NTNU

Norwegian University of
Science and Technology

Finding Security Patterns to Countermeasure Software Vulnerabilities

Ole Gunnar Borstad

Master of Science in Computer Science

Submission date: May 2008

Supervisor: Torbjørn Skramstad, IDI
Co-supervisor: Per Håkon Meland, Sintef IKT
Lillian Røstad, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Increasing connectivity and complexity of software make security more important than ever. Despite this, security is often an afterthought governed by an ad hoc penetrate and patch paradigm. Security is rarely taken into account throughout the entire software development lifecycle. There are however methods and security knowledge available that can help developers build more secure software. The increase in reported software vulnerabilities shows that these are not being utilised efficiently. Security modelling is a method that can help software developers prevent security problems at an early stage of development through identification of threats, vulnerabilities and countermeasures. Security patterns are known solutions to recurrent security problems.

The student is to explore how security modelling can be used to utilise expert security knowledge such as security patterns to include security throughout the development lifecycle.

Assignment given: 15. January 2008
Supervisor: Torbjørn Skramstad, IDI

Abstract

Software security is an increasingly important part of software development as the risk from attackers is constantly evolving through increased exposure, threats and economic impact of security breaches. Emerging security literature describes expert knowledge such as secure development best practices. This knowledge is often not applied by software developers because they lack security awareness, security training and secure development methods and tools. Existing methods and tools require too much effort and security is often given less priority in the trade-off between functionality and security.

This thesis defines a tool supported approach to secure software analysis and design. Possible vulnerabilities and their causes are identified through analysis of software specifications and designs, resulting in vulnerability cause graphs. The security modelling tool SeaMonster is extended to include security activity graphs; this technique is used with vulnerability cause graphs to model vulnerabilities and security improvement activities. A security activity graph is created to identify activities that keep the vulnerabilities from instantiating in the final software product. The activities in the security activity graph can be the use of security patterns. This way the above approach is used to find a security pattern as a countermeasure to a vulnerability, and can be used with the security pattern design templates implemented in a preliminary project (13). This is a way of providing coupling between security expertise and software developers to apply security knowledge in software development practice. The approach and tools are tested and demonstrated through a development case study of a medical patient journal system.

The main contributions of this thesis are an approach to secure software analysis and design, an extension of the security modelling tool SeaMonster, a case study of the approach and tools that show how security can be incorporated in early stages of software development. The contributions are intended to improve availability of security knowledge, to increase security awareness and bridge the gap between software experts and software developers.

Preface

This Master thesis extends the work of (13) and concludes my MSc in Computer Science at the Department of Computer and Information Science (IDI) at the Norwegian University of Technology and Science (NTNU). It consists of a preliminary study of current secure development practice, a developed tool-supported approach to perform secure software analysis and design, and a case study of this approach.

I would like to thank my subject teacher Lillian Røstad (IDI) and supervisor Per Håkon Meland (SINTEF) for good support and advice.

Trondheim, 29. May 2008

Ole Gunnar Borstad

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Part I

Introduction

Chapter 1

Introduction

Security has become relevant for all parties involved with software development due to increased risk from malicious users and the fact that software systems are continuously growing in size, complexity and connectivity (46). The increase of Web applications contribute to increased exposure to attacks and software vulnerabilities are likely to be exploited. Developing secure applications is a complex process and requires an in-depth understanding of potential security risks. Security modelling is a process of identifying threats, which may give the required understanding to design or identify vulnerability countermeasures. Security patterns are widely accepted security solutions that are fitted to a general representation, allowing the solution to be used in many different instantiations. The increase in cataloged vulnerabilities reported by CERT statistics [1] and security pattern publications show that there is clearly a need for better software security methods and tools.

1.1 Background and Motivation

Software security has received a lot of interest over the last several years. The field of computer security has experienced a change of focus from reactive network-based security approaches to the idea of engineering software to function correctly under malicious attack (46). Common design flaws and implementation bugs are leaving

[1] Carnegie Mellom University's Software Engineering Institute's CERT Coordination Center, http://www.cert.org/stats/fullstats.html

critical software systems open to attacks. The ad-hoc afterthought paradigm of security is not sufficient. To improve the security of software systems, security must be built in from the ground up. Most software developers are not trained in software security, and security experts do not necessarily know much about development (46). There is a need to bridge this gap, by giving developers and security experts proper tools and methods. Security vulnerabilities and risks need to be mitigated by countermeasures. Security patterns are among the countermeasures as a way of reusing expert security knowledge in software designs and implementations. The process of finding the right pattern is however not trivial and requires a thorough understanding of threats and exposures to software systems. A study performed by the author (13) suggests that security pattern support in software design tools might help developers design more secure software. Security pattern design templates were implemented in a state-of-the-art software design tool and tested through a case study of a medical journal system. It is pointed out that there is a need for methods and tools to support the software security modelling process to be able to identify suited security patterns and countermeasures in general.

There is a need for methods and tools that give developers assurance with respect to acceptable security risk levels. What vulnerabilities are prevented and to what degree? Security modelling tools are practically non-existent. SeaMonster (61) is an open source security modelling tool designed for all parties involved in secure software development. SeaMonster was initiated by NTNU/SINTEF and currently supports threat modelling with attack trees, misuse case diagrams and vulnerability cause graphs. SeaMonster is a prototype which with further development could become an in-depth security modelling tool. Sharing of modelling results between security experts and developers may increase security awareness and enable security work in the entire software development process. Information about known software vulnerabilities can already be found in several publicly available repositories or databases, such as National Vulnerability Database (60) or SecurityFocus (74). They provide information on vulnerabilities regarding system administration, and are therefore not very useful at development time. Allowing a security modelling tool to share models with development tools, i.e. design tools, by the use of repositories is expected to bridge the gap between security

professionals and developers, and most importantly help developers create more secure software.

1.2 Problem Statement

The overall goal of this thesis is to explore and suggest how software security can be improved by including security throughout the development lifecycle. Emphasis will be put on the early stages of software development, i.e. analysis and design phases, and on using security design patterns. A preliminary study is performed to survey current practice in the field of software security. The work will develop an approach to software security analysis and design and implement security improvement tools to support this approach.

1.3 Thesis Outline

This thesis is organised in the following chapters:

2 Software Security Practices
This chapter describes state-of-the-art software security practices. This is done by describing trends in software that increase the importance of secure development and currently documented security practices. Risk management, touchpoints and countermeasures, and knowledge management are described as pillars of software security.

3 Security Modelling
This chapter describes security modelling practice; approaches to analysing security issues in software. The described modelling techniques are threat modeling, attack trees, abuse cases, vulnerability cause graphs and security activity graphs.

4 Countermeasures
This chapter describes security patterns and risk analysis as countermeasures to security attacks.

5 Sharing Security Knowledge
This chapter describes technologies and methods for sharing of security knowledge. Ex-

amples of vulnerability repositories are given and research on sharing security knowledge that is applicable during development is described.

6 Research Agenda

This chapter describes the research design of the thesis. This includes research hypothesis, objectives, method and work plan.

7 Using a Countermeasure Model During Development

This chapter describes an approach to improve secure software development practice based on security modelling and countermeasures in design. A countermeasure model is introduced as a formal representation of a vulnerability countermeasure.

8 Extending SeaMonster

This chapter describes the current status and possibilities for further development of the prototype security modelling tool SeaMonster.

9 Realisation

This chapter describes the realisation of a security modelling tool. SeaMonster is used as a baseline.

10 Case Study

This chapter is a case study of the approach in Chapter 7 and the implemented tool described in Chapter 9. The case study works as a test and evaluation of the contributions in the thesis.

11 Evaluation and Discussion

This chapter evaluates the contributions and discusses other initiatives in security improvement tools and methods.

12 Conclusion

This chapter concludes the thesis. The contributions are summarised and possibilities for further work are presented.

Part II

Prestudy

Chapter 2

Software Security Practices

Software security is receiving increased interest among researchers and practicioners. The increased interest is a result of increase in reported security-related software vulnerabilities and incidents of security breach (48) as shown in Figure 2.1 [1]. This chapter describes current activities in the discipline of software security. The need for change in computer security practice is described by three important trends in modern software; connectivity, extensibility and complexity. Risk management, security touchpoints and knowledge management are described as the three pillars of software security. Any organisation developing software can implement these pillars to build a cost-effective software security program.

2.1 Introduction to Software Security

Software security is about engineering software to function correctly under malicious attack. Reducing the risk of security breaches by reducing consequences of attacks, i.e. isolating information from its users, might not be feasible in today's software where availability of information is key. The practice of software security tries to reduce exposure to threats by reducing vulnerabilities to an acceptable level. The integration of system and security engineering, leading to software security, is still in its infancy. Some of the first books on software security practice were published in 2001 (5; 86). Progress

[1] Carnegie Mellom University's Software Engineering Institute's CERT Coordination Center, http://www.cert.org/stats/fullstats.html

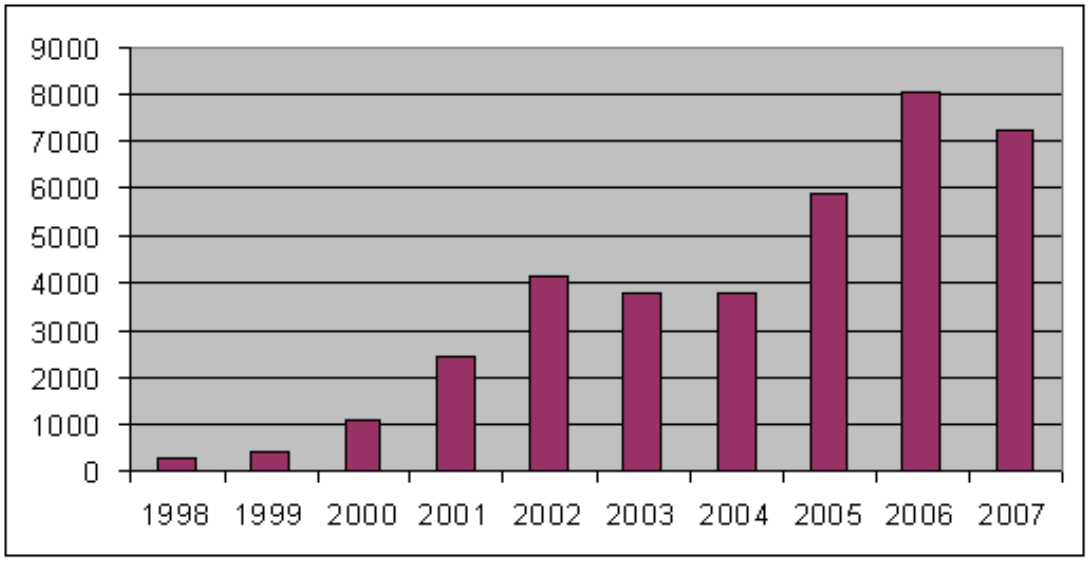


Figure 2.1: **Reported Software Vulnerabilities** - Security-related vulnerabilities reported to CERT/CC over several years

is however being made; big companies like Microsoft are recognizing software security as an important part of software engineering by developing the Security Development Lifecycle (53) and threat modelling techniques (80).

Computer system security is not a new field of computer science. Over three decades ago, the Multics operating system was designed as a secure system by the use of design principles such as access control lists and identification and authentication of users (68). Computer security has often been regarded as a branch of communication security which is based on cryptographic techniques and network theory (42). This view has produced several reactive network-based security approaches, such as firewalls, intrusion detection and cryptographic protocols (1). The term reactive in this setting means the approaches are added during system deployment or production; they are not designed as part of the software. Why is this no longer sufficient to produce secure computer systems? Three trends in computing systems have a large influence on the growing security issues (48):

Connectivity: The Internet has increased the connectivity of computing systems. A few decades ago, software systems were internal to organisations, limiting the number of users to employees and potentially customers of the organisation. Most of today's software is however exposed to the Internet through a Web browser, introducing billions

of potentially malicious users. The exposure to threats is increased dramatically which increases the probability that software vulnerabilities will be exploited.

Extensibility: Systems have to a larger degree become extensible. Extensible systems evolve in an incremental fashion by accepting updates or extensions, sometimes referred to as mobile code (50). Examples are Web browser plug-ins that allow viewing of different document formats. Word processors, spreadsheets, e-mail clients are extensible through scripts, applets and plug-in components. Extensible software is attractive because it may increase market share through rapid deployment. Extensible systems are however prone to create software vulnerabilities as unwanted extensions.

Complexity: Modern software systems are growing in size and complexity. Defect rate tends to increase with code size (32). This holds for security vulnerabilities as well (46). Analysing, developing and proving that a software system is free of problems, is probably not feasible at some point of complexity.

Along with the above trends, security is often a trade-off versus features and time to market. For users and system administrators, security interferes with usability and effectiveness. Different groups involved with software think of security in different terms. Software developers might think of code quality while administrators think of system management and network-based security approaches. With the trends of modern computing systems that make security harder to achieve, there is a need for methods and tools to deal with security at a higher level. All phases of the software development lifecycle should be included to build security in from the ground-up. Studies performed on e-business applications show that 70% of identified security defects are design related (37). Computer security must deal with *software vulnerabilities*.

Software security problems can be classified with some basic terminology. McGraw (46) proposes the following categorisation:

Defect: Defects include both implementation vulnerabilities and design vulnerabilities. A defect is a problem that may pass unnoticed or possibly surface in a fielded system and impose major consequences.

Bug: A bug is an implementation-level software problem. Bugs are fairly simple implementation errors that can easily be discovered and fixed. The use of automated tools to detect bugs is quite common and is a security best practice (45).

Flaw: Flaws are introduced at a higher level than bugs. A flaw will be instantiated in code, but is usually introduced at the architecture or design level. Automated tools to detect flaws do not yet exist.

Practice shows that at least 50% of software security problems are flaws (37; 46). This shows that security is about more than coding issues, fixing code-level bugs will only solve half the problem at best.

2.2 Pillars of Software Security

McGraw (46) defines applied risk management, software security touchpoints (best-practices applied to software artefacts) and knowledge as the three pillars of software security. The pillars can be applied by any organisation involved with developing software to any software development methodology. A cost-effective security program can be built by applying the pillars in a gradual manner. The pillars are described in the following sections.

2.2.1 Risk Management

A risk management strategy seeks to find a cost-effective way to practical security. There is no perfect defense in software simply because it would be too expensive (42). A cost-effective risk management strategy balances the cost of applying a security program and the risk of loss when software is compromised by attackers. Risk management is a business-level decision support tool with an underlying goal; avoiding potential software flaws before they materialise into vulnerabilities. The cost of software flaws tend to increase as they are discovered later in the development lifecycle (47).

A complete risk management framework is composed of architectural risk analysis and a management strategy. Applying risk analysis at the architectural level is a best-practice and one of the to be mentioned touchpoints in Section 2.2.2. In security communities

it is often referred to as threat analysis, security design analysis or security modelling, the latter is described in Chapter 3. A risk management strategy tracks and mitigates risk through the whole software development lifecycle (SDL).

McGraw (46) defines a Risk Management Framework (RMF) as the heart of building secure software systems. The RMF identifies and tracks risk through a software development project as touchpoints are applied and the risk landscape is changing. The notion of risk management is not characteristic for software security alone, most financial firms have a formal risk management department (28). Risk is defined by the following relationship:

$$R = PxI$$

The probability (P) is the likelihood that a risk will materialise and become a problem. The impact (I) is a metric for the business consequence of the problem. Quantification of risk is essential in any risk management process.

The RMF defined by McGraw consists of five stages:

1. **Understand the business context.** Risk is tied to an organisation's business goals. Understanding these goals is important in order to be able to make decisions and do prioritisation. If a risk does not impact the goals of the organisation, it is not worth analysing.

2. **Identify Business and Technical Risk**. Technical risks are grounded in software artefacts and are identified by the use of touchpoints. A technical risk should be mapped to business goals through business risk such as financial loss, loss of reputation etc. This process identifies what software flaws might potentially harm the business.

3. **Rank the Risks**. This stage creates an output list of weighted risks that can be used as a priority scheme. Risks are ranked according to their probability and impact.

4. **Define the Risk Mitigation Strategy**. Security analysts are good at finding technical problems, but not at finding the correct solutions. This is apparent by the use of checklists of what not to do instead of describing what should

be done (46). A risk analysis is worthless without a good mitigation strategy. This strategy defines a cost-effective set of activities described by metrics such as implementation cost and time or probability of success. The mitigation strategy should also include validation techniques that are used to assure that mitigation is successful. The validation techniques employ financial metrics such as return on investment and risk coverage.

5. **Mitigate and Validate**. The last stage executes the mitigation strategy of phase 4. Software artefacts are validated to assure that they do not bear unacceptable risk.

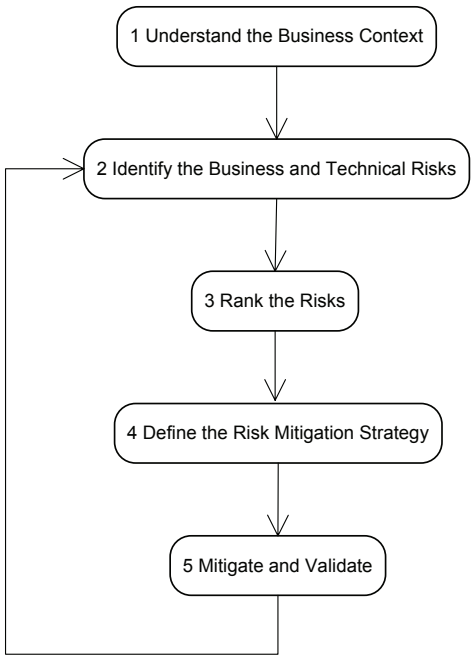


Figure 2.2: **Risk Management Framework** - Five stages make up the looped RMF process

The stages above should be applied in a continuous loop, as shown by Figure 2.2. There are risks connected to all stages of software development, and so the output of the five stages of the RMF should be kept updated. For example the RMF can be executed once for every stage of the SDL.

2.2.2 Security Touchpoints and Countermeasures

Software security is more than traditional security software features like cryptographic protection of communications. Security is a system-wide property; it requires a combination of security features and design for security. This is one of the reasons security has to be built in. Software developers have adopted security best practices to build more secure software systems (45). Some of these practices are referred to as software security touchpoints. These touchpoints apply a best practice to software artefacts during development and are independent of the applied software development process. McGraw et al. (85) defines seven touchpoints that includes common software development artefacts; requirements and use cases, architecture, design, testing, code, test results and deployment. The touchpoints are: Code Review, Architectural Risk Analysis, Risk-Based Testing, Penetration Testing, Abuse Cases, Security Requirements and Security Operations. The touchpoints are described briefly in the following paragraphs:

Code Review

Artefact: Code

Code is reviewed for implementation bugs. Static analysis tools are often applied to discover common vulnerabilities (17). Higher level problems, i.e. architectural, are hard to discover during code review, and requires architectural risk analysis.

Architectural Risk Analysis

Artefact: Design and specification

Security analysts search designs for flaws in software architecture. Possible attacks against the system, weaknesses and ambiguity in specifications are identified. The identified risks are then handled by a risk management process.

Penetration Testing

Artefact: System in target environment

Architectural risk analysis should give input to a penetration test of fielded software in its target environment. Penetration testing uncovers environment and configuration problems which can be fixed late in the development cycle. It is the most commonly applied software security best practice (9).

Risk-Based Security Testing

Artefact: Units and System

Security testing should include two strategies: 1) testing of security functionality by functional testing and 2) risk-based security testing. The latter should be based on attack patterns, risk analysis and abuse cases. To combine the knowledge of software architecture and common attacks, it is essential to think like an attacker when performing the tests.

Abuse Cases

Artefact: Requirements and use cases

Use cases describe the system's behaviour and interaction with legitimate users. Expanding the use case model with abuse cases describes how the system reacts to an attack by malicious users. Applying abuse cases allow focus to be put on security in early stages of software development. Abuse cases are further described in Section 3.2.3.

Security Requirements

Artefact: Requirements

Security must be grounded in software requirements and specify system functions in all possible circumstances of use, legitimate or malicious (51). The security requirements should cover both security functionality, such as access control, and non-functional characteristics relevant to security.

Security Operations

Artefact: Fielded system

Secure design can not replace the need for security operations such as network security. Attacks happen regardless of secure design and implementation. Security operations in the fielded system allow feedback of knowledge on attacks and exploits, logging procedures is a good example. This knowledge should be cycled back into the software development process.

Figure 2.3 shows where the touchpoints are applied. Although the figure resembles a traditional waterfall model, iterative approaches are very common today which means the touchpoints could be applied several times during a development project.

Applying the touchpoints is a way of putting software security into practice by changing the way organisations develop software. In order to reach a state of security, there need to be some protection or prevention of vulnerabilities. A general term for the

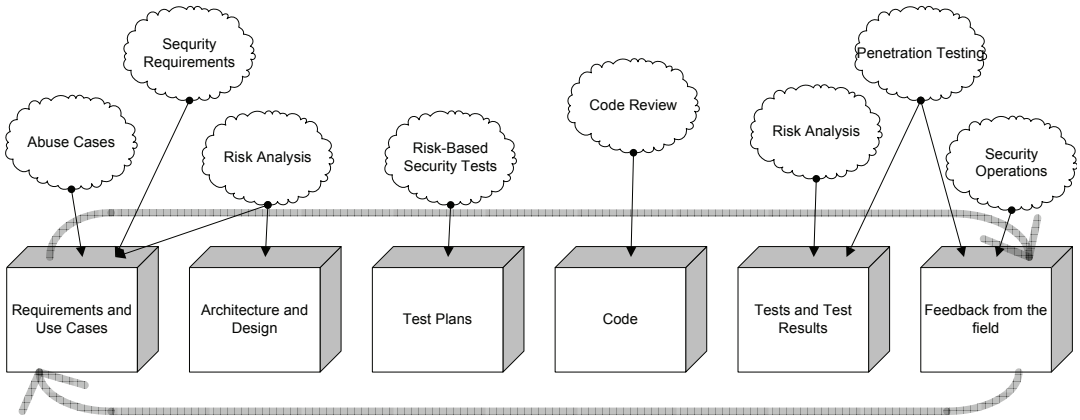


Figure 2.3: **Touchpoints and Software Artefacts** - The touchpoints are mapped to their related software development artefacts

protection and prevention is a countermeasure. Touchpoints are countermeasures to software vulnerabilities. Countermeasures are described further in Chapter 4.

2.2.3 Knowledge Management

McGraw (46) defines knowledge management as the third pillar of software security. With the recent changes in computer security towards a paradigm of building security in, there is a lack of practitioners and experts. Training in security practices and sharing of security knowledge is increasingly important, meanwhile there are not enough experienced practitioners to apprentice software architects and developers. Knowledge management can reform low-scale training techniques by compiling security knowledge and sharing it widely.

A security knowledge taxonomy groups security knowledge into related catalogues. This organisation seeks to improve understanding and provide a standard for security knowledge artefacts. The catalogues also help in mapping knowledge to phases of the software development lifecycle. Figure 2.4 shows a knowledge schema relating seven catalogues (11). Principles, guidelines and rules are advice of things to do and not to do when developing secure software systems. These catalogues span from high-level architectural principles such as the principle of least privilege (67), to code-level rules

such as avoiding insecure string functions in C. Attack patterns, exploits and vulnerabilities help recognise and to deal with problems related to security attacks. They are useful in security analysis and development for example in abuse cases. Historical risk includes risks and vulnerabilities. These are descriptions of specific incidents from real software systems and projects. Historical risks include business impact and are useful for finding similar problems during development of software.

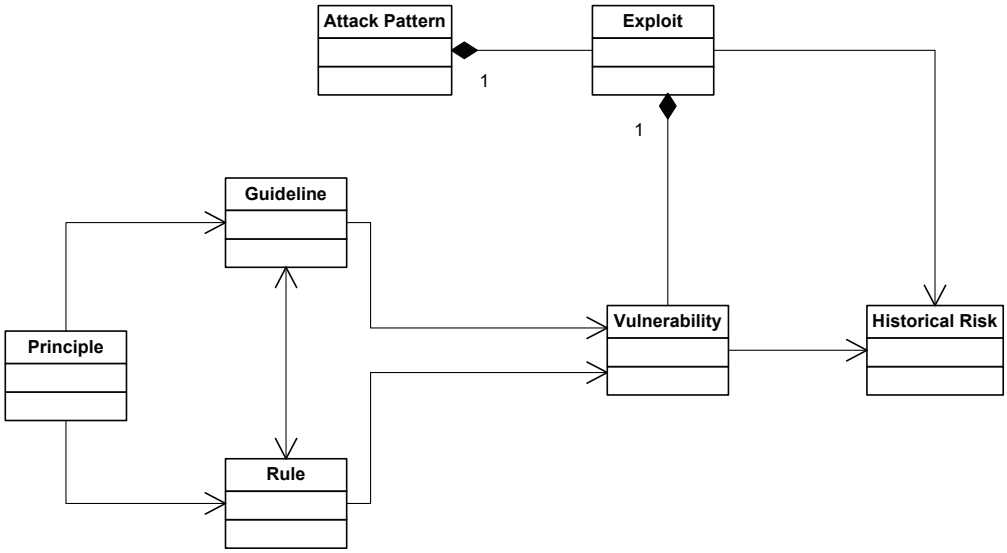


Figure 2.4: **Software Security Knowledge Taxonomy** - Organisation of software security knowledge

The catalogues are described in the following paragraphs.

Principles

A principle is a statement of general security wisdom. Principles are derived from experience, and are useful for diagnosing architectural flaws and practicing secure software development in general.

Relevant artefacts:

- Security requirements
- Security architecture

- Software design

Guidelines

A guideline is a recommendation for things to do or to avoid during software development. Guidelines are defined for a specific technical context such as operating system or programming language. Guidelines can uncover and prevent architectural flaws and implementation bugs.

Relevant artefacts:

- Security requirements
- Software design
- Code

Rules

A rule is a recommendation for things to do or to avoid at the level of syntax. Rules are for specific programming languages and can be used in code analysis tools. Rules uncover implementation bugs.

Relevant artefacts:

- Code

Attack patterns

An attack pattern is developed by studying large sets of software exploits. The patterns can identify the risk that a given exploit will occur in a system. Attack patterns are useful in designing abuse cases and security tests.

Relevant artefacts:

- Abuse cases
- Software design
- Security test plan
- Penetration tests

Historical risks

Historical risks are identified in actual software development. These risks are useful in software development to identify potential security issues at an early stage, because little analysis is needed since actual risks in similar situations are already documented.

Relevant artefacts:

- Software architecture
- Software design
- Test plans
- Deployed software

Vulnerabilities

Vulnerabilities are the result of defects in software that can be exploited by malicious users to abuse the system and its resources. They are useful for understanding how software vulnerabilities impact the security of a computer system.

Relevant artefacts:

- Software architecture
- Software design
- Penetration test
- Fielded system

Exploits

An exploit is an instance of an attack on a computer system that takes advantage of a specific vulnerability.

Relevant artefacts:

- Penetration test
- Fielded system

Documentation of security knowledge is currently dominated by low-level checklists and taxonomies (46). McGraw et al. presents a taxonomy of security coding errors (84) with seven groups of errors referred to as kingdoms. The taxonomy is designed to help programmers avoid security coding errors and to be able to identify possible sources of errors. They are also expected to increase security awareness and help developers understand how their work affects security. The taxonomy is a set of rules which are best fit as input in an automated tool such as static code analysers. Static analysis tools analyse source code without running the program, highlights coding errors or proves properties of the code (24).

The following are the seven kingdoms of security coding errors, in order of importance to software security:

1. Input Validation and Representation
2. API Abuse
3. Security Features
4. Time and State
5. Error Handling
6. Code Quality
7. Encapsulation

Each kingdom consists of a collection of coding errors that share a common theme. The kingdoms are briefly explained in the following paragraphs. The coding errors described below are a small excerpt from the taxonomy. See (46) for the complete taxonomy.

Input Validation and Representation

Client input should not be trusted because it may contain malicious data. Input validation and representation problems are caused by metacharacters, alternate encodings, numeric representations or lack of input validation in general. Input validation should be done by a white list technique (33).

- **Buffer Overflow:** The result of writing outside the bounds of allocated memory. This can lead to corrupted data, software crash or cause the execution of an attack payload.
- **Command Injection**: Executing commands from an untrusted source. This can lead to system misuse or attacks to a third party such as legitimate system users.

API Abuse

Application Programming Interface (API) abuse is caused by violations of the contract between caller and callee of an API. Examples are making false assumptions about the behaviour of the API callee, or when the API callee does not correctly implement the defined interface.

- **Dangerous Function**: Never use unsafe functions, i.e. functions that have documented security vulnerabilities.
- **Exception Handling**: Incorrect handling of exceptions can cause the software to crash.

Security Features

This kingdom includes features such as authentication, access control, confidentiality, cryptography and privilege management. The assumption that software security is about security features alone is a big misconception. Implementing security features incorrectly is quite common (27).

- **Password Management**: Storing a password in plaintext.
- **Missing Access Control**: Access control checks are not performed in all execution paths.

Time and State

Time and state is relevant to distributed computing. Modern computer systems include multi-core, multi-CPU or distributed computers interacting and interchanging information. Time and state errors are related to unexpected interaction between different threads and processes.

- **Deadlock**: Inconsistent locking routines can lead to deadlocks potentially stalling or crashing the system or making resources unavailable.
- **Race Conditions**: There is a time window between a resource property is checked and when the resource is used. This can be exploited to launch an attack or corrupt data.

Error Handling

Security defects related to error handling are very common (48). Not handling errors at all or insufficient handling introduces great security risk because of two reasons: 1) errors may give away too much information to possible attackers and 2) improper error handling may lead to unexpected system state.

- **Empty Catch Block**: Failing to provide handling of exceptions enables attackers to induce malicious system behaviour unnoticed.
- **Catch NullPointerException** : Catching of *NullPointerException* should not be used instead of programmatic checks to prevent dereferencing a null pointer.

Code Quality

Code quality affects the system behaviour. Poor code quality can lead to unpredictable system behaviour, providing attackers a way of stressing the system in unexpected ways.

- **Memory Leak**: Failing to free memory results in resource exhaustion.
- **Null Dereference**: Dereferencing a null pointer causes the program to raise a *NullPointerException*.

Encapsulation

Encapsulation with respect to security is about drawing boundaries and setting up barriers between programming entities. Example boundaries are between classes with different methods.

- **Debug Code**: Programmers use debug code to quickly test small units of code. Debug code might create unintended entry points if not removed before deployment.

- **Comparing Classes by Name**: Comparing classes by name is not sufficient to decide whether they are the same or differing classes.

There are other collections and taxonomies of security knowledge in security literature (12). The Seven Kingdoms define a middle ground between rigorous academic studies and ad hoc collections based on experience (46). The focus on simplicity and practicality is important with software security taxonomies because there is a lack of experienced practitioners, and increasing security awareness among software developers is beneficial. Another popular and useful list is the OWASP Top Ten (26). This list presents a consensus about what the most critical Web application security flaws are. The list is developed by a variety of security experts from around the world. The book '19 Deadly Sins of Software Security' (52) is a collection of 19 programming flaws and shows how to fix them.

Sharing of security knowledge between different parties involved with secure systems development is an ongoing topic in software security research. It is likely that this will become an increasingly important part of security knowledge management. Sharing of security knowledge is of special interest to the work described in this thesis so it is devoted a chapter of its own, see Chapter 5.

Chapter 3

Security Modelling

Security modelling is a collective term for modelling techniques of security concepts such as threats, attacks and vulnerabilities. These techniques improve understanding of security issues so they can be dealt with throughout the development lifecycle. This chapter describes security modelling as a method and its rationale with respect to software development. The following modelling techniques are described: attack trees, abuse and misuse cases, vulnerability cause graphs, security activity graphs and threat modelling.

3.1 The Need for Methods and Tools

Complex software is dealing with critical information and controlling systems in military, financial, medical and governmental organizations, making consequences of a security breach very dangerous. The complexity of software is related to attack ability. Security faults are a subset of quality faults, and quality faults tend to be a function of code complexity which is again proportional to code volume (32; 46). Developers need to understand and deal with how their complex programs affect security.

Studies performed on e-business applications show that 70% of found security defects are design related (37). The same study also showed that 47% of the application security defects should be regarded as severe design flaws, which means they are exploitable and could cause loss of revenue. This is connected with the fact that security is rarely

introduced in early stages of software development. The current focus is to detect problems rather than preventing them (49). Developers are not creating insecure software because they do not know how to code, flaws are introduced at a higher level of software development. A thorough understanding of security at a higher level is needed and abstractions other than code units should be introduced to enable developers to gain control on security at the level of a complete software system.

A lot of different terminology is used in security literature; in the following some core security concepts are defined to remove any ambiguity. The definitions (70) are essential to software security and security modelling, and will be used extensively throughout the thesis:

- Asset - Assets are information or resources which have value to an organization or person.
- Stakeholder - an organization or person who places a particular value on assets.
- Security Objective - a security objective is a statement of intent to counter threats and satisfy identified security needs.
- Threat - a potential for a security breach of an asset.
- Attack - an attack is an action that violates the security of an asset.
- Vulnerability - is a flaw or weakness that could be exploited to breach the security of an asset.
- Countermeasure - action taken in order to protect an asset against threats.
- Risk - a risk is the product of two parameters: the probability (P) that a successful attack occurs and Impact (I), i.e. expected loss. This is expressed by the formula R = P * I

The definitions are related and together they form a framework where software developers are able to gain an understanding of their software and its environment with respect to security. Security modelling is a method where these concepts are evaluated by the use of various modelling techniques in a systematic fashion. Security modelling

integrated in the software development process, is expected to improve security in software by helping developers prevent potential attacks (8). The importance of security modelling is based on the following assumptions:

Increased security awareness. Security failure data should be fed into some quality assurance process, as is usually the case with other software quality requirements. Software engineers generally do however not use security failure data to improve the security of the software they develop (59). Providing up-to-date information on security problems to developers is an important step towards building secure software. This information should be combined with practices to reduce the security problems. Software developers can not design, develop and test secure software systems without knowing the security issues (34). Developers need to be aware of how security faults are introduced in artefacts throughout the whole software development lifecycle, including the early stages with requirements and design, and the threats to a deployed software product.

Need for security methods and tools. Most current approaches to software security apply some best practices, e.g. secure programming techniques, in a rather ad hoc manner. Experience shows that these approaches help prevent software vulnerabilities, hence the term best practice, there is however a need for completeness in understanding and dealing with security issues. An ad hoc security program will not give assurance regarding what threats and vulnerabilities that are dealt with by each security practice. Security practices, techniques and tools should be applied throughout the whole development process in a systematic fashion to help development of secure software systems (7). Systematic prevention of software vulnerabilities requires an understanding of the causes of vulnerability, the threats they expose and how they can be exploited by attackers. Without this understanding, developers have no assurance that security risks are in fact mitigated by the chosen countermeasures. Structuring security practices, techniques and tools is a way of achieving a predictable and measurable secure development process.

Security modelling is a method that may change the 'penetrate and patch' behaviour of security, because it applies to all stages of development and its output is meant to

be kept updated as threats change. Increased security awareness is a direct benefit of security modelling. It is unlikely that implementing vulnerability countermeasures without a thorough security model will produce secure software, because choosing the right countermeasures requires an understanding of threats and exposures. This is backed up by The Open Web Application Security Project (OWASP), which states that threat modelling is essential for design of new Web applications to be able to mitigate the important security risks (66). Security modelling provides a context for use of software security tools because it defines a foundation that can be used to select proper tools and methods.

Security modelling tools can be used to model security effectively and enables sharing of security knowledge between parties involved with secure software development. The specific functionality of security modelling tools is not defined in security literature because security modelling tools are practically non-existent. Security modelling is done with general-purpose drawing tools (8), such as UML (31) tools. Work is ongoing to make software development tools make use of security models from security modelling tools. This is very relevant to the work presented in this thesis and will be further described in Chapter 5. The lack of tools is based on the fact that security modelling is mainly practiced by a small number of security experts and researchers. Software developers often use models, e.g. when working with architecture, design and deployment, but security rarely takes part in these models because languages and tools do not support security modelling. Microsoft's Threat Modeling Tool (55) is an exception to this, and allows users to create threat models for software applications. The tool captures threat models in machine-readable form for storage and updating. The Threat Modeling Tool is designed for Microsoft's threat modeling method, which narrows the potential use cases and makes it less suited for general security modelling as described in this thesis. Prototype tools such as SeaMonster, which is an open source security modelling tool, shows that there is interest among security researchers to develop new modelling tools for security. SeaMonster is described in detail in Chapter 8.

3.2 Modelling Techniques

There are several security modelling techniques, which in general can be grouped by their output: vulnerabilities and their causes, threats, attacks or countermeasures. The aim of the techniques is to gain an understanding of security issues to be able to deal with them throughout the software development lifecycle.

3.2.1 Threat Modeling

Threat modeling (80; 83) is a part of the Microsoft Trustworthy Computing Security Development Lifecycle (43). Threat modeling is usually performed in the design or specification phase of a development process, to understand a product's threat environment and defend against potential attacks. Six activities define the threat modeling process:

1. **Scoping the process.** Threat modeling an entire product is often too complex. Logical groups of functionality is grouped into *components*, which are then subject to threat modeling.

2. **Gathering background information.** Information from the component's specifications are gathered. Examples are use cases, deployment configuration, dependencies on other components, features or technologies and implementation assumptions.

3. **Describing the component.** This activity gives a security-focused description of the component's design. Entry points (interfaces with other software, hardware and users) are identified and given a trust level which describes the degree to which the entry point can be trusted to send or receive data. A list should be made of trust levels required to gain access to protected assets. Microsoft recommends the use of data-flow diagrams (DFDs).

4. **Obtaining threats.** Threats can be obtained through a brainstorming meeting, analysing DFDs to identify possible threats. The acronym STRIDE (spoofing, tampering, repudiation, information disclosure, denial of service, elevation of

privilege) can be used to help remember the types of threats a component might be exposed.

5. **Resolving threats.** Decide what actions are needed to resolve the threats, and execute them.

6. **Following up.** Tracking your own dependencies, verifying your assumptions and communicating security requirements. Changes made to the design should also be tracked throughout the product development cycle to ensure that the changes do not introduce holes in the security threat mitigations.

3.2.2 Attack Trees

Attack trees (AT) (69) describe possible threats or attacks to a system. By identifying how a system can be attacked and understanding the attacker, we may be able to design and implement proper countermeasures to thwart the attacks. Attack trees model attacks as a tree structure, with the goal as the root node and possible ways of achieving the goal as leaf nodes. The goal is a successful attack. Alternatives can be represented by OR nodes, AND nodes express that several steps are needed to perform an attack. A metric representing the cost or feasibility of an attack can be added to perform calculations on likelihood. Possible metrics are possible/impossible and monetary cost. Figure 3.1 shows an example attack tree with three ways of achieving unauthorized access to a system; by guessing a legitimate user's password, performing a buffer overflow attack and by performing a cross-site scripting attack (XSS).

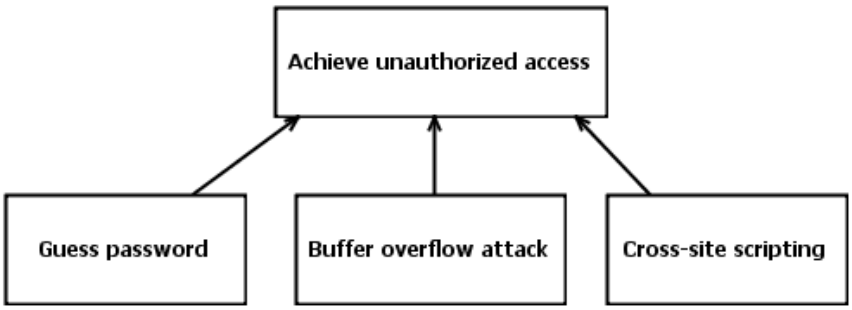


Figure 3.1: **Attack Tree Example** - Achieve unauthorized access

You create an attack tree by selecting a goal as the root node. Identify possible attacks for the root node, and then continue down the tree. Repeat for all nodes until you have exhausted possible attacks, and possibly pass the tree to someone else so they can add their attacks. Making useful attack trees requires thinking like an attacker, often referred to as a black hat approach in security literature (46). A black hat means applying desctructive thinking to abuse a computer system, to understand the motive, method, tools and desired results of attacks.

3.2.3 Abuse and Misuse Cases

Use cases (23) are used during elicitation and documentation of functional requirements to capture a system's normative behaviour, but there is little support for other types of requirements such as security. McDermott and Fox (44) suggested abuse cases in 1999 to express threats using standard UML use case notation. Sindre and Opdahl (76) suggested misuse cases which are negative use cases specifying undesirable behaviour in a system. Security requirements are elicited by documenting how the system is to react on exceptional cases such as illegitimate use. A use case can describe a countermeasure that mitigates a threat; the misuse case. A misuse case will exploit or hinder legitimate use cases. Abuse and misuse cases force developers to think like attackers, what do they want to achieve and how. Alexander (4) advocates using abuse and misuse cases to conduct threat analysis during requirements analysis. The differences in abuse and misuse cases are not important for this work, so the term misuse case will be used to cover these techniques. Røstad (77) has suggested to further specify the misuser, the actor that initiates misuse cases, as inside or outside attacker. This is adopted in further misuse cases in this thesis. Figure 3.2 shows an example misuse case diagram. An insider, the *Developer*, has forgot to remove debug code before software release, leaving a backdoor entry point to the program, shown as *Backdoor* in the figure. Vulnerabilities can be exploited thus introducing a threat to the system. The threat is represented by the misuse case *Steal payment credentials* which exploits the Backdoor vulnerability.

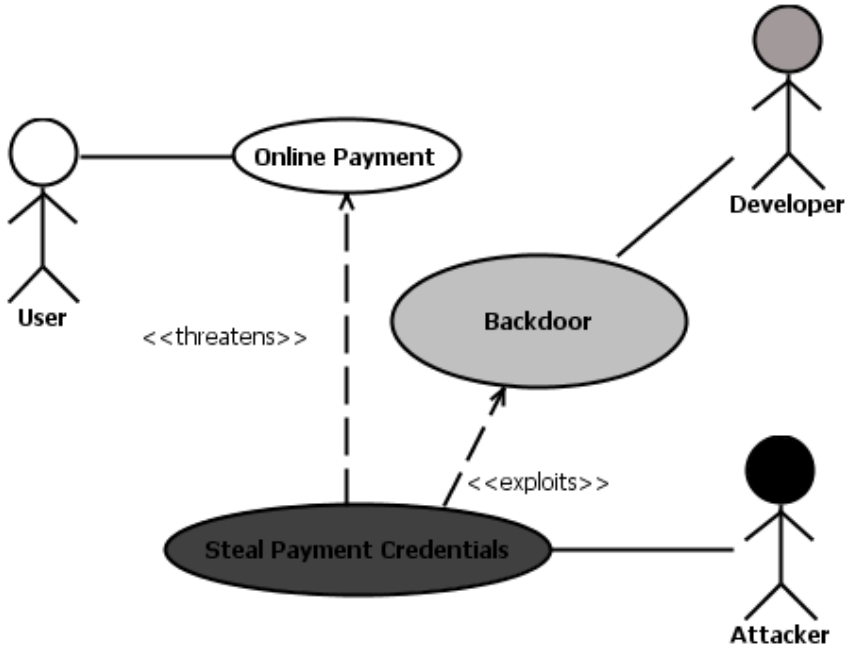


Figure 3.2: **Misuse Case Example** - Online payment

3.2.4 Vulnerability Cause Graphs

A vulnerability cause graph (VCG) (15) is used to analyse causes of vulnerabilities to prevent them in future software development. The causes of a vulnerability and their relationships are represented as a directed acyclic graph, with a single vulnerability as the root node. Internal nodes are causes, events and conditions during the software development process that may be the cause of a vulnerability. The semantics of a VCG is based on the predecessor-relationship: to prevent a cause C, either C itself or all predecessors of C must be mitigated. To prevent a vulnerability, all predecessors of the root node in the vulnerability's VCG must be prevented. There are four types of nodes in a VCG:

- Simple node: represents a condition or event that may contribute to a vulnerability.
- Compound node: combination of causes that form a VCG on their own.
- Conjunction: conjunction of two or more nodes.

- Root node: the vulnerability modelled by the VCG.

Countermeasures can be derived by identifying activities that mitigate causes in the graph. The goal is to prevent the vulnerability to materialise by using the information in the structure of the VCG (6; 7). VCGs model vulnerabilities and need a supplementary technique to identify countermeasures. Identifying causes to software vulnerabilities is however an important first step towards more secure software.

Figure 3.3 shows an example VCG of the vulnerability group *Injection flaws*. These flaws allow user-supplied data to trick the system into executing unintended commands. In the example, there are three main causes: 1) The cause *Accepts malicious input* allows attackers to pass malicious data to the system, including executable code and data formats not supported by the system. An underlying cause for this liberal input scheme is *Lack of input restrictions* in system documentation. 2) *Unrestricted privileges* give attackers the ability to perform system operations they are not intended to. Injected commands should not be able to perform restricted operations such as file access. 3) The third cause, *Use of unsafe APIs*, are programming interfaces that are not considered secure. Examples of these are exploited character escaping functions. *Code complexity* is identified as an underlying cause. Abundant code complexity makes the program code difficult to review by security engineers, allowing use of unsafe APIs to slip through quality assurance.

3.2.5 Security Activity Graphs

A security activity graph (SAG) (7) relates activities during software development to prevent potential vulnerabilities. By creating a SAG for every vulnerability, the goal is to identify countermeasures to prevent potential security vulnerabilities. SAG is a graphical representation of terms in first order predicate calculus, the graphs allow better readability and annotation with metadata (7).

The graph contains a tree structure, with the vulnerability as the root node. Other nodes are activities that influence security and take on boolean values. There are no restrictions on activities other than that they are meant to improve the security of a software product. The security touchpoints in Section 2.2.2 are good examples

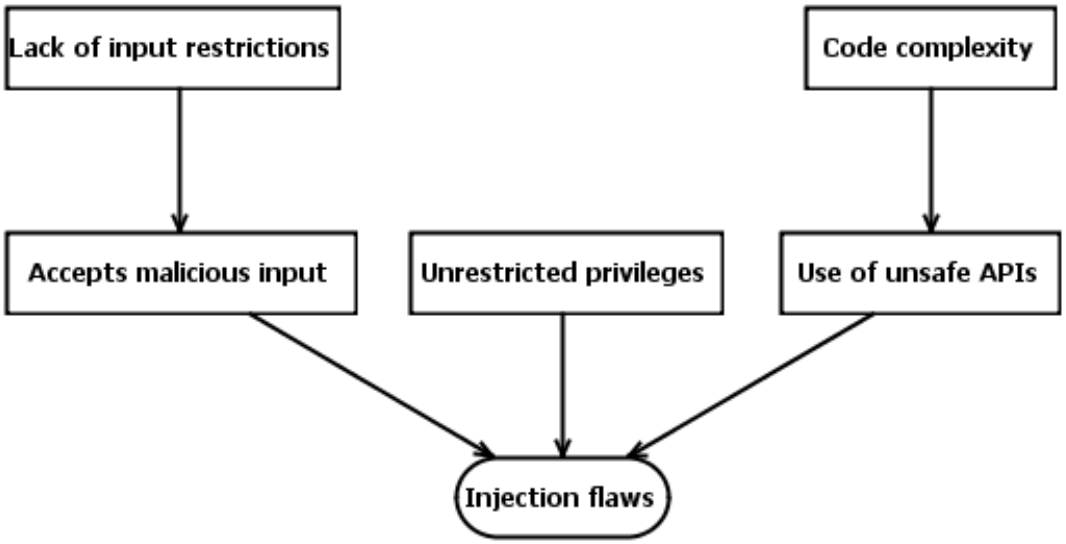


Figure 3.3: **Vulnerability Cause Graph Example** - Injection flaws

of possible activities. An activity is true if it is implemented perfectly during software development and is false otherwise. Activities can be arranged by software development phases such as analysis, design, implementation, testing and deployment. It is possible to assign cost metrics to activities; this might help produce a feasibility or ranking scheme. Relationships between nodes can be defined using logic gates which are a function of their input. Table 3.1 defines the graph elements and their logic equivalents.

Security activity graphs are tightly coupled with vulnerability cause graphs, and can be created from a VCG by a structured method following three simple steps:

1. Enumerate mitigation techniques for each cause.
2. Create SAG fragments for each set of mitigation techniques.
3. Combine the fragments to create a complete graph.

The following is an example which follows the above method to produce a SAG from the vulnerability cause graph in Figure 3.3. Figure 3.4 shows the resulting graph.

Step 1:
Mitigation techniques are identified for each cause of the VCG, shown in Table 3.2.

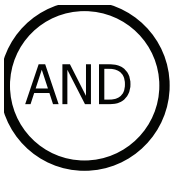
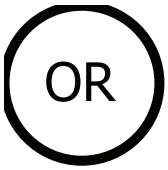
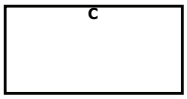
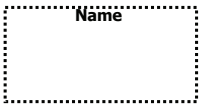
Graphical representation	Function	Logic Equivalent
	AND gate	$F = I_1 \cap I_2 \cap ... \cap I_n$
	OR gate	$F = I_1 \cup I_2 \cup ... \cup I_n$
	SPLIT gate	$F_1 = I, F_2 = I, ..., F_n = I$
	Activity	$F = C$
	Vulnerability	N/A

Table 3.1: **Security Activity Graph Elements** - Gates are a logical function F of their input I

Step 2:

SAG fragments are created for each set of mitigation techniques. One fragment is created for the input validation techniques and one for the changes in the use of APIs. Validation of input and rejection of invalid input are connected and must be applied in conjunction. Enforcing least privilege is added because all system access should be governed by a minimal privilege policy. The first fragment is an AND-node connecting these three techniques. The second fragment is a conjunction of static code analysis and use of safe APIs. Enforcing least privilege is equally important, so it is combined with

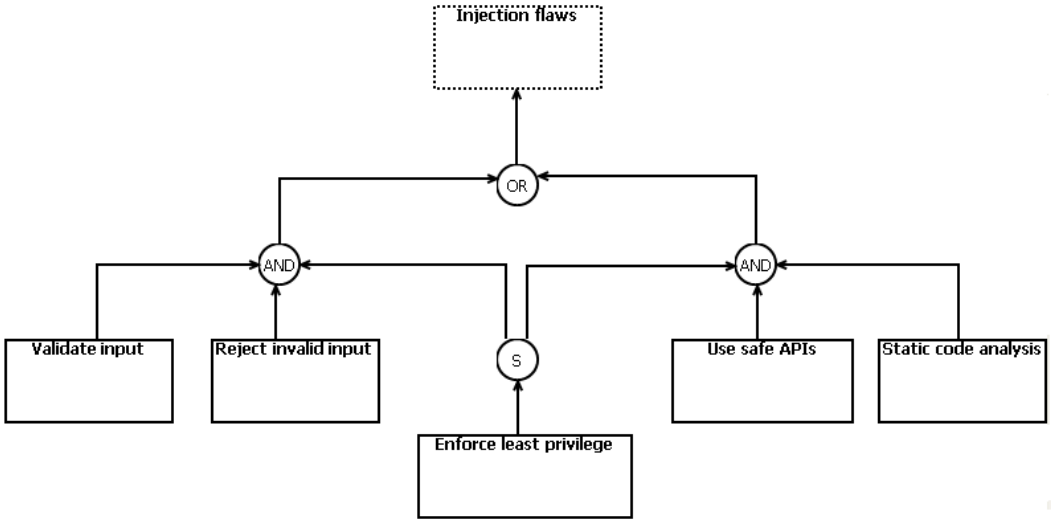


Figure 3.4: **Security Activity Graph Example** - Modelling injection flaws

the other two techniques. As in the first fragment, the resulting graph is an AND-node connecting the three techniques.

Step 3:

The fragments are combined into a security activity graph. The graph shows that to mitigate injection flaws, two paths are possible: 1) a conjunction of input validation, invalid input rejection and enforcing least privilege activities, or 2) a conjunction of static code analysis, use of safe APIs and enforcing least privilege. It should be noted that this example is not a complete analysis of injection flaw vulnerabilities. The resulting SAG is only intended as an example of this modelling technique.

For a given SAG there are likely several different sets of activities that satisfy the semantic function, ideally one wants to find the optimal set. A set could be optimal in terms of cost, efficiency, implementation speed etc. Automatic selection would require tool support for cost modelling and finding an optimal solution to the semantic function of the SAG.

VCGs and SAGs primary use is to understand vulnerabilities such that developers are able to identify and implement countermeasures, and finally producing secure software. These modelling techniques require tool support because the graphs are likely to be

Cause	Mitigation	Description
Lack of input restrictions	Validate input	All possible legal input to the system is specified through requirements.
Accepts malicious input	Reject invalid input	Input which do not pass the validation mechanism is rejected.
Unrestricted privileges	Enforce least privilege	All users should have a minimal set of privileges when connecting to backend systems such as databases or communication modules. A minimal set defines the needed privileges to perform the operations the system is designed for.
Code complexity	Static code analysis	Analysing complex code can effectively be done by code analysis tools. These tools can detect use of unsafe APIs or implementation bugs.
Use of unsafe APIs	Use safe APIs	Strongly typed parameterized query APIs with placeholder substitution markers. These handle all data escaping so use of unsafe escaping functions is not necessary.

Table 3.2: **Creating a Security Activity Graph** - Step 1

large and hard to maintain or keep track of (7).

Chapter 4

Countermeasures

In Section 3.1 a countermeasure was defined as *an action taken in order to protect an asset against threats and attacks*. Such actions are not limited to technical software artefacts, but can refer to any action designed to reduce the risk of attacks and software misuse. As an example, a countermeasure to the threat of inside attacks is to dismiss any disloyal employee. The countermeasures in this chapter is limited to those relevant to the software development process and related artefacts, and includes security patterns and reviews.

4.1 Security Patterns

A software pattern describes a recurring problem that arises in specific contexts, and presents a well-proven generic solution. The solution can form multiple concrete instantiations, and has become a pattern through many successful implementations, proving itself as a best practice. The use of patterns has been adopted in the security field. A substantial collection of security patterns has been developed (41; 71; 78). The definition of a security pattern (72) has the same key characteristics as general software patterns:

> A security pattern describes a particular recurring security problem that arises in specific contexts and presents a well-proven generic scheme for its solution.

Patterns appear to be valuable to secure systems development due to several advantages:

- Security knowledge is structured in an effective and understandable way. The knowledge in a pattern is constructive rather than presenting a laundry list of what not to do.
- There is currently a large gap between theory and practice in the field of software security, because developers do not know much about security and security experts do not necessarily know much about development (46). Security professionals are primarily concerned with the security of a system while system developers main concern is to get the system working. Patterns can be used to bridge the gap by making developers able to analyse the trade-offs between security and other requirements without the presence of a security professional (40).
- The pattern representation is known to most software developers (71). This enables patterns to be used as an effective medium for learning security knowledge.
- Much of the focus on computer and information security until now has been on low-level implementation and reactive network-based countermeasures, such as firewalls and cryptographic techniques. Patterns can be applied at any level from enterprise issues to architecture and design or low-level implementation such as programming language specific constructs. The pattern approach can help extend the security focus to include all phases of secure systems development.

Proper documentation of patterns is necessary to be able to understand and discuss them. This documentation should provide all the needed information to know when the pattern can be applied and how to implement the pattern. There are many variations of pattern descriptions, the one adopted here is taken from (14) because it is thorough and descriptive. This format suits a wide range of patterns, including security patterns:

Name
The name of the pattern.

Also Known As
Possible other known names.

Example

An example that demonstrates the problem in practice and shows why this pattern is useful.

Context

The situations in which the pattern may apply.

Problem

The problem addressed by the pattern.

Solution

The fundamental solution provided by the pattern. Fundamental implies a generic principle which can be utilised in different instances.

Structure

A detailed specification of the structural aspects of the pattern. This often refers but is not limited, to software components and their relations in some appropriate notation.

Dynamics

Scenarios describing run-time behaviour of the pattern.

Implementation

Guidelines for implementing the pattern.

Example Resolved

Discussion of important aspects with respect to the example.

Variants

A brief description of variants or specialisations of the pattern.

Known Uses

Examples of use in existing systems.

Consequences

Benefits and possible liabilities by using this pattern.

See Also

References to related patterns. This could be patterns that solve similar or related problems or patterns that improve potential liabilities introduced by the applied pattern.

It should be noted that not all elements of this description is needed for every pattern. The following is an example security pattern from (41), dealing with input in Web applications. The pattern representation adopted in this example is inspired by (14).

Name
Client Input Filters

Also Known As
Untrusted Client, Server-Side Validation, Sanity Checking

Example
Internet banking require customers to fill in account numbers and other personal information through forms. The server receiving the user supplied information does not trust the client side. When the client initiates a transaction, the server checks that all the parameters are valid before executing it.

Context
Any application accepting user input. The main concern is Web applications.

Problem
Web sites are highly interactive applications which require the user to submit some type of input. The server-side application logic can not treat the input as trusted because there are no guarantees that the input follows the format and range originally intended by application designers. This is because input from clients can be tampered with and it may be submitted by non-legitimate users.

Solution
All data provided by the client should be treated as malicious and filtered at the server. Client-side input validation can be used to give proper feedback through the user interface, but all validations must be repeated on the server. The server-side filter must deal with the following (the list is not exhaustive):

- Text input submitted by the user should be filtered to eliminate scripting tags and other questionable content.
- Suspiciously long URLs and header fields should be dropped and possibly logged.
- Calculated fields provided by the client should be ignored and recomputed at the server when the data is processed.
- Sensitive data that must be stored on the client should be kept in an encrypted, tamper-proof form.

Client filters should be able to modify requests before delivering them to the intended object. If the data cannot easily be fixed, the client filter should reject or simply drop the request. All filtering events should be reported to the central

logging mechanism, although many will be benign, they might indicate a pattern of attempted misuse. If a filter detects an obvious attempt to sidestep the security of the system, the request should be blocked and the event reported.

Structure
Input validation can be integrated into the requested objects, or they can be implemented as separate objects. The second alternative is less efficient but easier to manage. In either case, the essential is that the filters are always invoked before any processing of the client input. The structure of the pattern is described in Figure 4.1. A logging mechanism is added with a dotted connection because this component is not a part of this pattern but is recommended as a related pattern.

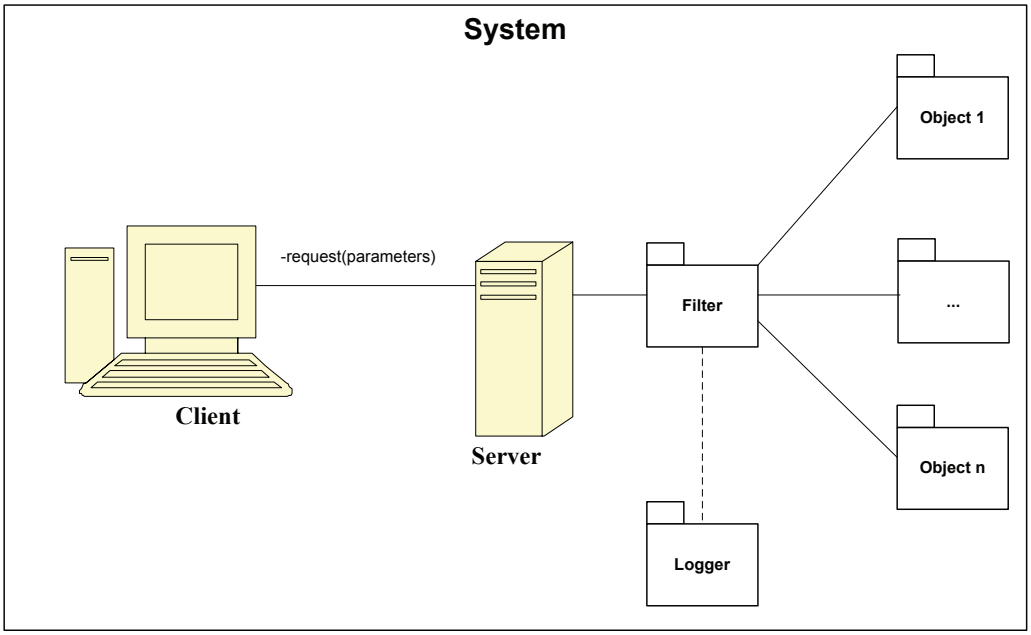


Figure 4.1: **Client Input Filters** - Pattern structure

Consequences
Consequences of using the pattern are provided benefits and potential liabilities. See Table 4.1.

Related Patterns
Log for Audit

Impact on	Consequence
Availability	If overly sensitive, this pattern can have an adverse effect on availability, preventing legitimate users from using the site.
Integrity	This pattern greatly enhances the integrity of the data processed by a web site.
Manageability	The management burden could be increased if overly sensitive sanity checks result in a high number of false reports of attacks that must be investigated.
Performance	This pattern will incur a small performance penalty, since it requires some time to perform checks. If data is stored in encrypted form on the client, encrypting and decrypting the data will also exact a performance hit.
Cost	This pattern has fixed implementation costs. However, if overly sensitive it could increase the customer service burden on the site.

Table 4.1: **Client Input Filters consequences**

Security literature describes a wide range of security expert knowledge as security patterns. Some categorisation scheme is useful to narrow scope and remove ambiguity when referring to different types of patterns. Security patterns can be organised in two groups (88):

- Structural patterns. These patterns will affect the implementation of the final product, and are represented as structures and interactions. They are used when defining software components.

- Procedural patterns. These patterns are used for process improvement to support development of enterprise level security strategy. Risk assessment often plays a central part in these patterns.

The organisation above is very coarse-grained and can be further defined by introducing dimensions of development stage or software abstraction level. Structural patterns are often defined as architectural or design patterns. These patterns deal specifically with problems at their respective levels of software development. This thesis is concerned with structural patterns, specifically *design patterns*. Unless otherwise is stated, security pattern will refer to security design pattern in following chapters.

4.1.1 Security Patterns as a Countermeasure

The work in (13) introduced security design patterns in software design tools to help developers create more secure software by increasing security awareness and making security a part of the design lifecycle. Design templates of two recognised security patterns were implemented in a design tool. The templates were applied in a case study, which showed how security vulnerabilities could be avoided at the design level by giving software developers security pattern tool support. This work is an important prerequisite for this thesis and is described in the following paragraphs.

Enterprise Architect (EA) (82) is a tool for designing and constructing software systems. EA supports UML design pattern templates which are UML documents that can be instantiated in software designs. The pattern support was extended with security pattern templates of the Intercepting Validator (78) and Role-Based Access Control (71) patterns. Figure 4.2 shows the Intercepting Validator template in EA. The template is a UML class diagram describing the static structures of the pattern. The pattern instantiation dialogue is shown in Figure 4.3. Pattern instantiation allows patterns to be merged with existing designs or creating a new design starting with the pattern structures.

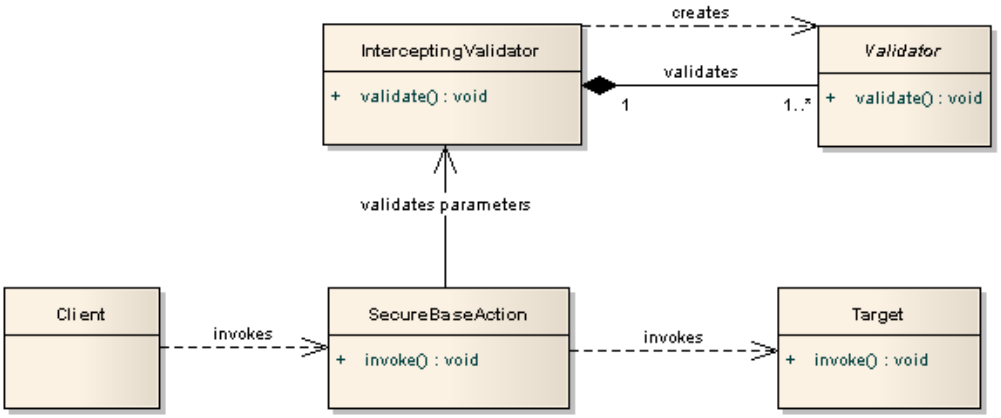


Figure 4.2: **Intercepting Validator** - Class diagram of the Intercepting Validator in Enterprise Architect.

The implemented pattern templates were tested in a case study. The case defined a system for administration of medical patient journals. The system requirements specified

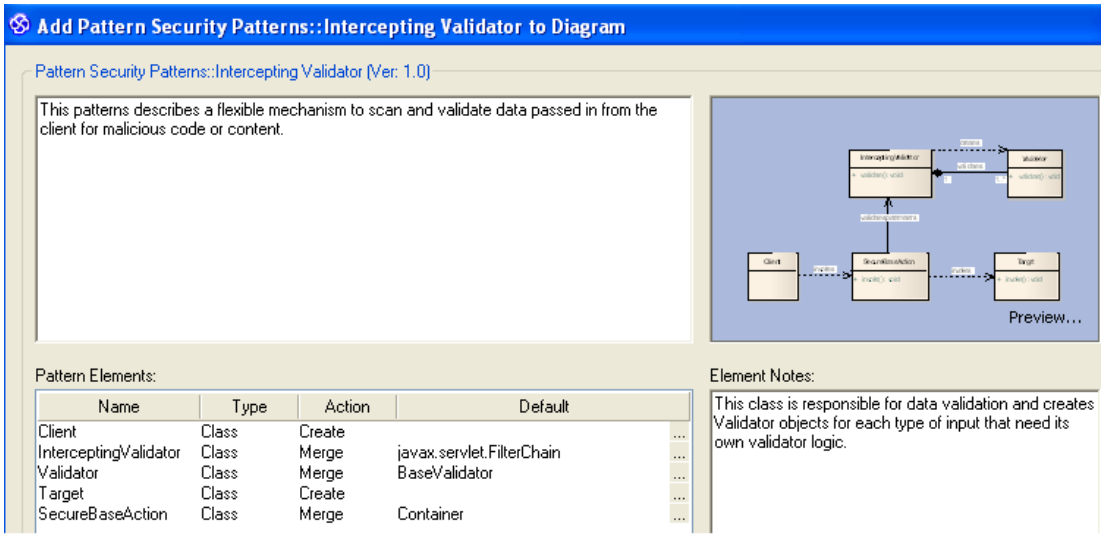


Figure 4.3: **Pattern Instantiation in Enterprise Architect** - Instantiation dialogue

that confidentiality and integrity of medical information must be ensured. The case study implementation was to show how security patterns can be instantiated through the software design tool in order to countermeasure vulnerabilities and mitigate security risk. An ad hoc instantiation process, shown in Figure 4.4, was designed to demonstrate the use of security pattern templates. Although this process can not be defined as a complete threat and countermeasure identification process, it resembles the stages followed and artefacts produced in most software development projects. Use cases, system requirements (already specified by the case) and initial design documents were used in a risk analysis where assets and threats and were identified. The STRIDE categories were used to identify the following threats: data disclosure and data tampering. Input validation and access control mechanisms were identified as countermeasures through the risk analysis. The process did not define any method or tool to identify these countermeasures based on threats. The countermeasures were simply chosen based on the author's software security knowledge and various vulnerability countermeasure sources (26; 71). After mechanisms to mitigate the risks were identified, the security patterns in the design tool were searched for a matching countermeasure mechanism and pattern. This was performed by browsing the descriptions of the pattern templates. The chosen pattern templates were then instantiated in a software design. The templates were instantiated in the medical journal system which was then implemented as a prototype

Web application. The mitigation of security risk was demonstrated by failed injection attacks to the system.

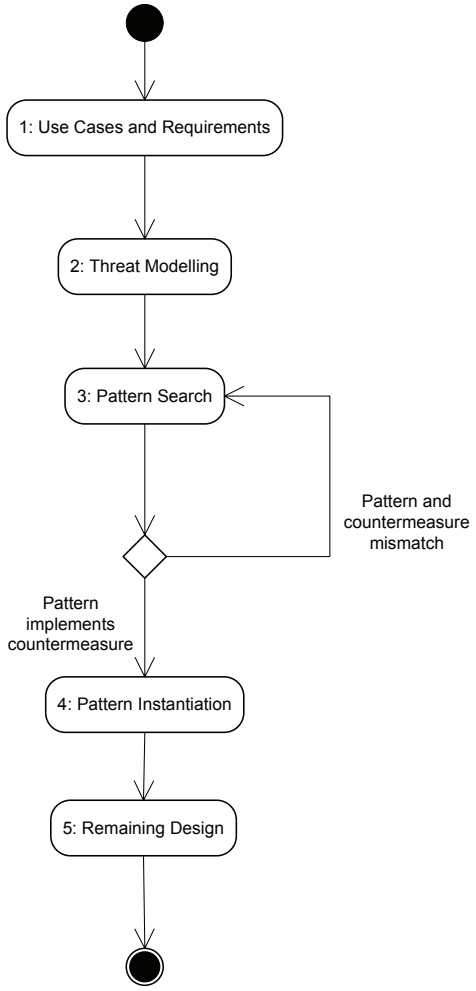


Figure 4.4: **Pattern Instantiation Workflow** - Pattern instantiation through design tools

The described case study displays a shortcoming in current secure systems development. There are methods to analyse and deal with security problems, i.e. risk management frameworks, threat modelling and security modelling, but there is a lack of methods and tools to tie these methods to activities in software development phases, i.e. design and implementation. Software developers do not have support for security modelling and

secure development techniques in development tools. This is contributing to the security knowledge gap. The author found that the security pattern templates of the described work will not provide enough assurance that the chosen patterns countermeasure the identified security risks. There is a need for a tool and method to assure that developers have chosen the right security patterns to counter vulnerabilities.

4.2 Architectural Analysis and Reviews

The analysis methods and reviews presented in this section are based on software architecture and design. The objective of these methods is to gather data to be able to make decisions based on knowledge about vulnerabilities, threats, impacts and probability. Modern approaches to risk analysis emphasise the importance of an architectural view and starting the analysis process early in the development lifecycle (22). This allows security defects to be avoided or mitigated at an early stage. Equally important is that security is meaningless without context. Security analysis and improvement activities must be coupled with the actual software product being built. Security can not be achieved just by adding some standard add-on security package. Founding security risk management on architecture and design artefacts allow explicit definitions of what needs to be secured, from who and what the software must be secured and for how long it can be considered secure.

When performing architectural analysis and reviews, choosing the right people is important (22). Risk analysis requires an understanding of potential business impact which depends on laws, regulations and business model. The system developers might build assumptions about the system and possible security risks, often based on implicit assumptions which might make them overlook possible sources of risk. Analysts external to the design and development team might help challenge these assumptions. System experts do however hold a lot of knowledge which is useful to analysis and reviews, advocating a combination of internal and external team members.

4.2.1 Architectural Risk Analysis

A distinction should be drawn between risk analysis and risk management (46). Risk analysis is about identifying risk and finding mitigation strategies. Risk management refers to tracking and mitigating risks through the development lifecycle. This section describes the basis of architectural risk analysis which has earned its importance to security through the high account of design-related security problems (22; 37).

There exists a wide range of risk analysis methods, some of these are briefly described in Table 4.2. Some methods calculate a value of an information asset and determine risk as a function of loss and probability; others use checklists of threats and vulnerabilities to determine a risk measurement.

Method	**Developer**	**Reference**
Commercial		
STRIDE	Microsoft	(35)
Security Risk Management Guide	Microsoft	(54)
ACSM/SAR	Sun	described in (30)
Cigital's architectural risk analysis process	Cigital	(46)
Standards-Based		
ASSET	NIST	http://csrc.nist.gov/asset
OCTAVE	SEI	(3)

Table 4.2: **Risk Analysis Methods**

McGraw (22) describes a prototypical analysis approach with a set of activities. These activities are rather general and can be fitted to suit a wider range of methods. It is valuable in capturing the essence of a risk analysis method, and describes the following activities:

Learn as much as possible about the target of analysis. Study the specifications, architecture and design documents. Include possible other artefacts, e.g. code. Identify threats and sources of attack. The learning activity is suited for brainstorming sessions and group discussion.

Discuss security issues surrounding the software. Discuss how the system works and note areas of disagreement or ambiguity. Identify vulnerabilities, exploits and possible fixes. Understand planned security controls because these may introduce new security risks.

Determine probability of compromise. Describe attack scenarios for exploits of vulnerabilities. The likelihood of compromise is a function of protection and threat capacity.

Perform impact analysis. Determine impacts on assets and business goals.

Rank risks. Risk analysis is a decision-support tool. Ranking the risks is a way of guiding the judgement calls to be made regarding risk mitigation.

Develop mitigation strategy. Recommend specific countermeasures to mitigate risks.

Report findings. Describe risks with attention to impact. It is important that decision makers have the information needed to spend limited mitigation resources effectively.

4.2.2 Architectural Reviews

Deliverables of software development activities can be made subject to software reviews which is a process examining the deliverable and providing approvals or comments. The review can be performed by any interested party, e.g. project personnel, managers, users, customers or owners. The deliverable can be any technical document such as project plans, budgets, requirements documents, specifications, design, source code etc. The goal of performing a review is to assess the quality of the software product and possibly identify defects or weaknesses. The review process may be governed by a set of written rules, the degree of formality in different review methods vary greatly as will be seen from the summary of review and inspection techniques. The IEEE Standard for

Software Reviews (36) defines a common set of activities for software reviews. These are closely linked to the software inspection process developed by Fagan (25). The activities are:

- Overview: Qualified personnel ensures that reviewers understand the goals of the review, procedures and the available materials.
- Preparation: Examination of the review materials for anomalies, i.e. defects according to the review goals.
- Examination: Reviewers pool their findings to agree on the status of the reviewed artefact.
- Rework: Assigned personnel, possibly the developer of the artefact, performs actions to correct anomalies and defects.
- Follow-up: Ensure that corrective actions are applied correctly.

There are obvious benefits to architectural software reviews, one is that it enables early detection of defects (63). It enables developers to identify defects earlier than what is possible during testing and operations feedback. Gilliam et al. (29) recommends introducing a form of security review in every phase of the software development lifecycle. McGraw (47) points out the importance of external analysis, i.e. by someone outside the design team. Having an independent view of the system is important because the designers and developers are often sceptical that their system may have flaws. Software reviews are a way of performing external analysis.

Performing architectural risk analysis and reviews is knowledge intensive (46). Reviewers need some security expertise to be able to identify anomalies. This knowledge may come from the experience of the reviewers or more explicit sources such as security checklists, vulnerability/threat/attack lists or compilations. Examples are the STRIDE (35) model of risk categories, attack patterns (33), design principles (86) or security issues in widely used frameworks such as .NET or Java.

The IEEE 1028-1997 standard defines five types of reviews:

- Management review - a review performed by management to evaluate work status.

- Technical review - qualified technical personnel reviews the software product for discrepancies between the product and specifications.
- Inspection - a formal review where reviewers follow a process to find defects.
- Walkthrough - an informal review where the product author leads reviewers through the product and the reviewers may point out defects or ambiguities through questions or comments.
- Audit - the goal of this review is to evaluate compliance with specifications, standards, contracts or other criteria.

Chapter 5

Sharing Security Knowledge

This chapter gives a rationale for sharing security knowledge and describes state-of-the-art mechanisms such as widely used vulnerability repositories. The chapter also describes past and present work in the connection between security knowledge repositories and development tools and methods. Sharing security knowledge and applying it during development is a hot topic in current software security research.

5.1 Why Sharing is Important

Security knowledge management is a set of activities concerning collection, encapsulation and sharing of expert security knowledge. The sharing of security knowledge is perhaps the most important because of the current state of software security. This is based on two observations: 1) the importance of expert knowledge in software security and 2) the knowledge gap between software practitioners and security experts. Being able to recognise situations where common attacks can be applied is crucial in effective risk assessment (47). This requires expert knowledge. The literature describes a security knowledge gap (46; 85) between software developers and security experts or engineers. There are sources of security knowledge, they are however not available or accessible to those who need them the most; those directly involved with software development. Sharing of security knowledge can build consensus on best practices and standardisation of secure development methods and tools. This collaboration is critical

to making software security a unified practice (46). Work in sharing security knowledge includes defining knowledge constructs such as taxonomies, catalogues etc., and tools and methods.

5.2 Vulnerability Repositories

There are several publicly available vulnerability repositories or databases. They provide informal descriptions, catalogues and information about patches to repair vulnerabilities. SecurityFocus (74) is among the most well-known sources of security information on the Web, with over 18 million page views a month. The SecurityFocus vulnerability database provides vulnerabilities that can be browsed by vendor, title, version and CVE. CVE is a dictionary of publicly known information security vulnerabilities and exposures (58). The dictionary provides a list of standardised names for vulnerabilities, and enables data exchange between security products by the use of CVE's common identifiers. BugTraq is a mailing list related to the SecurityFocus community for disclosure and discussion of computer security vulnerabilities. The National Vulnerability Database (60), hosted by NIST, is a U.S. government repository of security vulnerability management data. Similar to SecurityFocus, vulnerabilities can be searched by several attributes; keywords, vendor, version, CVE etc. The database contains over 29.000 CVE vulnerabilities and 13.000 vulnerable products. The Open Source Vulnerability Database (OSVDB) (62) is an independent vulnerability database aiming to provide accurate and unbiased technical information about security vulnerabilities. OSVDB keeps mappings to CVE and can be exported to XML so that anyone can keep a local copy of a OSVDB snapshot. The advanced search interface of the database is very sophisticated with its vulnerability classification scheme, see Figure 5.1.

Other vulnerability repositories and mailing lists are the Secunia Vulnerability Archive (73), ISS X-Force Alerts and Advisories (81), Packet Storm Advisories (2), CA Vulnerability Information Center (10), Cerias' CoopVDB (16) and the Open Web Application Security Project (65). Each of these repositories does not individually cover all aspects of a vulnerability, and they lack a common format (64). The chosen representations of vulnerabilities are varying from repository to repository, and the informal descriptions

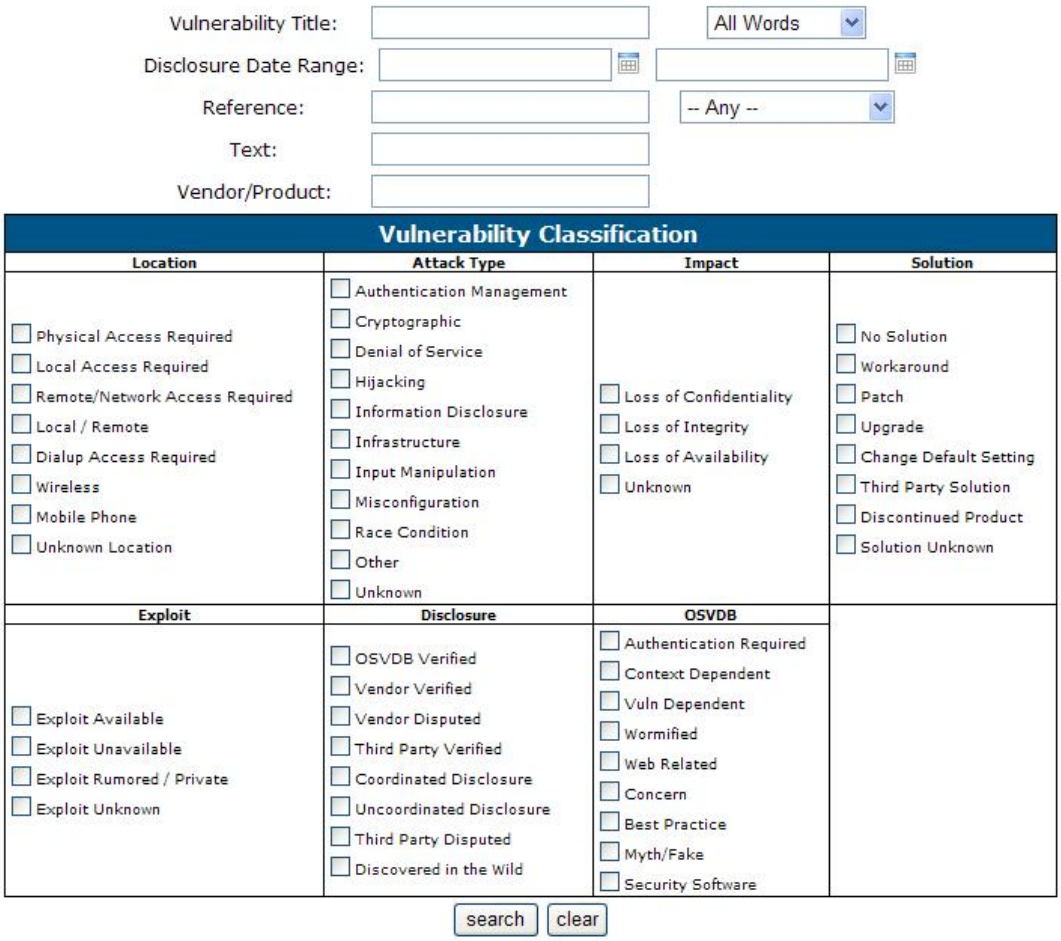


Figure 5.1: **OSVDB Vulnerability Search Interface** - Advanced search

are difficult to utilise in automated tools. The vulnerabilities are usually system specific and are counter measured by patches, making the repositories more useful for system administrators than software developers. There is a lack of knowledge sharing repositories and architectures for general software security knowledge that can be applied at development time.

5.3 Security Knowledge at Development Time

Schumacher et al. (72) describes a prototype security pattern search engine as a security improvement tool. The engine is implemented as an expert system with three main components:

- Security Knowledge Base - contains core security concepts and taxonomies together with a mapping to the structure of security patterns.
- Inference Engine - processes the user queries, applies the query to the knowledge base according to a set of inference rules, and presents answers.
- User Interface - Web interface receiving user input and presenting output.

Figure 5.2 shows the use cases of the security pattern search engine. The system seeks to help users solve typical security problems at different layers of abstraction. Users can search and explore patterns and simulate pattern scenarios which explore the consequences of applying a pattern. The knowledge base is maintained by security experts, shown as *Expert* and *Pattern Author* in the figure. A pattern author can access a library of predefined pattern elements, e.g. threats or countermeasures. A *Hierarchy Builder* is intended to identify relations between patterns to compute a pattern hierarchy.

SHIELDS (75) is an EU project launched in January 2008 concerned with detection and elimination of software vulnerabilities through security models. The objective is to increase software security by providing software practitioners methods and tools to prevent occurrences of known vulnerabilities. The project will conduct research and development on modelling software security vulnerabilities and countermeasures, creating a repository for storage and distribution of the models, and extending security

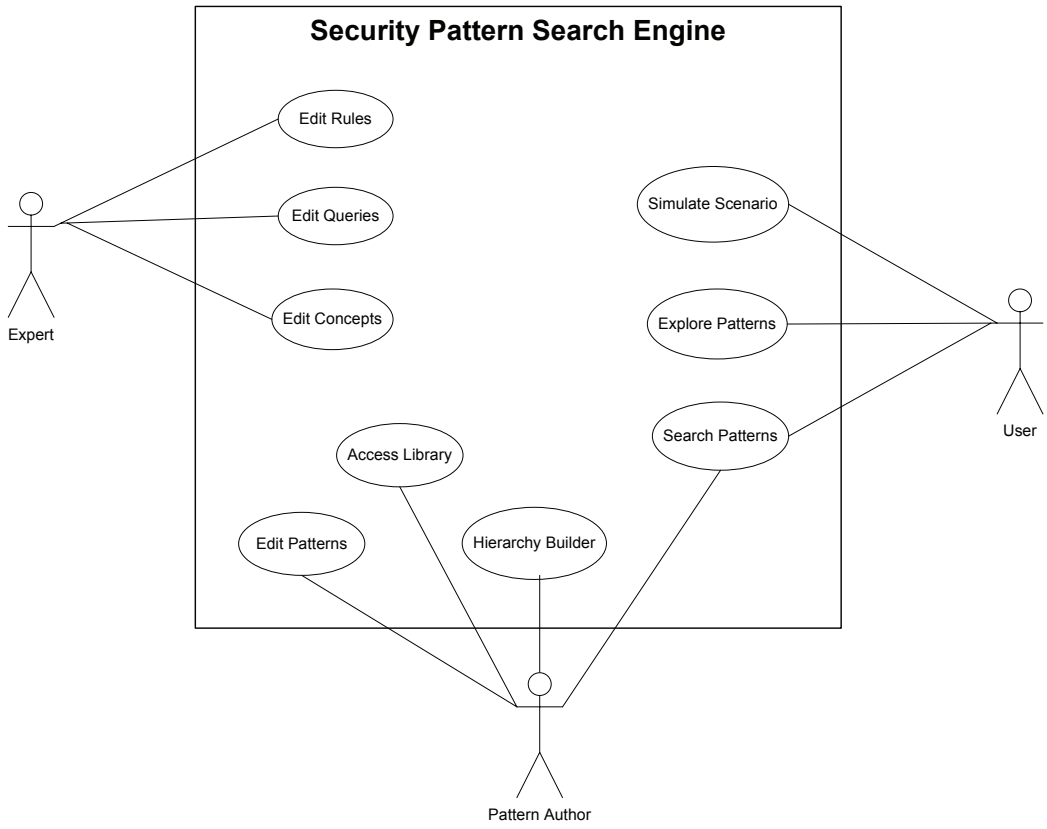


Figure 5.2: **Security Pattern Search Engine** - Use cases

development tools and methods to use the repository. The technical approach of the project is based on a knowledge repository known as the Security Vulnerability Repository Service (SVRS) (8). The service is to be made internet-accessible and defines the following use scenarios through supported security tools:

- Security experts can represent and publish formal vulnerability models that can be used by development and security tools.
- Help developers detect and remove software vulnerabilities from within their development tools.
- Gather metrics about vulnerabilities detected and removed, to be used in process improvement intitiatives.
- Provide information about design and code conformance to security policies.

Figure 5.3 shows the SVRS and suggested tools for software security.

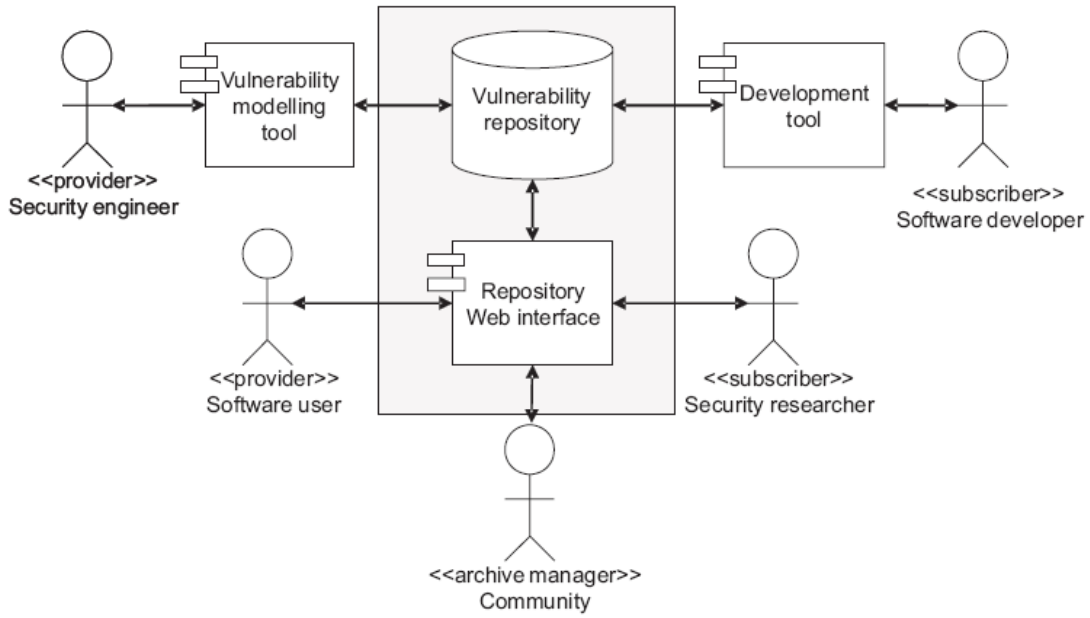


Figure 5.3: **Security Vulnerability Repository Service** - Main components and actors

The SVRS enables software practitioners and development tools to stay up-to-date with the latest security knowledge. The repository approach is designed for fast dissemination of security information from security experts to software developers, hoping to bridge the security knowledge gap. The SHIELDS project will also create a SHIELDS Verified certification to help development organisations document that known security vulnerabilities are not present in their software products. The certification refers to quality processes and mechanisms.

Part III

Contribution

Chapter 6

Research Agenda

This chapter describes the research design of the thesis. A hypothesis and research objectives are stated in the light of previous work and tendencies in current software development practices. A research process is tailored to fit the hypothesis and a work plan is included which describes milestone activities.

6.1 Hypothesis and Objectives

The preliminary study showed a shortcoming in state-of-the art secure software development practice; security is rarely taking part throughout the entire software lifecycle. Many companies treat security as an add-on typically done after an application is deployed. This is confirmed by studies performed on e-business applications, showing that 70% of found security defects are design related and 47% of these defects can be regarded as severe design flaws (37). This suggests that the old security approach based on security software and a patch-and-repair strategy is not enough. We need to focus on the applications themselves. The lack of security work in early stages of development is partly based on lack of methods, tools and security awareness in general, as pointed out in the preliminary study. Although design flaws impose severe security risks, fixing them does not have to be expensive. We believe that the flaws can be avoided in the design phase if software developers possess secure development methods and tools. This thesis will introduce new security improvement tools to improve current software

security practice. The following hypothesis is stated:

> Including security improvement tools and methods in early software development lifecycle phases will contribute to development of more secure software by mitigation of vulnerabilities during design, increasing software security awareness and bridging the software security knowledge gap.

The research objectives (RO) of this thesis are described in Table 6.1.

RO1	Develop a security modelling tool that enables software security experts and practitioners to create models of software security vulnerabilities and countermeasures.
RO2	Develop software design tool artefacts that provide reusable security expert knowledge in a format that is applicable at development time. This tool should apply the output of the tool described in RO1 to guide design of vulnerability countermeasures in software.
RO3	Document the tools as a proof of concept and evaluate contributions by performing a case study of the tools in objectives 1-2.

Table 6.1: **Research Objectives**

It is hypothesised that the use of these tools will contribute to development of more secure software by systematic prevention of software vulnerabilities. A security modelling tool is expected to increase security awareness by making security a part of early software development stages and supporting the identification process of vulnerability countermeasures. A design tool supporting security will bridge the gap between security professionals and software developers. The use of these tools in combination should contribute to assure that the designed software countermeasures possible vulnerabilities.

Note that measuring the assurance given by performing security modelling is without scope of this work. This work is focused on the benefits of *tool support* to increase awareness and accessibility of security knowledge.

6.2 Research Process

This thesis is an extension of previous work by the author (13). Figure 6.1 shows an activity diagram of the research process. The Prerequisite Work phase provides the basis for the research to be performed and contains the previous work and a preliminary study. A hypothesis is stated after the prerequisite work is completed. The implementation phase will develop a security modelling tool as described in research objective 1. Along with the implementation, a model describing how the modelling tool can be combined with software design tool artefacts such as security pattern templates will be defined. The implementation is followed by a case study which is designed to document the use cases of the implemented tools, and ultimately describe how security can be incorporated in early stages of a software development lifecycle. The case study functions as a tool supported secure development proof of concept. The hypothesis is finally evaluated by a qualitative discussion. This is a notable limitation of the research process, which is necessary to limit the scope. The chosen research process will not be able to quantify the achieved results and effects of made contributions. Validation is made subject to further work.

6.3 Work Plan

Table 6.2 shows a work plan with milestone activities, a workload estimate in number of weeks and percentages of total workload, and activity documentation references. The plan is used to implement the research process by guiding the work and limiting scope.

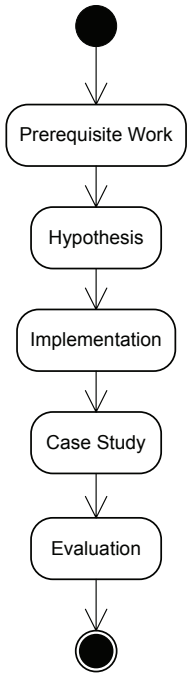


Figure 6.1: **Research Method** - Activity diagram

Activity	**Estimate**		**Reference**
Review previous work	0.5	2.5%	Found in (13)
Problem description	1	5%	Chapter 1
Preliminary study	6	30%	Chapter 2 through 5
Summary of previous work and need of further research	0.5	2.5%	Section 4.1.1
Define hypothesis and research process	1	5%	Chapter 6
Define tools context of use and requirements	1	5%	Chapter 7, Section 9.2
Implementation	4	20%	Chapter 9
Case study	2	10%	Chapter 10
Evaluation of method and results	2	10%	Chapter 11
Concluson and further work	2	10%	Chapter 12
Total	**20**	**100%**	

Table 6.2: **Work Plan**

Chapter 7

Using a Countermeasure Model During Development

This chapter gives a high-level description of suggested security improvement methods and tools of this thesis. The approach is based on applying security modelling to identify possible weaknesses in early software development artefacts, which may lead to security vulnerabilities. The output of the security modelling is used to apply countermeasures in software design. Countermeasure modelling is introduced as a model to describe vulnerabilities, countermeasures and their relations. This thesis will use the suggested approach to identify suited security design patterns.

7.1 Approach and Countermeasure Modelling

The need to incorporate security throughout the software development lifecycle (SDL) has been identified to improve software security practice in general. This thesis seeks to do this by providing methods and tools that enable developers to apply security information and knowledge during development. Specifically, the goal is to avoid and mitigate possible software vulnerabilities early in the development lifecycle. Developers should have a sense of assurance that they have implemented countermeasures that target the right issues.

This thesis suggests a process to fill the security void early in the SDL. There exist

methods to analyse and handle security in specification and design phases, e.g. architectural risk analysis, security requirements, abuse cases etc. The output of these activities is to a lesser extent used in software development to produce more secure software. There is a lack of tools and integration between these security improvement activities and software development activities, creating a gap between security expertise and secure software development practice. We need methods and tools to integrate security improvement activities throughout the SDL.

The approach in this work is based on a simple model shown in Figure 7.1. Security experts compile security knowledge in various formats to be shared in some repository. Software development tools with security improvement features are developed to make use of the repository, helping developers to create more secure software.

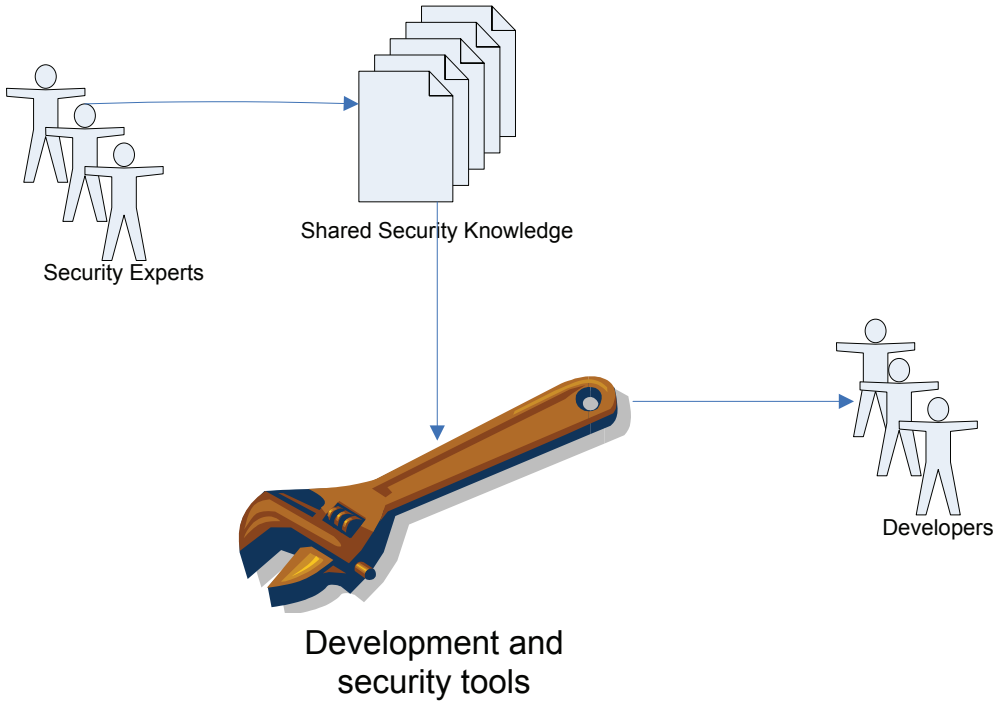


Figure 7.1: **Security Knowledge Tools Approach** - Security knowledge and tools at development time

The model defined in Figure 7.1 is a general architecture that can be used for a wide range of security knowledge formats and tools. The focus of this work is to identify and mitigate software vulnerabilities. The approach is based on tool supported security modelling of possible vulnerabilities and countermeasures and applying the output of the modelling during software design. Figure 7.2 visualises this approach as an add-on to a simple development lifecycle which resembles the classic Waterfall model (87). Although this figure shows a sequential process, the approach is suited for an iterative development process as well. After requirements are specified, security modelling is performed to gain an understanding of possible security issues with the software to be built. The security modelling activity takes requirements and possibly design documents as input, as shown by the backwards arrow from *Design* to *Security Modelling*. The security modelling identifies potential vulnerabilities, threats and countermeasures. The countermeasures are used in the design phase to prevent security problems.

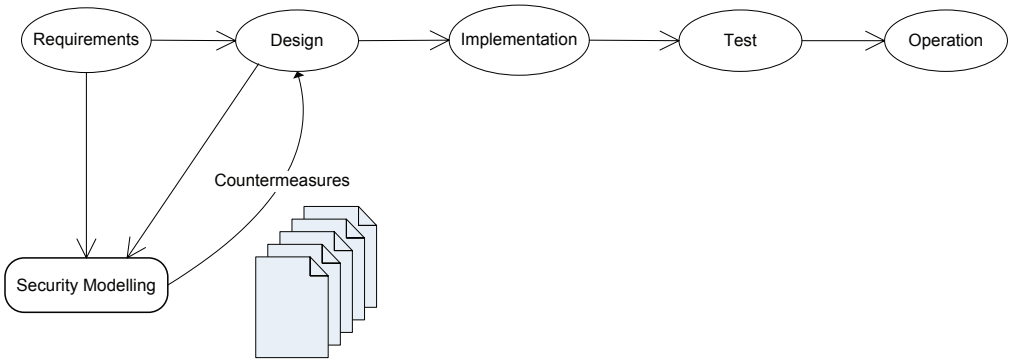


Figure 7.2: **Software Development Lifecycle With Security Modelling** - The chosen approach incorporates security modelling in the SDL

The Security Modelling activity in Figure 7.2 produces output which is based on **countermeasure modelling**; a modelling activity which seeks to analyse software artefacts to identify possible vulnerabilities and determine suited countermeasures. Figure 7.3 shows a meta-model describing countermeasure modelling concepts and relations. The *Vulnerability* class describes a software vulnerability; a flaw or weakness in a software artefact that could be exploited. A vulnerability can be a class of vulnerabilities describing a group of similar vulnerabilities that stem from the same type of flaw or weakness,

or a specific instance. A vulnerability is described in a *VulnerabilityModel* which is made to understand the vulnerability, i.e. what are the causes and how this vulnerability is instantiated in this specific instance. A VulnerabilityModel is realised by a vulnerability cause graph. The VulnerabilityModel is related to a *CountermeasureModel* which describes how the vulnerability can be avoided or mitigated. A CountermeasureModel is realised by a security activity graph. The VulnerabilityModel and CountermeasureModel classes can both be decomposed, as shown by the 'part of'-relation. There are possibly several countermeasures that can be combined to make a CountermeasureModel, these are defined as *CountermeasureItems*. A CountermeasureItem is a security improvement strategy, activity or artefact such as the touchpoints described in Section 2.3 or countermeasures in Chapter 4. A CountermeasureItem is further divided into a *ToolItem* or a *DocumentItem*. ToolItems are used by tools, examples are static analysis rules, test configurations and design templates. DocumentItems are used by humans, examples are: security requirements, checklists and inspection techniques. Countermeasure modelling is performed by using security modelling techniques to map these meta-modelling concepts to instances in the software being analysed.

7.2 Using Security Design Patterns

This thesis focuses on security design patterns as the countermeasure and limits the scope of CountermeasureItems to security pattern templates as described in (13). Security patterns are made available to developers through security pattern design templates, countermeasure modelling is meant to help developers identify and apply the right pattern before a flaw is instantiated in software design. The suggested approach with tool support is designed to provide coupling between security expertise and development practice as described in Figure 7.1.

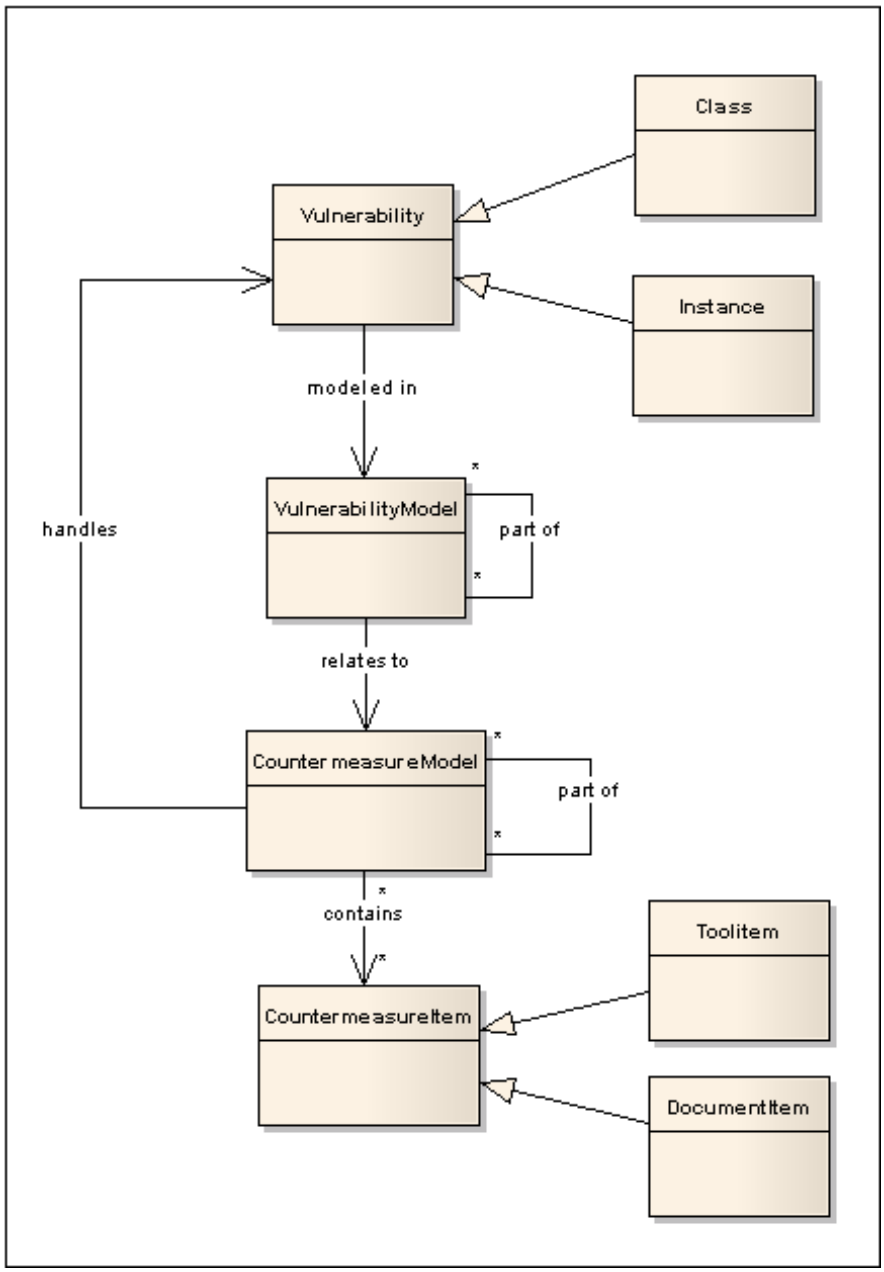


Figure 7.3: **Countermeasure Modelling Domain Model** - Concepts of countermeasure modelling (Use of this model is permitted by SINTEF)

7. USING A COUNTERMEASURE MODEL DURING DEVELOPMENT

Chapter 8

Extending Seamonster

SeaMonster (61) is an open source security modelling tool developed by students at NTNU for SINTEF through a software development project. SeaMonster is designed to help security experts and software developers model security through any phase of software development. This chapter describes the current status of SeaMonster; functionality and technical details. Possible further development is suggested. The chapter serves as an examination of SeaMonster as a baseline for the countermeasure modelling tool needed in the thesis.

8.1 Current Status

SeaMonster allows users to model threats, attacks and vulnerabilities in different diagram notations and tie the diagrams together in one security model. This security model is used to summarise a set of diagrams aimed at describing different views of a particular software security issue. The currently implemented modelling techniques are attack trees, vulnerability cause graphs and misuse cases. These were described in Section 3.2. Figure 8.1 shows a screenshot from SeaMonster. The centre frame shows the current model, in this case a security model diagram consisting of three sub diagrams: an AT, a VCG and a misuse case. These are connected together showing that there is a relationship between the models, i.e. the vulnerabilities can be exploited trough attacks, and the attacks can be performed in the misuse cases. The user interface is

based on a palette of modelling components, which is different for each type of diagram. Component properties such as descriptions or implementation cost can be used to further detail the components. These properties are mainly limited to textual names and descriptions so far.

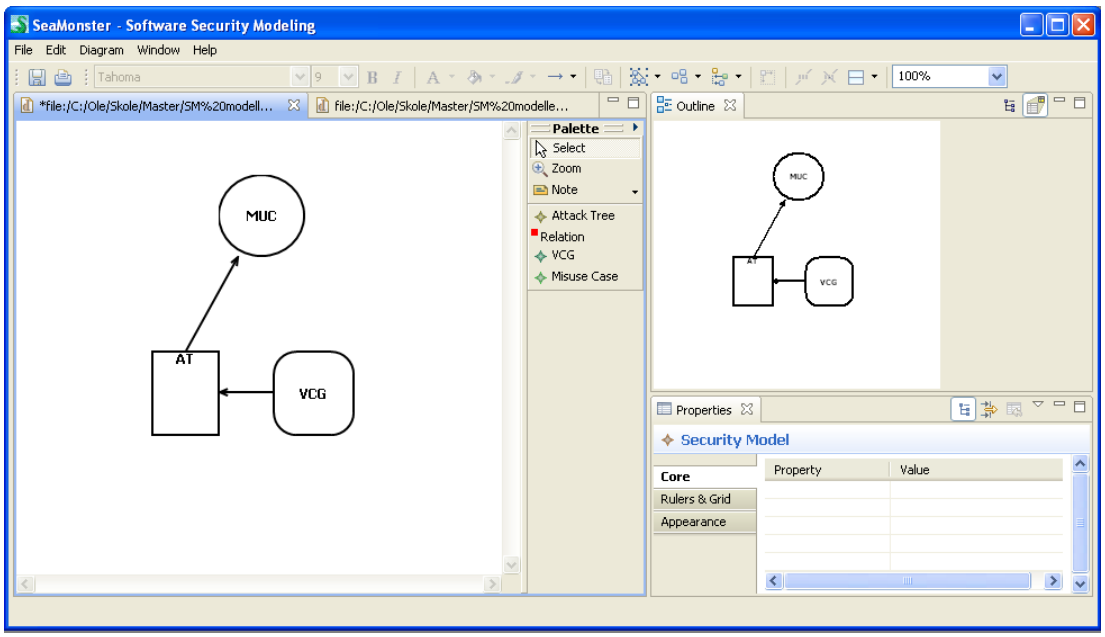


Figure 8.1: **SeaMonster Overview** - Security model diagram

SeaMonster is distributed as open source and is therefore free to use and modify. It is built using the Eclipse Graphical Modeling Framework (GMF) (19) which to a large degree defines the system architecture. A thorough understanding of this framework and its dependencies is needed to further develop and improve SeaMonster.

The Eclipse (21) platform is an open source development platform comprised of extensible frameworks and tools. GMF is a framework for developing Eclipse graphical editors, and functions as a bridge between the Eclipse Modeling Framework (EMF) (20) and Graphical Editing Framework (GEF) (18). The relations between these frameworks are described in Figure 8.2. EMF is a modelling and code generation framework for building applications based on a structured data model, often referred to as a domain model. GEF is a framework that takes an existing application data model and creates a graphical editor for this model. It provides layout and rendering for displaying

graphics.

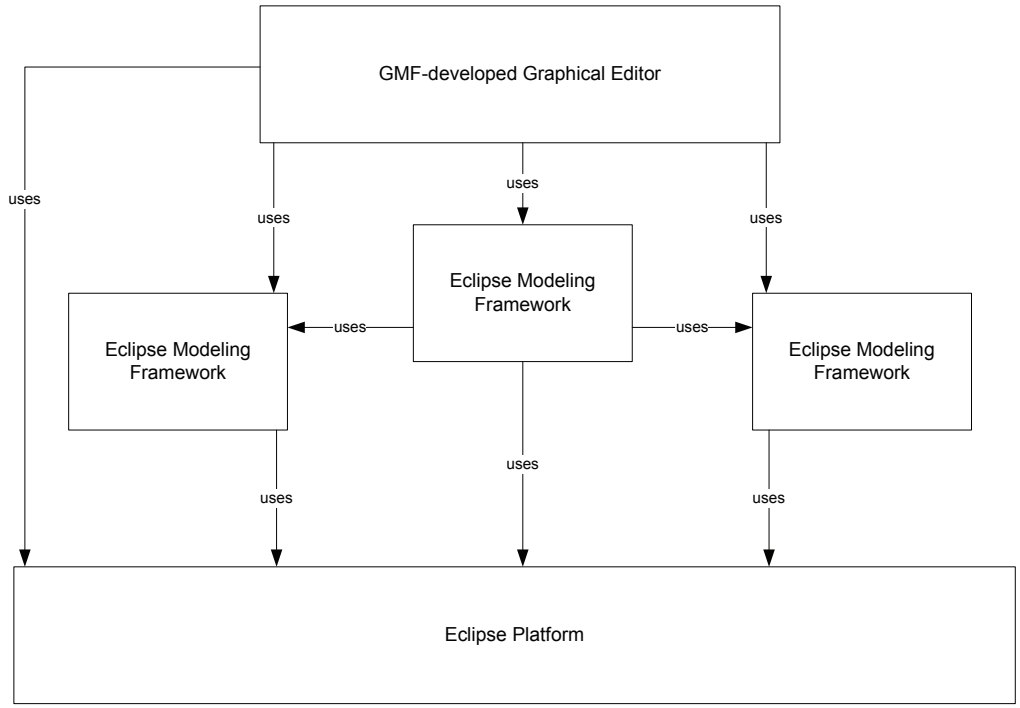


Figure 8.2: **GMF Dependencies** - Eclipse frameworks relations

GMF-based development

Figure 8.3 shows an overview of the GMF development process and models used during development. The domain model, graphical definition and tooling definition can be developed in arbitrary order; an intuitive approach might however assume the domain model is developed first. The domain model defines the structure of the application data and can be modelled using a UML (31) modelling tool. The structure defines data, their attributes and relations. The graphical definition model is central to GMF and contains the graphical elements that are to appear in the final editor. Shapes, colours, labels etc. can be defined here. The optional tooling definition model specifies a palette and other periphery such as menus and toolbars. These tools are used to create diagram components. Once the above definition models are created, a mapping model is used to link the graphical and tooling definitions to the domain model(s). The mapping of definitions enables reuse of components, i.e. a graphical or tooling definition may

work well for several domain models. The mapping model defines what parts of the data model are to be included as components in the diagram. Each component has a graphical representation from the graphical definition and optionally a tool such as a creation tool. Finally, a generator model specifies implementation details for the code generation. This includes packaging properties, application settings and more. Once the generator model is defined, the graphical editor can be generated. The resulting application code is a working graphical editor for the domain model. To add a feature not supported by the Eclipse modelling frameworks, it is possible to apply any wanted changes to the code.

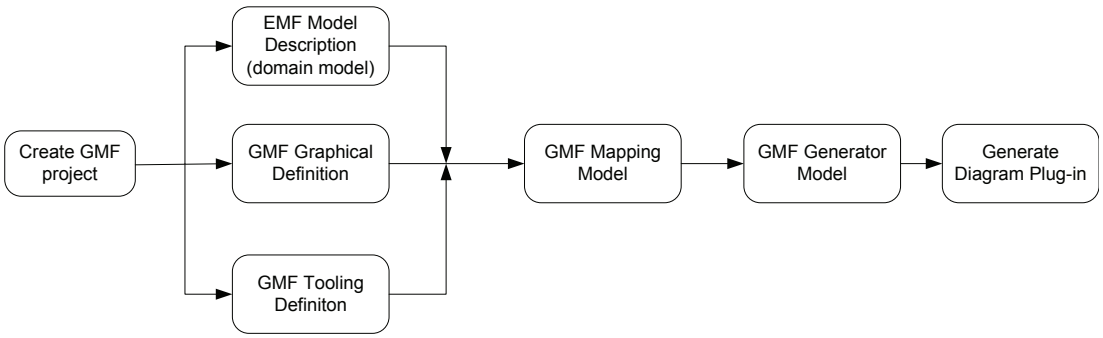


Figure 8.3: **GMF Overview** - basic development process

SeaMonster models and GMF components

The concepts to be modelled in a GMF-based editor must be included in the domain model. In the case of SeaMonster, any concept used in the security modelling approaches must be included in the domain model. Figure 8.4 shows a high-level class diagram of the domain model.

The *SecurityModel* class represents the high-level security model which ties together diagrams of different modelling approaches. The SecurityModel may contain one or more sub diagrams of the types *VCG*, *AttackTree* or *MisUseCase*. These define the modelling approaches available in the editor. The extentions of *SubDiagram* are reduced to single classes in the figure to increase readability. To expand SeaMonster with other modelling approaches, the domain model has to be expanded by creating a new extension of SubDiagram. The class *VCG* is expanded in Figure 8.5.

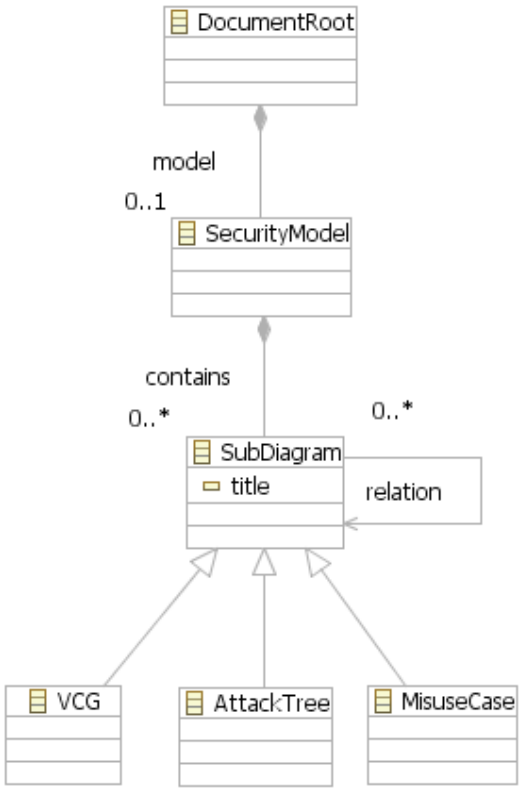


Figure 8.4: **SeaMonster Domain Model** - Class diagram of the domain model

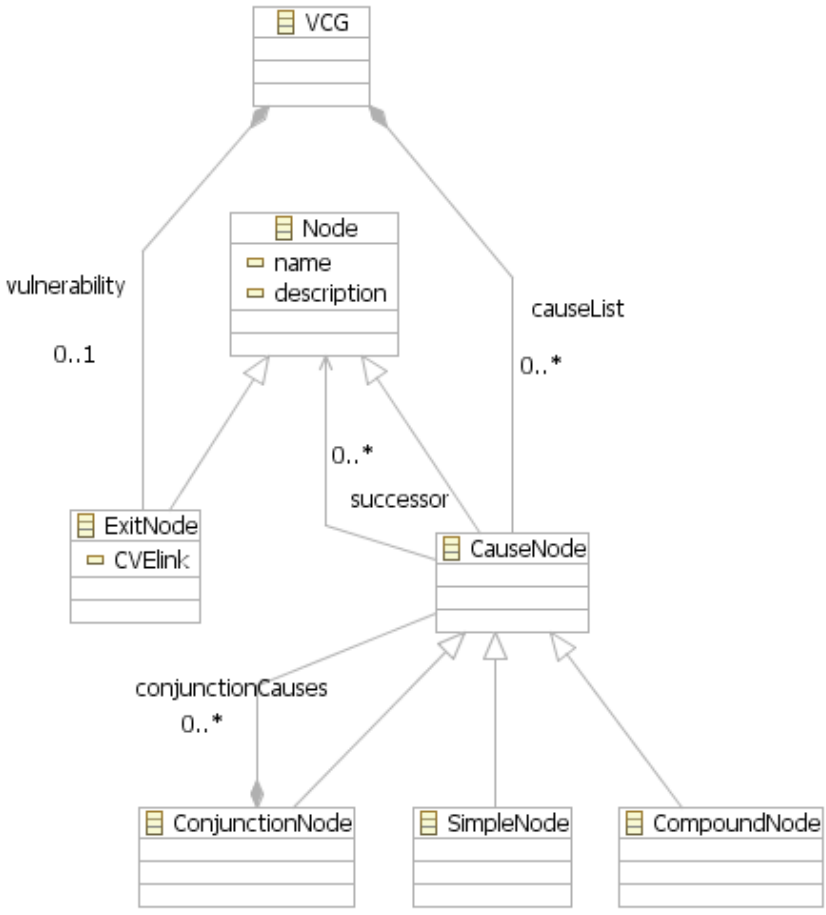


Figure 8.5: **VCG Data Definition Model** - Vulnerability cause graphs defined in the domain model

Refer to Section 3.2.4 for a detailed explanation of vulnerability cause graphs. The data definition of a VCG is the most complex of the diagrams because of numerous relations including constraints between components. VCG nodes are divided into *ExitNodes* and *CauseNodes*. An ExitNode represents the vulnerability described by the graph, and there is only one of them in each graph. All other nodes are a subtype of CauseNode. The CauseNode is subtyped by three classes; *SimpleNode*, *CompoundNode* and *ConjunctionNode*. The difference between the first two is the visual representation. The ConjunctionNode differs from the other two in that it is a container component and therefore can contain other CauseNodes. Some classes are given attributes, e.g. *name*, *description* and *CVElink*. These can be used to describe diagram components or enable graph computations in the resulting editor. The CVElink attribute is currently not implemented in SeaMonster, but it is designed to link modelled vulnerabilities to vulnerability repositories such as CVE (58). The AttackTree and MisUseCase definitions are fairly similar to the VCG definition and are left out here.

The GMF graphical definitions define the diagram notations. There is a variety of predefined symbols and external images can be utilised. The technical details of the graphical, tooling and mapping definition will not be explained here because they are rather straightforward with respect to GMF-development.

Components and deployment

SeaMonster is arranged as a set of plug-ins running on the Eclipse platform. Each type of diagram is implemented as a plug-in to improve modularity and extendibility. SeaMonster consists of the following plug-ins:

- seaMonster: contains the domain objects.
- seaMonster.edit: contains the domain objects.
- seaMonster.secmod: contains the security model diagram code. This plug-in also defines the SeaMonster application startup.
- seaMonster.attackTree: contains the attack tree diagram code.
- seaMonster.vcg: contains the vulnerability cause graph diagram code.
- seaMonster.misuseCase: contains the misuse case diagram code.

The implementation of SeaMonster is highly modular due to the plug-in structure. A plug-in can be added or removed independent of other components.

8.2 Possible Further Development

The design of SeaMonster using independent plug-ins and choice of implementation technology (Eclipse frameworks) make it highly modular. For instance, a goal of GMF is to allow the graphical definition to be reused for several domains. This is possible by using a separate mapping model to link the graphical and tooling definitions to a domain model. The technical feasibility of improving and extending SeaMonster is not an issue.

A quick study of SeaMonster is performed to identify possible improvements that will contribute to the research in this thesis. These are described in the following paragraphs.

Adding other modelling techniques. SeaMonster currently includes attack trees, vulnerability cause graphs and misuse case diagrams. Adding support for other security modelling types and security diagrams would increase the degree of completeness with respect to security modelling. The modelling of countermeasures is not strongly supported. Identifying countermeasures during development is central to this thesis, implementing some countermeasure modelling technique is very relevant and feasible. Security activity graphs are a viable option.

Tighter coupling between diagrams. A security diagram currently connects different security diagrams in SeaMonster. This diagram gives a high-level overview of a security model with single nodes representing a diagram and connections describing their relations. The relations are simple links and do not provide any functionality except describing that there is a relationship between to diagrams. For instance, the goal of an attack tree exploits the vulnerability of a vulnerability cause graph. Improvements of the links between diagrams could include navigation between diagrams by clicking in the high-level security model.

Diagram decomposition. Security models can be large and complex. Implementing a diagram decomposition mechanism would increase usability and enable security modelling at different abstraction levels. This would enable (different) users to model with a choice of granularity, e.g. security experts might provide more detailed knowledge than developers who are not trained in software security.

Model calculations. Some modelling techniques are suited for model calculations. The cost of performing an attack can be calculated in attack trees if each node is described with a cost metric. Model calculations can be useful in a risk management process, i.e. for prioritising and ordering schemes.

Preferred paths. Security modelling techniques often describe several options to threat, attack or counter attacks. For instance, there are several ways to achieve unauthorised access: using a backdoor entry point, guessing or brute-forcing a password. Highlighting parts of a security model can be used to describe preferred solutions, most likely attack etc. This can currently be done by using appearance properties such as node colour, but creating tools for this is better for usability.

Resolved nodes. Marking a node as 'Resolved' is useful in tracking security issues. This requires additional diagram tools.

Model validation. Modelling techniques have constraints on diagram components (nodes and connections). Model validation ensures that all constraints are followed in models created in the editor.

Use of external knowledge repositories Chapter 5 described mechanisms for sharing security knowledge through tools. Use of external vulnerability repositories is a possible improvement. The repositories can be used to pull information on common vulnerabilities that are modelled in a SeaMonster diagram.

Standard model format and diagram interchange Following a standard format for diagram export could enable easier diagram interchange between SeaMonster and other modelling tools. The UML meta-model is a possible candidate.

Decisions regarding further development are made in Chapter 9.

Chapter 9

Realisation

This chapter describes the realisation of the security improvement tool suggested in this thesis. A security modelling tool for modelling of vulnerabilities and countermeasures is defined through 22 requirements. The tool design and implementation is based on extending SeaMonster (61).

9.1 Realisation Description

Chapter 7 described an approach based on countermeasure modelling and sharing security knowledge between security experts and software developers through security improvement tools. A security modelling tool for modelling of software vulnerabilities and countermeasures (Research Objective 1) will be implemented to be used in combination with software design tool artefacts (Research Objective 2) as shown in Figure 9.1. Security experts and practitioners perform countermeasure modelling in the security modelling tool. A complete realisation would store the security models in a security knowledge repository which could then be utilised from the software design tool. In the following realisation, the repository will be short-circuited to limit implementation workload. The countermeasure models will not be accessible outside the security modelling tool because of the short-circuit, this is however a small technical limitation that does not affect the contribution of the countermeasure modelling process. Software developers use the countermeasure models to find the right security

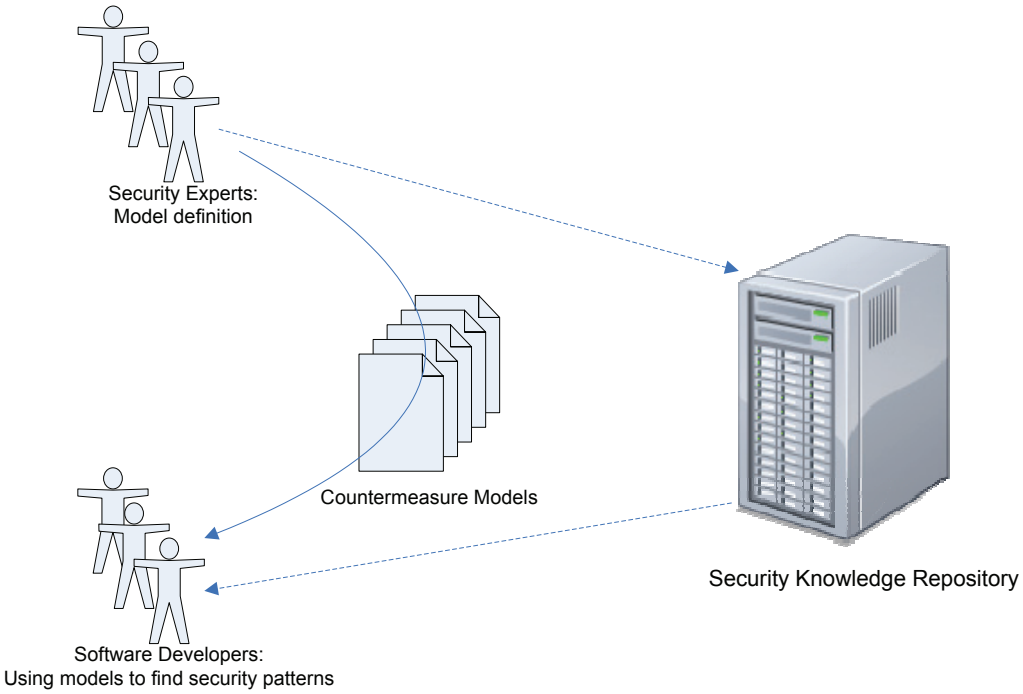


Figure 9.1: **Realisation** - The repository is short-circuited

pattern design templates.

The security pattern design templates are developed in previous work (13), the coming sections describe the realisation of the security modelling tool.

9.2 Requirements

Security is not yet an established niche in the commercial software modelling industry. It is therefore unfeasible to base the requirements of a security modelling tool on previous solutions. The author has created a list of requirements which are inspired by existing general-purpose modelling and design tools.

The following are suggested requirements for a graphical editor specialised for software security modelling:

1. There must be a canvas that expands as the user builds models exceeding the current modelling area.
2. There must be a tool palette from which the user can select between a predefined set of tools and modelling components.
3. The tool palette must be configured to the type of security diagram currently being modelled.
4. An unlimited number of predefined components can be added to the canvas.
5. There must be zoom in and zoom out functionality.
6. It must be possible to have several diagrams loaded simultaneously.
7. There must be a model selection pane to switch between the currently loaded diagrams.
8. Textual notes can be placed anywhere on the canvas. These are independent of the modelling components.
9. There must be a mechanism for highlighting parts of a diagram.
10. The editor must include an undo-operation that cancels the last operation.

11. The editor must include a cut-operation that removes the selected item(s).
12. The editor must include a copy-operation that copies the selected item(s).
13. The editor must include a clipboard and a paste-operation that pastes modelling components currently on the clipboard.
14. The editor must support modelling of software vulnerabilities through vulnerability cause graphs.
15. The editor must support modelling of vulnerability countermeasures through security activity graphs.
16. Modelling components must have the following textual properties: name and description.
17. The editor must support model decomposition.
18. The notation used in vulnerability cause graphs must follow the one used in (15).
19. The notation used in security activity graphs must follow the one used in (7).
20. It must be possible to save diagrams to file.
21. It must be possible to open diagram from file.
22. It must be possible to generate security activity graphs automatically from predefined vulnerability causes.

9.3 SeaMonster

SeaMonster will be used as a baseline for the described security modelling tool. SeaMonster was described and briefly evaluated for further development in Chapter 8. The decision to use SeaMonster is based on three aspects: fulfilment of requirements, technical feasibility of further development and potential for further development.

Fulfilment of requirements. SeaMonster fulfils the majority of requirements defined in the previous section so it is aligned with the security modelling tool described in this thesis. Table 9.1 shows a summary of an evaluation of fulfilled requirements in SeaMonster. Using SeaMonster as a baseline for the security modelling tool will relieve the workload and possibly improve the contributions of the thesis.

Requirement	Fulfilled	Comment
1	OK	The canvas can expand in all directions.
2	OK	
3	OK	
4	OK	
5	OK	
6	OK	
7	OK	
8	OK	
9	OK	Diagram components can be highlighted using a colour palette in the appearance properties.
10	OK	
11	OK	
12	OK	
13	OK	
14	OK	
15	N/A	Security activity graphs are not supported.
16	OK	
17	N/A	No decomposition mechanism exists.
18	OK	
19	N/A	Security activity graphs are not supported.
20	OK	
21	OK	
22	N/A	Security activity graphs are not supported.

Table 9.1: Requirements Fulfilled in SeaMonster

Technical feasibility. This refers to the feasibility of extending the SeaMonster implementation artefacts. SeaMonster has a highly modular design with one plug-in for every type of modelling diagram. This enables maintainability because changes in one plug-in will usually not introduce changes to other plug-ins. GMF-development is well documented through online tutorials and examples (19). The Eclipse platform pro-

vides model editors for GMF-artefacts, which should make it easier to understand the existing SeaMonster implementation artefacts.

Potential for further development. Some possible directions for further development were described in Section 8.2. This shows that SeaMonster has potential as a thorough security modelling tool. Requirements 15, 17 and 19 should be implemented as a contribution of this thesis.

9.4 Design

SeaMonster is developed using Eclipse Graphical Modeling Framework (GMF) which was described in Chapter 8. The architecture of SeaMonster is to a large degree defined by GMF and will remain unmodified. Some design modifications and additions are done in the data model and composition of system modules.

Figure 9.2 shows the system modules after adding the package seaMonster.sag. The packages seaMonster and seaMonster.edit contain the domain model, i.e. Java classes describing the security concepts to be modelled in the diagrams. These packages are used by the five packages defining one security diagram each. Adding security activity graphs in SeaMonster is done by implementing the package seaMonster.sag.

The domain model is expanded to define security activity graphs (SAG). The extension is designed to include SAGs in the security model diagram as is already done with attack trees, vulnerability cause graphs and misuse cases. Figure 9.3 shows the modified domain model of SeaMonster. The domain model is a data definition model describing concepts and relations to be modelled in the editor. The *SecurityModel* class represents the high-level security model which ties together diagrams of different modelling approaches. The SecurityModel may contain one or more sub diagrams of the types *VCG*, *AttackTree*, *MisUseCase* or the added *SAG*. These types define the modelling approaches available in the editor. The extensions of *SubDiagram* are reduced to single classes in the figure to increase readability. The class *SAG* is shown in full-scale in Figure 9.4. A SAG diagram contains one *Vulnerability* which has a name and a description. A vulnerability has one predecessor node, a *SAGNode*, as its root activity.

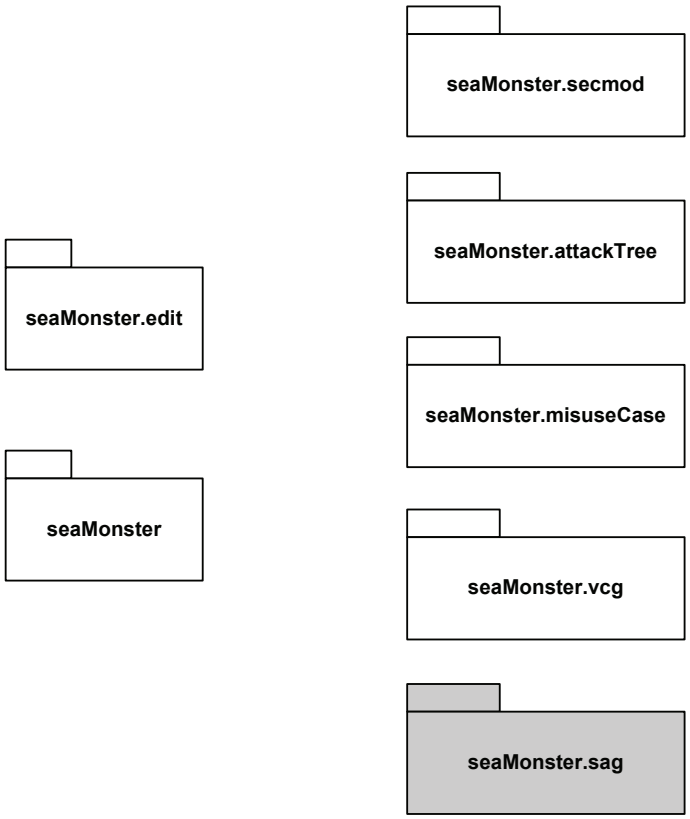


Figure 9.2: **SeaMonster Plug-ins Extended** - SeaMonster.sag implements security activity graphs

The root activity can be any node allowed in a SAG; or, and, split or activity. These nodes are represented by their respective classes in the figure.

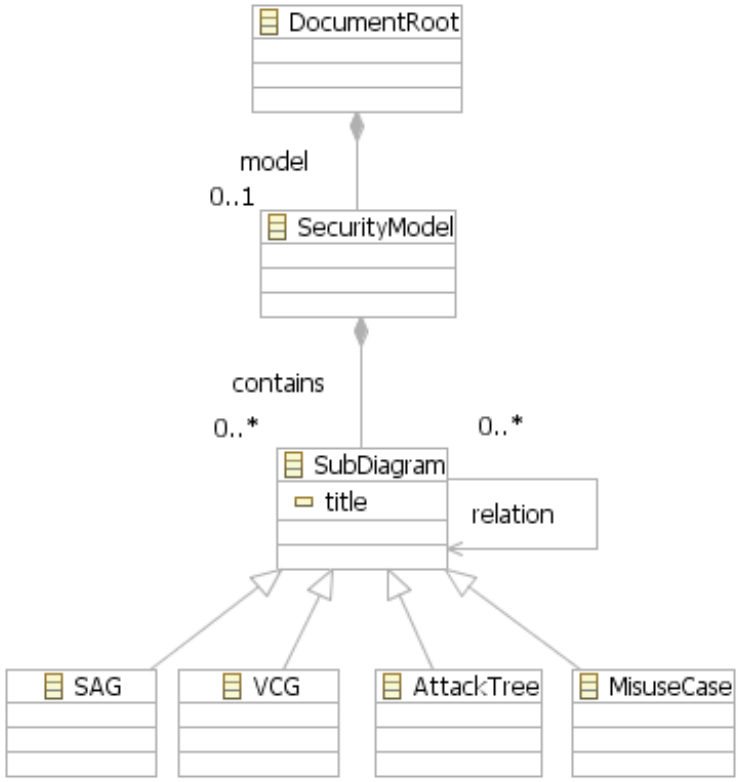


Figure 9.3: **Extended SeaMonster Domain Model** - SAG is added as a type of diagram

A SAG is a directed acyclic graph which means that no relation between modelling components in the graph should introduce a cycle. A cycle detection algorithm is used to detect possible cycles and should be applied each time a relation is created. The algorithm takes two nodes as parameters and returns true if a relation between the nodes does not introduce a cycle, and false if it does. The algorithm is the same as the one used in the existing attack tree plug-in. Pseudo-code is given below.

```
cycleDetection(nodeA, nodeB){
  X <- nodeB;
  while X not empty:
    Y <- remove first element of X
```

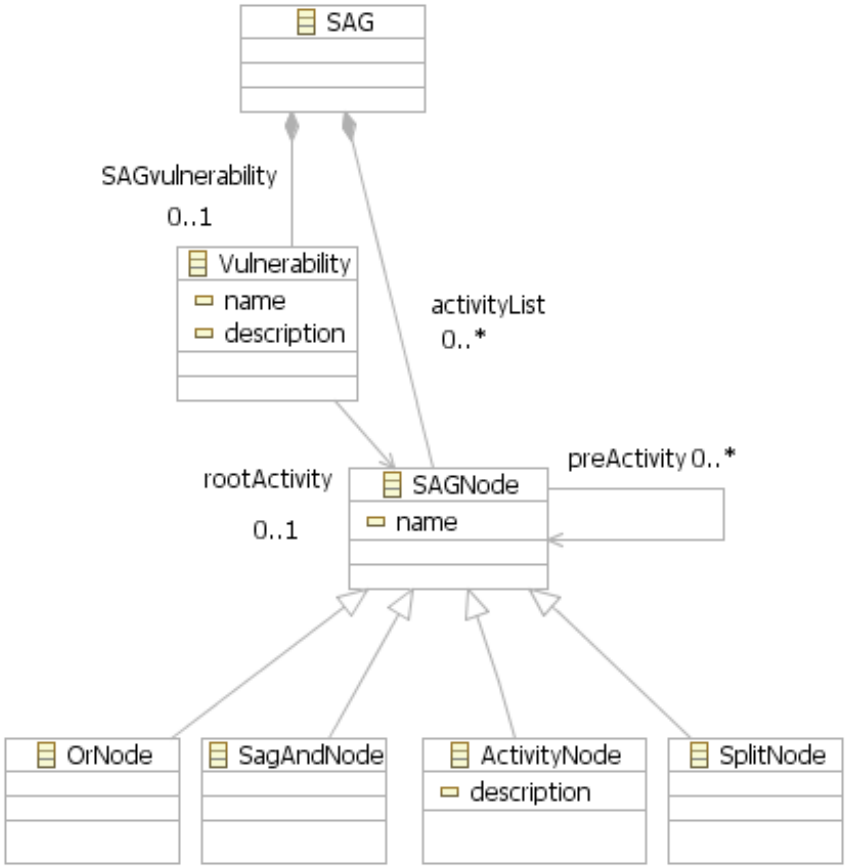


Figure 9.4: **SAG Data Definition Model** - SAG defined in the domain model

```
    if Y = nodeA:
      return false;
    add all predecessors of Y to X
  return true;
}
```

9.5 Implementation

This section describes the modification of existing SeaMonster plug-ins and the implementation of a new plug-in for security activity graphs. The details in this section require knowledge of GMF which was introduced in Chapter 8. GMF generates Java code from a set of GMF models. This set contains these files that need to be created or modified:

- seaMonster.ecore. This is the EMF data definition model.
- *.gmfgraph. The GMF graphical definition model defines the graphical elements of the editor.
- *.gmftool. The GMF tooling definition model defines the available tools.
- *.gmfmap. The GMF mapping model relates the data definition, graphical elements and tools.
- *.gmfgen. The GMF generator model defines the code generation process. Example variables are package names and structures, copyright information and file extensions.

9.5.1 Security Activity Graph Plug-in

The SeaMonster data definition model (seaMonster.ecore) is updated to include security activity graphs as shown in Figure 9.3 and 9.4 in Section 9.4. It is modified using the UML based built-in Eclipse EMF editor. The packages seaMonster and seaMonster.edit were regenerated to reflect the changes in the data definition model. The security model in SeaMonster is a diagram that connects different security diagrams in

a high-level perspective. This diagram is defined in the plug-in seaMonster.secmod and is modified to include security activity graphs. This required changes to the graphical, tooling and mapping model. Security activity graphs are implemented in the plug-in *seaMonster.sag*. The following paragraphs describe the steps involved in the implementation:

Graphical definition

The graphical definition which defines the visual appearance and properties of the diagram is shown in Figure 9.5. Modelling components that are to appear in the diagram are defined as *Node*s. Connections between nodes are defined as *Connection*s. *Figure Descriptor*s are created to define the visual representation of nodes and connections. The *Diagram Label*s define node labels, allow modelling components to have short names or descriptions. The nodes in the graphical definition follow the SAG syntax described in Section 3.2.5.

Tooling definition

The tooling definition which defines the tool palette is shown in Figure 9.6. The model defines six tools, one for the creation of each SAG element. The tool icons are set as bundled images. These images are also used on diagram elements to follow the specified modelling syntax.

Mapping definition

The mapping definition is shown in Figure 9.7. This model defines a mapping between the data definition, graphical and tooling models. A *Top Node Reference* is created for any modelling component that can be placed directly on the modelling canvas,e.g. the first element in the diagram or elements not connected to any other elements. This is created for a vulnerability, activity, and-node, or-node and split-node. Each top node is given a *Feature Label Mapping* which maps the name attribute of each class specified in the data definition to the node. The *Link Mapping*s relate the connections from the graphical definition to tools from the tooling definition and target nodes. One of the link mappings is given a *Link Constraint* which is set to be implemented in the generated Java code. This constraint contains the cycle detection algorithm. The code generation framework creates a code skeleton which has to be implemented by the developer.

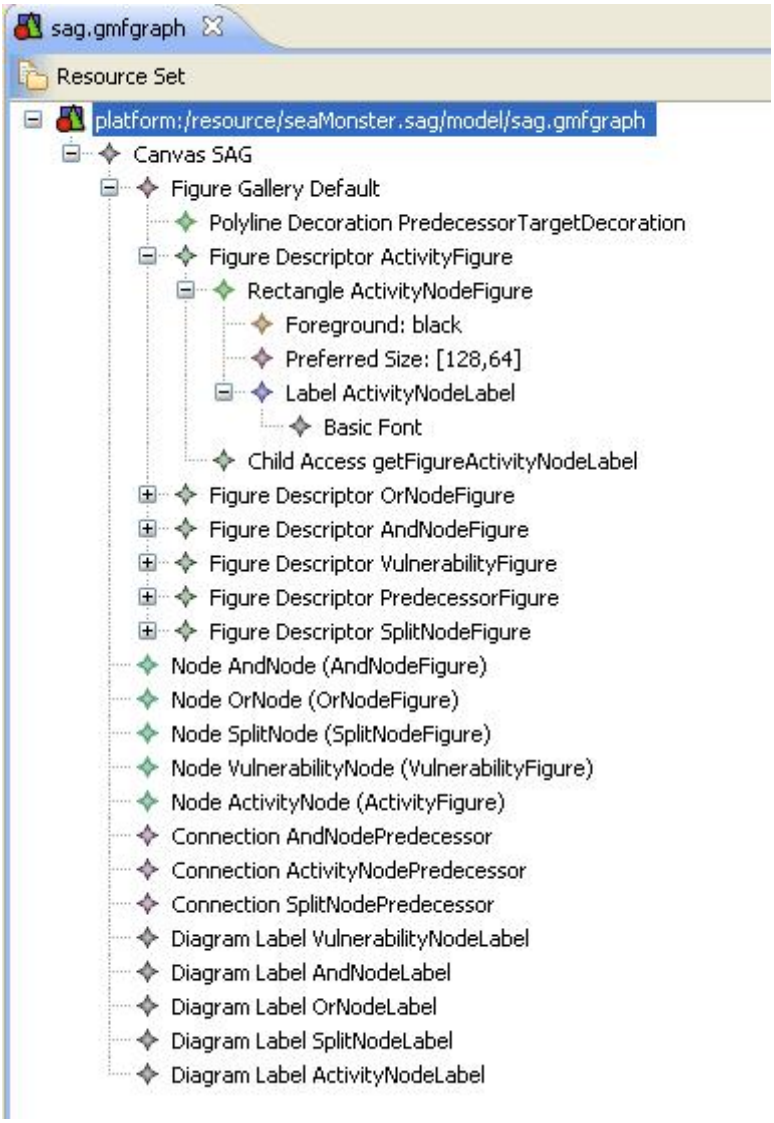


Figure 9.5: **SAG Graphical Definition Model** -

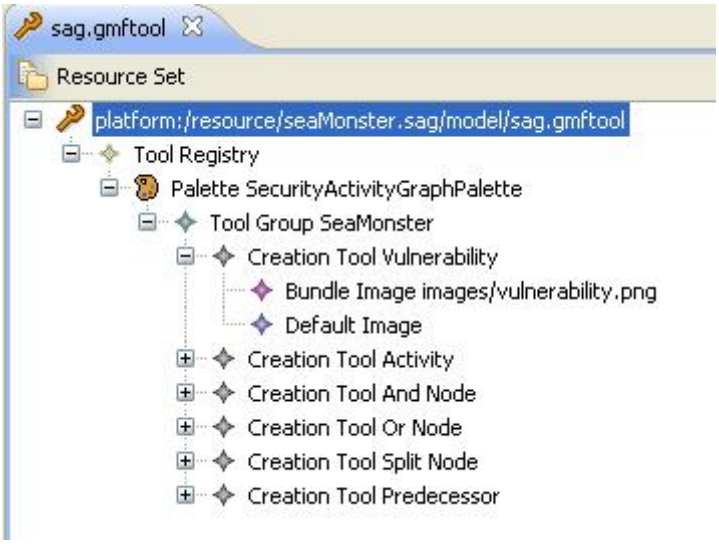


Figure 9.6: **SAG Tooling Definition Model -**

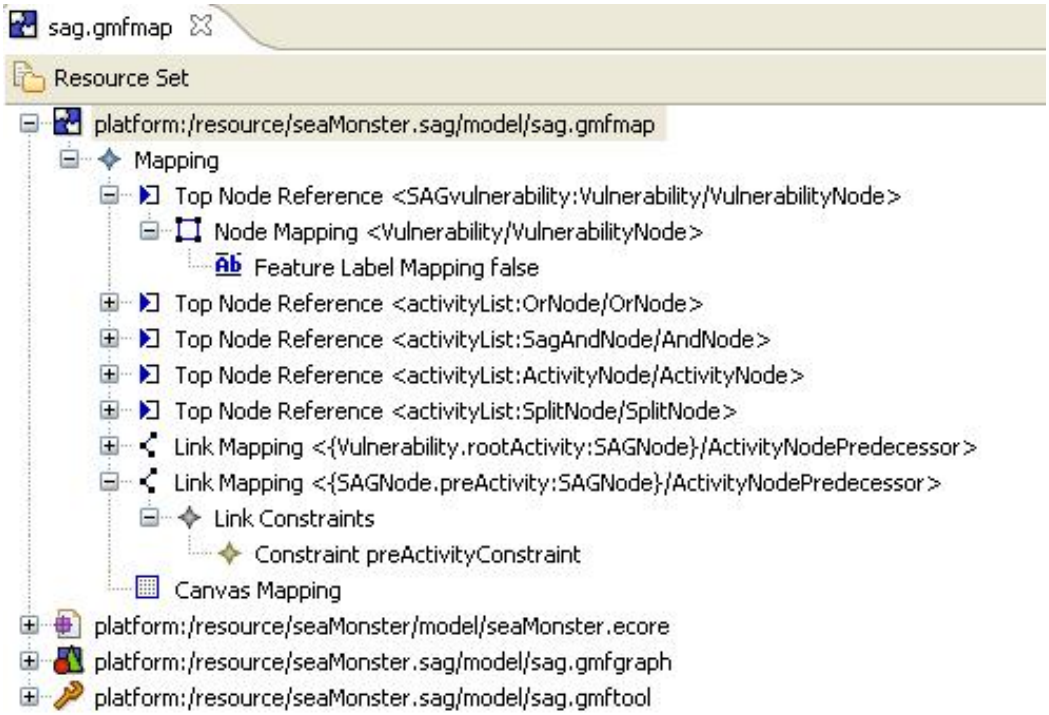


Figure 9.7: **SAG Mapping Definition Model -**

Modifications to generated code

The cycle prevention algorithm specified in Section 9.4 was implemented in SeaMonsterBaseItemSemanticEditPolicy.java found in the sag.diagram.edit.policies package. The method *preActivityConstraint* implements the link constraint of the LinkMapping(SAGNode/ActivityNodePredecessor) from the mapping definition model. This modification is made to adhere to the SAG syntax and to enable further work with model computations. The constructors of the classes SAGAndNodeEditPart.java, OrNodeEditPart.java and SplitNodeEditPart.java in the sag.diagram.edit.parts package were modified to use a CenterStackLayout. This change puts the diagram node labels to the centre of the node.

The product configuration was modified to add plug-in dependency of the new plug-in.

9.6 Testing

The testing of SeaMonster with a security activity graph plug-in is performed as an internal quality assurance. The goal is to identify possible discrepancies between requirements and implementation, to discover and being able to correct errors and flaws. The testing is meant to document that the application is suited for a case study and thereby assuring the value in made contributions. The tests are simple functional tests that can be mapped to the requirements in Section 9.2. Detected errors are corrected unless otherwise is stated. The test suite contains 13 tests and is described in Tables 9.2 through 9.14.

Table 9.2: Test 1

Test ID	1	
Test name	Canvas	
Requirements	1, 4	
Test description	**Expected Result**	**Result**
Build an arbitrary diagram. Extend the diagram with a new component that stretches outside the current canvas. Repeat the last step placing a component above, under, left and right of the initial diagram.	The canvas expands in the direction specified by the user.	Approved

Build an arbitrary diagram with 20 components and 20 connections.	The components are placed in the diagram without any interruptions or degrade of performance.	Approved

Table 9.3: Test 2

Test ID	2	
Test name	Palette	
Requirements	2, 3	
Test description	**Expected Result**	**Result**
Create a security activity graph diagram. Use the tool palette to create the following modelling components: vulnerability, and-node, or-node, split-node and activity. Draw a connection between the vulnerability and the activity.	The tool palette is configured for security activity graphs with tools for the components described in the test description. The tools can be used to create the mentioned components and connections.	Approved
Create a vulnerability cause graph diagram. Select the ExitNode creation tool from the palette and create the node.	The tool palette switches to vulnerability cause graph mode. An exit node is created.	Approved

Table 9.4: Test 3

Test ID	3	
Test name	Notes	
Requirements	8	
Test description	**Expected Result**	**Result**
Create a security activity graph diagram with a vulnerability. Create a note which is not connected to the vulnerability and give it the text 'this is a test'. Delete the vulnerability.	A note is created with the specified text. The note remains when the vulnerability is deleted.	Approved

Table 9.5: Test 4

Test ID	4	
Test name	Highlighting	
Requirements	9	
Test description	**Expected Result**	**Result**
Create a security activity graph diagram with a vulnerability connected to an or-node which is then connected to two activities. Give one activity and the or-node a background colour different from the vulnerability.	The specified diagram is selected and specified nodes are given a background colour.	Approved

Table 9.6: Test 5

Test ID	5	
Test name	GUI	
Requirements	5, 6, 7	
Test description	**Expected Result**	**Result**
Create an arbitrary diagram. Zoom in so one modelling component covers the canvas. Zoom out until the complete diagram fits the canvas.	Zooming functionality should behave as specified in the test description.	Approved
Create a vulnerability cause graph, security activity graph and an attack tree.	The different diagrams should be created without closing the other diagram(s).	Approved
Continue from the above step. Switch between the diagrams using the diagram selection pane.	SeaMonster shall switch between the diagrams and change the user interface accordingly.	Approved

Table 9.7: Test 6

Test ID	6
Test name	Standard editor functionality

Requirements	10, 11, 12, 13	
Test description	**Expected Result**	**Result**
Create an arbitrary diagram with 4 components.		
Create a connection between two components. Undo the operation from the Edit menu.	The connection is undone.	Approved
Create a connection between two components. Select the connection and cut it using the Edit menu.	The connection is cut from the diagram.	Approved
Select a component and copy it using the Edit menu. Paste the copy into the diagram using the Edit menu	The component is copied and pasted into the diagram.	Approved

Table 9.8: Test 7

Test ID	7	
Test name	Vulnerability Cause Graphs	
Requirements	14, 18	
Test description	**Expected Result**	**Result**
Create a vulnerability cause graph diagram.	All components of a vulnerability cause graph is available in the palette.	Approved
Create the diagram shown in Figure 9.8.	The diagram should be created without errors and with matching notation.	Approved

Table 9.9: Test 8

Test ID	8	
Test name	Security Activity Graphs	
Requirements	15, 19	
Test description	**Expected Result**	**Result**
Create a security activity graph diagram.	All components of a security activity graph is available in the palette.	Approved

Create the diagram shown in Figure 9.9.	The diagram should be created without errors and with matching notation.	Approved

Table 9.10: Test 9

Test ID	9	
Test name	Decomposition	
Requirements	17	
Test description	**Expected Result**	**Result**
Create an arbitrary diagram with at least one modelling component. Select a component, click it with the right mouse button and decompose it.	The component should be decomposed by initiating a dialogue where a new diagram can be instantiated. The new diagram is connected to the originial component and can be accessed by double-clicking.	Failed

Table 9.11: Test 10

Test ID	10	
Test name	Properties	
Requirements	16	
Test description	**Expected Result**	**Result**
Create a security activity graph diagram with one component of each type from the palette. Edit properties and give them a name and description	The properties are be stored with the component.	Approved

Table 9.12: Test 11

Test ID	11	
Test name	Persistence	
Requirements	16	
Test description	**Expected Result**	**Result**
Create a security activity graph diagram. Save the diagram to file, close and restart SeaMonster. Open the saved model.	The diagram should be saved to the specified path. The diagram should look the same after SeaMonster has restarted.	Approved

Table 9.13: Test 12

Test ID	12	
Test name	Cycle prevention	
Requirements	N/A	
Test description	**Expected Result**	**Result**
Create a security activity graph diagram. Create three and-nodes and connect them in a circular manner.	The third connection completing the cycle should not be allowed.	Approved

Table 9.14: Test 13

Test ID	13	
Test name	SAG generation	
Requirements	22	
Test description	**Expected Result**	**Result**
Create a vulnerability cause graph with a minimum of 3 causes. Highlight the complete graph and generate a security activity graph.	The generated SAG should be composed following the algorithm in (6).	Failed

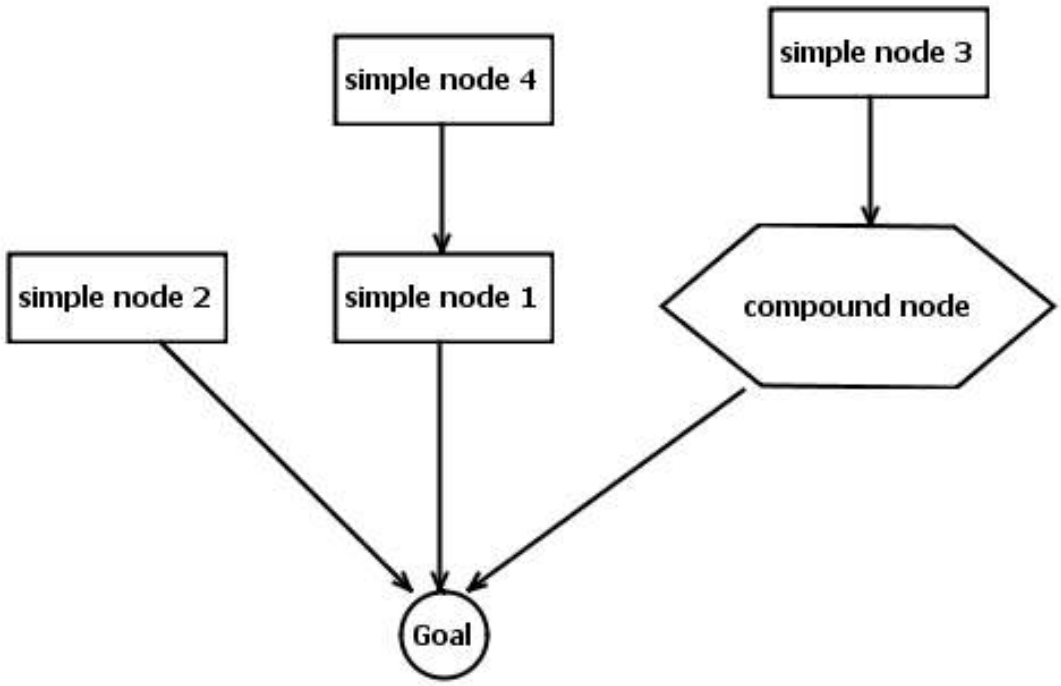


Figure 9.8: **Vulnerability Cause Graph Test -**

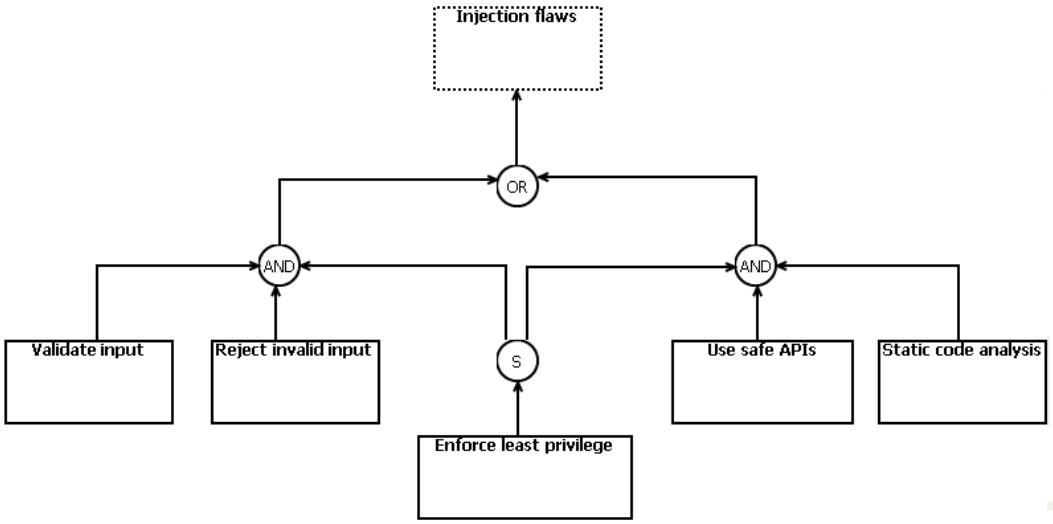


Figure 9.9: **Security Activity Graph Test -**

9.6.1 Test Summary

All tests were approved except Test 9 (Decomposition) and Test 13 (SAG Generation). Model decomposition and SAG generation is not implemented due to time constraints, therefore requirement 17 and 22 will not be fulfilled during this thesis. This is not viewed as a severe shortcoming of the implementation because this functionality is not essential to the research objectives and the countermeasure modelling approach. Model decomposition was stated as a requirement because it was intended to improve usability and readability of security model diagrams. Automatic generation of security activity graphs is likely a useful tool, especially for users not trained in security or the use of SAGs. The state of the implemented security modelling tool is satisfactory in regards to performing a case study.

Chapter 10

Case Study

This chapter contains a case study of the suggested security improvement method and tools. The case study is performed to document and evaluate the contributions of this thesis. A medical patient journal system is described as the case software system. The system's specifications are analysed through security modelling and vulnerability countermeasures are applied in the system design.

10.1 Approach and Case Description

Research objective 3 is a case study designed to document the suggested security improvement approach and tools. The value of the case study is twofold; as a proof of concept it shows how the contributions are applied and possible benefits. Second, the case study enables an evaluation of the contributions through experiences with applying them. The case study will demonstrate how security improvement tools can be applied in early stages of the software development lifecycle. The approach is based on an analysis of system specifications and designs, followed by mitigation of software vulnerabilities in the design phase.

The case study should deal with software and security risks that are relevant to the current trends in software and security. Increasing connectivity has a large influence on security issues. Systems connected to the Internet are vulnerable to software-based attacks, because the growing connectivity of computers increase the number of attack

vectors and decreases the level of sophistication that is needed to perform an attack (46). The Web is a special case of the client-server model, which introduces the concept of a user not trusted by the system. Deploying an application on a Web server introduces security risks which should be considered during design. The software described in the case study will therefore be specified as a Web application. Security becomes increasingly important as software handles valuable or confidential assets. The software described in the case study should have strict requirements to confidentiality and integrity of information.

Following the above guidelines, a system for administration of medical patient journals is chosen for the case study. The system should be available through a Web interface. Figure 10.1 shows a UML use case diagram of the system. The diagram describes interactions between users and the system. The system has several types of users which should have access to different functions and information. Medical staff is able to write diagnoses and comments to a patient profile. Patients use the system to view diagnoses and personal information stored in the system. The system administrator has access to all functions.

System requirements are made to define assets and security objectives. These requirements are created by the author solely for this case study and are not complete for a real medical information system. The system has the following requirements:

1. There are five actors in the system: patients, system administrator, medical staff, doctors and nurses. Medical staff includes doctors and nurses.

2. Each patient has a medical journal which includes the following: name, age, address, phone number, social security number, diagnoses and medical comments from doctors and nurses.

3. A patient should be able to view the personal information specified in Requirement 2 and the diagnosis linked to her.

4. Medical staff should be able to view any patient journal.

5. Doctors should be able to post comments to each patient's journal. These are only accessible by medical staff.

Figure 10.1: **Case Study Use Cases** - Connections from the *System Administrator* to the use cases are left out to increase readability

6. Nurses should be able to post comments to each patient's journal. These are only accessible by medical staff.

7. A system administrator should be able to view the names of all users of the system.

8. A system administrator should have access to all system functions.

9. The system must perform access control to ensure that the specified information are accessible only to those specified in these requirements.

10. The integrity of stored information must be ensured.

11. The system must be accessible through the Internet and a Web browser.

10.2 Analysis and Design

This section documents the case study execution and describes the produced artefacts. The specified journal system is analysed using the approach in Chapter 7 and designed using security pattern design templates.

Countermeasure modelling can be performed using only system specifications as input, but is perhaps more valuable if design is included because this enables a deeper and more specific analysis of possible threats and vulnerabilities. A high-level design and system deployment is created before the countermeasure modelling is performed. Figure 10.2 shows a deployment diagram of the journal system. The system is deployed on a Web server and accessed using a Web browser. The user interfaces are built using Java servlets (56) which can generate a dynamic response with Web-content based on a request. The servlets get their data from a data package which is responsible for communicating with the underlying database.

Figure 10.3 shows a high-level design with the main modules of the journal system. The *Client* package represents the client-side (Generated HTML pages and a Web browser). The *Container* class is a Web container which is a specialised Web server that supports servlet execution. All requests to the system are handled by this component.. The

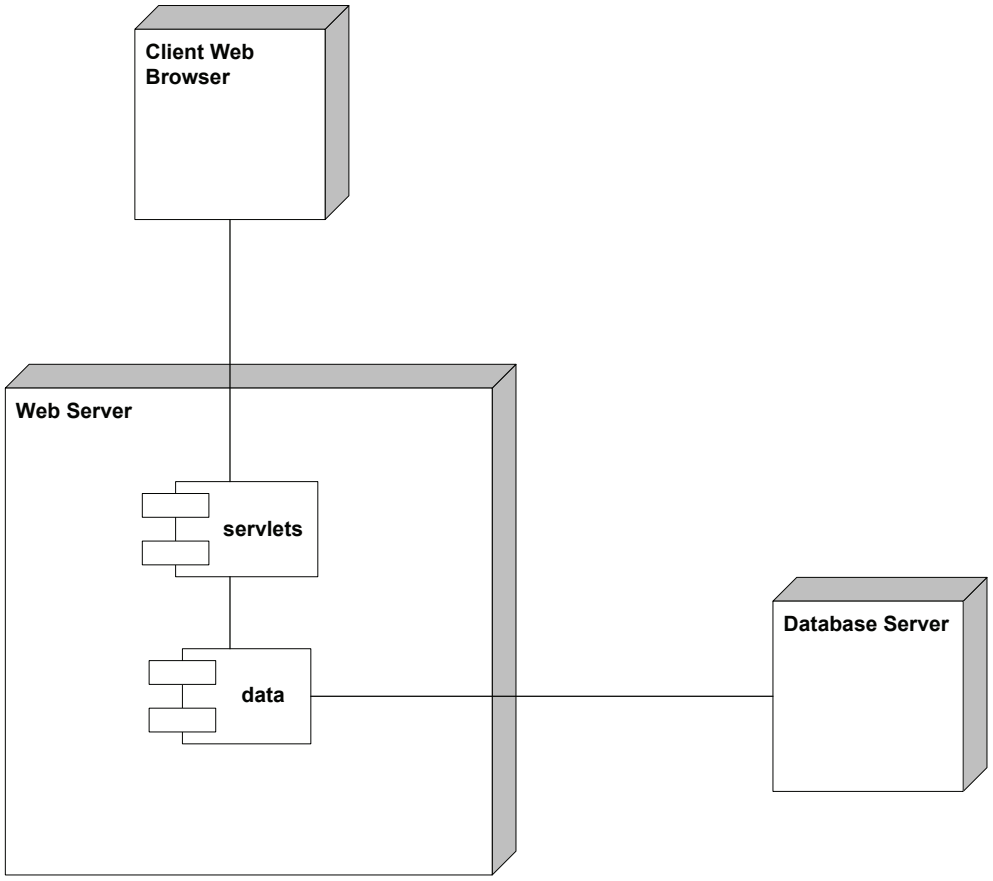


Figure 10.2: **Case Deployment** - The journal system is deployed on a Web server

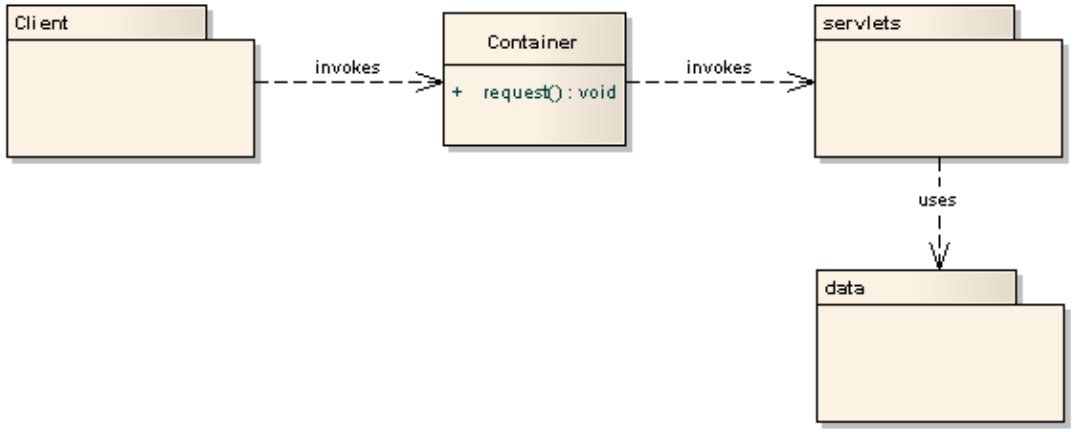


Figure 10.3: **High-Level Design I** - Module structure

servlets package holds the business logic of the application and communicates with the *data* package which provides persistence.

Figure 10.4 shows a design of the data package. This model does not support access rights, and would leave access control to the implementations of higher application layers.

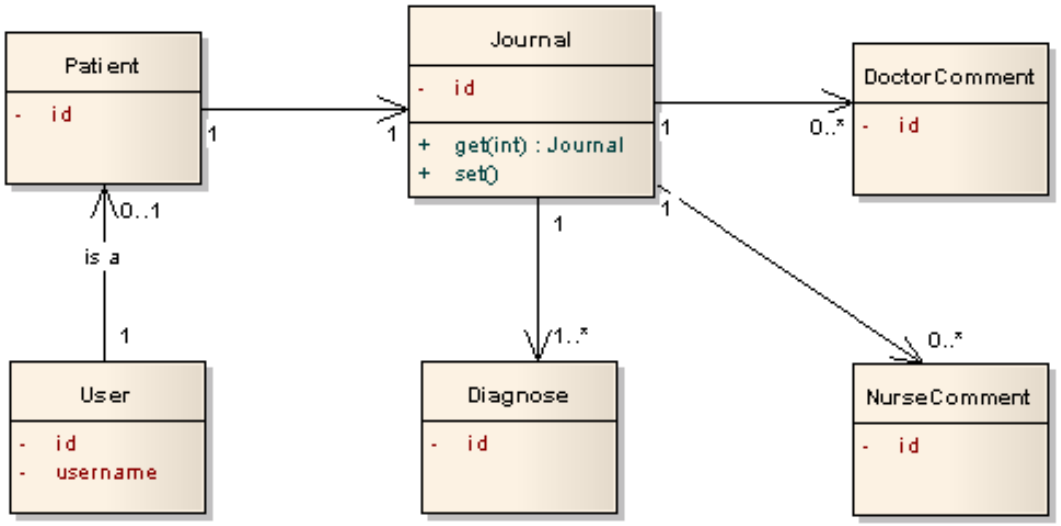


Figure 10.4: **High-Level Design II** - Data layer

The medical journal system is specified to deal with a large number of users and informa-

tion resources. So far the design is made without much effort in security. Implementing and deploying this design is likely to produce a very insecure software system.

10.2.1 Countermeasure Modelling

Proper countermeasure modelling requires security knowledge and experiences with security risks, attack patterns and common vulnerabilities and should therefore be performed by security experts and professionals. The system specifications and design are analysed, potential vulnerabilities and countermeasures are modelled using the extended SeaMonster. The modelling performed in this case study includes only technical vulnerabilities and causes because the scope of countermeasures is limited to those applicable during software design. Human factors are a big part of security risks (70) and should be considered in a real-world scenario. Figure 10.5 shows a screenshot from SeaMonster where a security activity graph diagram is created.

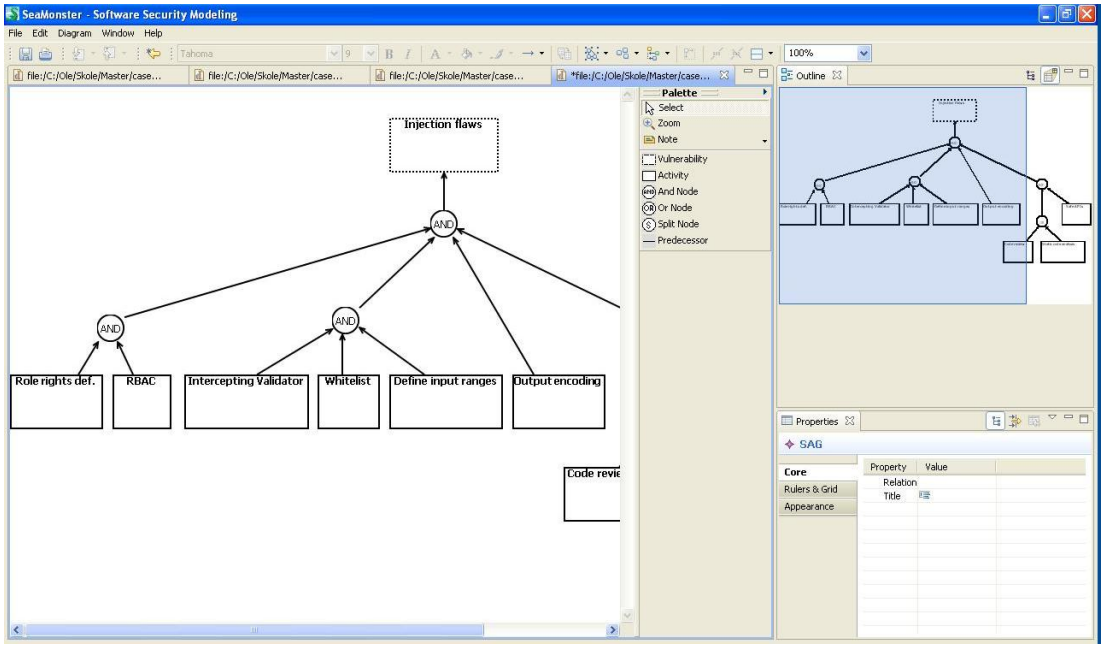


Figure 10.5: **Extended SeaMonster Screenshot** - Security Activity Graph Diagram

Medical records are highly sensitive data with strict privacy requirements. The disclosure of this information is a severe threat and should be counter measured during design. The system specifications are analysed using knowledge on information disclo-

sure issues and a vulnerability cause graph (VCG) is created as shown in Figure 10.6.

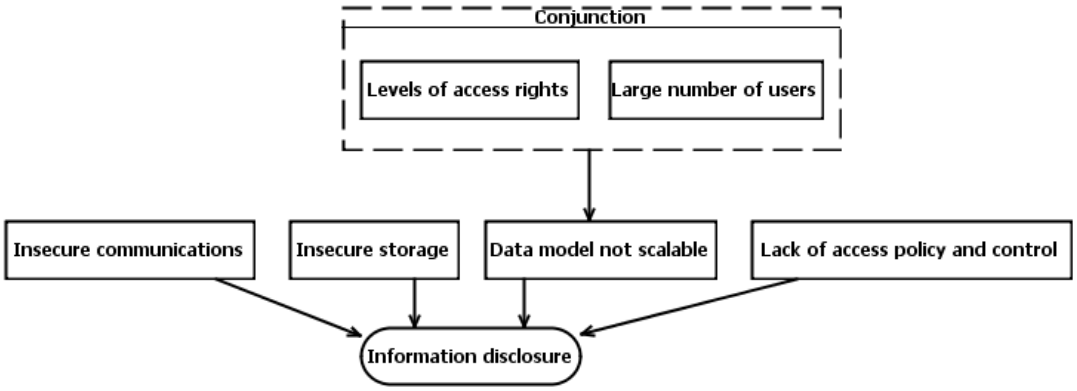


Figure 10.6: **Case VCG I** - Vulnerability: Information disclosure

These causes of possible information disclosure issues in the journal system are identified:

- Insecure communications. Requirements state that the information handled in the system is highly sensitive and its privacy must be ensured. Failure to encrypt communications means that an attacker who can sniff traffic from the network will be able to access the communication payloads.
- Insecure storage. The healthcare records and personal information of system users must be stored in a secure manner. The high-level designs do not specify any use of cryptographic storage. Preventing cryptographic flaws takes careful planning and must be included in design.
- Data model is not scalable. The system requirements state that the system must handle a large number of users mapped to different acting roles with levels of access rights. This is modeled as a conjunction of two causes in Figure 10.6. The data layer design defines access rights per user, which is highly redundant and error prone. Managing the rights per user account may not be feasible.
- Lack of access policy and control. A system of this size should have a centralised mechanism which defines access policies and performs access control.

A countermeasure model is now created for the information disclosure vulnerability. To create a security activity graph (SAG) from the VCG, recall from Section 3.2.5 that the first step is to enumerate mitigation techniques for each cause. The techniques are expressed as logic combinations of concrete actions:

Insecure communications:
Vulnerability scanning tools can verify that SSL is used on the front end. Scanning tools may not have access to check backend connections between infrastructure elements such as between Web server and database server. The use of code review is efficient to verify encryption on backend communications.
Graph fragment: (Vulnerability scanning tools AND Use SSL) AND (Encrypt communications between infrastructure elements AND Code review)

Insecure storage:
Sensitive data must be persisted using cryptographic storage. Failing to encrypt sensitive data or using poorly designed cryptography can lead to information disclosure. Verifying use of cryptographic storage should be done by a code review because scanning tools can only detect use of cryptographic APIs, not verify that they are being used properly.
Graph fragment: Code review AND Safe use of cryptography for storage of sensitive information

Data model is not scalable:
Assigning rights to roles based on job functions is a way of supporting the least privilege principle (67). Creating the role abstraction removes the access rights definition redundancy in the data model and ensures that users have the specified rights only.
Graph fragment: Role rights definition

Lack of access policy and control:
Role-Based Access Control (RBAC) (71) is a security pattern that defines how access rights are mapped to users via roles in an environment with a large number of users, information sources and resources. The roles should be reflected in the data layer design to enable a centralised access control mechanism. Any use of information resources through the system should be verified by performing access control according to the

roles.

Graph fragment: RBAC AND Centralised access control

Step 2 of the SAG generation process is to create graph fragments for each set of mitigation techniques. This is straightforward when the sets are expressed in predicate logic as done above. Finally, the fragments are combined to create the final graph. This can be seen in Figure 10.7. Some post-generation simplification could result in a simpler graph, e.g. reducing the number of AND-nodes, but the original graph preserves the relationships between the activities. An example is the use of code review to check the use of encryption of communications between infrastructure components. These activities are highly connected.

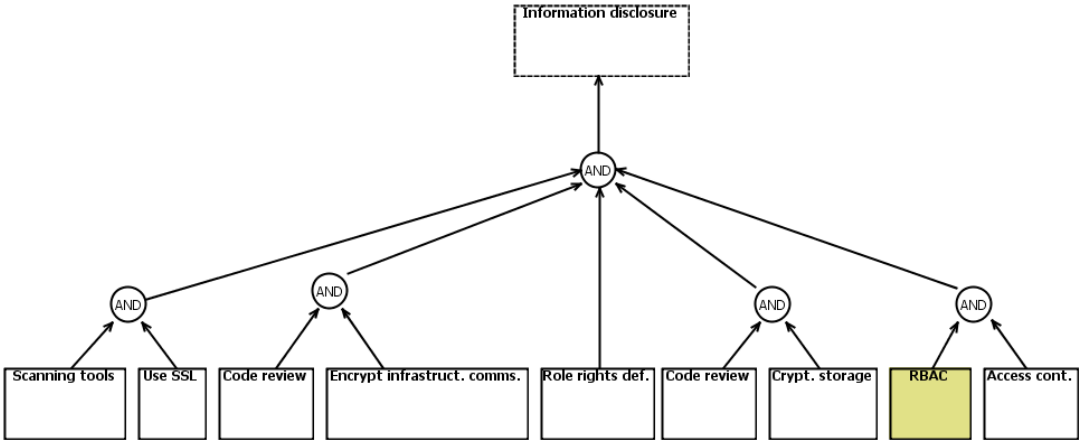


Figure 10.7: **Case SAG I** - Vulnerability: Information disclosure

The assets of the journal system are healthcare records and personally identifiably information. The integrity and confidentiality of this data must be ensured, as specified by requirements. Tampering or disclosure of this information is a severe threat and should be counter measured during design. Injection flaws enable attackers to create, read, update, or delete data. The system specifications are analysed using knowledge on injection flaws and a VCG is created as shown in Figure 10.8.

These causes of possible injection flaws in the journal system are identified:

- Unrestricted privileges. If user privileges are unrestricted, performing an injection attack will enable attackers to perform a wide range of operations they are not

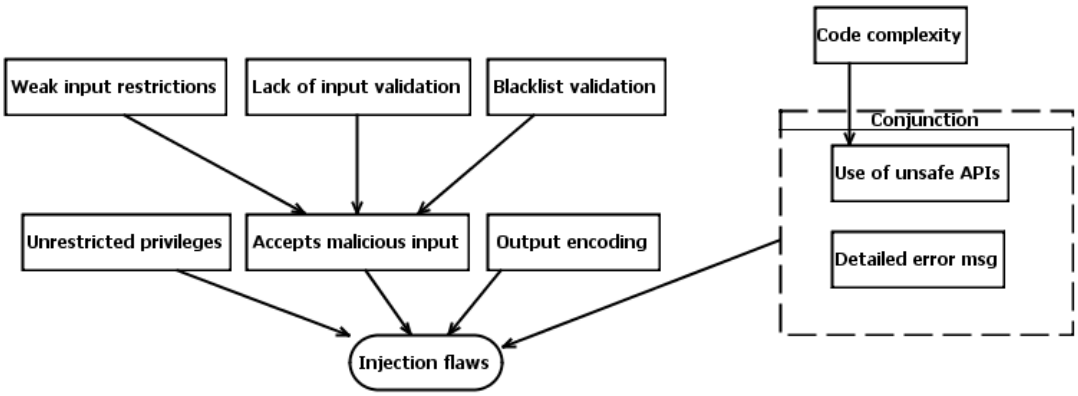


Figure 10.8: **Case VCG II** - Vulnerability: Injection flaws

meant to. Examples are file access, invoking sub-systems or system calls.

- Accepts malicious input. If the system accepts and trusts user-supplied data as safe, attackers may inject commands to compromise the system and integrity of stored information. Mitigating this is extremely important because there are many entry points to the software through user-supplied input. This cause has three underlying causes. Weak input restrictions enable attackers to supply input that is out of valid range or of different data types. Lack of input validation means the software accepts user-supplied data without verifying that it is safe. Blacklist validation is a flawed validation technique that specifies and removes a set of invalid character strings. It is not sufficient because it is difficult to cover all possible malicious inputs.

- Output encoding. Unspecified or lacking output encoding enables command injections to be run in browsers.

- Use of unsafe APIs and Detailed error messages. The use of unsafe Application Programming Interfaces (API) in combination with giving too detailed error messages allows attackers to exploit known weaknesses in the system. Error messages that reveal details of the system structure and technologies help attackers design their attacks. Examples of use of unsafe APIs are unsafe character escaping functions of database APIs.

A countermeasure model is now created for the injection flaw vulnerability. Mitigation techniques for each cause are enumerated:

Unrestricted privileges:
Two security patterns are used to restrict privileges; rights are defined using roles and RBAC is used to implement the access control. This should keep attackers from being able to perform system functions or access resources.
Graph fragment: Role rights definition AND RBAC

Accepts malicious input:
The security design pattern Intercepting Validator (78) is used to validate input. This pattern defines a centralised and simple yet flexible solution to validate all data passed in from the client. System designers should carefully define valid input ranges; data types, bounds and formats, to be able to reject and accept user-supplied input. Whitelist validation is a validation technique where input is filtered according to a set of rules that define /textitvalid input. The alternative to whitelist validation is blacklist.
Graph fragment: Intercepting Validator AND Define input ranges AND Whitelist validation

Output encoding:
Specifying and ensuring strong output encoding should be done to prevent a successful script injection from running in the client browser. This activity is named output encoding for brevity.
Graph fragment: Output encoding

Use of unsafe APIs and Detailed error messages:
Code review or static code analysis should be performed to detect use of unsafe APIs. Any instances of unsafe code should be replaced with safe APIs.
Graph fragment: (Code review OR Static code analysis) AND Use of safe APIs

Step 2 of the SAG generation process is to create graph fragments for each set of mitigation techniques. This is straightforward when the sets are expressed in predicate logic as done above. The resulting graph can be seen in Figure 10.9. Some post-generation simplification could result in a simpler graph, e.g. reducing the number of AND-nodes, but the original graph preserves the relationships between the activities better.

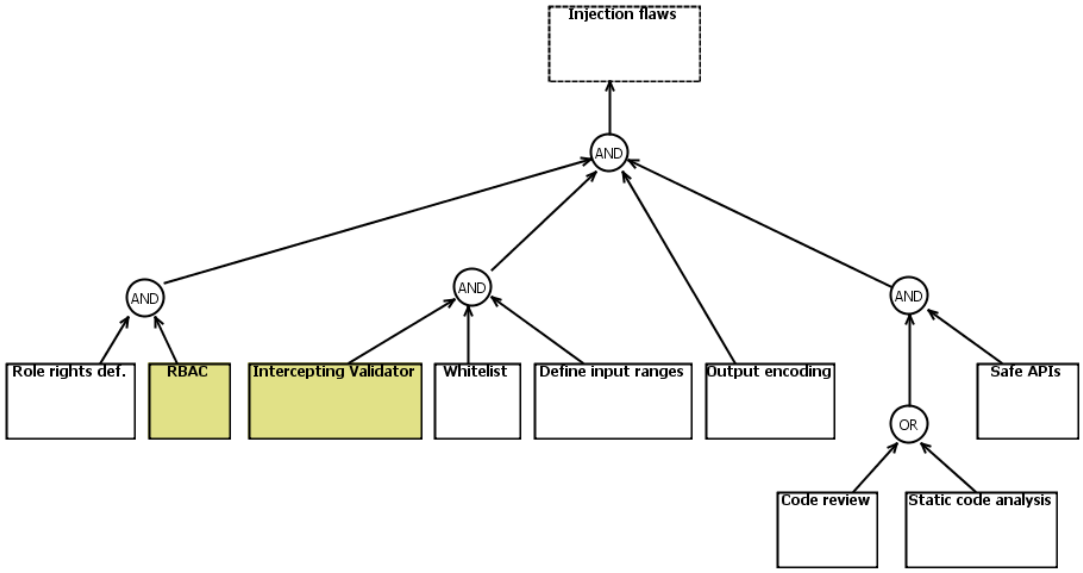


Figure 10.9: **Case SAG II** - Vulnerability: Injection flaws

The two security activity graphs point to a set of activities which in a real-world application should be implemented to mitigate security risk. Two activities will be followed in this case study: RBAC from the Information Disclosure vulnerability and Intercepting Validator from the Injection flaws vulnerability. The described patient journal system is likely to have more possible vulnerabilities; to limit scope these will not be explored. The next step of the development process is to incorporate the two security improvement activities in the system design.

10.2.2 Design with Countermeasures

The software system has been analysed through vulnerability cause graphs and security activity graphs in the countermeasure modelling phase. The security expert has identified countermeasures that should be introduced already in the software design phase. This work limits its scope of countermeasures to security design patterns, so the case study will describe the use of two patterns as shown in Figure 10.7 and 10.9. The activities (patterns) *Intercepting Validator* and *RBAC* are highlighted with colour. A software developer will now use the security activity graphs to guide his selection of countermeasures. In a real development case the selection could be performed by

the security expert highlighting a preferred path of the security activity graphs. Only two activities are highlighted here because of limited scope. The countermeasures are implemented in software design by the developer through security pattern templates in the software design tool Enterprise Architect (82). These are the templates developed in previous work (13). The applied security patterns in the case study are Intercepting Validator and Role-Based Access Control (RBAC).

Intercepting Validator

This pattern describes how to scan or validate data which is passed in from the client. Client requests may contain malicious content, especially in open environments like the Web where a client can not always be trusted. The pattern is described in (78), the following representation is restructured to fit the pattern representation adopted in this work:

Name
Intercepting Validator

Example
Online bookstores are very interactive and requires a lot of input from clients. This results in requests with parameters possibly containing malicious content. The sources and types of input are diverse, and there is a need to ensure that all requests are validated properly before it is processed in the business logic.

Context
Any environment where the system accepts data passed in from a client who can not be guaranteed safe or trustworthy.

Problem
You need a simple and flexible mechanism to scan and validate data passed in from the client for malicious code or malformed content. The data could be form-based, queries or even XML content. Several well-known attack strategies involve compromising the system by sending requests containing invalid data or malicious code. Such attacks include injection of malicious scripts, SQL statements, XML content, and invalid data using a form field

that the attacker knows will be inserted into the application to cause a potential failure or denial of service. The embedded SQL commands can go further, allowing the attacker to wreak havoc in the underlying database. These types of attacks require the application to intercept and scrub the data prior to its use.

The solution to this problem must balance the following forces:

- You want to validate a wide variety of data.
- You want a common mechanism for validating various types of data.
- You want to dynamically add validation logic as necessary to keep your application secure against newly discovered attacks.
- Validation rules must be decoupled from presentation logic.

Solution

Use an Intercepting Validator to cleanse and validate data prior to its use within the application, using dynamically loadable validation logic. The intercepting validator controls a chain of validators which validate the input in turn. The validators should be applied declaratively based on URL, allowing different requests to be mapped to different validator chains. The validation logic in each validator determines whether or not the request should continue or be aborted. Validation must always be performed on the server-side.

Structure

A class diagram for Intercepting Validator is shown in Figure 10.10. The *Client* class represents the client who sends requests to a target within the system. In Web applications, this is usually a Web browser. All requests are handled by *SecureBaseAction*. This class is used by the client to generically enforce request validation. Invoking this component should be secure, i.e. no processing of request parameters should be done. A request is relayed from SecureBaseAction to this class which is responsible for configuring chains of validators and running the requests through these chains. *Validator* is a generic representation of an input validator holding validation logic. An instance of the Intercepting Validator pattern implements one or more

validators, one for each type of input that needs special validation logic. *Target* represents the client requested resource. This can be any resource within the system.

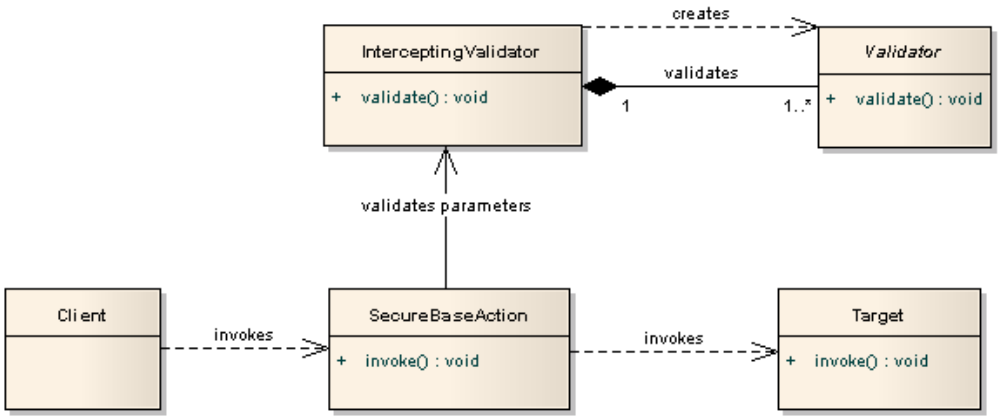


Figure 10.10: **Intercepting Validator Class Diagram** - Pattern structure

Dynamics

A sequence diagram is shown in Figure 10.11. The sequence follows these steps:

1. *Client* makes a request to a particular resource specified as the *Target.*
2. *SecureBaseAcition* uses the InterceptingValidator to validate the data for the target service request.
3. *InterceptingValidator* retrieves the appropriate validators according to the configuration for the target.
4. *InterceptingValidator* invokes a series of validators as configured.
5. Each validator validates the request data. If validation fails, the sequence aborts and the request never reaches the target.
6. Upon successful validation, the *SecureBaseAction* invokes the target resource.

Implementation

Implementations of the pattern will have different instances of the Validator class depending on the input and its targets. Different validators are used

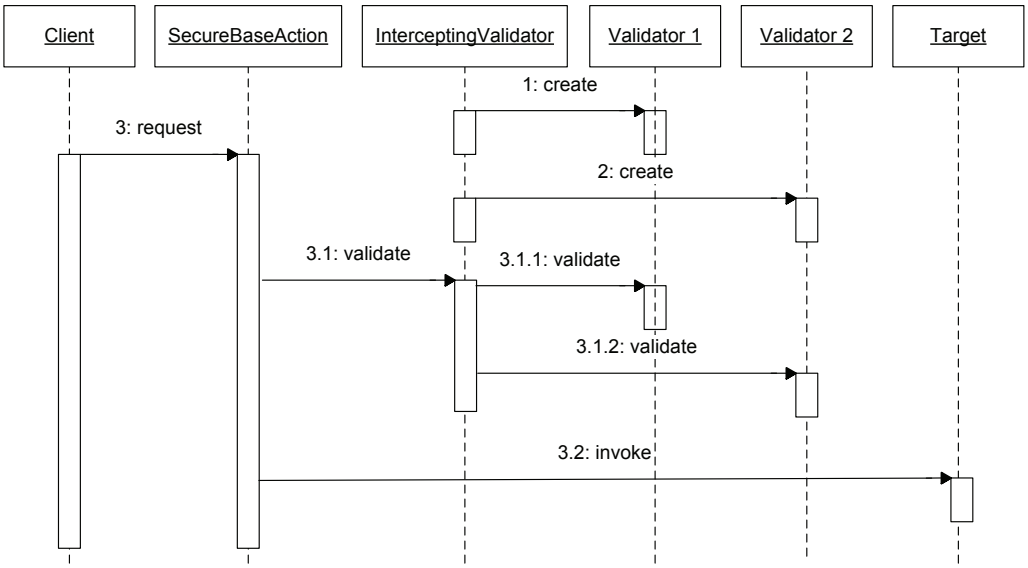


Figure 10.11: **Intercepting Validator Sequence Diagram** - Interaction sequence

to validate different data types and validators may be coupled with different components in the system. For example, data that will become part of an SQL statement should be validated to ensure that embedded SQL commands cannot be entered.

Example Resolved

All requests are mapped to proper validators, and input validation is handled in a centralized and uniform manner.

Consequences

The following benefits may be expected from applying this pattern:

- Malicious code and injection attacks are identified before the business logic processes the request.
- Centralizes security validations, giving more maintainable and reusable code.
- Decouples validations from presentation logic, giving better software manageability and reduces redundancy.

- Simplifies addition of new validators. As new data-based attacks are discovered, new validators can be implemented and installed without requiring redeployment of the application.

Potential disadvantages from applying this pattern:

- Increased processing overhead.

Role-Based Access Control

This pattern describes how to assign rights in a system with a large number of users, information types or resources, and access rights, based on roles, job functions or tasks. The complete pattern description can be found in (71). The following description is limited to the most essential parts:

Name
Role-Based Access Control

Example
A hospital has many patients, doctors, nurses and other personnel. The specific individuals also change frequently. Defining individual access rights has become a time-consuming activity and is error-prone.

Context
Any environment in which we need to control access to computing resources where there is a large number of users, information types, or a large variety of resources.

Problem
For convenient administration of authorization rights we need to have ways to factor out rights. Otherwise, the number of individual rights is just too large, and granting rights to individual users would require storing many authorization rules, and it would be hard for administrators to keep track of the rules. How do we assign rights based on the functions of tasks of people?

The solution to this problem must balance the following forces:

- In most organizations people can be classified according to their functions or tasks.
- Common tasks require similar sets of rights.
- We want to help the organization to define precise access rights for its members according to a need-to-know policy.

Solution

Most organizations have a variety of job functions that require different skills and responsibilities. For security reasons, users should get rights based on their job functions or their assigned tasks. This corresponds to the application of the need-to-know principle, a fundamental security policy (79). Job functions can be interpreted as roles that people play in performing their duties. In particular, Web-based systems have a variety of users: company employees, customers, partners, search engines and so on.

Structure

A class diagram for Role-Based Access Control is shown in Figure 10.12. The *User* and *Role* classes describe the registered users and the predefined roles respectively. Users are assigned to roles which are given rights according to their functions. The association class *Right* defines the access types that a user within a role is authorized to apply to the protection object.

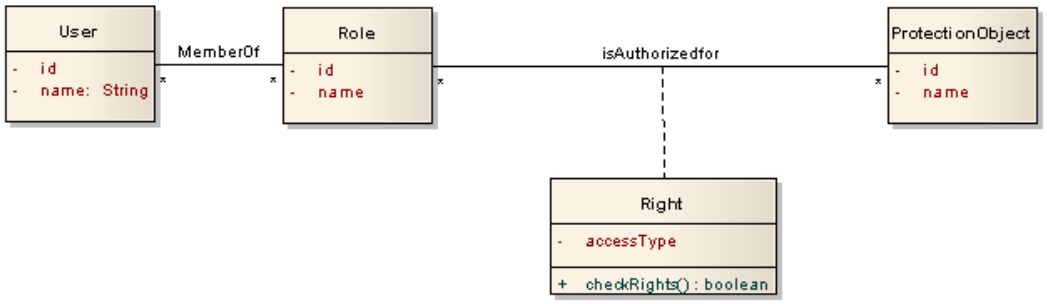


Figure 10.12: **Role-Based Access Control Class Diagram** - Pattern Structure

Implementation

Roles may correspond to job titles, for example manager or secretary. A finer approach is to make them correspond to tasks - for example, a professor

has the roles of thesis advisor, teacher, committee member, researcher, and so on.

Example Resolved

The hospital now assigns rights to the roles of doctors, nurses and so on. The number of authorization rules has decreased dramatically as a result.

Known Uses

Some type of role-based access control is implemented in a variety of commercial systems, including Sun's J2EE, Microsoft Windows 2000 and several database systems.

Consequences

The following benefits may be expected from applying this pattern:

- The complexity of security is reduced because there are much more users than roles.
- Organization policies about job functions can be reflected directly in the definition of roles and the assignment of users to roles.
- It is very simple to accommodate users arriving, leaving or being reassigned. All these actions require only manipulation of the associations between users and roles.
- Groups of users can be used as role members, further reducing the number of authorization rules and role assignments.

Potential disadvantages from applying this pattern:

- The pattern may add conceptual complexity with the new concepts of roles and assignments.

A pattern presents a generic solution and has to be adapted through an instantiation process. This means mapping and integrating design elements from the pattern onto elements of the application design. Figure 10.13 shows the pattern instantiation dialogue in Enterprise Architect.

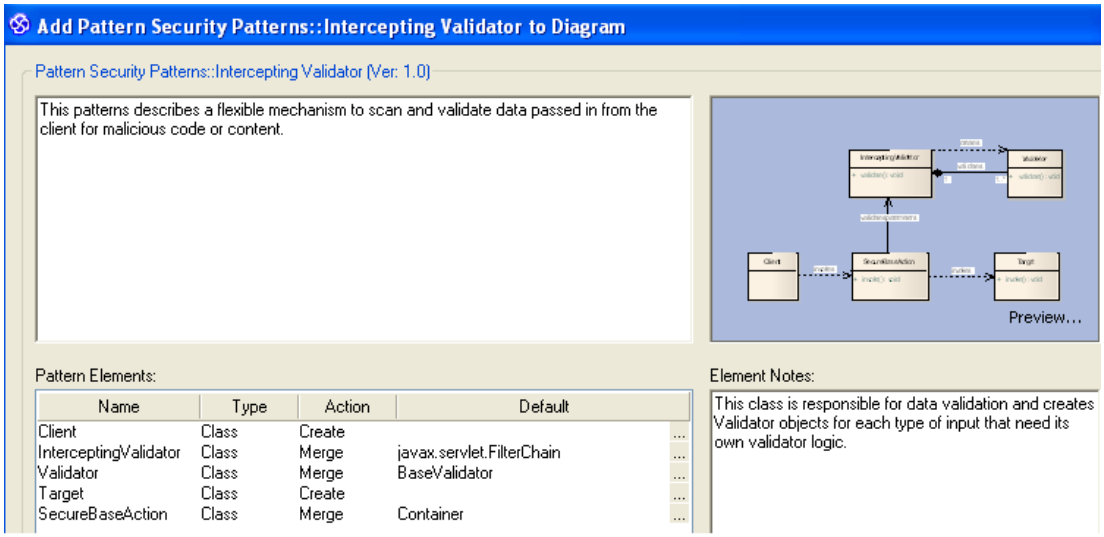


Figure 10.13: **Pattern Instantiation in Enterprise Architect** - Instantiation dialogue

The patterns are instantiated and merged with the existing design modules described in Section 10.2. Figure 10.14 shows the high-level design from Figure 10.3 with the Intercepting Validator pattern. Requests from the client-side were passed directly to the servlets through a Web container in the original design. These requests contain parameters which impose severe security risks. The parameters may include malicious content such as program code, and need to be validated before passed to the servlets. Any request from the client is now intercepted by the *FilterChain* class, and all parameters are validated by one or two validators based on the request address. *ParamValidator* is a general validator for all request parameters. *SQLValidator* validates requests that will be used in components that connect to an SQL database.

Figure 10.15 shows the high-level data layer design from Figure 10.4 with the RBAC pattern. Users are now members of roles which give access to data. The *Journal, Diagnose, DoctorComment and NurseComment* classes are instances of ProtectionObject from the pattern structure, which means access to these resources are controlled by the pattern. The *Role* class implements the Role and Right classes which define the access a user within a role is allowed to apply to a protected object. User access rights to system resources are now defined in a uniform manner, improving the maintainability of rights and asserting that resources are protected by the given security policy.

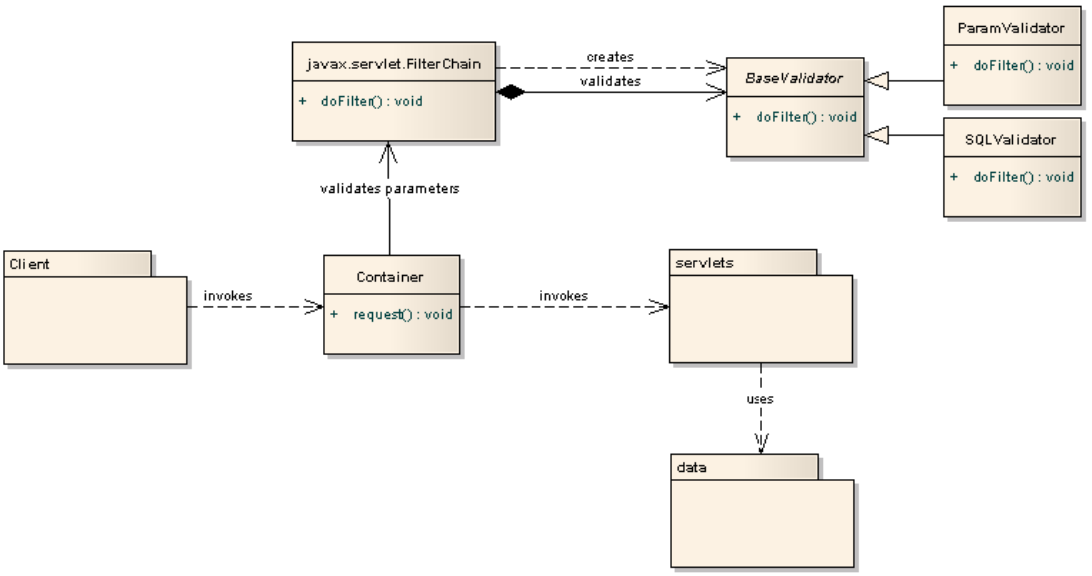


Figure 10.14: **High-Level Design with Security Pattern** - Intercepting Validator is instantiated

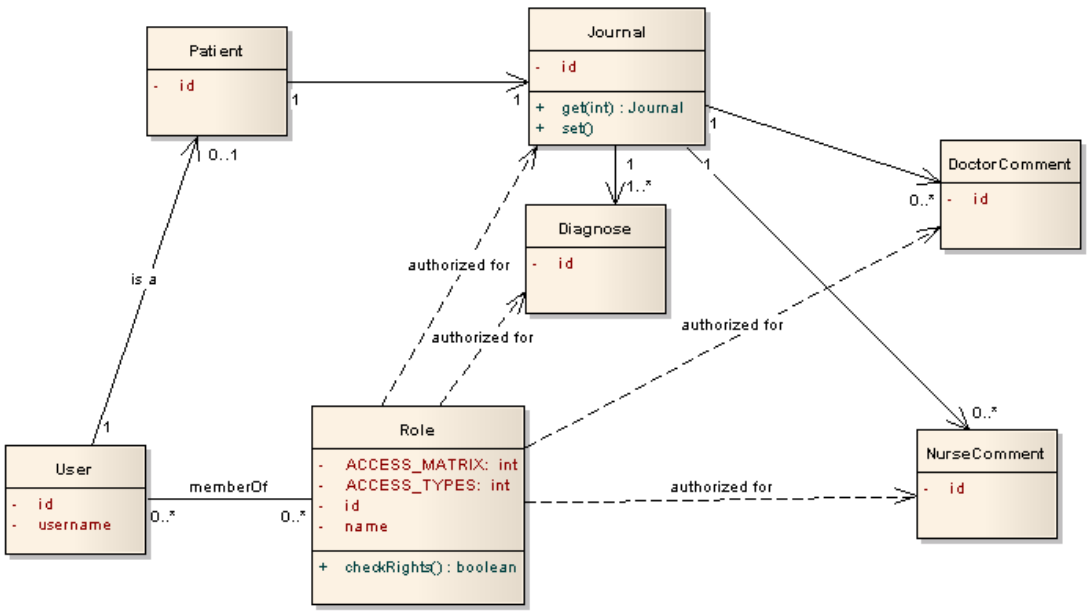


Figure 10.15: **Data Layer Design with Security Pattern** - RBAC is instantiated

The resulting design implements countermeasures by instantiating security patterns and making small adjustments to existing design elements. The countermeasures were identified through the countermeasure modelling process.

Part IV

Evaluation and Conclusion

Chapter 11

Evaluation and Discussion

This chapter evaluates the results, contributions and the research method. The completion of each research objective is evaluated and flaws or shortcomings of the research method are discussed. Other research initiatives are described and related to this thesis.

11.1 Contributions

This section discusses the significance of the contributions and possible flaws or shortcomings in the results. The discussion is broken down to each research objective. The objectives are restated below for convenience.

RO1	Develop a security modelling tool that enables software security experts and practitioners to create models of software security vulnerabilities and countermeasures.
RO2	Develop software design tool artefacts that provide reusable security expert knowledge in a format that is applicable at development time. This tool should apply the output of the tool described in RO1 to guide design of vulnerability countermeasures in software.
RO3	Document the tools as a proof of concept and evaluate contributions by performing a case study of the tools in objectives 1-2.

11.1.1 Extension of SeaMonster

The extension of SeaMonster takes it one step closer to being a complete security modelling tool that can be utilised by the software development industry. There are still several possibilities for improvement in SeaMonster when it comes to general usability and functionality. Examples are model decomposition, better support for highlighting of diagram elements, diagram interchange between other modelling and software development tools. The testing and case study has shown that SeaMonster is a functional security modelling tool for modelling of attacks, threats, vulnerabilities and finally countermeasures through security activity graphs. The preliminary study supports that this is a tangible contribution to secure software development. This completes Research Objective 1.

11.1.2 Security Pattern Design Templates

The security pattern design templates described in Section 4.1.1 and applied in the case study were developed and evaluated in (13). To summarise, there are some limitations to these:

- Only two security patterns are currently implemented as templates.
- The template format is short on descriptions which may decrease their usability. They should preserve the typical pattern representation to avoid losing information.
- The templates support only one software design tool (Enterprise Architect).

The lack of security improvement tools for software development was identified in the preliminary study. The security pattern design templates are a tangible contribution to software design tools as a concept, and it is believed that they will contribute to increased utilisation of security expert knowledge. This is a step towards bridging the gap; to include software developers in secure development. These templates complete Research Objective 2.

11.1.3 Case Study and Related Process

The case study was included to test the contributions and document them as a proof of concept. The case describes the implementation of two security improvement activities; two security pattern design templates. This could favourably be increased to create a more complete and realistic case. The analysis and design of the patient journal software system could be more thorough to resemble a complete software design cycle. This was however not possible in the restricted time. The research method is discussed in Section 11.2, this also includes some thoughts on the case study design. The author was able to test the tools more thoroughly through the case study and as such it was useful for evaluation purposes. The significance of this contribution lies in the possible increased security awareness among software developers who apply security improvement tools. It puts the spotlight on software security and involves the developers by utilising key security personnel and expert knowledge where it is needed; in the software *development* process.

The case study applied the approach and definitions described in Chapter 7. The countermeasure domain model defines software security concepts that are highly relevant to software developers. This model allows developers to discuss security concepts in an unambiguous manner, hopefully to increase security awareness. Including countermeasure modelling early in the software development lifecycle is a contribution to change the old 'penetrate and patch' ad-hoc security process. It is believed that the described approach with tools can help identify and mitigate possible security vulnerabilities before they are realised in software products. Although the case study functions as a proof of concept and test of contributions, it is not sufficient to state the effects and significance of these contributions. Research Objective 3 is completed but there is a need for more validation.

11.2 Research Method

This section discusses the chosen research method and the corresponding validity of the performed research. This is done in order to assess the work quality and better understand the results.

The evaluation phase of the thesis consists of a case study followed by a discussion, both performed by the author. This is a very subjective evaluation approach where there is a possible bias towards accepting the results and the hypothesis. A possible improvement would be to perform a case study with experiments using randomly selected software development practitioners. The experiments could include usability tests and case studies with the implemented approach and tools. This kind of study was not included in the research design because it is out of scope for one person with the available time. The software system in the case study was limited for the same reason. A more thorough study could include a more complete software development lifecycle to see the effects of introducing security in design through the described contributions. Performing a complete design, implementation and security testing of the patient journal system could be used for research and results validation.

The lack of validation makes it difficult to state the implications of the contributions in this thesis. The implemented approach and tools are based on trends and needs within secure software development identified in the preliminary study. The results are pointed out by mere observations from the author. Improvements to this work could be done in further work through a thorough validation.

11.3 Other Initiatives

Jürjens (39) presents methods and tools for model-based security engineering with UML. This method provides tool support for analysis of UML models against security requirements A verification framework uses security analysis on models of the security extension of UML, UMLsec (38). The aim of this work is to contribute towards usage of UML for secure systems development in practice. Examples show that this approach is able to find and correct several serious design flaws. This approach requires the use of and tool support for UMLsec as modelling language. The security modelling approaches in this thesis have no prerequisites or require prior knowledge of modelling techniques such as UML. An automated verification framework relies on the underlying analysis routines to provide and apply the security knowledge. The threats to software are constantly changing, and mitigating security risk requires an evolving

set of countermeasures. The scheme presented by Jürjens is more static than the approach in this thesis where the security knowledge lies with the security expert applying the approach. This approach and tools provide a way to translate the knowledge of a security expert into a format that is usable during software development. This way the security knowledge is not subject to being outdated. Using security modelling to identify security issues does not introduce a complex process or the need of analysis engines. A verification framework is a more complex technical solution.

Byers and Shahmeri (6) describe an approach to prevent vulnerabilities from being introduced during software development, which resembles the work of thesis. Their approach is based on formal modelling of vulnerability causes through VCGs. Developers can select a set of activities that will prevent the vulnerabilities. The activities are modelled using SAGs. This approach is independent of software development process and its strength lies in flexibility and support for evolution. A modelling tool named GOAT (57) has been developed to support this process. This tool can not be considered a complete security modelling tool because it is designed to support the process defined by Byers and Shahmeri (6) and therefore limited to VCGs and SAGs. SeaMonster covers a wider range of security modelling techniques although the approach in this thesis only applies VCGs and SAGs. This makes SeaMonster useful to a wider range of users and software security applications. According to the developers of GOAT, its usability is not satisfactory (57). It lacks an undo/redo operation and removed graph elements are not deleted from the underlying database directly which means a database administration tool is needed. Testing shows that the user interface is difficult for inexperienced users and sometimes cumbersome; e.g. creation of a new vulnerability requires a sequence of dialogue boxes. GOAT uses a client-server architecture which means it is more complex than SeaMonster. This architecture does however give possibilities for sharing of information. Information on vulnerabilities is stored in the database and can be shared across several GOAT clients. GOAT must be run with a server and a client; this is a disadvantage to users that want to model locally on a single computer because it adds complexity to the use cases. SeaMonster runs on a single desktop computer without any setup or configuration process. The approach in this thesis includes a central storing mechanism as described in Section 9.1, this was however not implemented

due to time restrictions. With further development, SeaMonster should utilise some security information repository to access and share security modelling data.

There are many approaches to software security, one of the bigger issues today however is that they are not applied and understood by software developers. This thesis arranges software security practice to be used by developers through security improvement tools. The intention is that tool support will increase security awareness and introducing security in software design tools will bridge the gap between developers and security experts. Software security practices are usually based on experience and best practices (34; 43; 45). The approach in this thesis relies on best practices to mitigate vulnerabilities, but it does not limit the set of best practices that can be applied. By analysing the causes of vulnerabilities, we can identify and apply any countermeasure without being limited to a predefined set of security practices. This approach does however rely on the best practices and thorough security knowledge is essential to perform sound vulnerability and countermeasure analysis.

Chapter 12

Conclusion

This chapter concludes the thesis by relating the results and contributions to the hypothesis. Possible further work is summarised.

12.1 Conclusion

A method to apply security knowledge during development has been defined. This method utilises two security improvement tools during software analysis and design; a security modelling tool and security pattern design templates. The overall goal of this thesis was to contribute to development of more secure software by improving three characteristics of current software development:

- **Mitigation of vulnerabilities during design.** The case study showed how vulnerabilities could be avoided during design by utilising the defined method and implemented tools to identify and apply best practice countermeasures. The contributions of this thesis help to identify countermeasures effectively and enable developers to apply them.

- **Increasing the software security awareness among developers.** The implemented security improvement tools increase the availability of security expert knowledge and therefore we believe they will contribute to increased security

awareness. This is not measured during the thesis and validation is needed to quantify the contribution to this characteristic.

- **Bridging the software security knowledge gap between security experts and developers.** The approach of tool supported use of a countermeasure model during development is a way to share security expert knowledge between security experts and software developers. The security experts have an approach based on security modelling of vulnerabilities and countermeasures, which produces security information in a format that can be utilised by developers; security activity graphs and security pattern templates. This enables tighter coupling between software security expertise and software development practice. Hopefully this will help to bridge the knowledge gap, however validation is needed to quantify the contribution to this characteristic.

The hypothesis can not be accepted or rejected based on the results of this thesis. There is a need for validation of results to quantify the effects of the defined modelling approach and implemented tools. There are however strong indications that favour acceptance.

12.2 Further Work

There is clearly a need for further work in security improvement tools and methods for software analysis and design. It is possible to improve and further develop the contributions of this thesis. Below is a list of ideas for further work:

- Validation of results. Some empirical validation should be performed to be able to validate results and accept or reject the hypothesis. The possible increase in security awareness and contribution to development of more secure software should be quantified.

- Extending functionality of SeaMonster diagrams. Diagram decomposition and improved connections between diagrams. This is to improve usability and increase the general modelling capabilities.

- Automatic composition. SeaMonster could implement algorithms to generate a SAG from a VCG automatically. This may benefit those who are not trained in security and will remove redundant modelling work. Generation of other diagram types could also be explored.
- Model validation. Study model checking and validation to determine if this is an efficient way of re-using expert security knowledge.
- Design tool artefacts. Study what other possibilities exist for design tool security artefacts except pattern templates.
- Security knowledge repository. Ideally, the security pattern templates and security models should be accessible from the design tool and SeaMonster respectively through a repository as described in Section 7.1.

Part V

Appendix

References

[1] M. Abadi and R.M. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Trans. Software Engineering*, 22(1):2–15, 1995.

[2] Packet Storm Advisories. http://packetstormsecurity.org/ - accessed 10.03.2008.

[3] C. Alberts and A. Dorofee. *Managing Information Security Risk: The OCTAVE Approach*. Addison Wesley, 2005.

[4] I. Alexander. Misuse cases: Use cases with hostile intent. *IEEE Software*, 20(1):58–66, Jan-Feb 2003.

[5] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley and Sons, 2001.

[6] S. Ardi, D. Byers, C. Duma, and N. Shahmehri. A cause-based approach to preventing software vulnerabilities. *(submitted)*, 2008.

[7] S. Ardi, D. Byers, and N. Shahmehri. Towards a structured unified process for software security. *Proceedings of the 2006 international workshop on Software engineering for secure systems*, 2006.

[8] S. Ardi, P.H. Meland, I.A. Tøndel, and N. Shahmehri. How can the developer benefit from security modeling? *Availability, Reliability and Security, ARES*, 2007.

[9] B. Arkin, S. Stender, and G. McGraw. Software penetration testing. *Security & Privacy, IEEE*, 3(1):84–87, Jan-Feb 2005.

[10] Computer Associates. Vulnerability information center. http://www3.ca.com/securityadvisor/vulninfo/ - accessed 10.03.2008.

[11] S. Barnum and G. McGraw. Knowledge for software security. *Security & Privacy, IEEE*, 3(2):74–78, March-April 2005.

[12] M. Bishop. *Computer Security: Art and Science*. Addison-Wesley, 2003.

[13] O. G. Borstad. Instantiation of security patterns during development. Depth Study, TDT4560 Safety and Security in IT Systems, Norwegian University of Science and Technology (NTNU), 2007.

[14] F. Buschmann, R. Meunier, H. Rohnert, and P. Sommerlad. *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley & Sons, 1996.

[15] D. Byers, S. Ardi, N. Shahmehri, and C. Duma. Modeling software vulnerabilities with vulnerability cause graphs. *In Proceedings of the International Conference on Software Maintenance, Philadelphia, PA, USA*, 2006.

[16] CERIAS. Coop vdb - the public vulnerability database. https://cirdb.cerias.purdue.edu/coopvdb/public/ - accessed 10.04.2008.

[17] B. Chess and G. McGraw. Static analysis for security. *Security & Privacy, IEEE*, 2(6):76–79, Nov-Dec 2004.

[18] Eclipse Community. Eclipse graphical editing framework. http://www.eclipse.org/gef/ - accessed 20.04.2008.

[19] Eclipse Community. Eclipse graphical modeling framework. http://www.eclipse.org/modeling/gmf/ - accessed 20.04.2008.

[20] Eclipse Community. Eclipse modeling framework. http://www.eclipse.org/modeling/emf/ - accessed 20.04.2008.

[21] Eclipse Community. Eclipse web site. http://www.eclipse.org/ - accessed 20.02.2008.

[22] D.Verdon and G. McGraw. Risk analysis in software design. *Security & Privacy, IEEE*, 2(4):89–84, July-Aug 2004.

[23] Jacobson et al. *Object-oriented software engineering: a use case driven approach*. Addison-Wesley, 1992.

[24] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *Software, IEEE*, 19(1):42–51, Jan-Feb 2002.

[25] M. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3), 1976.

[26] OWASP Foundation. Owasp top 10 - the ten most critical web application security vulnerabilities. *http://www.owasp.org/images/e/e8/OWASP_Top_10_2007.pdf - accessed 23.01.2008*, 2007.

[27] D. Geer. Just how secure are security products? *Computer*, 37(6):14–16, Jun 2004.

[28] D. Geer. Risk management is where the money is. *The Digital Commerce Society of Boston*, Nov 1998.

[29] D.P. Gilliam, T.L. Wolfe, J.S. Sherif, and M. Bishop. Software security checklist for the software life cycle. *Enabling Technologies: Infrastructure for Collaborative Enterprises. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on*, pages 243–248, June 2003.

[30] M. Graff and K. van Wyk. *Secure Coding: Principles and Practices*. O'Reilly and Associates, 2003.

[31] Object Management Group. Unified modeling language. http://www.uml.org - accessed 20.02.2008.

[32] L. Hatton. Reexamining the fault density - component size connection. *IEEE Software magazine, March/April*, 2, 1997.

[33] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code*. Addison-Wesley, 2004.

[34] M. Howard. Building more secure software with improved development processes. *Security & Privacy, IEEE*, 2(6):63–65, Nov-Dec 2004.

[35] M. Howard and D. LeBLanc. *Writing Secure Code*. Microsoft Press, 2003.

[36] IEEE. Standard for software reviews. http://ieeexplore.ieee.org/servlet/opac?punumber=5362 - accessed 10.04.2008.

[37] A. Jaquith. The security of applications: Not all are created equal. *Research report, @stake*, 2002.

[38] Jan Jürjens. Umlsec: Extending uml for secure systems development. *UML 2002 - The Unified Modeling Language*, 2460 of LNCS:412–425, 2002.

[39] Jan Jürjens. Sound methods and effective tools for model-based security engineering with uml. *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 322–331, 2005.

[40] D.M. Kienzle and M.C. Elder. Final technical report: Security patterns for web application development. *DARPA, Washington DC*, 2002.

[41] D.M. Kienzle, M.C. Elderm, D. Tyree, and J. Edwards-Hewitt. Security patterns repository version 1.0. *DARPA, Washington DC*, 2002.

[42] B.W. Lampson. Computer security in the real world. *Computer*, 37(6):37–46, June 2004.

[43] S. B. Lipner. The trustworthy computing security development lifecycle. *Proceedings of the 20th Annual Computer Security Applications Conference, Tucson, AZ, USA*, pages 2–13, 2004.

[44] J. McDermott and C. Fox. Using abuse case models for security requirements analysis. *In ACSAC Š99: Proceedings of the 15th Annual Computer Security Applications Conference, Washington, DC, USA, IEEE Computer Society*, page 55, 1999.

[45] G. McGraw. Software security. *IEEE Security & Privacy 2*, 2004.

[46] G. McGraw. *Software Security: Building Security In*. Addison-Wesley, 2006.

[47] G. McGraw. Software assurance for security. *Computer*, 32(4):103–105, Apr 1999.

[48] G. McGraw. From the ground up: the dimacs software security workshop. *IEEE Security & Privacy*, 1(2):59–66, Mar-Apr 2003.

[49] G. McGraw. From the ground up: The dimacs software security workshop. *Security & Privacy, IEEE*, 1(2):59–66, Mar-Apr 2003.

[50] G. McGraw and E.Felten. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley & Sons, 1999.

[51] N. Mead. Requirements engineering for survivable systems. *Technical Report CMU/SEI-2003-TN-013, Carnegie Mellon University*, 2003.

[52] M.Howard, D. Leblanc, and J. Viega. *19 Deadly Sins of Software Security*. McGraw-Hill, 2005.

[53] M.Howard and S.Lipner. *The Security Development Lifecycle*. Microsoft Press, 2006.

[54] Microsoft. Security risk management guide. http://www.microsoft.com/technet/security/guidance/complianceandpolicies/secrisk/default.mspx - accessed 10.04.2008.

[55] Microsoft. Threat modeling tool. http://www.microsoft.com/downloads/details.aspx?familyid=62830f95-0e61-4f87-88a6-e7c663444ac1&displaylang=en - accessed 23.01.2008.

[56] Sun Microsystems. Java servlet technology. http://java.sun.com/products/servlet/ - accessed 30.04.2008.

[57] J. Millving and M. Pedersen. Tool support for a software security process. MSc Thesis, Division for Databases and Information Techniques, Department of Computer and Information Science Linköpings universitet, Linköping, Sweden, 2007.

[58] MITRE. Common vulnerabilities and exposures (cve). http://cve.mitre.org - accessed 10.03.2008.

[59] A. P. Moore, R. J. Ellison, and R. C. Linger. Attack modeling for information security and survivability. *Dependable Systems and Networks Conference, Gothenburg, Sweden*, 2001.

[60] National Institute of Standards and Technology (NIST). The national vulnerability database. http://nvd.nist.gov/ - accessed 10.03.2008.

[61] Open-Source. Seamonster - security modeling software. http://sourceforge.net/projects/seamonster/ - accessed 22.05.2008, 2007.

[62] OSVDB. The open source vulnerability database. http://osvdb.org/ - accessed 10.03.2008.

[63] D.L. Parnas and M. Lawford. The role of inspection in software quality assurance. *Software Engineering, IEEE Transactions on*, 29(8):674–676, Aug 2003.

[64] S. Polepeddi. Software vulnerability taxonomy consolidation. *Technical Report UCRL-TH-208822, Lawrence Livermore National Laboratory*, 2005.

[65] The Open Web Application Security Project. Owasp category:vulnerability. http://www.owasp.org/index.php/Category:Vulnerability - accessed 10.03.2008.

[66] The Open Web Application Security Project. Threat risk modeling. http://www.owasp.org/index.php/Threat_Risk_Modeling - accessed 23.01.2008.

[67] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 9:1278–1308, Sep 1975.

[68] J.H. Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM Volume 17, Issue 7*, 1974.

[69] B. Schneier. Attack trees: Modeling security threats. *Dr. Dobb's Journal*, 1999.

[70] M. Schumacher. *Security Engineering with Patterns: Origins, Theoretical Models, and New Applications*. Springer, 2003.

[71] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons, 2005.

[72] M. Schumacher and U. Roedig. Security engineering with patterns. *Proceedings of Pattern Language of Programs*, 2001.

[73] Secunia. Secunia vulnerability archive. http://secunia.com - accessed 10.03.2008.

[74] SecurityFocus. The securityfocus vulnerability database. http://www.securityfocus.com - accessed 10.03.2008.

[75] SHIELDS. Shields project web site. http://er-projects.gf.liu.se/projectweb/473186f5c1d60/Index.html - accessed 10.03.2008.

[76] G. Sindre and L. Opdahl. Eliciting security requirements with misuse cases. *Requirements Engineering*, 10(1):34Ű–44, 2005.

[77] L. Røstad. An extended misuse case notation: Including vulnerabilities and the insider threat. *In Proceedings of The Twelfth Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQŠ06), Luxembourg*, 2006.

[78] C. Steel, R. Nagappan, and R. Lai. *Core Security Patterns, Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Prentice Hall, 2006.

[79] R.C. Summers. *Secure Computing: Threats and Safeguards*. McGraw-Hill College, 1997.

[80] F. Swiderski and W. Snyder. *Threat Modeling*. Microsoft Press, 2004.

[81] Internet Security Systems. X-force alerts and advisories. http://xforce.iss.net - accessed 10.03.2008.

[82] Sparx Systems. Enterprise architect web site. http://www.sparxsystems.com.au/ - accessed 07.03.2008.

[83] P. Torr. Demystifying the threat modeling process. *Security & Privacy, IEEE*, 3(5):66–70, Sept-Oct 2005.

[84] K. Tsipenyuk, B. Chess, and G. McGraw. Seven pernicious kingdoms: a taxonomy of software security errors. *Security & Privacy, IEEE*, 3(6):81–84, Nov-Dec 2005.

[85] K.R. van Wyk and G. McGraw. Bridging the gap between software development and information security. *Security & Privacy, IEEE*, 3(5):75–79, Sept-Oct 2005.

[86] J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, 2001.

[87] R. Winston. Managing the development of large software systems. *Proceedings of IEEE WESCON 26*, 1970.

[88] J. Yoder and J. Barcalow. Architectural patterns for enabling application security. *Proceedings of PLoP*, 1997.

Appendix A

Glossary

Actor

Active entity external to the system. An actor interacts with the system and can be a human being or other computing system.

Application Programming Interface (API)

Source code interface that defines the services provided by a code library. The API defines a contract between a caller and the callee, i.e. the code interface.

Asset

Information or resources which have value to an organization or person. An asset is to be protected by the system.

Attack Tree (AT)

An attack tree is a way of modelling attacks to a computer system. The tree describes the different ways in which the goal attack (the root node) can be achieved.

Common Vulnerabilities and Exposures (CVE)

The CVE is a dictionary of publicly known information about security vulnerabilities and exposures.

Countermeasure

Action taken in order to protect an asset against threats. In a software system, this refers to features that are implemented to prevent attacks, not to fulfill a functional requirement.

Countermeasure model

A formal representation of a countermeasure.

Data-flow Diagram (DFD)

A data-flow diagram is a diagram describing the data flow through an information system. These diagrams contain data flow between external entities, processes and data stores.

Eclipse Modeling Framework (EMF)

The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model.

Graphical Editing Framework (GEF)

Eclipse Graphical Editing Framework is a framework providing rendering and layout functions to create a graphical editor from an existing application model.

Graphical Modeling Framework (GMF)

The Eclipse Graphical Modeling Framework provides a generative component and runtime infrastructure for developing graphical editors based on EMF and GEF.

Injection flaw

Common type of security flaw in Web applications. Injection occurs when user-supplied data is sent to and accepted by an interpreter as part of a command or query.

The Open Web Application Security Project (OWASP)

The Open Web Application Security Project is a worldwide free and open community focused on improving the security of application software.

Pattern instantiation

Pattern instantiation refers to the process of implementing a software pattern.

Privilege

The right to perform a certain action in a system.

Risk Management Framework (RMF)

RMF is used in two contexts through the thesis:

1) as some process for handling risk in software development projects.

2) Cigital's risk management process, the Risk Management Framework (22).

Security activity

An activity performed during the software lifecycle to fulfill one or more security goals.

Security Activity Graph (SAG)

A security activity graph is a tree structure of activities that can be implemented during software development to prevent vulnerabilities. An activity is any measure that seeks to improve the security of the final software product.

Security pattern

A well-understood solution to a recurring security problem.

Security Vulnerability Repository Service (SVRS)

Security Vulnerability Repository Service is part of the technical approach of the EU project SHIELDS. The SVRS is an internet-accessible knowledge repository to be used by security tools.

Software Development Lifecycle (SDL)

The software development lifecycle is a general term for a complete software development process. A number of different SDL models have been created.

Spoofing, Tampering, Information disclosure, Denial of Service, Elevation of privilege (STRIDE)

STRIDE is a security threat modelling practice where potential security threats are categorised using six threat categories: spoofing, tampering, information disclosure, denial of service and elevation of privilege.

Tampering

A category of attacks based on changing information without the right to do so.

Threat

A potential for a security breach of an asset. A threat is a potential attack, it may or may not be applicable to a given system.

Unified Modeling Language (UML)

The Unified Modeling Language is a family of graphical notations used to describe and design software systems. The language is particularly suited for systems built using the object-oriented style.

Vulnerability

A feature of a system that may be exploited in a way that violates the system security.

Vulnerability Cause Graph (VCG)

A vulnerability cause graph is a security modelling technique used to analyse causes of software vulnerabilities. The causes of a vulnerability and their relationships are represented as a directed acyclic graph.

Vulnerability model

A formal representation of a vulnerability.

XML Metadata Interchange (XMI)

XMI is a standard for exhanging metadata via XML. In the context of this thesis it can be viewed as an interchange format for UML models.