets between a given source and destination and replay them from own IP address
  - A2 - Modify and suppress any number of IP packets from a given source and destination
  - A3 - Send IP packets from any IP address and receive packets

Fuzzing / security testing

- Identification of inputs
- Controlled input mutation
- Input injection
- Result analysis

Identification of inputs

- Application decomposition
- Identification of the interfaces
- Enumeration of data inputs:
  - Sockets
  - Pipes
  - Registry
  - Files
  - RPC (etc.)
  - Input parameters
  - Etc.
- Enumeration of the data structures
  - C/C++ Data structures
  - HTTP headers
  - HTTP body
  - Search strings
  - Flags
  - Etc.
- Establish the valid constructs

Fuzzing example

- Filename too long (On:Ll)
- Link to another file (Ol)
- Deny access to file (Oa)
- Lock file (Oa)

OnHand.xml

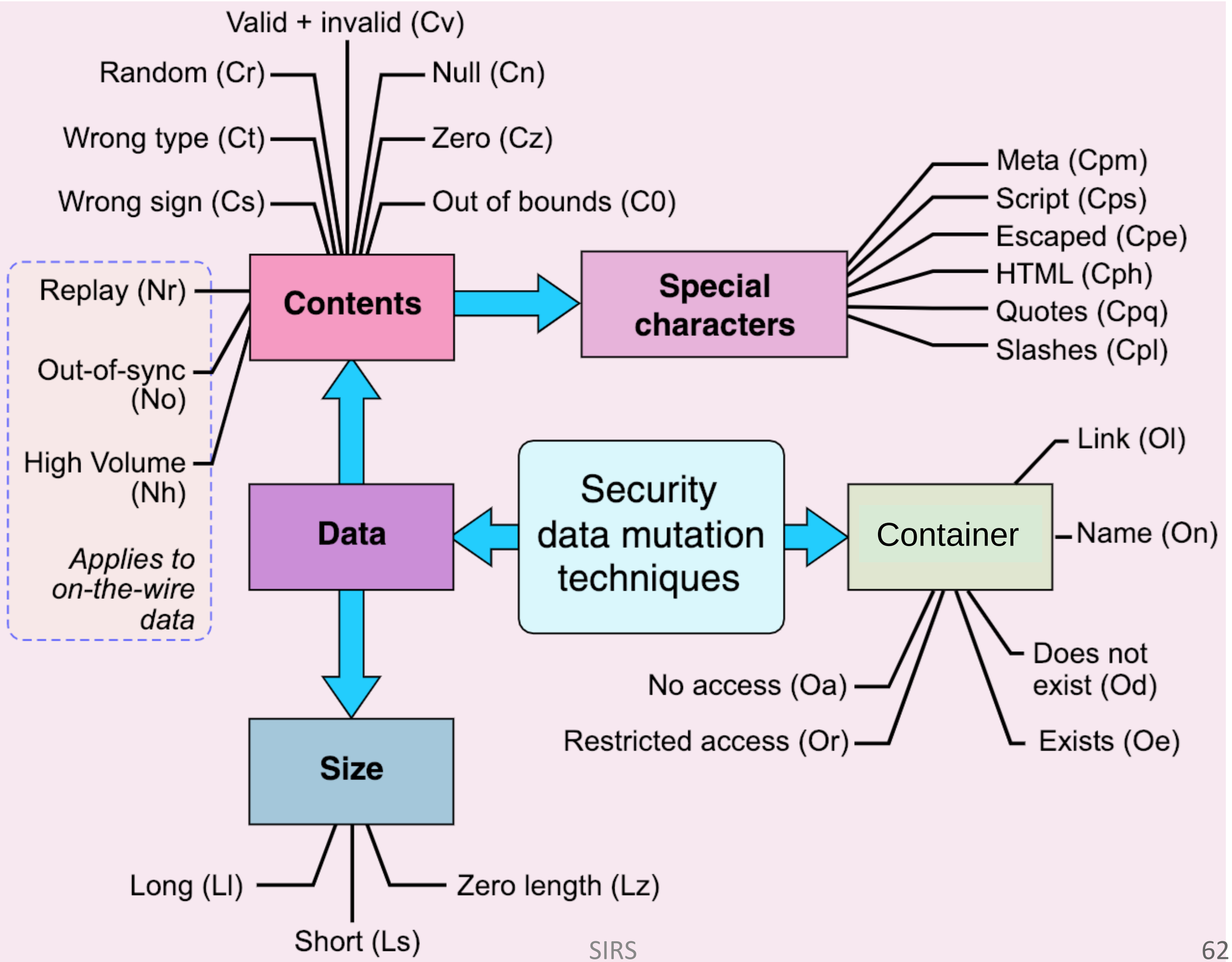
- No data (Lz)
- Junk (Cr)

```
<?xml version="1.0" encoding="utf-8"?>
<items>
  <item name="Foo" readonly="true">
    <cost>13.50</cost>
    <lastpurch>20020903</lastpurch>
    <fullname>Big Foo Thing</fullname>
  </item>
  ...
</items>
```

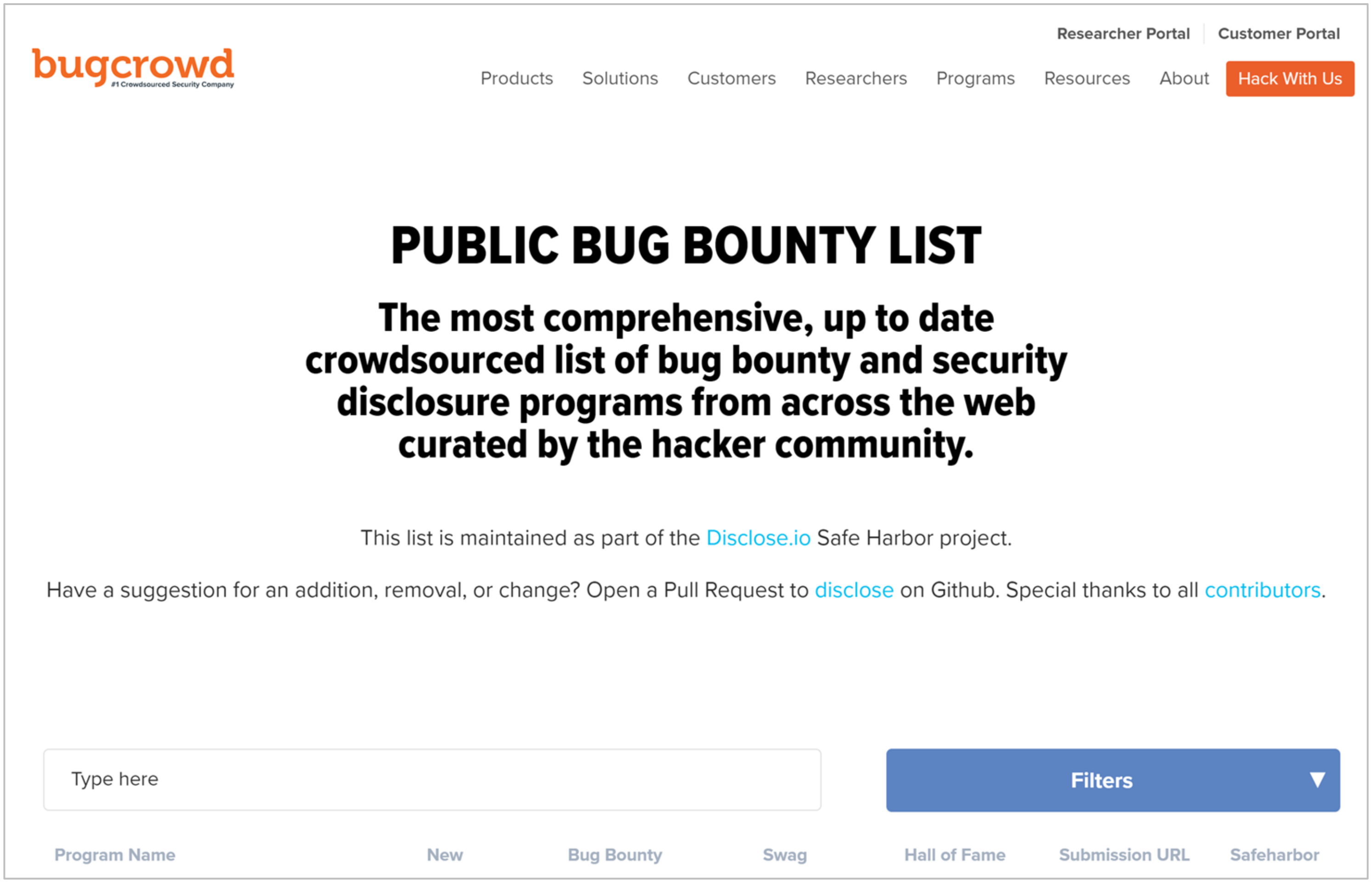
- Different version (Cs & Co)
- No version (Lz)

- Escaped (Cpe)
- Junk (Cr)

Controlled input mutation



Incentives for ethical hackers: Bug bounty programs

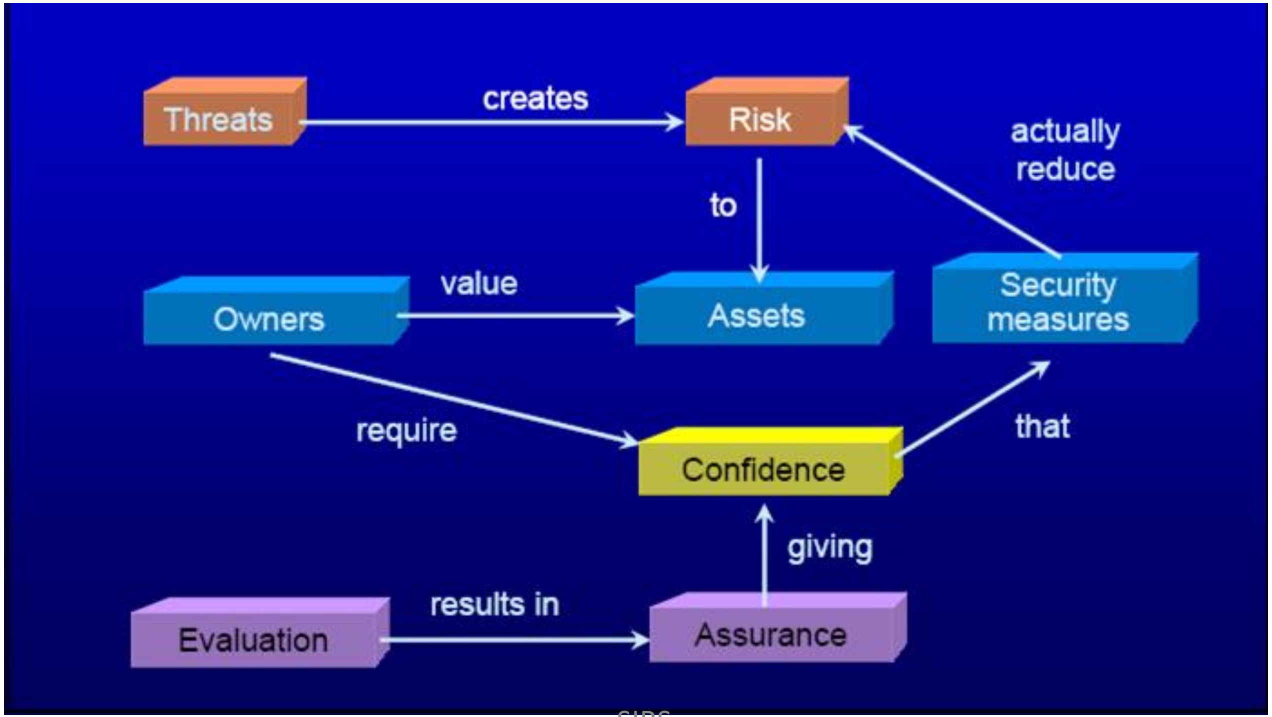


Roadmap

- Security Architecture
- Development cycle
- Recommendations
- **Certification of applications and systems**

Trust and Certification

- Assurance: ways of convincing others that a model, design or implementation is correct (trustworthy)



Certification

- For operating systems
  - TCSEC (Orange Book) (US)
  - ITSEC (UK, France, Germany, Nederland's)
- For computer networks
  - TNI - Trusted Network Interpretation (US)
- For cryptography
  - **FIPS 140** (US)
- For applications
  - ITSEC
  - **Common Criteria** (international)

Most relevant today

TCSEC (US standard)

- Evaluation:
  - Design analysis
  - Test analysis
  - Final review
- Evaluation done by independent evaluators
- Assigns a class: C1, C2, B1, B2, B3, A1
- Problems:
  - Focused heavily on confidentiality, disregarding other security properties
  - Narrow scope: mainly military operating systems
  - Based on the documentation, no access to the source code
  - Long evaluation time
  - Evaluation mixes assurance with functionality

ITSEC

- European standard
- Functionality and assurance are evaluated separately
- Applicable to systems and applications
  - TOE – *Target of Evaluation*

TCSEC	ITSEC
C1	F1+E2
C2	F2+E2
B1	F3+E3
B2	F4+E4
B3	F5+E5
A1	F5+E6

better

- Problems:
  - No validation that security requirements make sense
  - Inconsistency in evaluations (not as well defined as in TCSEC)

Common Criteria

- International standard ISO/IEC 15408
- Parts:
  - CC documentation
  - Evaluation methodology of CC (CEM)
  - National schemas (Country specific): evaluators selection; certification attributions; interaction between evaluators and vendors, etc.
    - e.g. in the USA NIST accredits commercial organizations
- CC Methodology (CEM)
  - Functional requirements
  - Assurance requirements
  - Evaluation Assurance Levels (EALs)
    - e.g. Java Smart Card has an EAL=5+ (maximum is 7)
- Types of evaluation
  - Protection Profile (PP)
  - Security Target (ST)

next slide

CC – PPs and STs

- Protection Profiles – product independent
  - for categories of products:
  - Operating systems
  - Firewalls (packet filter, application-level gateway)
  - SmartCards
- Security Targets – specific for each product
  - Hitachi Universal Storage Platform V – EAL2
  - Cisco PIX Firewall – EAL4+
  - GemXplore Xpresso V3 Java Card Platform – EAL5+
- List of PPs http://www.commoncriteriaportal.org/pps/
- List of STs (certified products) http://www.commoncriteriaportal.org/products/

CC – PP and ST

Protection Profile (generic)

- Introduction
- **Description of** the class/family of target products, a.k.a. Target of Evaluation (**TOE**)
- Description of the execution environment
  - Assumptions regarding the system operation
  - Threatened resources
  - Security policy of the target organization
- Security objectives
  - Product/system objectives
  - Environment objectives
- Security requirements
  - Functional
  - Assurance
- Rational
  - Interconnects the previous points

Security Target (specific)

- Introduction
- TOE description
- Description of the execution environment
  - Assumptions regarding the system operation
  - Threatened resources
  - Security policy of the target organization
- Security objectives
  - Product/system objectives
  - Environment objectives
- Security requirements
  - **Functional**
  - **Assurance**
- TOE specification
  - Security mechanisms
  - Description on how to assure security
- **PP claims**
  - How the PP objectives /requirements are fulfilled
- Rational

CC security requirements

Functional Requirements

- Product/system behavior definition regarding security
- 11 classes divided in families that contain components
- Components have:
  - Requirements definition
  - Dependencies from other requirements
  - Requirements hierarchy
- Predefined classes:
  - Audit (FAU)
  - Cryptography Support (FCS)
  - Communications (FCO)
  - User Data Protection (FDP)
  - Identification and Authentication (FIA)
  - Security Management (FMT)
  - Privacy (FPR)
  - Protection of the TOE Security Functions (FPT)
  - Resource Utilization (FRU)
  - TOE Access (FTA)
  - Trusted Path/Channels (FTP)

Assurance Requirements

- Establish confidence in the security features
- Correction of the implementation
- Fulfillment of the security objectives
- 10 classes
  - 1 – Evaluation of PPs
  - 1 – Evaluation of STs
  - 1 – Maintenance of Assurance
  - 7 – Product assurances
- Assurance classes:
  - Development
    - TOE design, Functional specifications, …
  - Delivery and Operation
  - Configuration
  - Product Documentation
  - Life cycle
    - Delivery, Flaw remediation, …
  - Testing
    - Depth, coverage, …
  - Vulnerability analysis

Evaluation Assurance Levels

- Derived from the assurance requisites

EAL1	Functionally Tested
EAL2	Structurally Tested
EAL3	Methodically Tested & Checked
EAL4	Methodically Designed, Tested & Reviewed
EAL5	Semiformally Designed & Tested
EAL6	Semiformally Verified Design & Tested
EAL7	Formally Verified Design & Tested

better

Roadmap

- Security Architecture
- Development cycle
- **Recommendations**
- Certification of applications and systems

IEEE Center for Secure Design recommendations

IEEE CENTER FOR SECURE DESIGN

Interested in keeping up with Center for Secure Design activities? Follow @ieeecsd on Twitter, catch up with us via cybersecurity.ieee.org, or contact Kathy Clark-Fisher, Manager, New Initiative Development (kclark-fisher@computer.org).

http://cybersecurity.ieee.org/blog/2015/11/13/avoiding-the-top-10-security-flaws/

Secure design

- Prevent security problems in **design** stage
  - Design flaws
    - Different from implementation bugs or defects
  - Avoiding flaws can significantly reduce the number and impact of security breaches

  - The goal of a secure design is to enable a system that **supports and enforces** the necessary authentication, authorization, confidentiality, data integrity, accountability, availability, and non-repudiation requirements, **even when the system is under attack**

Top 10 recommendations

1. Earn or give, but never assume trust
2. User authentication that cannot be bypassed or tampered
3. Authorize after you authenticate
4. Strictly separate data and control instructions
5. All data must be explicitly validated
6. Use cryptography correctly
7. Identify sensitive data and how to handle it
8. Always consider the users
9. Understand how external components affect attack surface
10. Be flexible when considering future objects and actors

1/10 Earn or give, but never assume **trust**

- Software systems rely on composition and cooperation of two or more software tiers or components

- Offloading security functions from server to client exposes those functions to a much less trustworthy environment

- 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

2/10 Use **authentication mechanism** that cannot be bypassed or tampered with

- Authentication is the act of validating an entity's identity
- A securely designed system should also prevent that user from changing identity without re-authentication
- Authentication techniques should require one or more factors for more sensitive operations
  - Factors:
    - something you know,
    - something you are, or
    - something you have

3/10 **Authorize after** you **authenticate**

- Authorization should be conducted as an explicit check
  - 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
  - Time, location, etc.
  - Handle revocation

4/10 Strictly **separate data** and **control** instructions

- Combining data and control instructions in a single entity, especially a **string**, can lead to injection vulnerabilities
  - Often leads to untrusted data controlling the execution flow of a software system
  - Concern at all levels: machine instructions, high-level instructions, domain specific languages

5/10 All **data** must be **explicitly validated**

- It is important to explicitly ensure that assumptions on data hold
  - Vulnerabilities frequently arise from implicit assumptions about data

- Design software systems to ensure that comprehensive data validation actually takes place and that all assumptions about data have been validated when they are used

6/10 **Use** cryptography **correctly**

- Through the proper use of cryptography, one can ensure the confidentiality of data, protect data from unauthorized modification, and authenticate the source of data
  - and more

- Common cryptography pitfalls:
  - Creating your own cryptographic algorithms or implementations
  - Misuse of libraries and algorithms
  - Poor key management
  - Randomness that is not random
  - Failure to allow for algorithm adaptation and evolution

7/10 Identify **sensitive data** and how it should be handled

- Data sensitivity is context-sensitive
  - Depends on regulation, company policy, contractual obligations, user expectation, etc.
    - Examples: user-input, data computed from scratch, data coming from external sensors, cryptographic material, and Personally Identifiable Information (PII)
- First step: create a policy that explicitly identifies different levels of classification
- Define most important property:
  - Confidentiality
  - Integrity
  - Availability

8/10 Always consider the **users**

- The security of a software system is inextricably linked to what its users do with it
- Always consider the users, and any other stakeholders, in the design and evaluation of systems
  - Factors
  - Trade-offs
- Make the most common usage scenario also secure
  - “secure by default”
  - Make relevant settings as easy to find

9/10 Understand how external components affect **attack surface**

- The attack surface are the different points where you can try to enter or extract data from the system

- You must assume that incoming external components are not to be trusted until appropriate security controls have been applied

- Align the component's attack surface and security policy with the overall system's

10/10 Be flexible when considering future changes to Objects and Actors

- Software security must be designed for change
  - Environments, threats and attacks
  - Rather than being fragile, brittle, and static

- Consider the security implications of future changes
  - Design for security updates
  - Design for security properties changing over time
  - Design for changes in components beyond your control
  - Design with the ability to isolate or toggle functionality
  - Design for changes to objects intended to be kept secret (keys)
  - Design for changes in entitlements (dynamic permissions)

Summary

- Security Architecture
- Development cycle
- Recommendations
- Certification of applications and systems