Secure Software Development

Marcus Bendtsen
Institutionen för Datavetenskap (IDA)
Avdelningen för Databas- och Informationsteknik (ADIT)
Agenda

• Securing the software development life cycle

• Example of a formal secure development method

• Secure architectural, design and implementation patterns
Introduction

We do not simply write code, and then as an afterthought test and patch it to ensure that it fulfills a functional requirement:

*If we want a piece of code that sums integers, then we state this before we start coding and specifically write the code to sum integers. We do not randomly write code and then try and patch the code to sum integers.*
Introduction

For non-functional requirements such as quality and security, the same logic applies: *We do not patch a piece of code to ensure it fulfills a non-functional requirement.*

- Non-functional requirements are met not only by stating the requirements, but activities are required.
- Security considerations must permeate all phases of the software development life cycle.
Software Development Life Cycle

1. Requirements
   - Gather requirements and use cases

2. Architecture and Design
   - Plan how the system shall work and how code should be written

3. Implementation
   - Code and make test plans

4. Verification
   - Test and ensure that requirements and design are fulfilled

5. Release & Maintenance
   - Release, patch, release, patch, ...

LiU EXPANDING REALITY
Software Development Life Cycle

- **Gather requirements and use cases**
- **Risk analysis**
- **Plan how the system shall work and how code should be written**
- **Code and make test plans**
- **Test and ensure that requirements and design are fulfilled**
- **Release, patch, release, patch, ...**

**Security requirements**

**Risk analysis**

**Risk-based security tests**

**Risk analysis and penetration testing**

**Static analysis**

**Architecture and Design**

**Implementation**

**Verification**

**Release & Maintenance**
Software Development Life Cycle

1. **Gather requirements and use cases**
2. **Plan how the system shall work and how code should be written**
3. **Code and make test plans**
4. **Test and ensure that requirements and design are fulfilled**

**Release & Maintenance**
- Release
- Patch
- Release
- Patch
- ...

**Static analysis**
- Risk-based security tests
- Risk analysis
- Security requirements

**Architecture and Design**

**Implementation**

**Verification**
Security Requirements

• Requirements are gathered during the initial phase of the software development life cycle.

• This is an opportunity to not only gather functional requirements, but also security requirements.

• Several methods exists for gathering security requirements.

• We will look at misuse cases, which can be seen as a method in itself, but also takes part in more elaborate methods (such as SQUARE).
Use cases and Misuse cases

- A use case illustrates required usage of a system – i.e. expected functionality.

- However it is equally important to illustrate how one should *not* be able to use the system.

- Misuse cases are used to identify threats and required countermeasures.
3.1 Examples of use of the extended notation

Emergency access control in healthcare systems

Figure 3 depicts an example misuse case model using the extended notation proposed in this paper. The model illustrates use and misuse of the access control mechanism in an Electronic Patient Record (EPR) system. As explained in the introduction, such healthcare systems often have emergency access control mechanisms designed to be able to override the standard access control mechanisms in situations where access to information is of vital importance but there is no time to register the patient in the system and link him/her to a specific ward - which is necessary for the standard access control to function properly. In these situations healthcare personnel are authorized, by their organization and the law, to use the emergency access control mechanism to gain access to information that they have a legitimate need and right to view. However, for such an emergency mechanism to be useful, it has to be available at all times. This effectively leads to a backdoor into the system that may be misused by insiders to snoop around when they should not. Most system users will not attempt misusing this mechanism although it is possible. But, it is important to be able to consider the possibility and map out potential consequences and apply proper countermeasures if the consequences are grave. And that is the reason why this addition to the misuse case notation is important. You cannot get a complete overview of potential risks and threats towards a system if you do not consider the complete picture.

By identifying emergency access as a vulnerability we are also able to consider proper countermeasures to apply in order to minimize the risk for misuse - in this case auditing (enables traceability and detection of misuse) and awareness training (e.g. making sure that system users are aware of the consequences of misuse - and what is considered misuse).
Misuse case example

- **Electronic Patient Record (EPR)**
  - Under normal circumstances patients should be registered in the system and linked to a specific ward – only personnel with access to the patients at this ward can then read the patients records.
  - During emergencies the organization and the law allows the use of an emergency access control function – which gives immediate access to any records needed.
  - For such an emergency control to be useful, it must be available at all time. This effectively creates a backdoor in the system that insiders can use to snoop around.
  - By identifying emergency access as a vulnerability we can also consider proper countermeasures – **auditing** (enables traceability and detection) and **awareness training** (making sure that users are aware of consequences of misuse).
User input in web-enabled systems

In an IT-system all input, from users or other systems, should be handled with caution. Figure 4 illustrates a generic login procedure for a web application - the user has to enter a username and password to log in. Identified attacks include (but are definitely not limited to):

- Injection - for instance sql-injections to tamper with database content or override password check.
- Overflow - entering unexpected or large quantities of data in the input fields to observe system reaction or possibly take control over the system.

Input validation is identified as a countermeasure that helps mitigate these threats. This model illustrates how the extended notation helps highlight vulnerabilities that may be exploited. An insider is not included because these attacks are typically performed by outside attackers. Highlighting vulnerabilities in this way may be particularly helpful in a risk analysis process, where the customers are involved. By visualizing vulnerabilities, attacks and what may happen it will hopefully be easier to get acceptance and resources to apply security measures.

An insider on the system development team

This example illustrates how the extended notation may be used not only on a system level, but also on a business- or organizational level. An insider may exist inside a development team.
User input in web-based systems

Authorized user

Enter username

Enter password

Use system

Injection attack

Input validation

Overflow attack

Attacker

<<exploit>>

<<exploit>>

<<exploit>>

<<threaten>>

<<threaten>>

<<mitigate>>

<<mitigate>>

<<mitigate>>

<<mitigate>>

3.2 A step-by-step approach: how to apply the extended notation

In [15] Sindre and Opdahl propose guidelines, a set of steps, to perform when using misuse cases to elicit threats and countermeasures. The approach described
An insider on the system development team

System developer

Implement system

Insider

Inject backdoor

Inject bug

<<mitigate>>

<<mitigate>>

<<mitigate>>

Code audit

Security testing

Fig. 4. Extended misuse case example: user input

Team or an organization. For example, a disgruntled employee working on a development project may inject code into a system that opens up a backdoor that attackers may exploit like Figure 5 illustrates.

Fig. 5. Extended misuse case example: insider in development team

3.2 A step-by-step approach: how to apply the extended notation

In [15] Sindre and Opdahl propose guidelines, a set of steps, to perform when using misuse cases to elicit threats and countermeasures. The approach described...
Requirements

- Misuse cases is one method of gathering requirements.

- Other more complex methods exists that range up to full-fledge risk analysis methods.

- Misuse cases are good due to their simplicity, this increases the probability that they will be used.

- When requirements have been gathered they are transferred to the design and architecture phase.
Software Development Life Cycle

**Requirements**
- Gather requirements and use cases

**Architecture and Design**
- Plan how the system shall work and how code should be written

**Implementation**
- Code and make test plans

**Verification**
- Test and ensure that requirements and design are fulfilled

**Release & Maintenance**
- Release, patch, release, patch, ...

**Static analysis**
- Risk-based security tests

**Risk analysis**
- Risk analysis and penetration testing

**Security requirements**
Risk analysis

- Risk analysis is used at the **architecture & design** phase and at the **verification** phase (to some degree also at requirements stage)

- Helps to find and quantify risks and then allows us to change our architecture and design.

- We will look briefly at CORAS (more in Info-Sec course) and in more detail about Attack Trees (overlap with Info-Sec course).
CORAS (overview)

Step 1 – Experts and clients decide upon which system is to be analyzed and what parts of the system that should be focused upon.

Step 2 – The system to be analyzed is formalized, assets are identified, high-level risk analysis.
**CORAS (overview)**

<table>
<thead>
<tr>
<th>Consequence value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catastrophic</td>
<td>1000+ health records (HRs) are affected</td>
</tr>
<tr>
<td>Major</td>
<td>100-1000 HRs are affected</td>
</tr>
<tr>
<td>Moderate</td>
<td>10-100 HRs are affected</td>
</tr>
<tr>
<td>Minor</td>
<td>1-10 HRs are affected</td>
</tr>
<tr>
<td>Insignificant</td>
<td>No HR is affected</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Likelihood value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Certain</td>
<td>Five times or more per year (50-<em>: 10y = 5-</em>: 1y)</td>
</tr>
<tr>
<td>Likely</td>
<td>Two to five times per year (21-49: 10y = 2,1-4,9: 1y)</td>
</tr>
<tr>
<td>Possible</td>
<td>Once a year (6-20: 10y = 0,6-2: 1y)</td>
</tr>
<tr>
<td>Unlikely</td>
<td>Less than once per year (2-5: 10y = 0,2-0,5: 1y)</td>
</tr>
<tr>
<td>Rare</td>
<td>Less than once per ten years (0-1:10y = 0-0,1:1y)</td>
</tr>
</tbody>
</table>

**Step 3 – Prioritize assets, create scales for consequence and likelihood values, create risk evaluation matrix.**

<table>
<thead>
<tr>
<th>Asset</th>
<th>Importance</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Health records</td>
<td>2</td>
<td>Direct asset</td>
</tr>
<tr>
<td>Provision of telecardiology service</td>
<td>3</td>
<td>Direct asset</td>
</tr>
<tr>
<td>Public’s trust in system</td>
<td>(Scoped out)</td>
<td>Indirect asset</td>
</tr>
<tr>
<td>Patient’s health</td>
<td>1</td>
<td>Indirect asset</td>
</tr>
</tbody>
</table>

**Consequence**

<table>
<thead>
<tr>
<th>Consequence</th>
<th>Insignificant</th>
<th>Minor</th>
<th>Moderate</th>
<th>Major</th>
<th>Catastrophic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Certain</td>
<td>Must be evaluated</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Likely</td>
<td>Acceptable</td>
<td>Acceptable</td>
<td>Acceptable</td>
<td>Acceptable</td>
<td>Must be evaluated</td>
</tr>
<tr>
<td>Possible</td>
<td>Acceptable</td>
<td>Acceptable</td>
<td>Must be evaluated</td>
<td>Must be evaluated</td>
<td>Must be evaluated</td>
</tr>
<tr>
<td>Unlikely</td>
<td>Acceptable</td>
<td>Acceptable</td>
<td>Must be evaluated</td>
<td>Must be evaluated</td>
<td>Must be evaluated</td>
</tr>
<tr>
<td>Rare</td>
<td>Acceptable</td>
<td>Acceptable</td>
<td>Must be evaluated</td>
<td>Must be evaluated</td>
<td>Must be evaluated</td>
</tr>
</tbody>
</table>

Images from Braber et al. – *Model-based security analysis in seven steps – a guided tour to the CORAS method*
**Step 4** – Create threat diagrams through structured brainstorming (workshop).

**Step 5** – Estimate risks (consequence and likelihood)

Images from Braber et al. – *Model-based security analysis in seven steps – a guided tour to the CORAS method*
CORAS (overview)

**Step 6** – Risk evaluation, estimates are confirmed or adjusted.

<table>
<thead>
<tr>
<th>Likelihood</th>
<th>Insignificant</th>
<th>Minor</th>
<th>Moderate</th>
<th>Major</th>
<th>Catastrophic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rare</td>
<td></td>
<td>CC1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unlikely</td>
<td></td>
<td></td>
<td>PR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Possible</td>
<td>CI1, SS2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Likely</td>
<td></td>
<td>SS1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Certain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Step 7** – Risk treatment

Images from Braber et al. – *Model-based security analysis in seven steps – a guided tour to the CORAS method*
Attack trees

Represent attacks against the system in a tree structure, with the goal as the root node and different ways of achieving that goal as leaf nodes.
Attack Trees

- **Open Safe**
  - Pick lock
  - Learn combo
  - Cut open
  - Install improperly
    - Find written combo
    - Get combo from target
      - Threaten
      - Blackmail
      - Eavesdrop
      - Bribe
        - Listen to conversation
        - Get target to state combo
Attack Trees

Open Safe

- Pick lock
- Learn combo
- Cut open
- Install improperly

- Find written combo
- Get combo from target

- Threaten
- Blackmail
- Eavesdrop
- Bribe

and

- Listen to conversation
- Get target to state combo
Attack Trees

- **Open Safe**
  - **P**
  - **I**
  - **P**
  - **I**
  - **P**
  - **I**
  - **P**
  - **I**

- Pick lock
- Learn combo
- Cut open
- Install improperly
- Find written combo
- Get combo from target
- Threaten
- Blackmail
- Eavesdrop
- Bribe

- **Listen to conversation**
- **Get target to state combo**

**and**
Attack Trees

- Open Safe
  - Pick lock
    - Find written combo
    - Threaten
  - Learn combo
    - Get combo from target
  - Cut open
    - Get combo from target
  - Install improperly

- Listen to conversation
  - Get target to state combo

Costs:
- Pick lock: $30
- Learn combo: $20
- Cut open: $10
- Install improperly: $100
- Threaten: $60
- Blackmail: $100
- Eavesdrop: $60
- Bribe: $20
- Listen to conversation: $20
- Get target to state combo: $40
Attack Trees

- We can annotate the attack tree with many different kind of Boolean and continuous values:
  - “Legal” versus “Illegal”
  - “Requires special equipment” versus “No special equipment”
  - Probability of success, likelihood of attack, etc.

- Once we have annotated the tree we can query it:
  - Which attacks cost less than $10?
  - Legal attacks that cost more than $50?
  - Would it be worth paying a person $80 so they are less susceptible to bribes? (In reality you need to also consider the probability of success)
Attack Trees

• First you identify possible attack goals.
• Each goal forms a separate tree.
• Add all attacks you can think of to the tree.
• Expand the attacks as if they were goals downwards in the tree.

• Let somebody else look at your tree, get comments from experts, iterate and re-iterate.

• Keep your trees updated and use them to make security decisions throughout the software life cycle.
Software Development Life Cycle

- **Security requirements**
  - Gather requirements and use cases

- **Risk analysis**
  - Plan how the system shall work and how code should be written

- **Architecture and Design**
  - Code and make test plans

- **Implementation**
  - Test and ensure that requirements and design are fulfilled

- **Verification**
  - Risk analysis and penetration testing

- **Release & Maintenance**
  - Release, patch, release, patch, ...
Software Development Life Cycle

1. Requirements: Gather requirements and use cases.
2. Architecture and Design: Plan how the system shall work and how code should be written.
3. Implementation: Code and make test plans.
4. Verification: Test and ensure that requirements and design are fulfilled.
5. Release & Maintenance: Release, patch, release, patch, ...

- Security requirements
- Risk analysis
- Risk-based security tests
- Static analysis
- Risk analysis and penetration testing
- Static analysis
Software Development Life Cycle

- **Security requirements**
  - Gather requirements and use cases

- **Risk analysis**
  - Plan how the system shall work and how code should be written

- **Architecture and Design**
  - Code and make test plans

- **Implementation**
  - Test and ensure that requirements and design are fulfilled

- **Verification**
  - Risk-based security tests
  - Static analysis
  - Risk analysis and penetration testing

- **Release & Maintenance**
  - Release, patch, release, patch, ...
Software development process

• The software development life cycle is generic, can be modified to fit into any development process:
  • Iterative (SCRUM, Kanban, etc)
  • Waterfall

• Adopting a secure software development process entails adding the security **touchpoints** discussed.

• Examples of formal development processes that include security touchpoints are: SDL, TSP, CLASP.
SECURITY DEVELOPMENT LIFECYCLE (SDL)
Security Development Lifecycle (SDL)

If a software development project is determined to be subject to the security development lifecycle (SDL) then the team must successfully complete sixteen mandatory security activities to comply with the Microsoft SDL process.

-Simplified Implementation of the Microsoft SDL
Pre-SDL: Security training

- All members must receive appropriate **training** to **stay informed** about security basics and recent trends in security and privacy.

- Topics include:
  - Threat modeling (e.g. design implications)
  - Secure coding (e.g. buffer overruns, cross-site scripting)
  - Privacy (e.g. types of privacy-sensitive data)

- This is only the baseline training, specialization and advanced training may be necessary.
Phase 1: Requirements

- Specify security requirements for the application as it is designed to run in its planned operational environment.

- A project team must define quality gates (e.g. all compiler warnings must be fixed before committing code), these are defined for each phase of the development and are negotiated with a security advisor.

- **Bug bars** must be defined which can be seen as quality gates for the entire project, e.g. no known vulnerabilities in the application with a “critical” or “important” rating at time of release.
Identify functional aspects of the software that require deep review:

- Which portions of the project will require security design reviews before release?

- Which portions of the project will require penetration testing by a mutually agreed upon group that is external to the project team?

- What is the privacy impact rating?
  - P1: High privacy risk, e.g. installs software
  - P2: Moderate privacy risk, e.g. one-time user initiated data-transfer
  - P3: Low privacy risk, e.g. no anonymous or personal data is transferred
Phase 2: Design

- All design specifications should describe how to securely implement all functionality provided by a given feature or function:
  - Attack surface reduction (giving attackers less opportunity to exploit a potential weak spot).
  - Threat modeling (risk analysis) of components or features that have meaningful security risks (can be defined by the security risk assessment during requirements).
  - Secure design patterns (discussed later)
Phase 3: Implementation

- Publish a list of approved tools and their associated security checks, such as compilers/linker options and warnings.
- List is to be approved by external security advisor.
- Teams should analyze all functions and APIs that will be used in conjunction with a software development project and prohibit those that are determined to be unsafe.
- Once a prohibited list is defined, all code should be scanned for these functions and APIs and modified accordingly.
- Static analysis of code should be performed.
Phase 4: Verification

- Dynamic program analysis, monitor application problems with memory corruption, user privilege issues, etc.

- **Fuzz testing**, deliberately introduce malformed or random data to an application during dynamic analysis.

- Update threat model and attack surface analysis, account for any design or implementation changes to the system, and assure that any new threats/attack are reviewed and mitigated.
Phase 5: Release

- An incident response plan must be in place:
  - A first point of contact in an emergency.
  - On-call contacts with decision-making authority that are available 24-hours a day.
  - Security servicing plans for code inherited from other groups in the organization.
  - Security servicing plans for third-party code (and if appropriate the right to make changes).
Final security review: Includes an examination of threat models, tool output, performance against quality gates and bug bars.

- Pass FRS – Good to go.
- Pass FSR with exceptions – Issues that can be fixed in the next release.
- FSR with escalation – Go back and address whatever SDL requirement that is not fulfilled or escalate to executive management for decision.

- Release to manufacturing (RTM) or release to web (RTW) conditional on FSR.
- If at least one component has privacy rating P1 then a privacy advisor must certify that the privacy requirements are satisfied.
- All specifications, code, binaries, threat models, plans, etc. must be archived so that service can be done on the product at a later stage.
• Security advisors can request that for some critical software additional activities are completed, e.g.:
  • Manual code review
  • Penetration testing
  • Vulnerability analysis of similar applications

• SDL is not a “one-size-fits-all” process, teams must implement SDL in a fashion that is appropriate to time and resources.

• There exists variants, such as SDL for Agile.
SECURE DESIGN PATTERNS
Software Development Life Cycle

- **Requirements**: Gather requirements and use cases.
- **Architecture and Design**: Plan how the system shall work and how code should be written.
- **Implementation**: Code and make test plans.
- **Verification**: Test and ensure that requirements and design are fulfilled.
- **Release & Maintenance**: Release, patch, release, patch, ...

- **Security requirements**:
- **Risk analysis**:
- **Risk-based security tests**:
- **Static analysis**:
- **Risk analysis and penetration testing**:

- **LiU Expanding Reality**
Secure design patterns

• **Descriptions** or **templates** describing a general solution to a security problem that can be applied in many different situations.

• The design patterns are meant to **eliminate the accidental insertion of vulnerabilities** into code or to **mitigate the consequence of vulnerabilities**.

• Categorized by abstraction: **architecture**, **design** or **implementation**.
Categories

- **Architectural-level patterns:** Focus on high-level allocation of responsibilities between different components and define the interaction between those high-level components.
  - Privilege separation (PrivSep)

- **Design-level patterns:** Address problems in the internal design of a single high-level component.
  - Secure factory
  - Secure chain of responsibility

- **Implementation-level patterns:** Low-level security issues, applicable to specific functions or methods in the system.
  - Secure logger
  - Clear sensitive information
Privilege separation (PrivSep)

- **Intent:** Reduce the amount of code that runs with special privilege without affecting or limiting the functionality of the program.

- **Motivation:** In many applications, a small set of simple operations require elevated privileges, while a much larger set of complex and security error-prone operations can run in the context of normal privileged user.
Privilege separation (PrivSep)

- **Request from unauthorized user**

  - `root`
    - Open socket and listen for connections

  - `root`
    - Spawn a child process that has least possible privilege

  - `Unprivileged`
    - Authenticate (complex code)

  - `User`
    - Do some work as user

  - `root`
    - Spawn a child with the privileges of the authorized user

  - `Return identity`
Privilege separation (PrivSep)

- The majority of the code is run without elevated privileges.
- If there is a vulnerability and somebody gets control of the process, then they are confined within the same level of privilege.
- Extra testing, verification, reviews etc. can be focused on the code that runs with elevated privileges.
Secure Factory

- **Intent:** Separate the security dependent logic involved in creating an object from the basic functionality of the created object.

- **Motivation:** An application may make use of an object whose behavior is dependent on the privileges of the user running the application.
Secure Factory

AbstractSecureFactory
+getInstance() : AbstractSecureFactory
+getObject(givenCredentials : SecurityCredentials) : SomeObject

ConcreteteSecureFactory1
+getObject(givenCredentials : SecurityCredentials) : SomeObject

ConcreteteSecureFactory2
+getObject(givenCredentials : SecurityCredentials) : SomeObject

Getting SomeObject is done by making the call:
AbstractSecureFactory.getInstance().getObject(securityCredentials)

The returned object, SomeObject, is an object that operates with the correct privileges.
Secure Factory

- Inside the factory:
  1. Using the current concrete implementation of AbstractSecureFactory
  2. Look at security credentials that were passed in the call
  3. Create an instance of the appropriate concrete version of SomeObject
  4. Further specialise settings in SomeObject

![Diagram showing the hierarchy of SomeObject, including LowPrivilegeSomeObject, MidPrivilegeSomeObject, and HighPrivilegeSomeObject]
Secure Factory

- The caller and SomeObject does not have to contain logic for checking privileges. It is always returned by the factory, and the factory picks the SomeObject with correct behavior.

- It is easy to change the security credentials by changing which concrete factory is used. (This is similar to another pattern which we will not discuss in class).

- Concrete versions of SomeObject does not have to implement code for functions that are not callable by the level of privilege to which it is developed.
  - The LowPrivilegeSomeObject does not need to implement the Write function.
Secure Chain of Responsibility

- **Intent:** Decouple the logic that determines privileges from the portion of the program that is requesting the functionality.

- **Motivation:** Applications sometimes need to allow and disallow certain functions depending on the role of the user.
Secure Chain of Responsibility

Supply request and user credentials

Manager Report Generator
Handle request
Pass request if credential check fails

Sale Analyst Report Generator
Handle request
Pass request if credential check fails

Sales Intern Report Generator
Handle or reject request
Secure Chain of Responsibility

- The selection of functionality is hidden from the caller, it will be selected based on the user credentials.
- The caller is not aware of which handler has dealt with the request.
- Easy to change the behavior of the system (add/remove handlers). Can even be done dynamically at runtime by changing the links.
Secure Logger

- **Intent:** Prevent an attacker from gathering potentially useful information about the system from system logs and to prevent an attacker from hiding their actions by editing system logs.

- **Motivation:** System logs usually contain a great deal of information about the system itself and its users.
Secure Logger

Application → Log → Secure Logger

Protected data

Log Reader → Unprotected data

Log Viewer
Secure Logger

- Standard mechanisms for reading log files will not work as the data will be somehow encrypted.
- The reader is necessary to access log files, and it requires authentication and authorization.
- Any adversary that gets a hold of log files can not use their content.
- (A possible implementation could use existing disk encryption systems).
Clear Sensitive Information

- **Intent:** It is possible that sensitive information has been stored in reusable resources after a user session or application has run. Sensitive information should be cleared from reusable resources.

- **Motivation:** In many cases the action of returning a reusable resource to the pool of resources simply marks the resource as available. The contents of the resource are left intact until the resource is actually reused. This could potentially lead to leaking of private information.

  (Resources include files, memory allocations, etc.)
Clear Sensitive Information

Application → Scrub data → Release data

Do not simply release back

Pool

Get resource

Return to pool
Clear Sensitive Information

```cpp
ClientInfo::~ClientInfo() {
    this->ipAddr = 0;
    this->trustLevel = BOGUS;
    this->numFaultyRequests = 0;
}
```

An example of clearing sensitive information in the destructor of an object. In this way the information stored in memory is made insensitive before destroying the object.
Secure Design Patterns

• Secure design patterns are important for all developers, regardless of platform or language.
• Their main purpose is to:
  • Eliminate the accidental insertion of vulnerabilities into code or to mitigate the consequence of vulnerabilities.
• Using design patterns you are taking advantage of many years of learning from mistakes made by others, and you are using best practices.
• It also helps when communicating about code with other developers.
• There are many more very useful patterns:
Software Development Life Cycle

**Misuse cases**
- Gather requirements and use cases
- CORAS/Attack Trees
  - Risk analysis
  - Risk-based security tests
- Static analysis
- Risk analysis and penetration testing
- SDL

**Secure design patterns**
- CORAS/
  - Misuse cases
  - CORAS/
  - Secure design patterns
  - CORAS/
  - Security requirements
  - CORAS/
  - SDL

**Requirements and Design**
- Architecture and Design
- Implementation
- Verification
- Release & Maintenance
  - Release, patch, release, patch, ...

**Verifying**
- Code and make test plans
- Test and ensure that requirements and design are fulfilled