esse

Advanced Security for Systems Engineering - VO 05: Security Architectures

Herbert Brunner und Christian Schanes

INSO - Industrial Software
Institut für Rechnergestützte Automation | Fakultät für Informatik | Technische Universität Wien

Agenda

Introduction to Software Architecture

Trade-off between IT Security Goals

Secure Design Principles

CLASP - Comprehensive, Lightweight Application Security Process

Security Modeling in UML

Security Patterns

Summary

Introduction to Software Architecture 1/3

- Usually developing software is done as follows [Dustdar 2003]
  - Start with a "wishlist" of requirements
  - Create a concept
  - Afterwards changes are needed
  - Deviation of the concept is necessary
- Result: Big Ball of Mud
- Most of the IT projects fail [Standish 1994, Yourdon 2004]

Introduction to Software Architecture 2/3

- Software architecture is [Dustdar 2003]
  - compareable to civil architecture
  - aims to manage the complexity of systems we build
- Architecture states [Dustdar 2003]
  - how to structure a building or software
  - the act of designing this structure (i.e., the arrangement of parts)

Introduction to Software Architecture 3/3

- Software architecture [Dustdar 2003]
  - Does not specify the details, but the coarse-grained components (e.g. subsystems)
  - Includes the analysis of the problem
  - Is the basis for the implementation of the solution
  - Aims to make complexity manageable
  - Consists of decisions that have system-wide consequences
- **Software architecture is about taking decisions related to structure of the software by viewing it from different viewpoints** (e.g., Information, Security, ...)

(Vergleiche http://www.sei.cmu.edu/architecture/start/glossary/community.cfm)

Secure Architectures

(CERT Secure Coding Standards)

Trade-off between IT Security Goals

- IT Security Goals: confidentiality, integrity, availability, authenticity, authorization
- Security Goals may conflict with each other, when designing software architecture, e.g.,
  - Confidentiality vs. availability
    (i.e., Encrypted data not recoverable, if key lost)
  - Authenticity vs. integrity
    (i.e., Attacker impersonates a user and modifies data)
  - Availability vs. authenticity
    (i.e., Slow hashing algorithms within authentication process)
  - Integrity vs. confidentiality
    (i.e., Attacker modifies encrypted data)

Secure Design Principles

- Least Privilege
- Separation of Duties
- Fail Secure
- Economy of Mechanisms
- Complete Mediation
- Least Common Mechanisms
- Psychological Acceptability
- Leveraging Existing Components

Least Privilege - Overview

- Software is operating with least privileges, if
  - Only the minimum level of access right has been given to it
  - For a minimum amount of time to perform its operation
- Examples of the least privilege principle:
  - Military security rule of "need-to-know"
  - Administrative privileges granted to a middleware server
- Goal is containment of the damage resulting from, for example, a security breach
- Best practise for managing resources (e.g., databases): have many admins with limited access to resources instead of one "superuser"

Least Privilege - Potential Vulnerabilities

- Failure to check if privileges were dropped successfully
  - Example: In some cases a system may record actions of impersonated user rather than the impersonator, i.e., user A is admin, impersonates no-admin user B and drops the database
- Failure to drop privileges when reasonable
  - Example: An attacker may access resources that should not have been accessible, i.e., access to /etc/passwd in conjunction with a injection vulnerability

Least Privilege - Design Techniques

- Modular Programming:
  - Split entire program into smaller subunits or modules
  - Each module is discrete with unitary functionality (cohesive)
  - Each module is designed to perform one logical operation
  - Each module is rather independent to others (coupling)
  - Because of single purpose per module, privileges and access are limited easier
- Example: CalcDiscount(), CalcShipCost() $\rightarrow$ restriction to different users
- Best practise suggests: Designed software modules are highly cohesive and loosely coupled

Separation of Duties - Overview

- Separation of duties is that
  - The entity that approves an action
  - The entity that carries out the action and
  - The entity that monitors that action must be separate
- Goal is to prevent a single user from carrying out and hiding prohibited actions
- Examples:
  - Equipment purchase can only be done by several users (e.g., ordering computers)
  - Admins should be able to, for instance, set password policy, but not be able to impersonate other users
  - Two-man rule

Separation of Duties - Potential Vulnerabilities

- Log forging

```
//read integer value from request object
String val = request.getParameter("val");
try {
        int value = Integer.parseInt(val);
}
catch (NumberFormatException) {
        log.info("Failed to parse val=" + val);
}
```

- If an attacker (i.e., a developer) submits the string "twenty-one%0a%0aINFO:+User+logged+out%3dbadguy" in production system, the result will be

```
INFO: Failed to parse val=twenty-one
INFO: User logged out=badguy
```

(Vergleiche `https://www.owasp.org`)

Separation of Duties - Design Techniques

- When designed and implemented correctly, reduction of damage by a person or resource is reduced
- Separation of duties should be found in application features (i.e., role-based access control) and software development life-cycle (i.e., deny access for developers on production systems)
- Example when dealing with cryptographic keys:
  - Splitting of cryptographic keys (i.e., registry and config file) might be a better idea, and storing the parts in different locations, instead of storing in a single location
- Best practise suggests to implement separation of duties with auditing

Defense in Depth - Overview

- Defense in Depth (layered defense) results from layering
  - Security controls = countermeasure to avoid / minimize security risk and
  - Risk mitigation safeguards into software design
- Seeks to delay, rather than to prevent attacks
- Should give an organization time to detect and respond to an attack
- Goal is that software doesn't get totally compromised, because of single security breach
- Example: it's not good to totally rely on a firewall for an internal-use-only application

Defense in Depth - Shortcomings

- Defense in Depth
  - Can add complexity to the software system
  - Contradicts to the principle of simple design
  - Therefore might introduce new risks
  - Can be reactive (i.e. detect malicious activities and block them) or preventive (e.g., awareness training, security patches)

Defense in Depth - Strategy

- Following layers might be used when designing defense in depth
  - Antivirus software
  - Firewalls
  - Demilitarized zones
  - Intrusion detection systems
  - Packet filters
  - Routers and switches
  - Proxy servers
  - VPNs, etc.
- Best practise suggests: Total risk to the system needs to be assessed (i.e., concerning new risks or conflicting security goals)

Fail Secure - Overview

- Fail secure is the principle that the software
  - Reliably functions when attacked
  - Is quickly recoverable into a normal business
  - Is into secure state in the event of design or implementation flaw
- Example:
  - User is denied access by default and locked out after max. number of access attempts
  - Error in traffic lights leads to red instead of green light
- Maintain the resiliency of software by defaulting to a secure state
- Fail secure implies secure by design, secure by default and secure by deployment

  (Vergleiche `http://msdn.microsoft.com/en-us/library/ms995349.aspx`)

Fail Secure - Bad Style Example

- No fail safe fall back mechanism after exception occurred

```
isAdmin = true;
try {
  codeWhichMayFail();
  isAdmin = isUserInRole( ''Administrator'');
}
catch (Exception ex) {
  log.write(ex.toString());
}
```

- Shortcoming: user is an admin by default, which is a security risk

(Vergleiche `https://www.owasp.org`)

Fail Secure - Potential Vulnerabilities

- Exceptions that occur in procession of a security control itself (i.e., user login process)
  - When exception occurred
    - Do not enable behaviour that countermeasure usually not allows
    - Follow same execution path as disallowing the operation (or throw exception)
- Exceptions in code that is not part of a security control
  - Security-relevant, if affects whether application properly invokes the control, i.e., an exception causes a security method not to be invoked when it should

Fail Secure - Design Techniques

- Do not allow exceptions to go unhandled
- Do not allow any exceptions to reach the GUI
- Inspect application's fatal error handler
- Check, if error handler is called frequently
  (i.e. there might be a security vulnerability)
- Use aspect-oriented design for cross-cutting concerns

Economy of Mechanisms - Overview 1/2

- Economy of mechanisms referred to as KISS (Keep It Simple Stupid)
- 80/20 rule
- Often developers design more functionality than necessary ("bells-and-whistles")
- Good indicator to identify unneeded functionality is reviewing requirements traceability matrix (RTM)
- Example of a RTM:

	funct. requ.	testing requ.
business requ. 1	...	...
business requ. 2	...	...
business requ. ...	...	...
business requ. n	...	...

Economy of Mechanisms - Overview 2/2

- Bells-and-whistles features will not be part of the RTM
- Such gimmicks may increase user experience and usability of software
- BUT, attack surface of the software increases
- Simple design supports a better understanding of programs
- Software is less error-prone and maintenance is easier
- Modular programming supports economy of mechanisms
- “Complexity is the worst enemy of security” (Schneier)

Economy of Mechanisms - Design Techniques

- Avoidance of unnecessary functionality and security mechanisms
  - Patching and configuration of software has been known to disable security features that were in implemented → better not to design such features instead of disabling them afterwards
- Strive for simplicity
  - Keep security mechanisms simple
  - Implementation should not be partial → otherwise security issues
  - Model the data simple (results in simpler validation routines)
- Strive for operational ease of use (e.g. SSO)

Complete Mediation - Overview

- Complete mediation states that access requests need to be mediated each time that authority is not bypassed in subsequent requests
- Enforcement of a system-wide view of control
- Protects against authentication and confidentiality threats
- Addresses the integrity aspects of software
- Cope with alternate path vulnerabilities
  (i.e., code paths that may disclose sensitive information)
- Central implementation of security functionality instead of duplicate functionality with different implementations
- Examples of complete mediation:
  - Not allow browser postbacks without validation access rights
  - Check transaction state to protect against data duplication
  - Java Security Manager

Complete Mediation - Potential Vulnerabilities

- Caching in applications
  - The results of the cached authority check might increase performance
  - May lead to an increased security risk (e.g., authentication bypass, session hijacking)
- Change value of query string parameter without additional validation
  - Example:
    - User A logged in and the URL in browser bar shows name value pair "user=A"
    - Changing the value to B would display user B's information without validation
    - Confidentiality of B's sensitive data gets violated

Complete Mediation - Design Techniques

- Avoid client-side, cookie-based caching of authentication credentials for access
- When dealing with web applications $\rightarrow$ disable submit button until transaction not has finished
- Identify code paths that access privileged and sensitive resources
- Locate the weakest link (code, service, interface or user) and protect surrounding software against it
- Be aware of social engineering attacks and security not aware users
- Keep in mind users that don't know how to use the software

Open Design - Overview

- The security of a software should not be based on the secret of the design and implementation
- Comparable to a crypto system where the security is only based on key and not on the secret of the cryptographic operations (Kerckhoff's principle)
- Example: GSM crypto algorithms
- Security by obscurity
- Always design a system in a way, that the design and implementation is known by the attacker
- Doesn't mean to publish the design and implementation

Open Design - Potential Vulnerabilities

- Example: Obscurity Through Alternate Port bindings
  - Using alternate port number for an otherwise insecure network service (i.e., Telnet on port 1023)
  - Ineffective security control, because most port scanners would detect the service on the alternate port number (i.e., better use SSH)
- Example: Obscurity through closed source code
  - Security should not rely on keeping the source secret (e.g., leftover debug code, hard-coded passwords)
  - Assume attackers have source code, they could easily extract sensitive information

Open Design - Potential Vulnerabilities

- Avoid security by obscurity
  - No hard coding of sensitive information in source code or binaries (e.g., cryptographic keys, passwords, connection strings)
  - Use hidden form fields in web application carefully, i.e., modified client
  - Hidden URLs for secret documents without authentication, i.e., google hacking

Least Common Mechanisms

- Principle states to minimize common mechanisms that are shared between users / processes
- Every shared mechanism represents a potential information flow between users / processes
- Shared mechanisms could be, for instance, variables or entity beans
- Design should isolate code (functions) by user roles $\rightarrow$ limits exposure of sensitive data
- Example:
  - Instead of sharing a function between superusers and nonsuperusers consisting of different code paths for each party, implement two separate functions to serve the different roles

Psychological Acceptability

- Security mechanisms should be designed make usage, adoption and automatic application as high as possible
- With psychological adoption in mind security protection mechanisms should
  - be easy to use
  - not affect accessibility
  - be transparent to the user
- Users should not be burdened by security mechanisms
- Example:
  - Password policy that requires min. 16 characters for passwords may enforce users to write down their passwords and decrease overall security

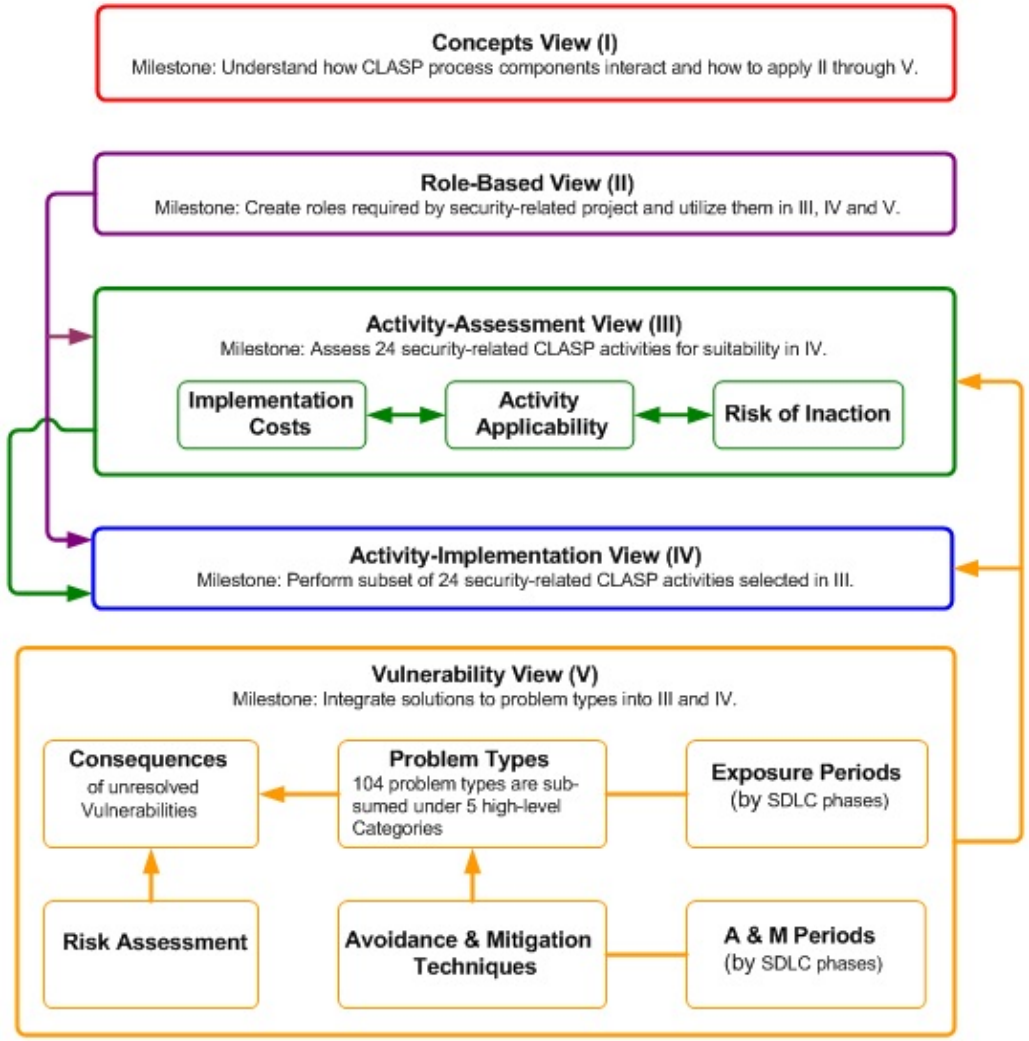
Leveraging Existing Components

- Promotes the reusability of existing components
- Service-oriented architecture (SOA) is widely used today
  - SOA provides communication between heterogeneous environments / platforms
  - Example: Many financial institutions use a common currency conversion service
- Tier architecture is advisable to design software that needs to fit high scalability
  - Software functionality can be broken into presentation, business and data access tiers
  - Use of a single data access layer supports scaling
- Enterprise application blocks are recommended over custom developing libraries
- Reuse tested, proven, existing common components

CLASP - Comprehensive, Lightweight Application Security Process

- Well-structured and organized approach for moving security concerns into the early stages of the SDLC
- Easy integration into an existing application development life-cycle
- Each CLASP activity can be divided into process components and linked to one or more project roles, e.g. architects, developers, security auditors, project managers
- Contains a vulnerability lexicon that acts as a kind of knowledge base
- Recommends to use static analysis of source code
- CLASP process
  - CLASP Views
  - CLASP Resources
  - Vulnerability Use Cases

CLASP Views



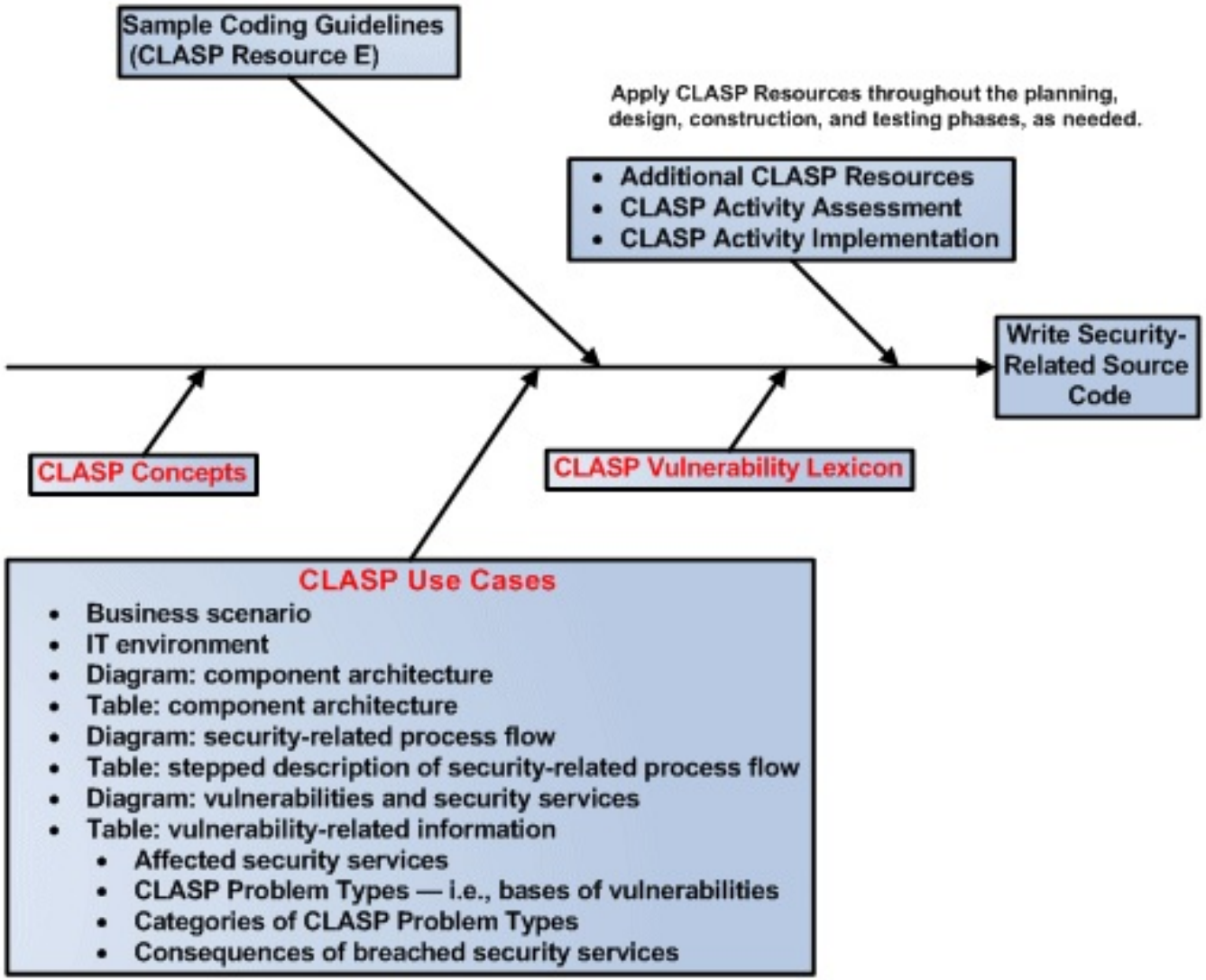
(Vergleiche `https://www.owasp.org`)

CLASP Resources

- Basic Principles in Application Security (all Views) → Resource A
- Example of Basic Principle: Input Validation (all Views) → Resource B
- Example of Basic-Principle Violation: Penetrate-and-Patch Model (all Views) → Resource C
- Core Security Services (all Views; especially III) → Resource D
- Sample Coding Guideline Worksheets (Views II, III & IV) → Resource E
- System Assessment Worksheets (Views III & IV) → Resource F
- Sample Road Map: Legacy Projects (View III) → Resource G1
- Sample Road Map: New-Start Projects (View III) → Resource G2
- Creating the Process Engineering Plan (View III) → Resource H
- Forming the Process Engineering Team (View III) → Resource I
- Glossary of Security Terms (all Views) → Resource J

(Vergleiche `https://www.owasp.org`)

CLASP Vulnerability Use Cases



(Vergleiche `https://www.owasp.org`)

CLASP Views - Role-Based View

- Provides a high-level view to project manager and their team concerning security issues
- Introduces basic responsibilities of project members
- 24 activities in the whole process, e.g.,
  - Project managers: monitor security metrics, institute security awareness program
  - Designer: identify attack surface, apply security principles to design
  - Implementer: implement interface contracts, elaborated security technologies
  - Security Auditor: perform threat modeling
  - Integrator: perform code signing

CLASP Views - Activity-Assessment View

- Lessens the burden on project managers and their process team and help to assess the appropriate CLASP activities
- For each activity there is given the following information
  - Information on activity applicability, i.e., some activities are only applicable for applications using a back-end database
  - Discussion of risks associated with omitting the activity, i.e., includes rating the overall impact of the activity, relative to other CLASP activities
  - Indication of implementation cost, i.e., the frequency of the activity, the man-hours per iteration and possible automation technologies for activities
- CLASP doesn't change the steps in software engineering process, it recommends extensions to common artifacts and provides implementation guidance

CLASP Views - Example Activity-Assessment View

Purpose:	• Ensure project members consider security to be an important project goal through training and accountability. • Ensure project members have enough exposure to security to deal with it effectively.
Owner:	Project Manager
Key contributors:	
Applicability:	All projects
Relative impact:	Very high
Risks in omission:	• Other activities promoting more secure software are less likely to be applied effectively. • Accountability for mistakes is not reasonable.
Activity frequency:	Ongoing
Approximate man hours:	• 160 hours for instituting programs. • 4 hours up-front per person. • 1 hour per month per person for maintenance.

(Vergleiche `https://www.owasp.org`)

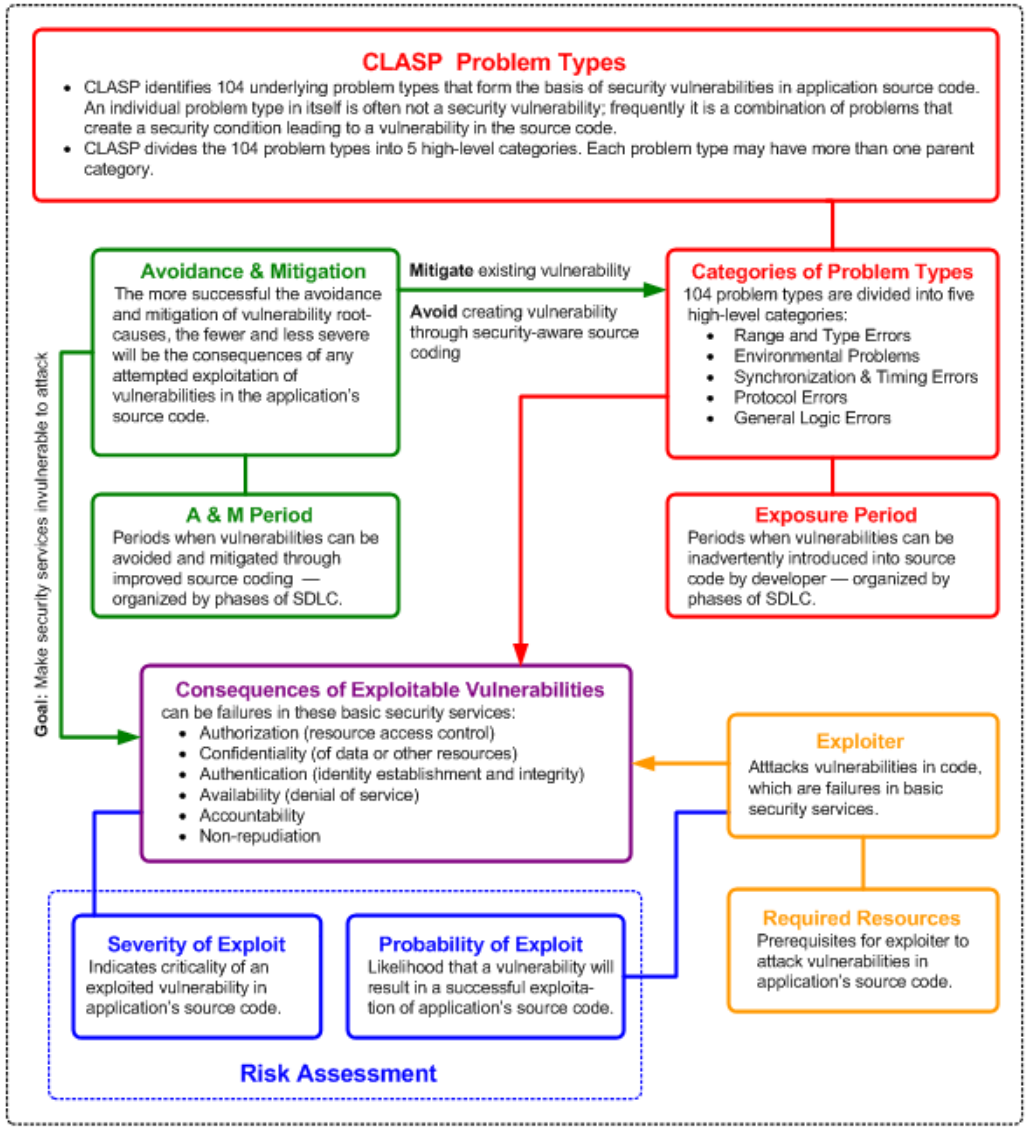
CLASP Views - Activity-Implementation View

- Integrates the 24 security-related activities into a software development process
- Activity phase translates into executable software
- Example: Project manager institutes security awareness training
  - Tasks
    - Provide security training to all team members
    - Promote awareness of local security setting
    - Institute accountability for security issues
    - Appoint a project security officer
    - Institute rewards for handling security issues

CLASP Views - Vulnerabilities View 1/2

- Depicts a vulnerability lexicon that categorises security problems at appropriate levels
- Within CLASP taxonomy security vulnerabilities are described by
  - Problem types (e.g. basis causes)
  - Categories (for diagnosis and resolution)
  - Exposure periods (e.g. SDLC phases)
  - Consequences (w.r.t security goals)
  - Platforms
  - Resources (for attack against vulnerability)
  - Risk assessment
  - Avoidance and mitigation periods

CLASP Views - Vulnerabilities View 2/2



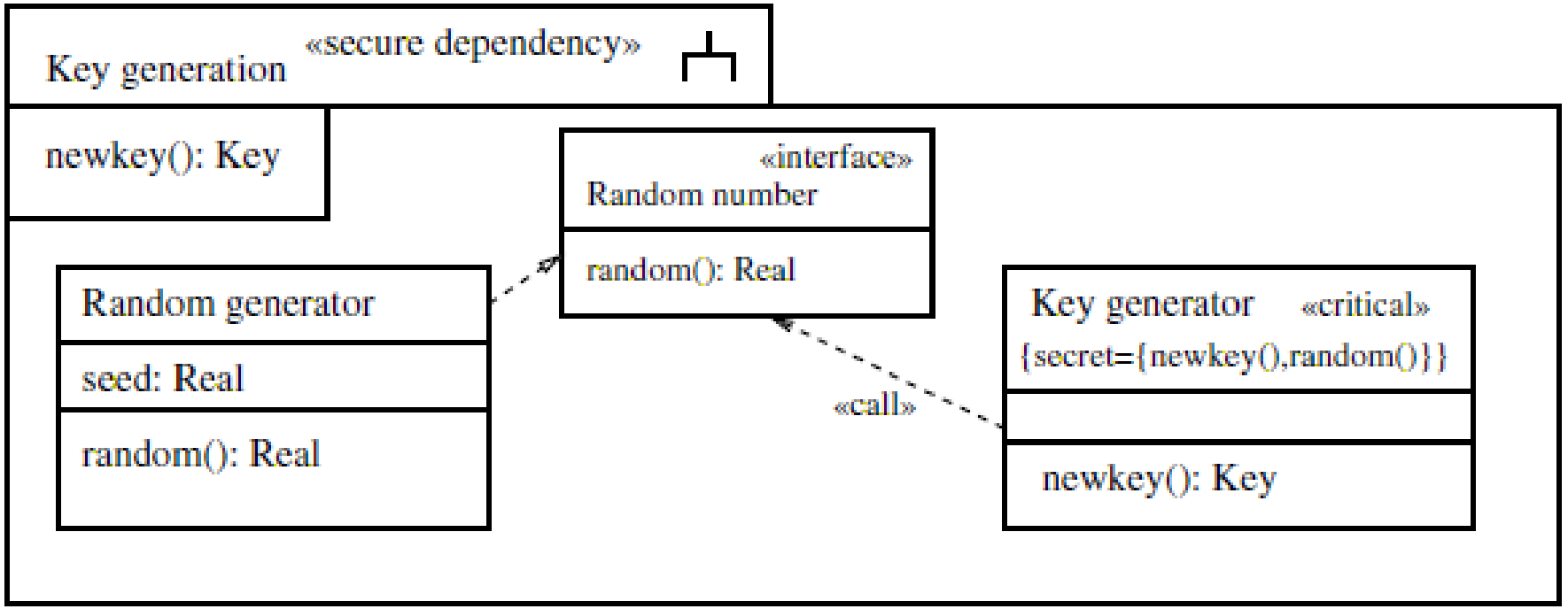
(Vergleiche `https://www.owasp.org`)

Security Modeling in UML

UML is widely used for modeling software systems. There are several extensions of the UML language. Special extension for modeling security requirements, e.g. UMLSec

- Extension via UML Profiles
- modelling of security relevant dependencies between components
- Extends data structures via annotations, e.g."secrecy" or "integrity"
- Extension offers additional devices, e.g. smart cards
- There exist extensions for connection types like "encrypted" or "LAN"

Security Modeling in UML - Example UMLSec



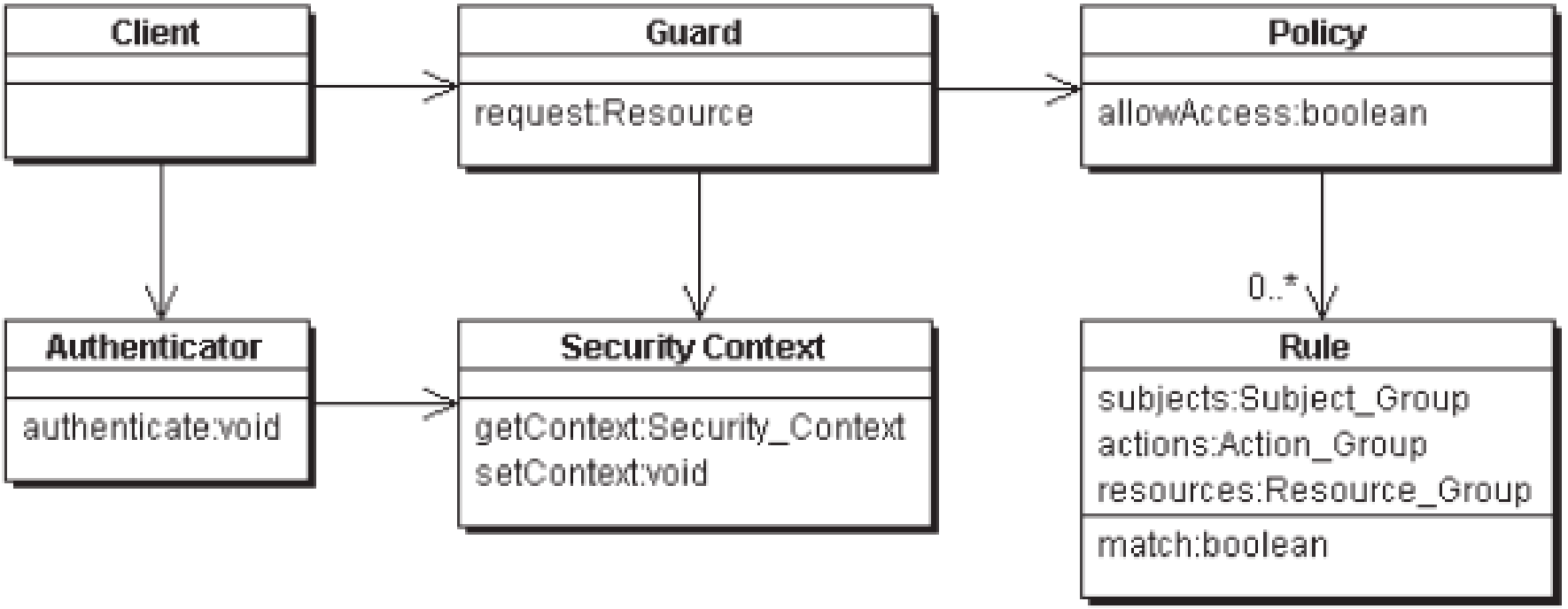
(Vergleiche Jürjens Jan, 2004)

Security Patterns

Security patterns provide paradigms / samples for designing secure software components

- Reusable prototype
- Based on best practise
- Patterns exist for technical and / or organizational design purpose
  - Technical, e.g. reference monitor (protected system)
  - Organizational, i.e., documentation of security goals, secure installation / configuration
- Different patterns available
  - `http://coresecuritypatterns.com/patterns.htm`
  - `https://www.owasp.org/index.php/Category:OWASP_Security_Analysis_of_Core_J2EE_Design_Patterns_Project`

Security Patterns - Beispiel "Policy"



(Vergleiche Bob Blakley und Craig Heath, 2004)

Summary

- Solid architecture is the basis of secure software
- IT security goals may conflict with another
- Due to business needs decide what design principles to use
- Balancing of different design principles is recommended
- CLASP is an process approach to integrate security in various stages of software projects
- Security modeling and security patterns offer concepts to design secure software components
- “Complexity is the worst enemy of security” (Schneier)

References and Bibliography

- Official (ISC)2 Guide to the CSSLP; Mano Paul; 2011
- https://www.owasp.org
- Computer and Information Security Handbook; Vacca, John R.; 2009
- The Protection of Information in Computer Systems;
  Saltzer, Jerome H. and Schroeder Michael D.; 1975

Thank you!

http://security.inso.tuwien.ac.at/