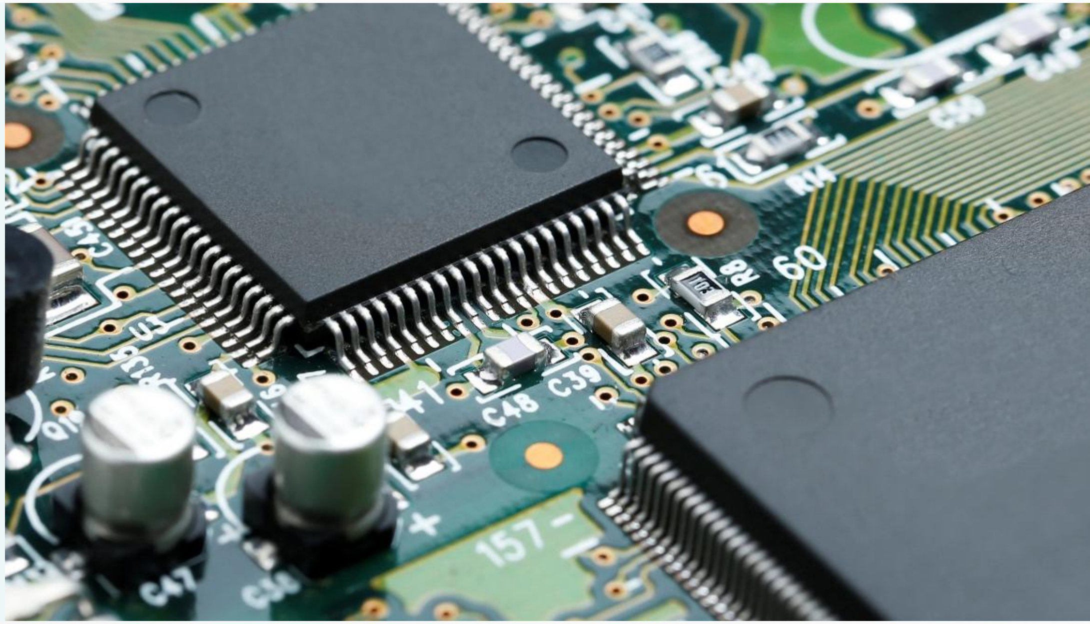
Cursul #9

Secure Software Development

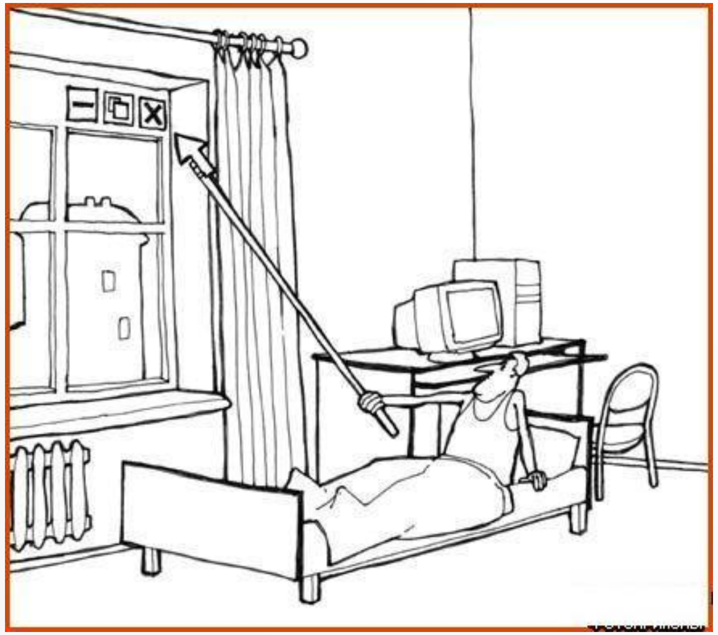
Intro



Introduction

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

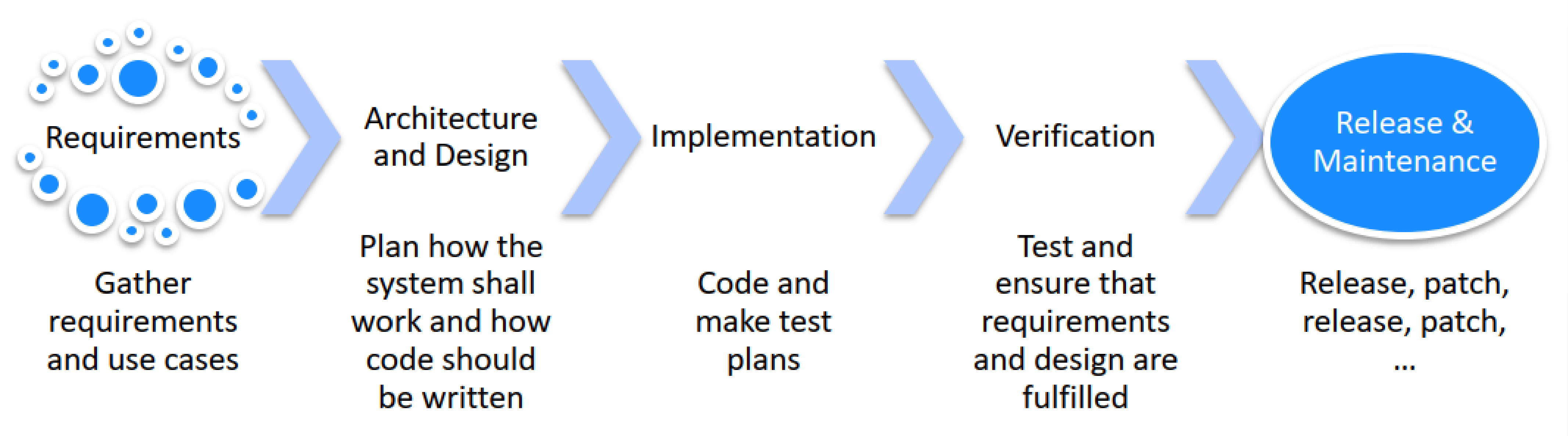
- If we want a piece of code that sums integers, then we state this before we start coding and specifically write the code to sum integers. We do not randomly write code and then try and patch the code to sum integers.



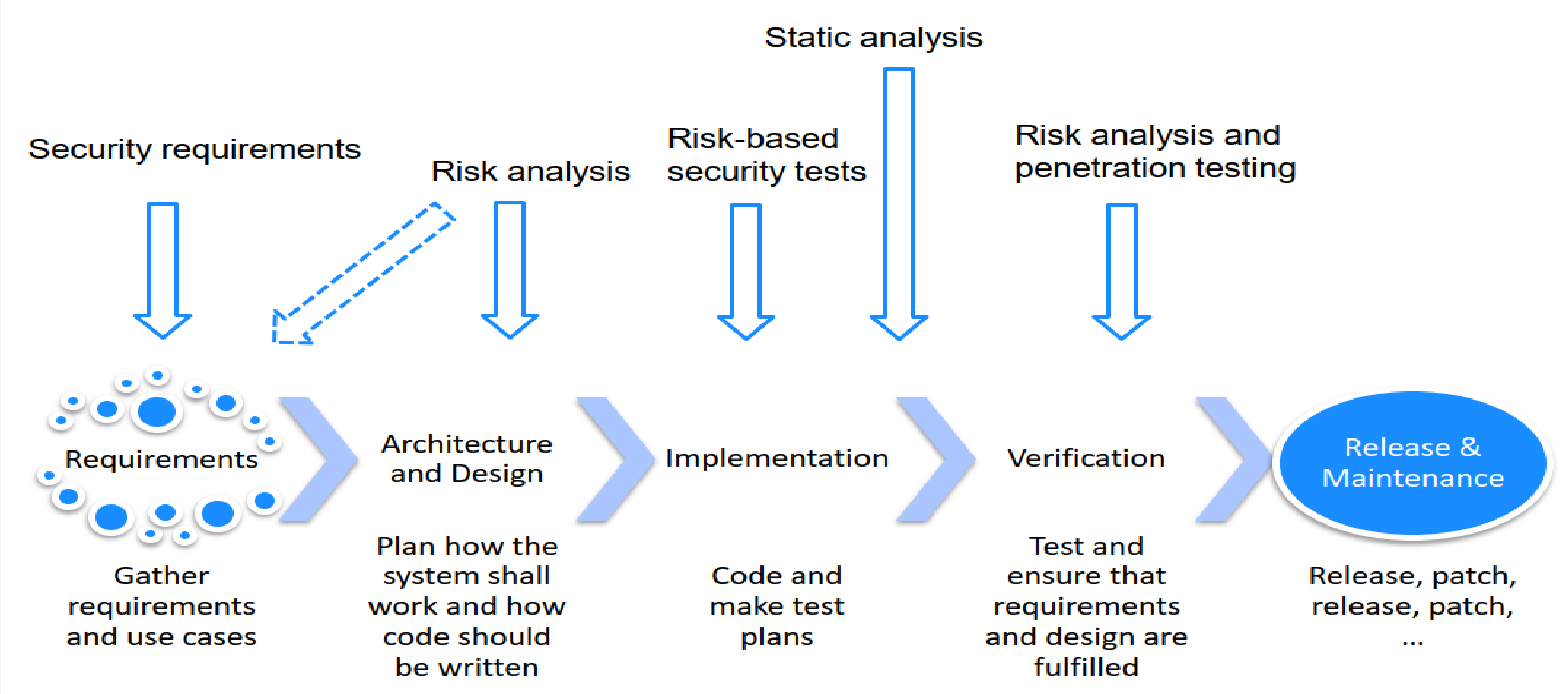
Introduction

- 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



Software Development Life Cycle



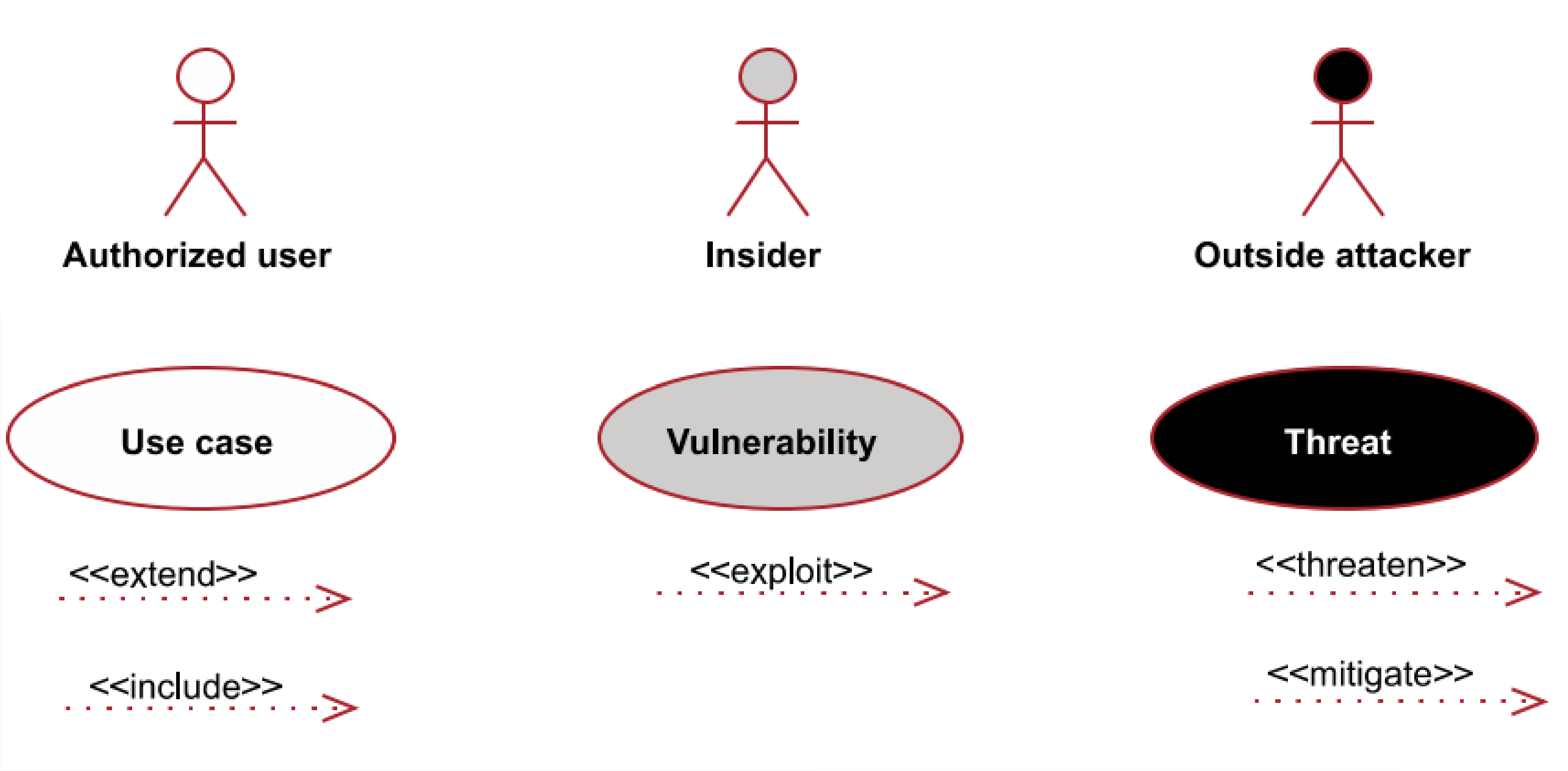
Security requirements

- Requirements are gathered during the initial phase of the software development life cycle.
- This is an opportunity to not only gather requirements on “business logic”, 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 can be used to **identify threats, assets, and** required **countermeasures**.

Misuse case legend



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).

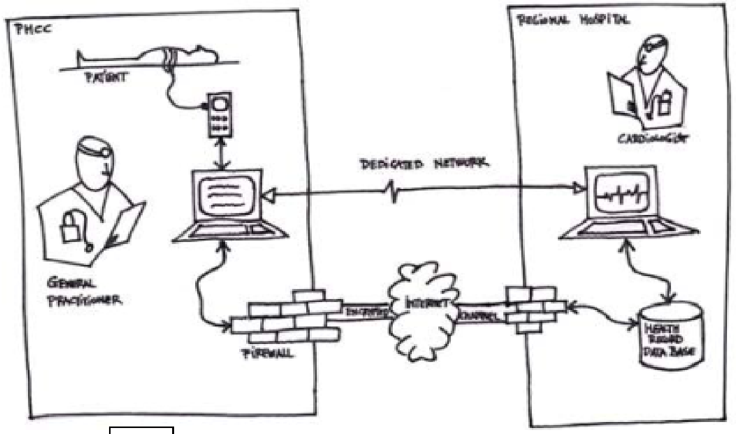
Requirements

- Misuse cases is one method of gathering requirements.
- Other more complex methods exists that range up to full-fledged 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.

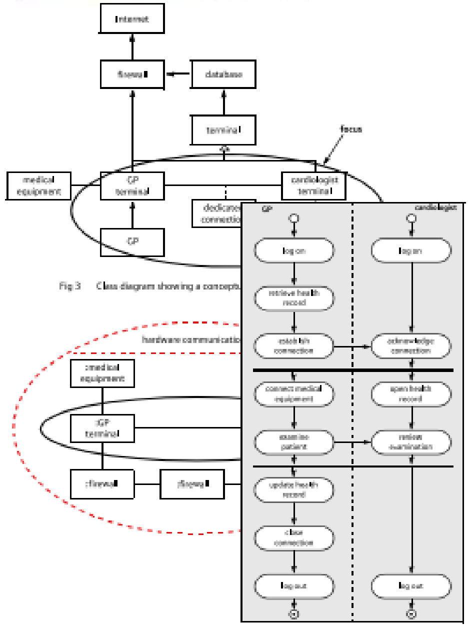
Risk analysis

- 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 and in more detail about Attack Trees.

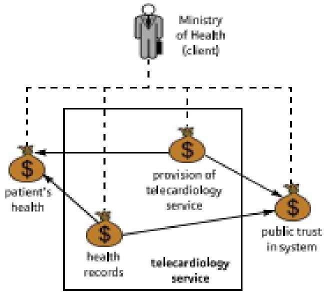
CORAS (overview)



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.



threat (accidental) / threat (deliberate) / threat (non-human)	threat scenario / unwanted incident / asset	vulnerability
Who/what causes it?	How? What is the incident? What does it harm?	What makes it possible?
Hacker	Breaks into the system and steals health records	Insufficient security
Employee	Sloppiness compromises confidentiality of health records	Insufficient training
Eavesdropper	Eavesdropping on dedicated connection	Insufficient protection of connection
System failure	System goes down during examination	Unstable connection/immature technology
Employee	Sloppiness compromises integrity of health record	Prose-based health records (i.e. natural language)
Network failure	Transmission problems compromise integrity of medical data	Unstable connection/immature technology
Employee	Health records leak out by accident — compromises their confidentiality and damages the trust in the system	Possibility of irregular handling of health records

CORAS (overview)

Consequence value	Description
Catastrophic	1000+ health records (HRs) are affected
Major	100-1000 HRs are affected
Moderate	10-100 HRs are affected
Minor	1-10 HRs are affected
Insignificant	No HR is affected

Asset	Importance	Type
Health records	2	Direct asset
Provision of telecardiology service	3	Direct asset
Public's trust in system	(Scoped out)	Indirect asset
Patient's health	1	Indirect asset

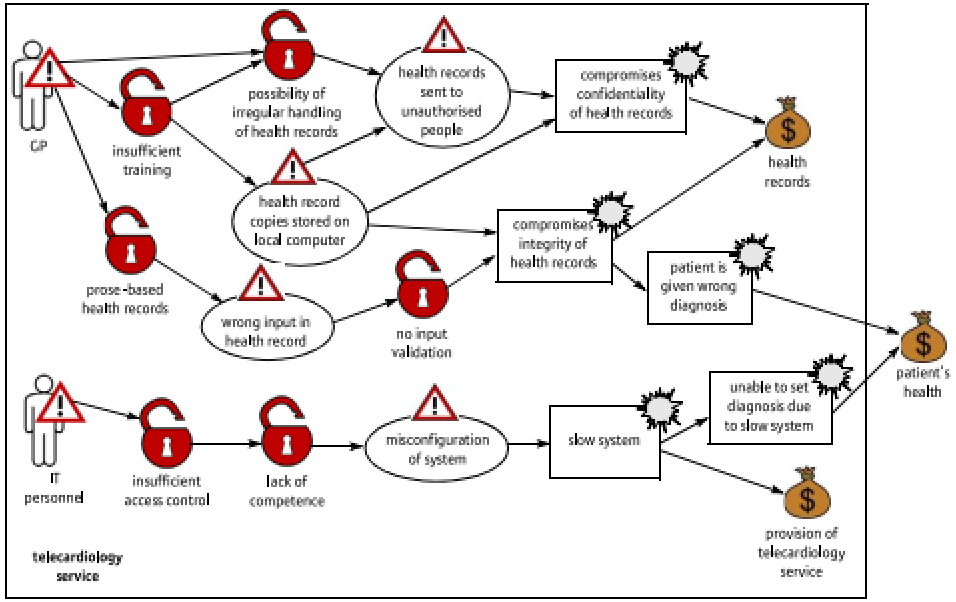
Likelihood value	Description[3]
Certain	Five times or more per year (50-*: 10y = 5-*: 1y)
Likely	Two to five times per year (21-49: 10y = 2,1-4,9: 1y)
Possible	Once a year (6-20: 10y = 0,6-2: 1y)
Unlikely	Less than once per year (2-5: 10y = 0,2-0,5: 1y)
Rare	Less than once per ten years (0-1:10y = 0-0,1:1y)

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

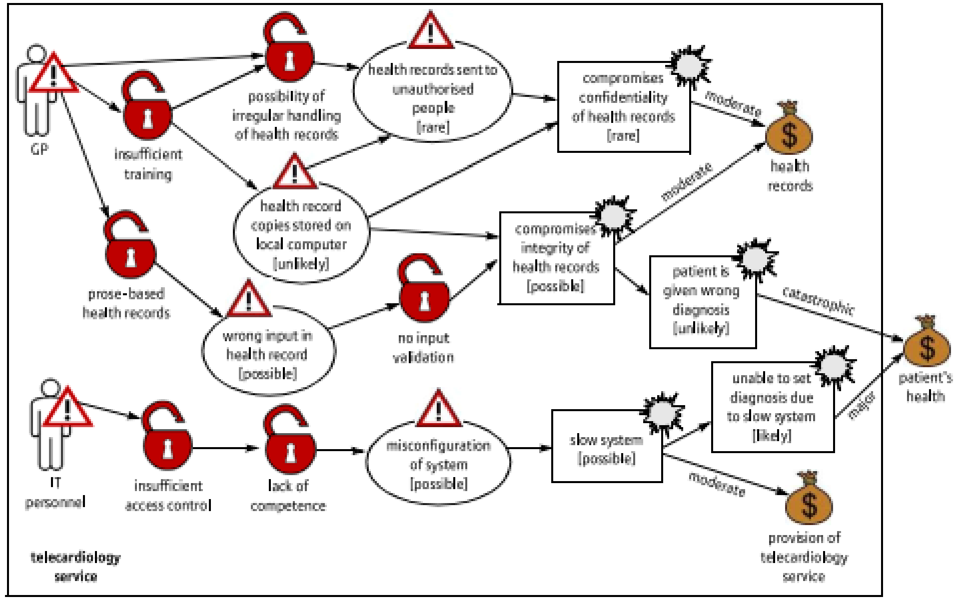
		Consequence				
		Insignificant	Minor	Moderate	Major	Catastrophic
Frequency	Rare	Acceptable	Acceptable	Acceptable	Acceptable	Must be evaluated
	Unlikely	Acceptable	Acceptable	Acceptable	Must be evaluated	Must be evaluated
	Possible	Acceptable	Acceptable	Must be evaluated	Must be evaluated	Must be evaluated
	Likely	Acceptable	Must be evaluated	Must be evaluated	Must be evaluated	Must be evaluated
	Certain	Must be evaluated	Must be evaluated	Must be evaluated	Must be evaluated	Must be evaluated

CORAS (overview)

Step 4 – Create *threat diagrams* through structured brainstorming (workshop).



Step 5 – Estimate risks (consequence and likelihood)

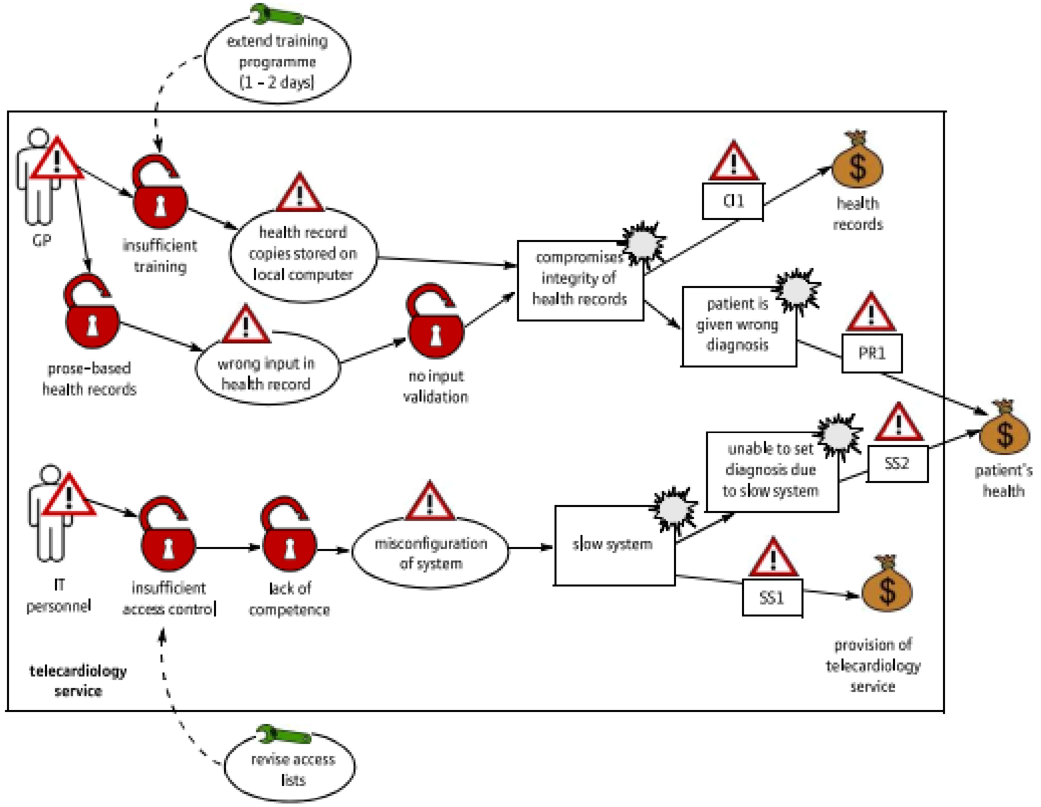


CORAS (overview)

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

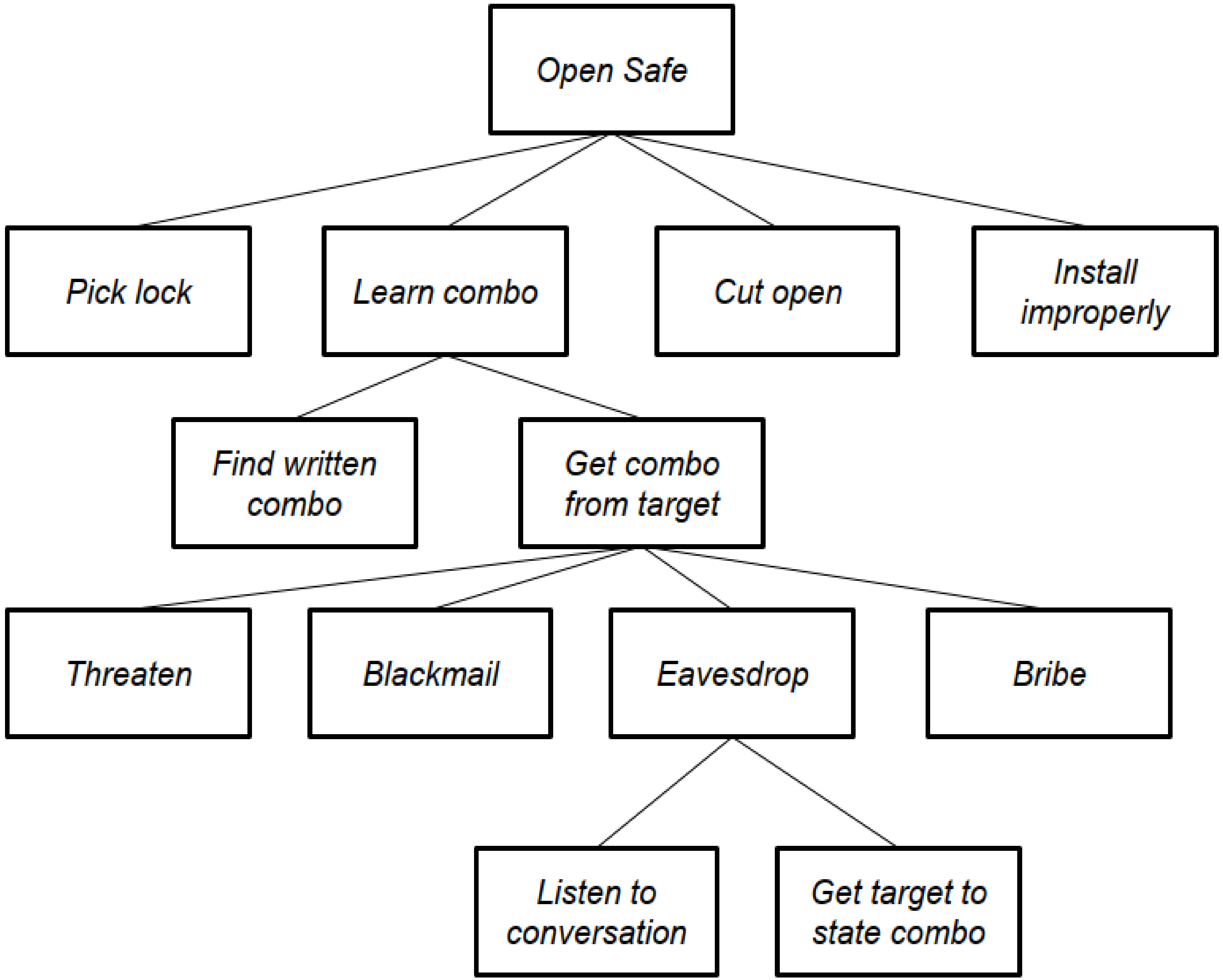
		Consequence				
		Insignificant	Minor	Moderate	Major	Catastrophic
Likelihood	Rare			CC1		
	Unlikely					PR1
	Possible			CI1, SS2		
	Likely				SS1	
	Certain					

Step 7 – Risk treatment

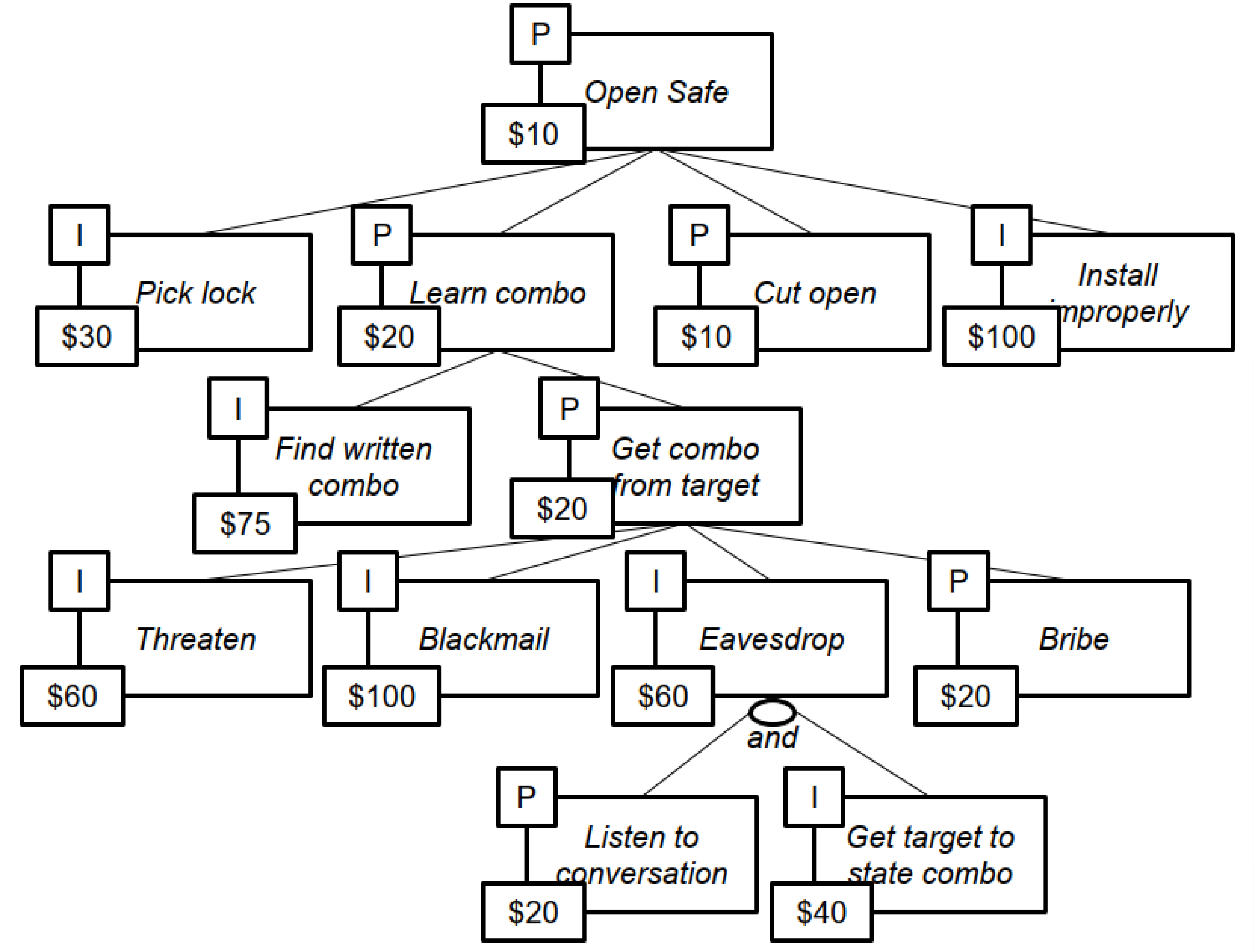


Attack trees

- 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



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)

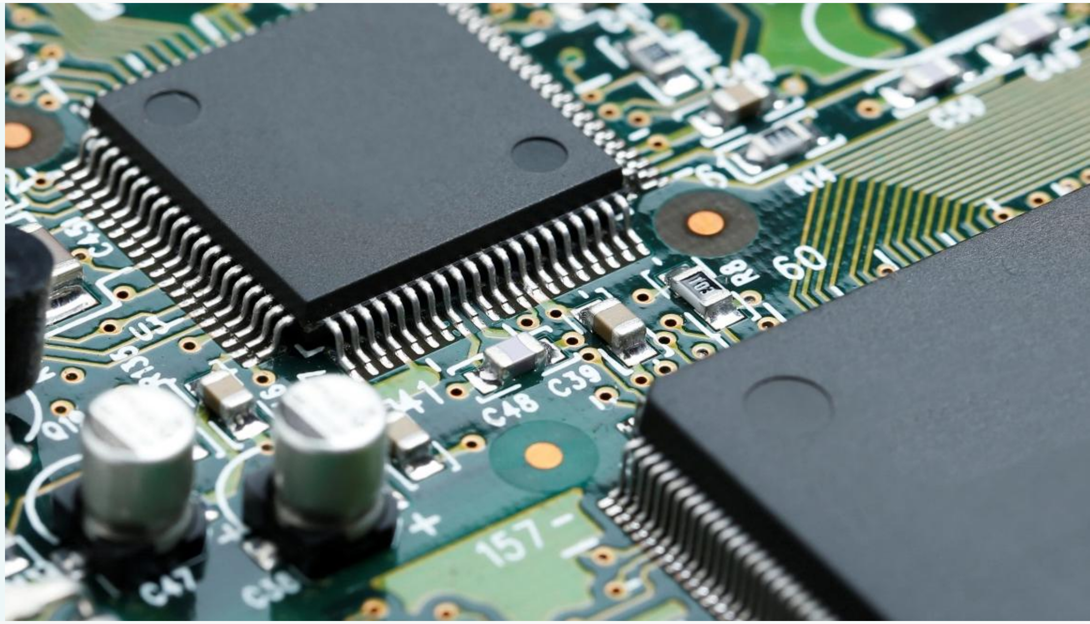
Attack trees

- 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 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.

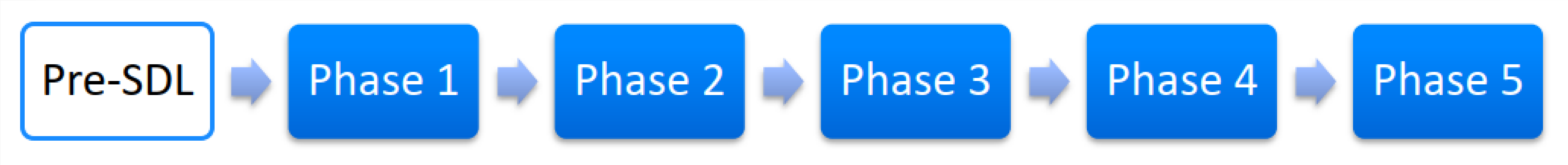
Security Development Lifecycle (SDL)



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.

Phase 1: Requirements (cont.)

- 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
    - Give attackers less opportunity to exploit a potential weak spot
    - For example, reduce amount of code that is exposed to untrusted inputs
  - Defense in depth
    - Design system with multiple layered **independent** defenses. (Design with the assumption that unknown vulnerabilities might exist.)
  - 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
  - Helps to catch certain bugs early
  - Helps to ensure that **secure coding policies** are followed

Example: SDL Banned C APIs

CharToOem, CharToOemA, CharToOemBuffA, CharToOemBuffW, CharToOemW, IsBadCodePtr, IsBadHugeReadPtr, IsBadHugeWritePtr, IsBadReadPtr, IsBadStringPtr, IsBadWritePtr, Makepath, OemToChar, OemToCharA, OemToCharW, StrCat, StrCatA, StrCatBuff, StrCatBuffA, StrCatBuffW, StrCatChainW, StrCatN, StrCatNA, StrCatNW, StrCatW, StrCpy, StrCpyA, StrCpyN, StrCpyNA, StrCpyNW, StrCpyW, StrLen, StrNCat, StrNCatA, StrNCatW, StrNCpy, StrNCpyA, StrNCpyW, _alloca, _fstrncat, _fstrncpy, _getts, _gettws, _i64toa, _i64tow, _itoa, _itow, _makepath, _mbccat, _mbccpy, _mbscat, _mbscpy, _mbslen, _mbsnbcat, _mbsnbcpy, _mbsncat, _mbsncpy, _mbstok, _mbstrlen, _snprintf, _sntprintf, _sntscanf, _snwprintf, _splitpath, _stprintf, _stscanf, _tccat, _tccpy, _tcscat, _tcscpy, _tcsncat, _tcsncpy, _tcstok, _tmakepath, _tscanf, _tsplitpath, _ui64toa, _ui64tot, _ui64tow, _ultoa, _ultot, _ultow, _vsnprintf, _vsntprintf, _vsnwprintf, _vstprintf, _wmakepath, _wsplitpath, alloca, gets, lstrcat, lstrcatA, lstrcatW, lstrcatn, lstrcatnA, lstrcatnW, lstrcpy, lstrcpyA, lstrcpyW, lstrcpyn, lstrcpynA, lstrcpynW, lstrlen, lstrncat, nsprintf, scanf, snscanf, snwscanf, sprintf, sprintfA, sprintfW, sscanf, strcat, strcatA, strcatW, strcpy, strcpyA, strcpyW, strcpynA, strlen, strncat, strncpy, strtok, swprintf, swscanf, vsprintf, vswprintf, wcscat, wcscpy, wcslen, wcsncat, wcsncpy, wcstok, wnsprintf, wnsprintfA, wnsprintfW, wscanf, wsprintf, wsprintfA, wsprintfW, wvnsprintf, wvnsprintfA, wvnsprintfW, wvsprintf, wvsprintfA, wvsprintfW

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).

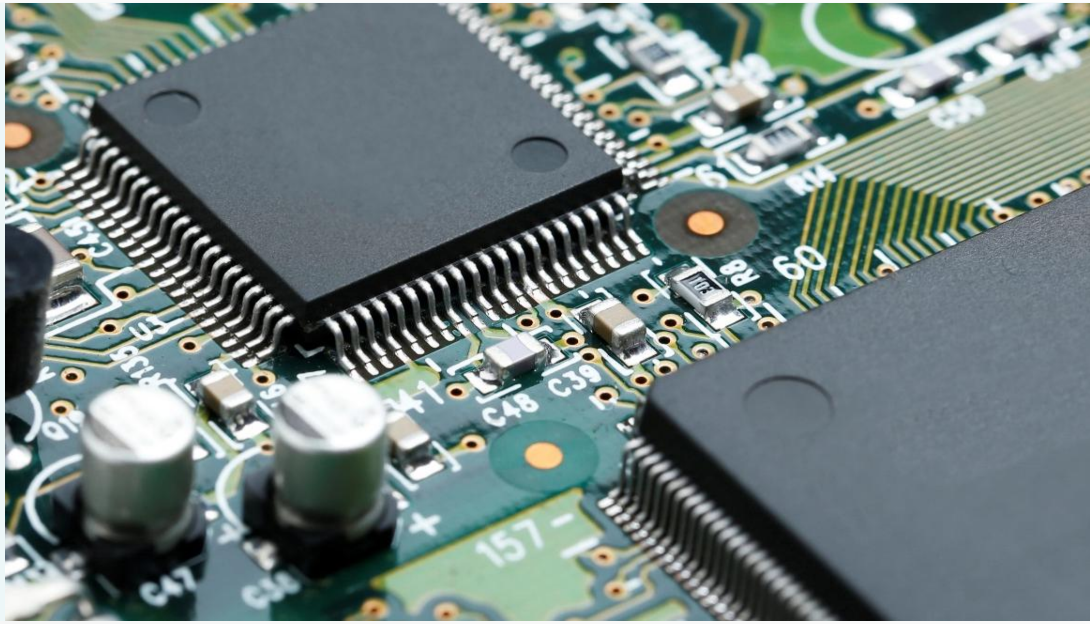
Phase 5: Release (cont.)

- **Final security review**: Includes an examination of threat models, tool output, performance against quality gates and bug bars.
  - Pass FSR – 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 product 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 Development Lifecycle (SDL)

- 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



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.

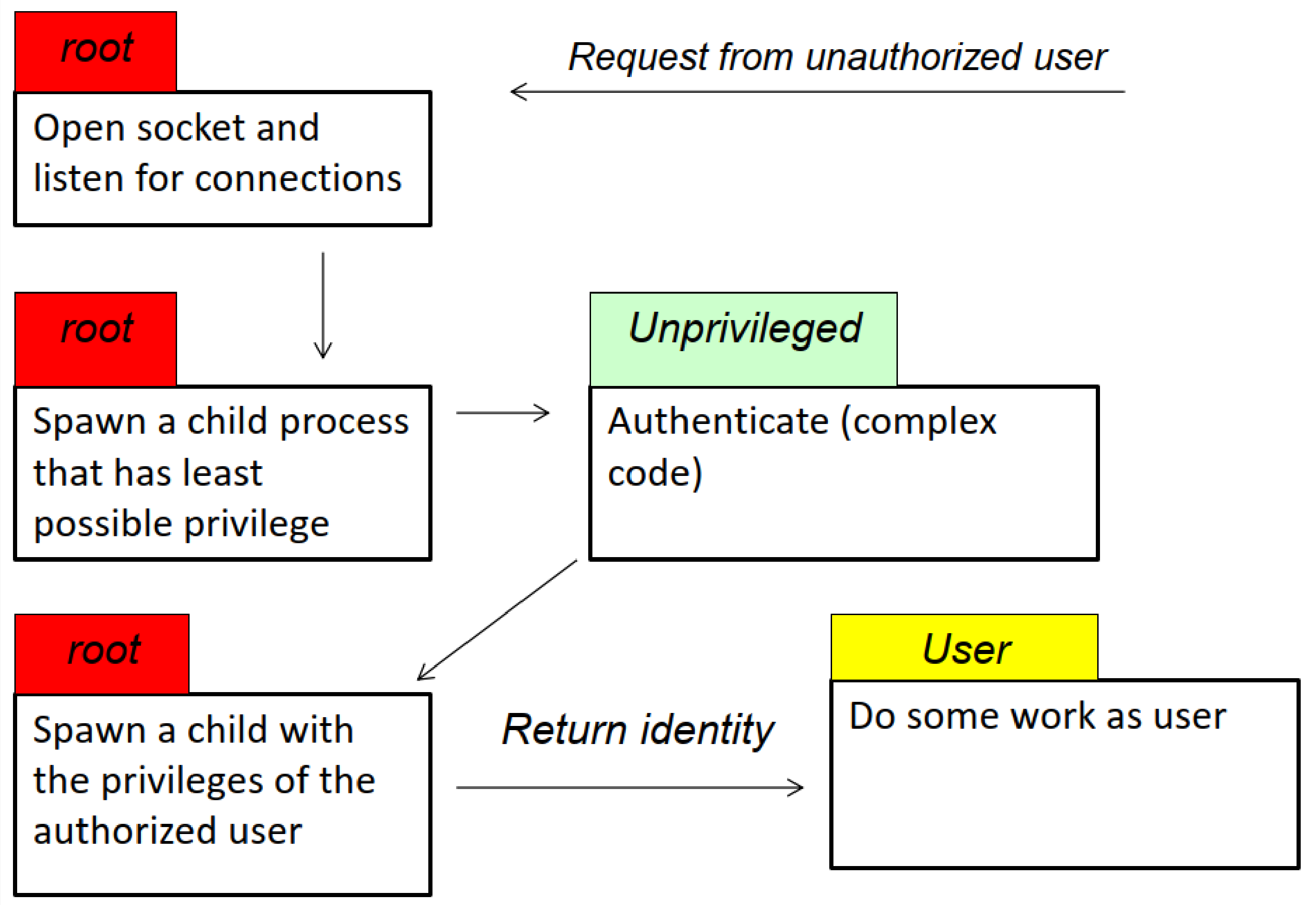
Categories

- **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
  - sClear sensitive information

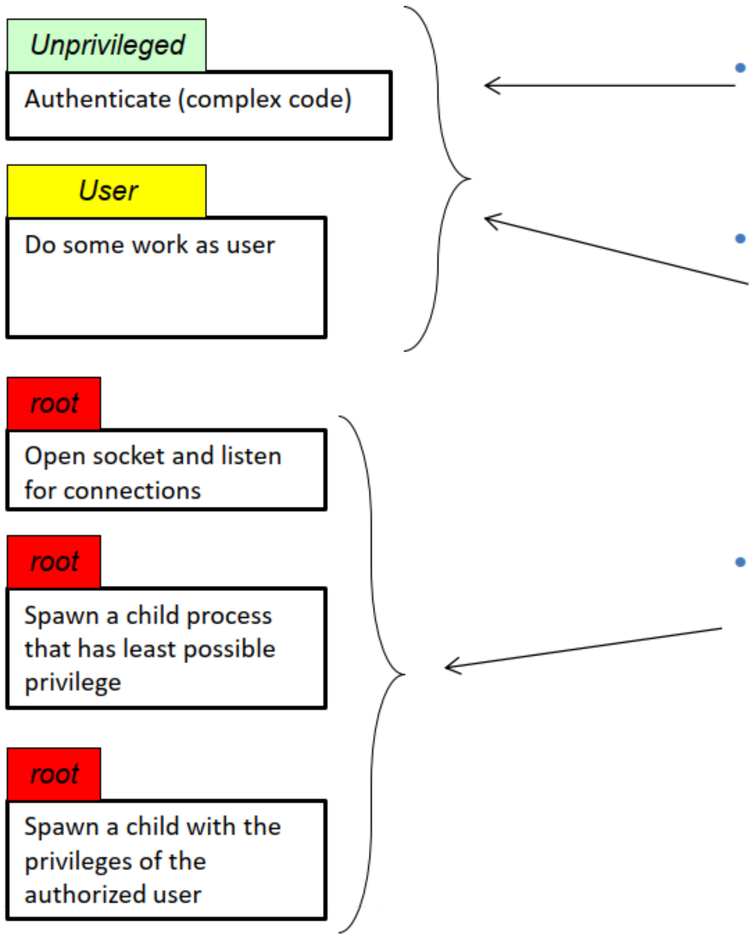
Privilege separation (PrivSep)

- **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.
  - PrivSep is applicable when high-privilege and low-privilege code can be separated into separate components, and there is a small amount of interaction between these components
  - If continuous interaction is required **Defer to Kernel** (see course literature) is more appropriate

Privilege separation (PrivSep)



Privilege separation (PrivSep)

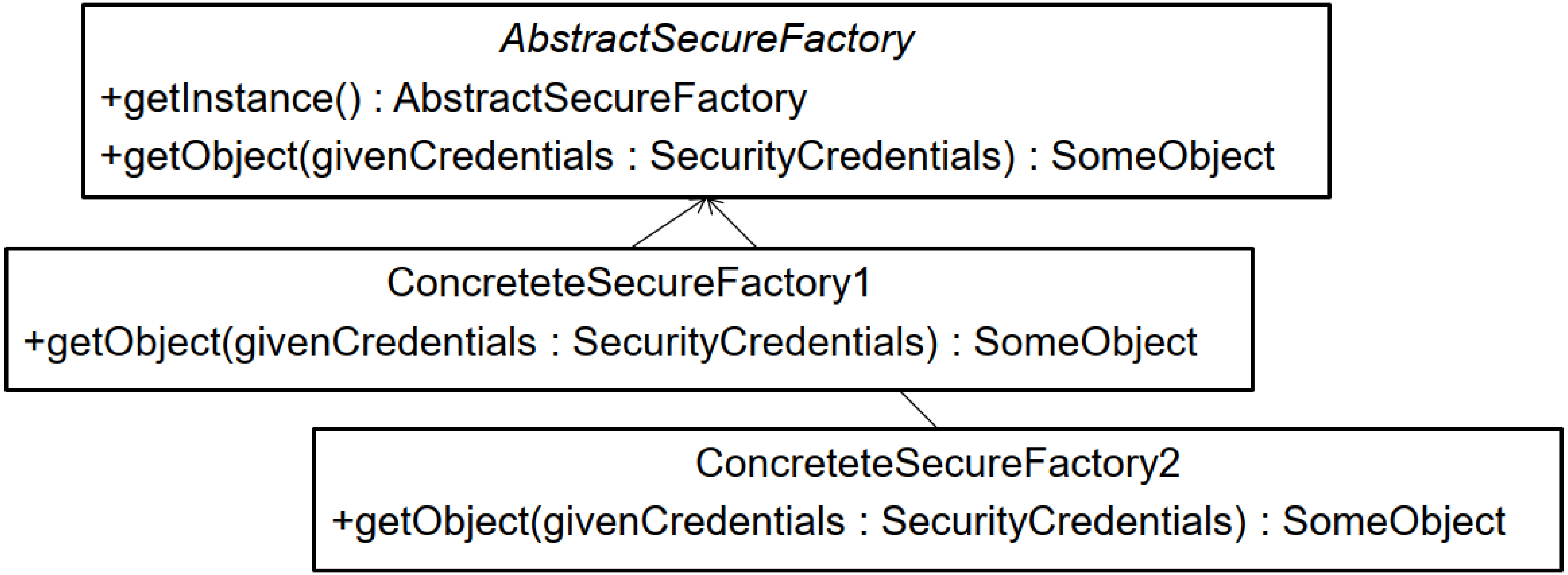


- The majority of the code is run without elevated privileges.
- If there is a vulnerability and somebody gets control of the process, then they are confined within the same level of privilege.
- Extra testing, verification, reviews etc. can be focused on the code that runs with elevated privileges.

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



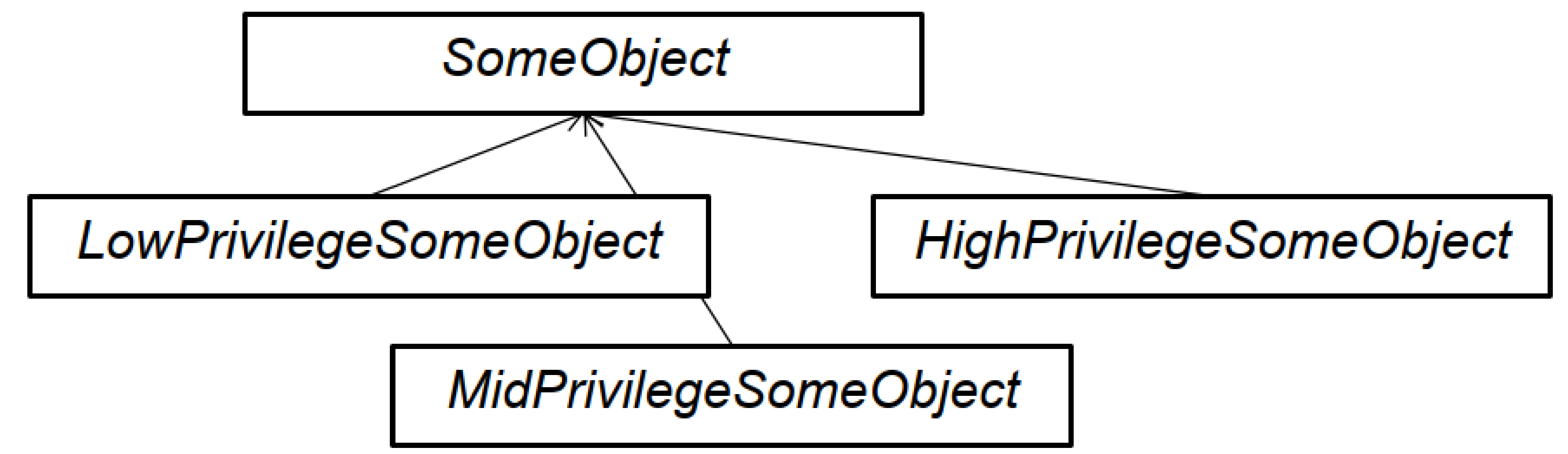
Getting SomeObject is done by making the call:

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



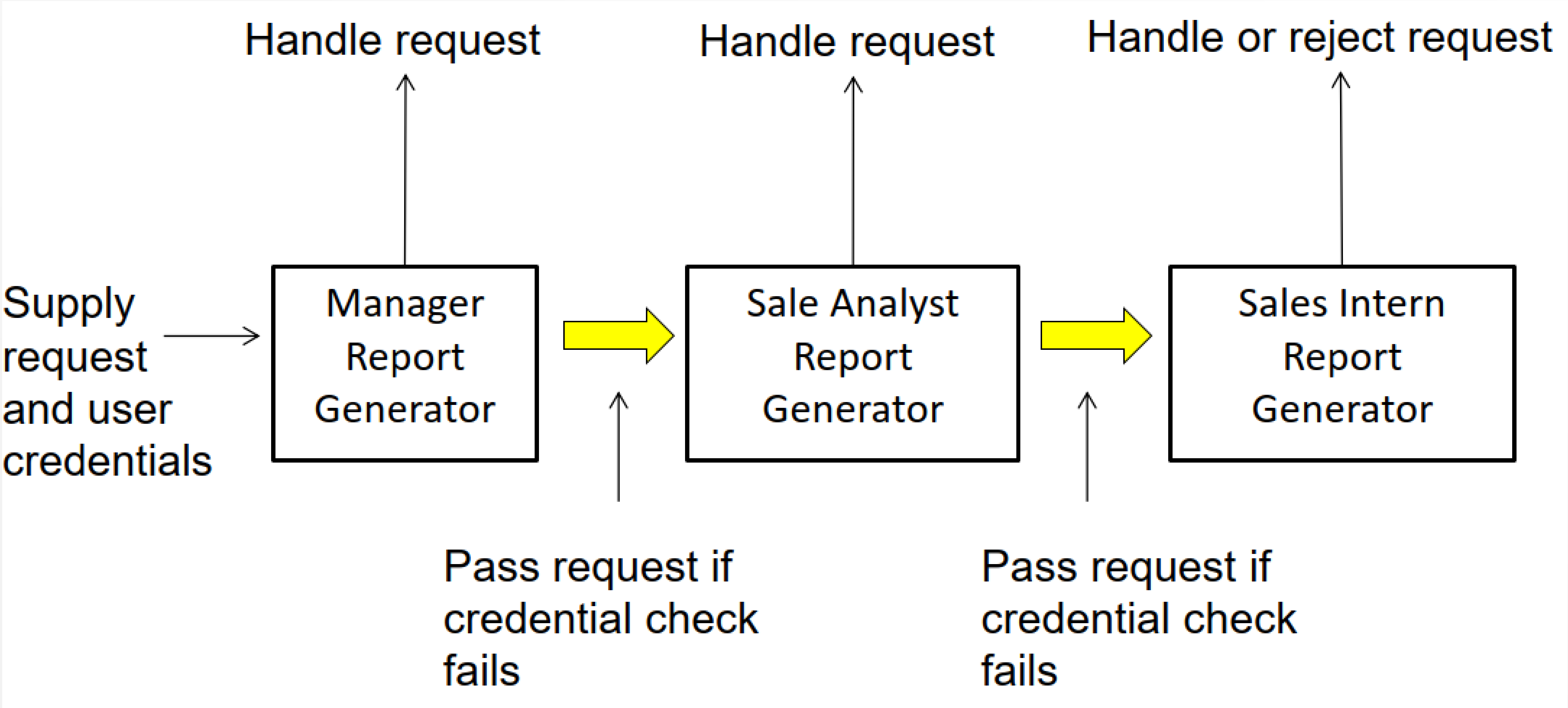
Secure Factory

- 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.
- 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

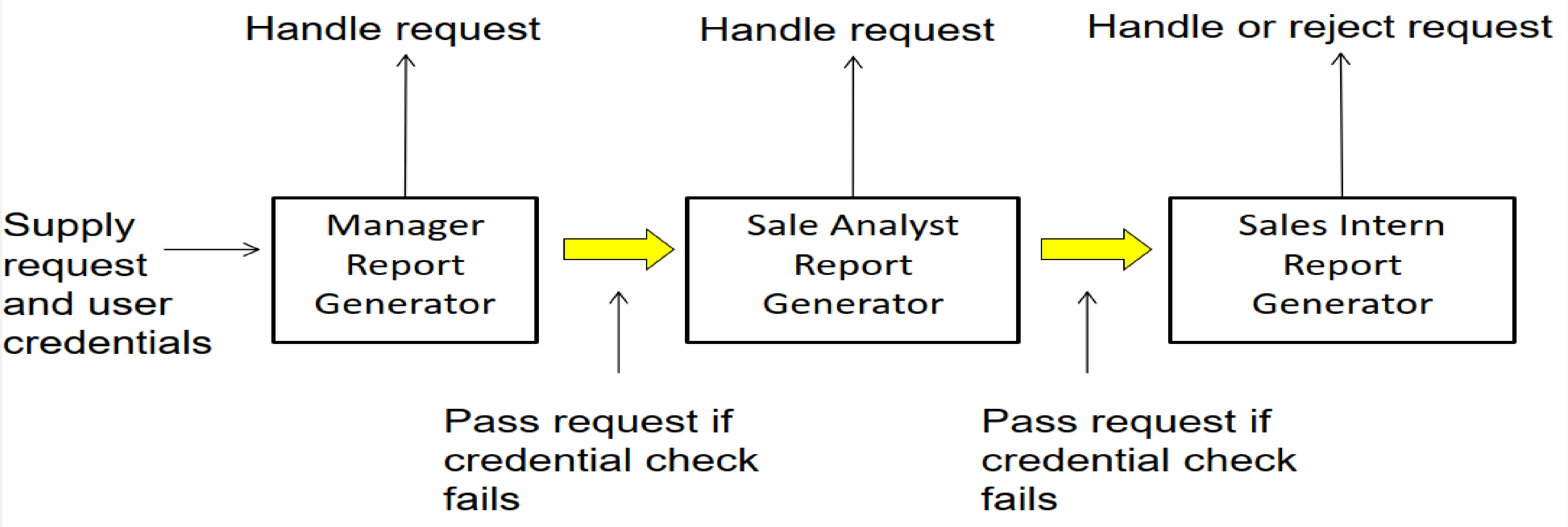
- **Intent**: Decouple the logic that determines privileges from the portion of the program that is requesting the functionality.
- **Motivation**: Applications sometimes need to allow and disallow certain functions depending on the role of the user.

Secure Chain of Responsibility



Secure Chain of Responsibility

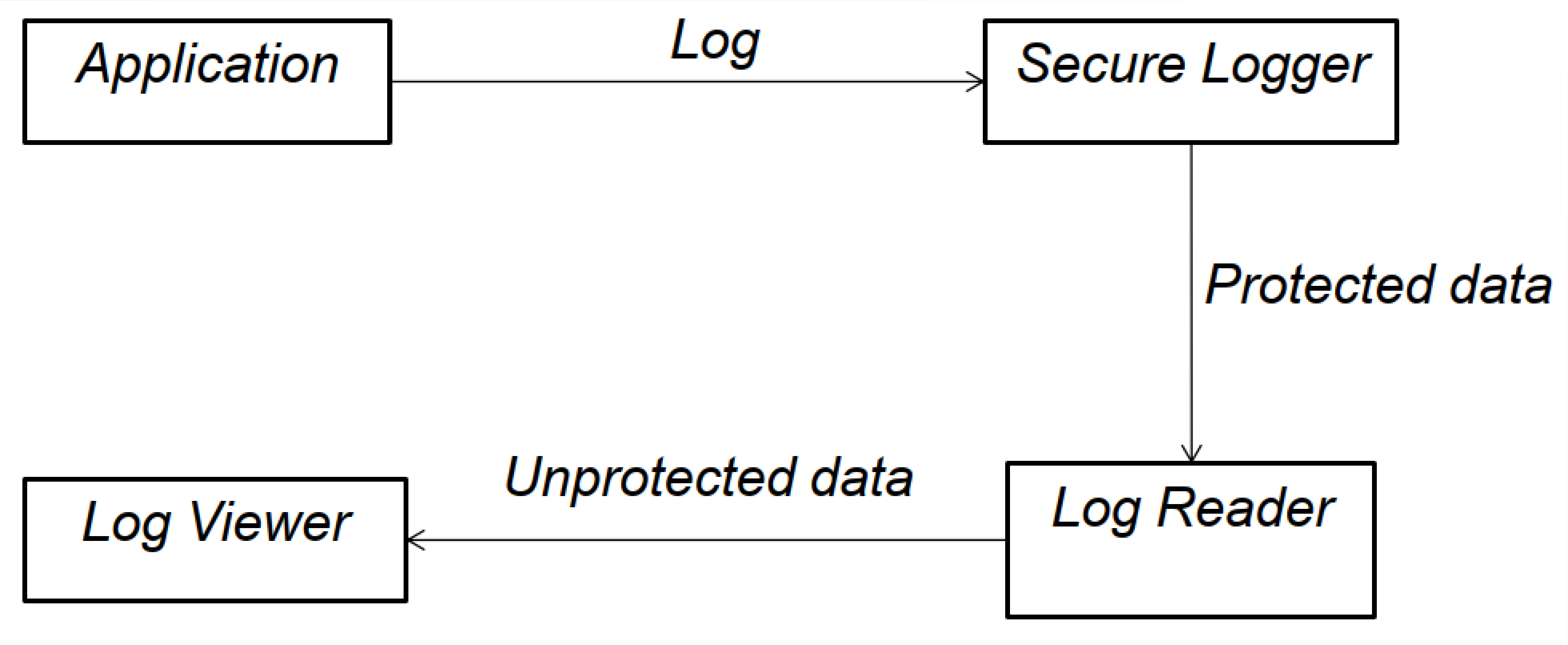
- The selection of functionality is hidden from the caller, it will be selected based on the user credentials.
- The caller is not aware of which handler has dealt with the request.
- Easy to change the behavior of the system (add/remove handlers). Can even be done dynamically at runtime by changing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



Secure Logger

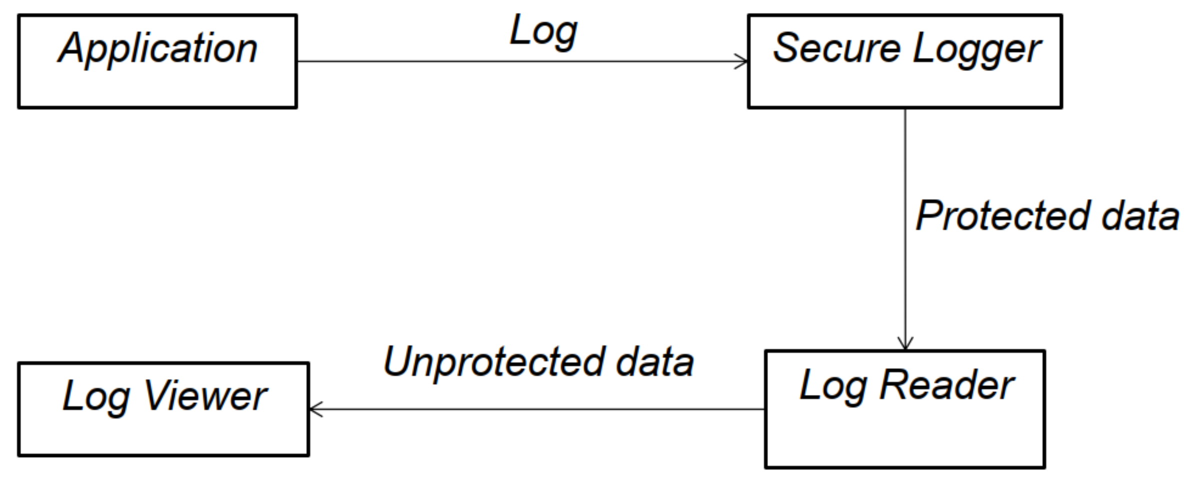
- **Intent**: Prevent an attacker from gathering potentially useful information about the system from system logs and to prevent an attacker from hiding their actions by editing system logs.
- **Motivation**: System logs usually contain a great deal of information about the system itself and its users.

Secure Logger



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).

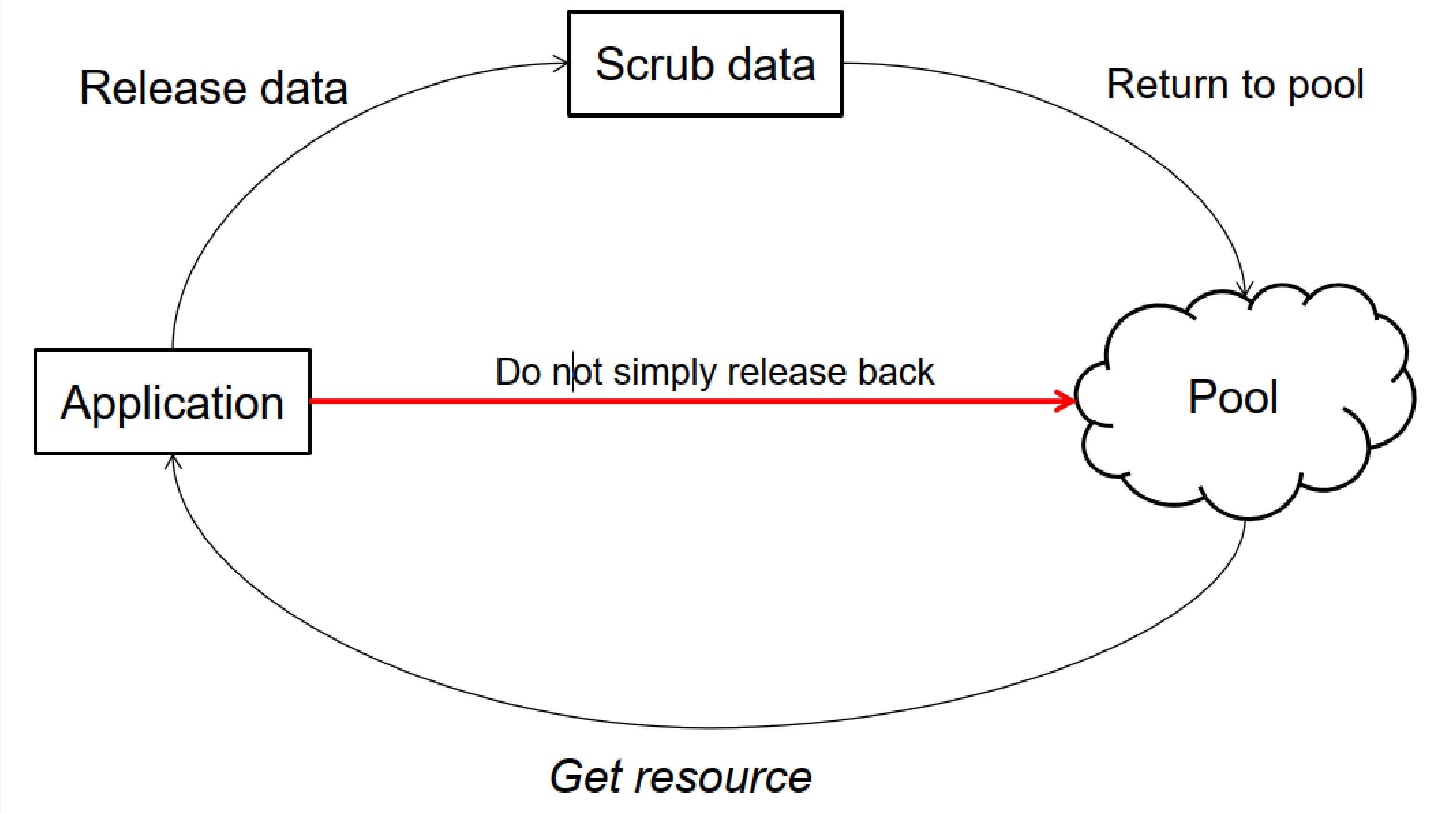


Clear Sensitive Information

- **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



Clear Sensitive Information

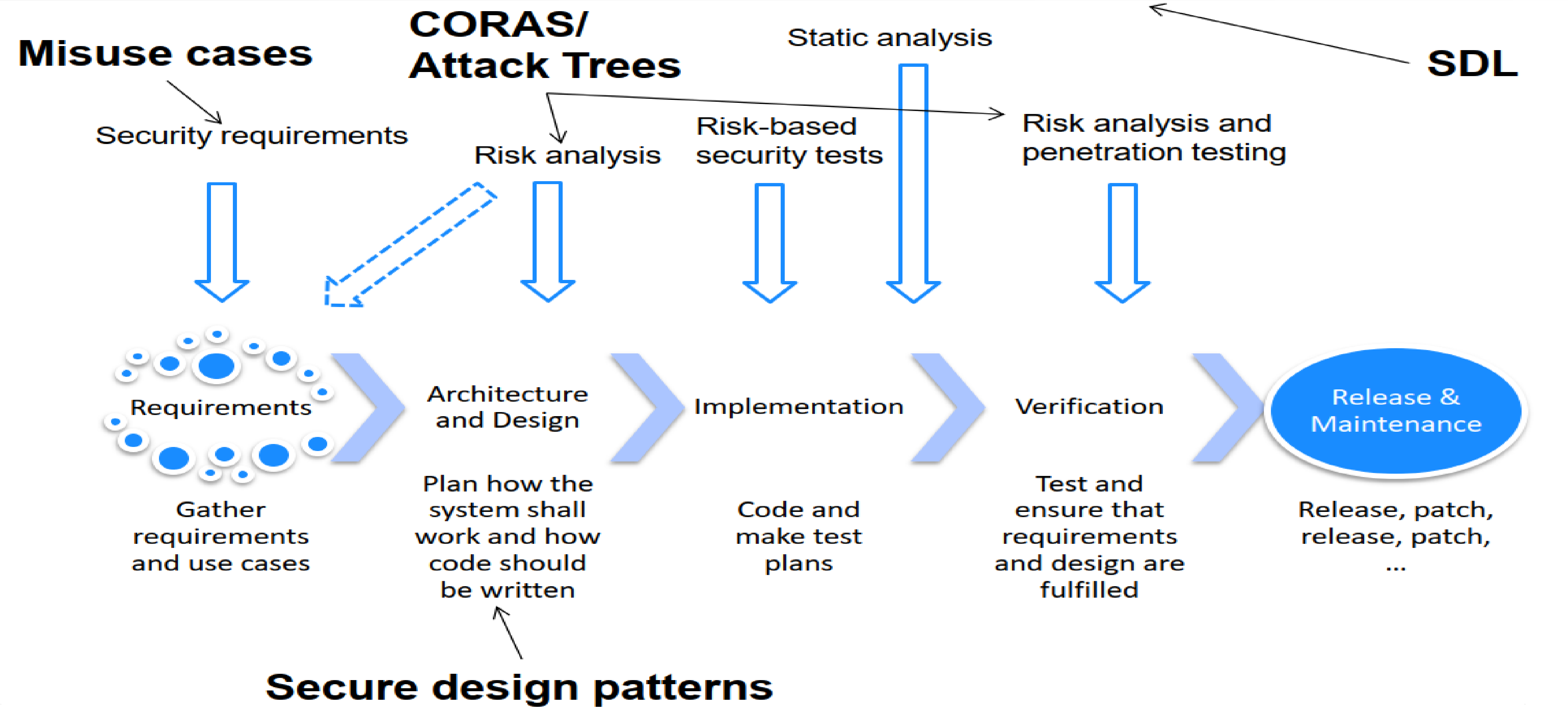
```
ClientInfo::~ClientInfo() {
 this->ipAddr = 0;
 this->trustLevel = BOGUS;
 this->numFaultyRequests = 0;
}
```

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:
  - C. Dougherty, K. Sayre, R. C. Seacord, D. Svoboda, K. Tagashi. Secure Design Patterns. Technical Report CMU/SEI-2009-TR-010.

Software Development Life Cycle



SS
automotive crunch

Q&A