A Framework for Secure Software Engineering:

A Knowledge Modeling based Approach for inferring Association between Source Code and Software Design Artifacts

K.A.I. Abeyrathna

13000022

B.N. Dahanayake

13000187

C.S. Samarage

13001078

Supervisor: Dr. Prasad Wimalaratne

Co-Supervisor: Mr. Chaman Wijesiriwardana

Submitted in partial fulfillment of the requirements of the

B.Sc (Hons) in Software Engineering 4th year Project

(SCS 4123)

January 2, 2018

Declaration

We certify that this dissertation does not incorporate, without acknowledgment, any material previously submitted for a degree or diploma in any university and to the best of my knowledge and belief, it does not contain any material previously published or written by another person or myself except where due reference is made in the text. We also hereby give consent for my dissertation, if accepted, be made available for photocopying and for interlibrary loans, and for the title and abstract to be made available to outside organizations.

Candidate Name:

……

Signature of Candidate Date:

Candidate Name:

……

Signature of Candidate Date:

Candidate Name:

……

Signature of Candidate Date:

This is to certify that this dissertation is based on the work of Ms. K.A.I. Abeyrathna, Mr. B.N. Dahanayake, and Mr. C.S. Samarage under my supervision. The dissertation has been prepared according to the format stipulated and is of the acceptable standard.

Supervisor Name: Dr. Prasad Wimalaratne

……

Signature of Supervisor Date:

This is to certify that this dissertation is based on the work of Ms. K.A.I. Abeyrathna, Mr. B.N. Dahanayake, and Mr. C.S. Samarage under my supervision. The dissertation has been prepared according to the format stipulated and is of the acceptable standard.

Co-Supervisor Name: Mr. Chaman Wijesiriwardana

……

Signature of Co-Supervisor Date:

Abstract

The popular approaches in securing software systems are operating system security, anti-virus, and firewalls. These approaches build security around the software system instead of integrating within the software system. However, it is not adequate since the root cause of software vulnerabilities reside within the software system. As a result, current approaches for Software Development have given a major focus on the integration of Security with the development process to develop secure and reliable software systems. Secure Software Engineering process integrates security in each phase of the software development lifecycle. A disconnected set of security-specific practices and tools are available to be used in each phase. Architecture-level security flaws arise in the design phase while security specific bugs are caused in the implementation level. Whenever a security issue in one phase is not resolved, it can be propagated to security ramifications in another phase. The unresolved architecture-level security flaws will create security bugs at the implementation level. A connectivity between the security bugs and architecture-level security flaws needs to be identified to solve the root cause of the security bug arise as a ramification.

This dissertation proposes a semi-automated approach to infer the association between security bugs and architecture-level security flaws by implementing a framework named Conexus as a proof of concept. The proposing approach uses static code analysis to identify the security bugs with respect to OWASP Top 10 vulnerability types and threat modeling to identify the architecture-level security flaws with respect to STRIDE threat categorization model. The identified security bugs and architecture-level security flaws are used as the input to the Conexus framework and the association between the two categories is derived using a Knowledge modeling based mechanism. The security controls violated by each STRIDE threat category and OWASP Top 10 vulnerability type are used in the Knowledge Base to identify the association between threat categories and bug categories through a semantic similarity matching model. Depending on the results generated from the Conexus framework, a software developer can revise the design to make a secure design followed by a secure code to eliminate and reduce security vulnerabilities in a software application.

Acknowledgments

This thesis is the result of nearly one year of devoted work by which we being fortunate to have the unconditional assistance of several people who have been extremely supportive in various ways. First and foremost we would like to offer my humble gratitude to Dr. Prasad Wimalartne, our supervisor, for been extremely supportive and for the guidance he had given us. He had been a great listener, a great advisor, and a great teacher.

We are also grateful to Mr. Chaman Wijesiriwardana, the co-supervisor, a lecturer at the University of Moratuwa and Mr. Lahiru Wijesekara, Software Engineer at Pearson Lanka for the tremendous encouragement, support and the guidance given throughout this project.

We would also like to acknowledge members of UMBC Ebiquity group for being generous to answer the project matters regarding UMBC Semantic Similarity Service we raised.

Our deepest gratitude goes to our beloved parents for their unconditional support, love and encouragement extended towards us throughout all the ups and downs of our life. Finally, we would like to thank all the people who had helped us throughout this project.

List of Acronyms

ASF	-	Application Security Frame
DFD	-	Data Flow Diagram
MS TMT	-	Microsoft Threat Modeling Tool
NVD	-	National Vulnerability Database
SDL	-	Security Development Lifecycle
OWASP	-	Open Web Application Security Project

Table of Contents

List of Figures

List of Tables

Chapter 1 : Introduction

The exponential growth of software consumption has raised many new challenges in the domain of security. The main reason behind securing software systems is it comprises of confidential and sensitive information [1]. In securing the software systems, popular approaches to security are operating system security, anti-virus and firewalls [1, 2]. By contrast, these approaches treat software as a black box ignoring the source code [1]. However, the source code is an important component of the software which needs to be secured [1, 2]. *Application Security* is another aspect of securing software systems which is based primarily on finding and fixing known security problems after they have been exploited in the fielded systems [2]. Since, *Application Security* follows naturally from a network-centric approach to security by embracing standard approaches, such as penetrate and patch and input filtering, it is not adequate to secure software [2]. Hence, *Software Security* which is a kind of computer security aims to address the aforementioned weaknesses directly by focusing on a secure design and implementation of software [2].

Security breaches begin by exploiting a vulnerability. A vulnerability is a weakness in a system, application, or network that is subject to exploitation or misuse [3]. In the context of software security, security vulnerabilities are security-relevant software defects that can be exploited to cause an undesired behavior [2]. Software defects occur in the software's design and its implementation where it can be categorized as design flaws and implementation bugs [1, 2, 4]. A bug is a defect in the implementation-level [1, 2, 4] and a flaw is a design-level or architecture-level software defect [1, 2, 4]. Despite the fact that defects categorized into two, there is an overlap between the two categories [5]. Further, 50% of the security problems found in software are architectural in nature [2, 4, 6]. An architecture-level security flaw may be instantiated in the source code as a security bug, though it is a defect in the architecture [4]. Thus, finding a solution for a security bug created due to an architecture-level security flaw requires identifying the association between the two categories. Hence, to ensure security, both security bugs and architecture-level security flaws must be eliminated and/or make them harder to exploit [2].

This dissertation proposes a semi-automated approach (Conexus Framework) to infer the association between security bugs and architecture-level security flaws of a particular software application. To achieve the goal, security bugs should be identified in the source code using a static code analysis tool and architecture-level security flaws using a threat modeling tool manually. The output generated from the static code analysis and threat modeling are used as an input to the Conexus Framework to derive the association. The association is limited to the security bug categories and architecture-level security flaw categories. Depending on the association, a software

developer can revise the design to make a secure design followed by a secure code to eliminate and reduce security vulnerabilities.

1.1 Motivation

Security vulnerabilities are not restricted to a few products but affect vendors and products available on the market [2]. A significant amount of software defects arise due to implementation bugs and architectural flaws [2, 4, 6]. The consequences of a software malfunction or a security breach might lead to a recall, millions in lost revenue or a loss of sensitive customer data [7]. The current challenge that software companies come across is to maintain the software quality with security while accelerating innovation [7]. Due to the necessity of security in a software application, security factor has been added as a characteristic rather than a sub-characteristic in ISO/IEC 25010:2011 Software Quality characteristic's [8].

In the traditional approach to software development lifecycle (SDLC), security is concerned in the final phases. Therefore, the effort and the cost needs to resolve a security vulnerability found in the final phases or after the release is relatively high [1, 2]. The current approach provided by vendors in solving security vulnerabilities is releasing security patches for the encountered issues. However, it is challenging to discover security vulnerabilities and release the security patches, before an attacker discover or exploit them and cause any harm [1, 2].

A research on Cyber Security Engineering from Carnegie Mellon University [9] have identified organizations that have focused on security in the early stages have seen major reductions in operational vulnerabilities, resulting reductions in software patching. Another case study from the aforementioned research [9] showed that the cost to fix requirement problems identified later in the project cost close to $2.5 million while the cost to fix these problems early in the lifecycle was $0.5 million. Thus, all these facts have driven to look for new ways in developing software over standard software development processes to further reduce overall software risk.

Building Security In paradigm introduced by Gary McGraw in 2004 [2] and Microsoft Trustworthy Initiative [10] started with Bill Gates Memo [11] in 2002 are the two main aspects which lead to a paradigm shift in the SDLC. This paradigm and the initiative introduce a Secure Software Development Lifecycle (Secure SDLC) which implies that security should be built in along with the development of the software by integrating into all phases of the SDLC [2, 10].

The general focus of all the aforementioned practices is to establish a set of practices in order to move developers into a Secure SDLC. This aimed at reducing the number and severity of security

vulnerabilities in software and hence identify and manage the security issues throughout the development instead of at maintenance.

1.2 Problem Definition

The *Building Security In* paradigm introduces seven touchpoints including code review, architectural risk analysis, penetration testing, risk-based security tests, abuse cases, security requirements and security operations to be used by integrating with the SDLC [2]. Touchpoints are process-agnostic software security best practices applied on a software artifact including requirements, use cases, design documents, architecture documents, code, test plans, and test results [2] which are aimed at identifying security issues in different phases of the SDLC. Security-specific built-in tools are available to be used with each touchpoint. However, there is a lack of connectivity between touchpoints and hence, tools are isolated to each phase of the lifecycle.

Whenever a security issue in one phase is not resolved, it can be propagated to security ramifications in a succeeding phase. In order to solve the security ramification, the real causes of the issue need to be solved. Hence, a connectivity between these touchpoints needs to be identified. Current tools are incapable of identifying the association between the software issues encountered at different phases of the life cycle due to the separation between the tools. A manual process has to be followed by the software developer to identify the association whenever in identifying the real causes of a security issue.

Considering the order of effectiveness, code review and architectural risk analysis are the two main touchpoints to be considered in Secure SDLC [2]. Security bugs are identified in the code review while architecture-level security flaws are identified in the architectural risk analysis [1, 2]. Whenever a security bug is encountered, the solution needs to be identified. The solution for the security bug can be a code-level or architecture-level remedy [1]. In case that, the solution is an architecture-level remedy, the software developer needs to identify the corresponding security flaw(s) for the relevant security bug manually. The process of identifying the relevant architecture-level security flaw(s) is challenging.

Hence, this dissertation proposes an approach (Conexus Framework) that targets in inferring the association between security bugs and architecture-level security flaws that are difficult to find manually. Conexus Framework is thus can be used in an iterative development environment or in a re-engineering process, or when a legacy code is used in a new software development environment, to reduce or remove software defects that are prone to security vulnerabilities.

1.3 Aims and Objectives

1.3.1 Goal of the Project

Build a framework for secure software engineering to aid the developers in finding the potential root-cause(s) for a security bug identified in the source code. The considered root-causes are the architecture-level security flaw(s) identified as threats.

1.3.2 Objectives

- Background study on Software Security and Secure SDLC.
- Finding the most appropriate threat modeling approach/tool by comparing the existing approaches/tools to be used in the architectural risk analysis to identify architecture-level security flaws.
- Finding the most appropriate static code analysis technique/tool by comparing the existing techniques/tools to be used in the code review to identify security bugs.
- Inferring the relationship between architecture-level security flaws and the security bugs using STRIDE Threat categorization [12] and OWASP Top 10 2013 [13] respectively.
- Determine the potential architecture-level security flaw(s) as the root cause(s) for the security bugs using the derived association.
- Evaluate the framework for the concept of secure design will lead to a secure software system by using sample project(s).

1.4 Scope

The proposed framework in this dissertation is only focused on analyzing associations of Java web-based projects. The Level-0 or Level-1 data flow diagram is the only design diagram required for the analysis process and the type of the architecture is not concerned. It is only focused on software security and not on any other approaches to computer security. The security bugs categorized irrespective of OWASP T10 will not be processed form the framework. Correspondingly, architecture-level security flaws categorized irrespective of STRIDE threat categorization model will not be processed.

1.5 Structure of the Dissertation

The rest of this document is as follows. Chapter 2 reviews the background and the existing literature related to the project. Chapter 3 describes the design architecture of the project in detailed and Chapter 4 presents the implementation of the project. The Chapter 5 illustrates the evaluation results and Chapter 6 concludes the dissertation with a conclusion and a discussion about the future work.

Chapter 2 : Background Study

2.1 Introduction

This section discusses the current research approaches and techniques that have been conducted related to the particular area of the study proposing in this dissertation. An analyzing of the relevant concepts and major tools used for architectural risk analysis and code review which considered as the two important practices in seven touchpoints in Secure SDLC are discussed in detail.

The specific aims of this background study are;

- To study security specific approaches and practices introduced to the general SDLC processes.
- To identify different types of analysis tools available for both code review and architectural risk analysis.
- To explore available tools and methodologies for aforementioned analysis with their advantages and disadvantages.
- To analyze the applicability of such tools for the proposed framework in this dissertation.
- To present the proposed framework which will be an aid for software developers in implementing security in a software application and evaluate the applicability of the framework during a Secure SDLC process.

2.2 Secure Software Development Processes

A growing body of research has been conducted in identifying how to integrate security within software development due to the increase in the number of software security problems. As aforementioned in Chapter 1, the pioneer approaches to solving security problems by applying a set of activities through SDLC are Seven Touchpoints introduced in *Building Security In* paradigm [2] and Security Development Lifecycle (SDL) introduced by Microsoft Trustworthy Initiative [1, 10].

Building Security In [2] is a collaborative effort that provides practices, tools, guidelines, rules, principles, and other resources that software developers, architects, and security practitioners can use to build security into software in every phase of its development. The *Building Security In* paradigm introduces seven touchpoints as aforementioned in Chapter 1. The touchpoints have been

integrated with Software Security knowledge organized into seven knowledge catalogs including Principles, Guidelines, Rules, Attack Patterns, Historical Risks, Vulnerabilities and Exploits as illustrated in Figure 2.1: Mapping of software security knowledge catalogs to various software artifacts and software security best practices [2].

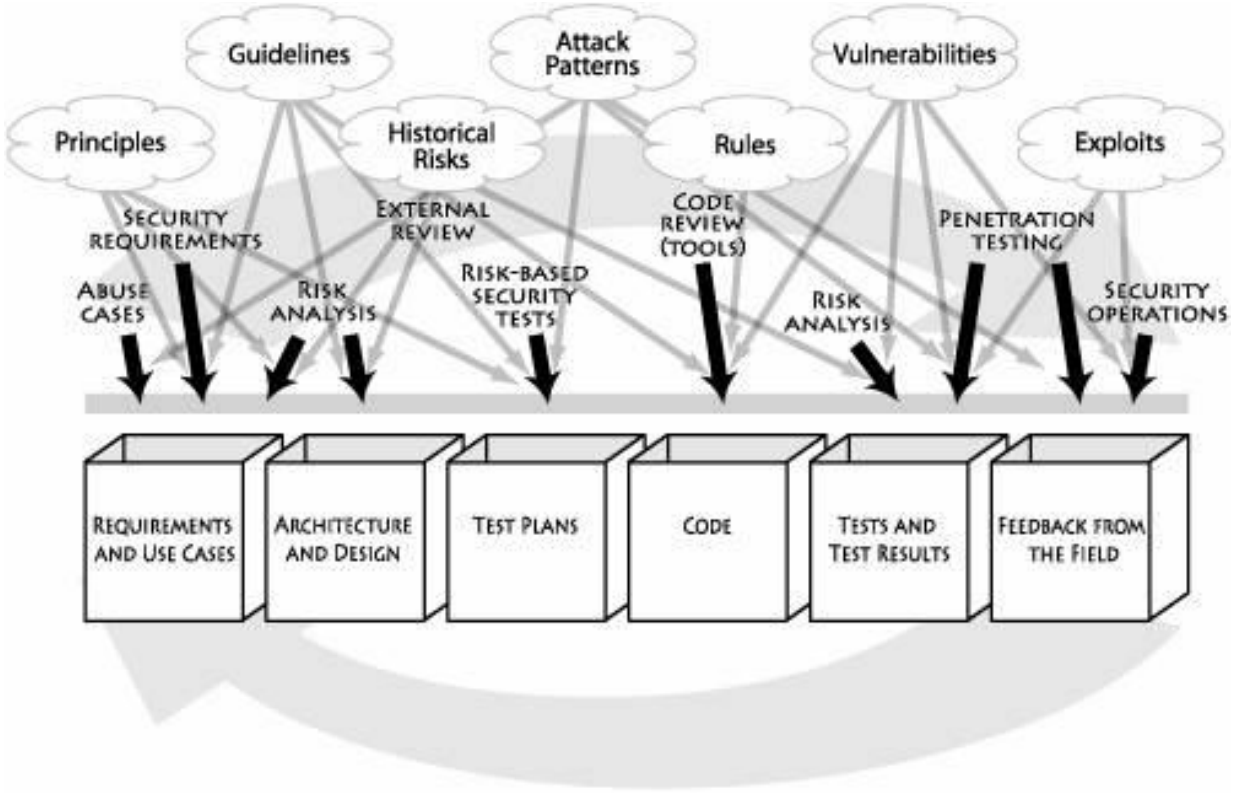


Figure 2.1: Mapping of software security knowledge catalogs to various software artifacts and software security best practices [2]

According to Gary McGraw, software security requires a careful balance by unifying the two sides of attack and defense, exploiting and designing and breaking and building into a coherent whole [2]. In order to make it easier for companies that follow best practices, different touchpoints are in ranking as illustrated in Figure 2.2: Touchpoints numbered according to effectiveness and importance [2].

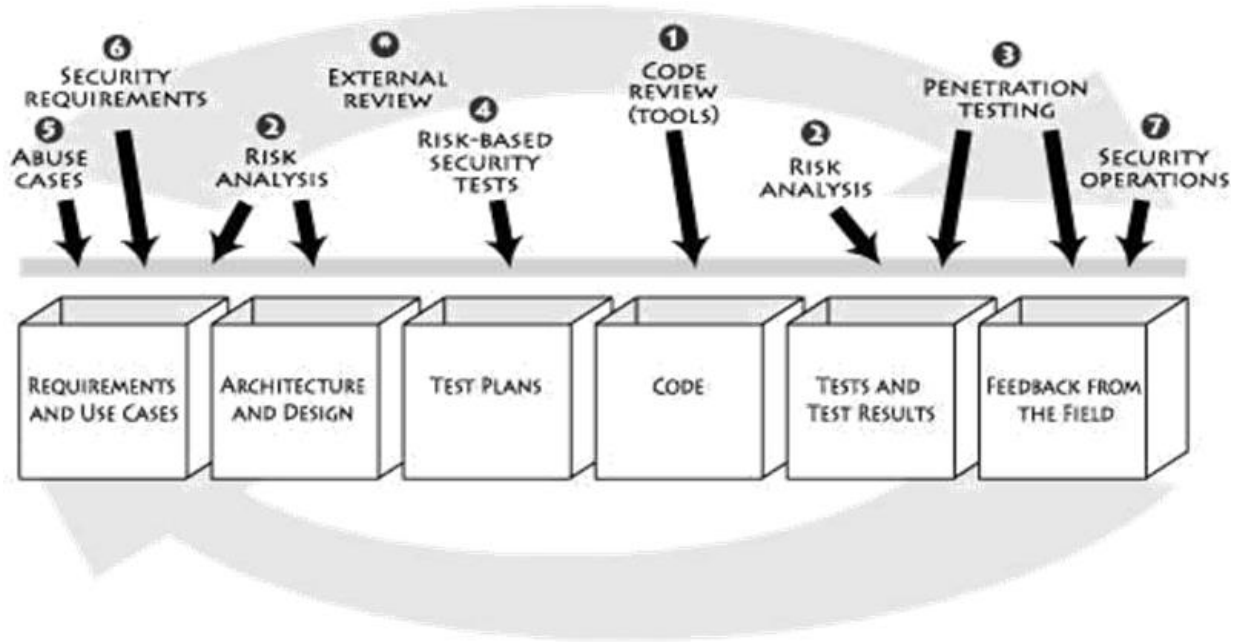


Figure 2.2: Touchpoints numbered according to effectiveness and importance [2]

Despite that Secure SDLC process introduced by Gary McGraw [2] conveys that it follows a traditional waterfall model, the current software development methodologies followed by most of the companies are iterative approaches. This process can be used in iterative approaches where security specific activities can be cycled through more than once as the software evolves. Thus, software security knowledge catalogs can be successfully applied to the SDLC by integrating with touchpoints regardless of the base software development model [2].

Correspondingly Microsoft has carried out a noteworthy effort under its Trustworthy Computing Initiative which focused on people, process, and technology to tackle the software security problem [1, 10]. On the people front, Microsoft trains every developer, tester, and program manager in basic techniques of building secure products. Microsoft's development process has been enhanced to make security a critical factor in design, coding, and testing of every product.

A key part of Microsoft’s Trustworthy Computing is the Security Development Lifecycle (SDL) [1, 10] which focuses on software development and introduces security and privacy throughout all phases of the software development process as illustrated in Figure 2.3: Microsoft Security Development Lifecycle . The Microsoft SDL combines a holistic and practical approach to reduce the number and severity of vulnerabilities in Microsoft products [1, 10].

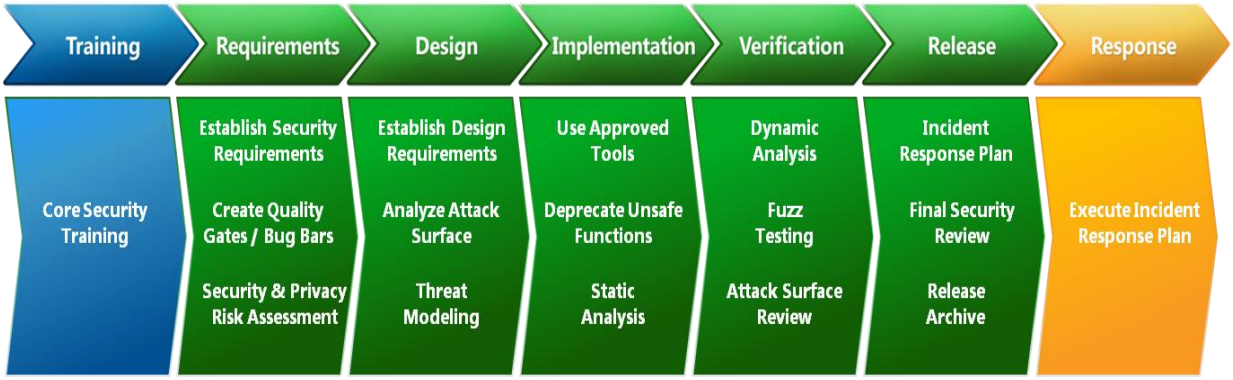


Figure 2.3: Microsoft Security Development Lifecycle [1, 10]

Conforming to the aforementioned approaches introduced to the SDLC, it conveys that Architectural risk analysis and Code review are two significant steps which should be conducted in a security specific SDLC process. The following sections 2.3 and 2.4 include a detailed description of the methodologies followed and tools used in architectural risk analysis and code review respectively.

2.3 Architectural Risk Analysis

Architecture-level security flaws account for 50% of security issues in software application [2, 4, 6, 10]. In accordance, Architectural risk analysis plays an important role in the software security process by explicitly identifying security risks in the architecture/design [1, 2]. The analysis can be started with the creation of a one-page overview of the system as the first step [14]. Consequently, apply a three-step process which includes Attack resistance, Ambiguity analysis, and Weakness analysis.

In the aforementioned first step of the creation of an overview of the system, a forest-level view of the application is created which allows global reasoning about security from the attacker perspective by checking security constraints and by assigning security properties to component instances [5]. Thenceforth, in the Attack resistance step of the three-step process is carried out to build an attack checklist to understand known attacks. The controls which are needed to prevent common attacks are considered in this step and STRIDE threat category model [12] for categorizing the attacks. STRIDE threat category model is a model introduced by Microsoft SDL which has been described in the succeeding section 2.3.1.1.

Ambiguity analysis is conducted to help in exposing an application's area of potential vulnerabilities. Trust modeling, Data Sensitivity modeling and Threat modeling are multiple modeling techniques considered for ambiguity analysis. Trust modeling is carried out to identify

the boundaries for security policy for function and data. Data Sensitivity modeling is used to identify privacy and trust issues for application data. Threat Modeling is conducted to identify the attacker's perspective and areas of weakness.

2.3.1 Threat Modeling

Threat modeling which is an architectural risk analysis method is a structured approach that enables to identify, quantify, and address the security risks associated with an application [1, 2, 15]. The inclusion of threat modeling in the Secure SDLC helps in ensuring that applications are developed with security-built-in from the beginning.

A DFD needs to be produced for the threat modeling process. There are different levels of DFD as Level-0, Level-1, and Level-2 etc. A paper published by Abi-Antoun, et al [16] states that drawing Level-0 or Level-1 DFD is sufficient to identify the architecture-level security flaws in the design. The Level-2 and higher DFD diagrams require looking at the internals of the application binaries which is difficult to achieve in the design phase. Hence, Level-0 or Level-1 DFD is acceptable for threat modeling process.

Threat modeling process can be decomposed into 03 steps as follows [15].

Step 1: Decompose the Application

This step concerned with gaining an understanding of the application and how it interacts with external entities which result in identifying entry points to see where a potential attacker could interact with the application, identifying trust levels which represent the access rights that the application will grant to external entities and identify assets that the attacker would be interested in. Thus, this information is used to produce data flow diagrams (DFDs) for the application.

Step 2: Determine and rank threats.

In this step, threats are determined and categorized according to a threat categorization methodology. The goal of threat categorization is to identify threats from both attacker's perspective and defensive perspective. DFDs produced in step 1 is used to identify potential threat targets from the attacker's perspective since Threat Modeling examines the system from a potential attacker's perspective. The identified threats are ranked using a risk model.

Step 3: Determine countermeasures and mitigation.

In this step, mitigations and countermeasures are identified for the ranked threats.

All the details gained by the aforementioned steps are documented and the resulting document is the threat model for the application.

2.3.1.1 STRIDE Threat Categorization

STRIDE [12] is a threat categorization model introduced by Microsoft. STRIDE helps to identify threats from the attacker's perspective by classifying attacker's goals into 06 threat categories. The set of STRIDE threat categories and the affected security controls due to each of the threat category is illustrated in Table 2-1: STRIDE Threat Categories and the affected security controls [15].

Table 2-1: STRIDE Threat Categories and the affected security controls [15]

	STRIDE Threat Type	**Security Control**
S	Spoofing	Authentication
T	Tampering	Integrity
R	Repudiation	Non-Repudiation
I	Information Disclosure	Confidentiality
D	Denial of Service	Availability
E	Elevation of Privilege	Authorization

2.3.1.2 ASF Threat Categorization

Application Security Frame (ASF) [15] is a threat categorization model which helps to identify the threats from the defensive perspective. It helps to identify the threats as weaknesses of security controls. In addition, ASF is known as a security control categorization model to identify threats. Security Control types included in the ASF [15] are Authentication, Authorization, Configuration Management, Data Protection in Storage and Transit, Data Validation / Parameter Validation, Error Handling and Exception Management, User and Session Management and Auditing and Logging.

2.3.2 Conceptual Analysis

A research article published by M. Frydman, et al [17] introduces an automated approach for Threat modeling by producing two data structures, Identification trees, and Mitigation trees, to identify threats in software designs and advise mitigation techniques while concerning specification requirements and cost concerns. Identification trees are used to identify threats in the software design and Mitigation trees describe countermeasures of threats and classify the set of software specifications that are required to mitigate a specific threat. The two data structures and ranking information of threats have been combined in a knowledge base called Attack Patterns.

The automated model is based on the Microsoft Threat modeling methodology and relies on the information contained in the attack patterns. It uses the Identification trees to find the potential threats of a given software model and uses the mitigation trees to compute the software specifications of least effort needed to mitigate the detected threats during the development lifecycle.

A research conference paper published by X. Yuan, et al [18] describes an effort being conducted to develop a tool to retrieve relevant Common Attack Pattern Enumeration and Classification (CAPEC) type attack patterns for software development. CAPEC attack patterns are valuable resources that can help software developers to think like an attacker and have the potential to be used in each phase of the secure software development lifecycle. The tool is capable of retrieving attack patterns most relevant to a particular STRIDE [12] type and can be used in conjunction with Microsoft Threat modeling tool (MS TMT). It has the capability to search for CAPEC attack patterns using keywords. A metric has been defined in this tool to measure the degree of usefulness of an attack pattern and the degree of its relevance to a particular STRIDE category.

The paper published by Bernhard J. Berger, et al [19] proposes a practical approach to Architectural risk analysis that leverages Microsoft Threat modeling approach. This proposed approach uses extended DFDs and a security knowledge base to be used as an aid for software developers in detecting vulnerabilities in software architectures. The knowledge base contains information on architectural weaknesses and possible mitigations. The extended DFDs are a refinement of DFDs which are a representation of the system architecture. The analysis resulted from the tool leads to a list of tackled problems and a list of not handled security flaws.

2.3.3 Threat Modeling Tools

Microsoft Threat Modeling Tool (MS TMT) [20] is a tool that helps in finding threats in the design phase of software projects. It is based on the STRIDE threat categorization model [12] and enables any developer or software architect to;

- Communicate about the security design of their systems.
- Analyze those designs for potential security issues using a proven methodology.
- Suggest and managing mitigations for security issues.

The tool graphically identifies processes and data flows (using DFDs) that comprise an application or a service. Despite the fact that the identified threats are categorized to STRIDE, the tool lacks in providing any prevention technique for them.

ThreatModeler [21] which is a web-based Threat Modeling tool can be used with agile development methodologies to create and utilize application threat models across the SDLC process. It automates the identification, enumeration, and prioritization of potential threats based on real-world intelligence and the organization's risk mitigation policies. The tool reform the traditional approach by automatically building threat models from the functional information user provide about their applications and systems. Integrated with an Intelligent Threat Engine (ITE) which is capable of automatically analyzing threat models and predict where the potential threat exists. The identified threats are ranked by risk and generate abuse cases from the ITE. The tool lacks in categorizing the identified threats into STRIDE or any other accepted threat categorization model and generating output reports which can be saved after the analyzing process.

A paper published by I. Williams, et al [22] has evaluated the effectiveness MS TMT using two parts. One is Threat modeling using a manual process and the other part is using the MS TMT for Threat Modeling. The evaluation has been conducted with the help of a set of university students and using a mock online shopping web application. The study of the evaluation shows that the students as a whole have improved their work on threat modeling with the use of tool compared with not using the tool. The evaluation shows that selecting different DFD elements in the Threat modeling tool will generate different threats and the abstraction level of the DFD affects the number of potential threats identified.

2.4 Code Review

Security Bugs which can be found in the implementation phase of a software project are identified in the code review process. Two approaches to code review have been defined as Dynamic Code analysis and Static Code analysis and this dissertation is only focused on security-specific approaches in static code analysis.

2.4.1 Static Code Analysis

Static Code analysis is a software testing method that can be performed in the different stages of the software development to ensure software is free of vulnerabilities introduced to the code due to programming errors [23]. In the context of security review perspective, Static Application Security Testing (SAST) is a well-known method for discovering vulnerabilities and it is classified into a white-box test [24].

The development processes and practices in developing secure software are primarily focused on the use of best practice recommendations which are aimed at addressing common mistakes within

a current development process [25]. These include perspective practices such as OWASP Top 10 [13] and *Building Security In Meta-Model* [2]. A paper published by N. Kaur, et al [26] describes that the efforts in the implementation of secure software have focused on studying implementation vulnerabilities like SQL Injections and Cross-Site-Scripting as listed in OWASP T10.

2.4.1.1 OWASP Top 10 and OWASP Proactive Controls

OWASP Top 10 [13] (hereafter OWASP T10) is the ten most critical web application security risks which provide a powerful awareness document for web application security. The different versions of OWASP T10 are focused on identifying the most common vulnerabilities which have always been organized around risks. It depicts how an attacker can potentially use many different paths through an application to do harm to an organization where each of the paths represents a risk.

OWASP Proactive Controls (hereafter, Proactive controls) [27] is the ten most important control and control categories. This is a developer-centric list of security techniques which can be included in every software project. Each proactive control helps in preventing one or more of the OWASP Top Ten web application security vulnerabilities.

The mapping between OWASP T10 2013 and Proactive Controls 2016 is illustrated in Table 2-2: OWASP T10 2013 mapping with Proactive Controls 2016 .

Table 2-2: OWASP T10 2013 mapping with Proactive Controls 2016 [27]

	A1: Injection	A2: Broken Authentication and Session Management	A3: Cross-Site Scripting (XSS)	A4: Insecure Direct Object Reference	A5: Security Misconfiguration	A6: Sensitive Data Exposure	A7: Missing Function Level Access Control	A8: Cross-Site Request Forgery (CSRF)	A9: Using Components with Known Vulnerabilities	A10: Unvalidated Redirects and Forwards
C1: Verify for Security Early and Often	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
C2: Parameterize Queries	✓									
C3: Encode Data	✓		✓							
C4: Validate All Inputs	✓		✓							✓
C5: Implement Identity and Authentication Controls		✓								
C6: Implement Appropriate Access Controls				✓			✓			
C7: Protect Data						✓				
C8: Implement Logging and Intrusion Detection	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
C9: Leverage Security Frameworks and Libraries	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
C10: Error and Exception Handling	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

2.4.2 Conceptual Analysis

The white paper published by Coverity [28] outlines a practical approach for implementing secure practices into the software development lifecycle. It has introduced a development testing platform which allows the development organizations to coherently integrate code testing into the software development process. Coverity development testing solutions train developers to address both security and quality when testing the code which leads to secure software development practices. The commonly found potentially critical security defects in the source code are identified from this platform and will be provided an aid for the developers to fix them. The major weakness of this platform is the lack of linking the root-cause with the design phase by limiting it to the implementation phase.

The paper published by Sultan S. Alqahtani, et al [29] have proposed, while known vulnerabilities and security concerns are reported in specialized vulnerability databases, these repositories often remain information silos. In this research, a modeling approach is introduced, which eliminates these silos by linking security knowledge with other software artifacts to improve traceability and trust in software products. A Security Vulnerabilities Analysis Framework (SV-AF) is introduced in this approach to support evidence-based vulnerability detection. Two case studies are presented to illustrate the applicability of the presented approach. In these case studies, the National Vulnerability Database (NVD) and the Maven build repository are linked to trace vulnerabilities across repository and project boundaries. In the analysis, 750 Maven project releases are identified as directly affected by known security vulnerabilities and by considering transitive dependencies, an additional 415604 Maven projects can be identified as potentially affected by these vulnerabilities. This approach for ensuring security in a software is limited to the code level and connecting the design phase with the identified bugs is not supported in the framework.

2.4.3 Static Code Analysis Tools

The Coverity Development Testing Platform [28] introduced by Coverity, provides development teams the ability to test code for defects in a non-intrusive manner. It integrates with IDEs like Eclipse or Visual Studio, and developers can identify quality and security defects from within their IDE, without disrupting the development workflow. The identified defects are automatically notified to the developers within the existing workflow, prioritized by risk and impact. Developers have one-click access to a rich defect knowledge base which takes the guesswork out researching unfamiliar defects and helps developers to find the root-cause of a defect in an efficient manner. Considering the fact that many organizations leverage shared code across projects and services,

Coverity Static Analysis will also show the development team all of the places across the shared code where that defect exists, so a fix can be applied in all these places. However, this is a commercial tool.

Find-Sec-Bugs [30] is a static analysis tool used to find security audits of java web applications. It can detect 113 different vulnerability types with over 689 unique API signatures. The plugin covers popular frameworks including Spring-MVC, Struts, and Tapestry etc. and available for Eclipse, IntelliJ, Android Studio and NetBeans. Command line integration is available with Ant and Maven. The plugin can be used with systems such as Jenkins and SonarQube and extensive references are given for each bug patterns with reference to OWASP T10 and Common Weakness Enumeration (CWE). A detailed report of the results of the analyzing process is provided from the plugin which can be saved in XML format. A set of predefined bug patterns are available in this tool which has been categorized in accordance with OWASP T10. Despite that, the output reports generated from the tool do not contain the detected bugs as a categorization of OWASP T10.

SonarQube [31] is another static code analysis tool used to collect and analyze source code, measuring quality and providing reports for the project. It combines static and dynamic analysis tools and enables quality to be measured continuously over time. Everything that affects the code base, from minor styling details to critical design errors, is inspected and evaluated by SonarQube, thereby enabling developers to access and track code analysis data ranging from styling errors, potential bugs, and code defects to design inefficiencies, code duplication, lack of test coverage, and excess complexity. The Sonar platform analyzes source code from different aspects and hence it drills down to the source code layer by layer, moving from the module level down to the class level. At each level, SonarQube produces metric values and statistics, revealing problematic areas in the source that require inspection or improvement.

A paper submitted by Harneet Kaur [32] has included a comparison conducted between Find-Sec-Bugs and SonarQube as listed in Table 2-3: A comparison of static code analysis tools (Find-Sec-Bugs and SonarQube) .

The automated categorization of security bugs into the OWASP T10 categorization is an advantage of the SonarQube [31] tool compared to Find-Sec-Bugs plugin [30]. Despite that fact, the OWASP T10 categorization of the SonarQube is limited to A1, A2, A5, A6, A7, and A9. The inability of generating a final report that can be saved after the analysis is a major drawback of the SonarQube tool. The number of vulnerability types identified for the supported languages by SonarQube tool and the supported OWASP T10 categories is illustrated in the following Table 2-4: No. of

Vulnerability Types identified and supported OWASP T10 categories for each supported languages from SonarQube .

Table 2-3: A comparison of static code analysis tools (Find-Sec-Bugs and SonarQube) [32]

	Find-Sec-Bugs [30]	**SonarQube** [31]
Purpose	Finding potential bugs	Managing overall quality assurance
Types of Verification	Code-level design flaws, bad practice, multi-threaded correctness	Bugs, duplications, vulnerabilities, code smell, technical debt, overall quality statistics, and metrics
Plugins and Integration with Jenkins	Plugin available within Jenkins	Plugin once integrated with Jenkins has an option to further integrate Find-Sec-Bugs as sub-plugin on SonarQube server
Custom Rules	132 rules written in Java and analyze Java code only	Customizable 1000+ rules supporting more than 20 languages
Analysis Results	Displayed on Jenkins server with no flexibility of customization for false positives	Displayed on SonarQube server with the flexibility to eliminate false positives, assign severity levels, close issues and check compliant code examples
Authorization and Accessibility	Non-private accessibility of results on Jenkins server	Only authorized users can access results by logging into SonarQube server

Table 2-4: No. of Vulnerability Types identified and supported OWASP T10 categories for each supported languages from SonarQube [31]

Language	**No. of Vulnerability Types**	**Supported OWASP T10 categories**
Java	33	A1, A2, A5, A6, A7, A9
PHP	10	A1, A2, A3, A5
JavaScript	9	A3, A6, A9
C#	6	A6
Flex	6	-
Python	1	-

The Table 2-4 depicts that identification of security bugs by SonarQube is comparatively high for the Java language.

If static code analysis is used at the right stage during the development of a project, it has the capability of identifying critical security vulnerabilities or security bugs which may not appear to

the surface during or after the project release [32]. In addition to that, with a tool like SonarQube, the generated false positives in an analysis can be eliminated more efficiently [32]. SonarQube is not only useful for maintaining and assuring the security of one project but the configuration can be used in many projects without the restriction of the language used to develop the project.

2.5 Summary

The aforementioned approaches for software security depict that Threat modeling is a well-known and accepted method used in architectural risk analysis. Static code analysis which is conducted in the code review process is a methodology for software testing used with the aid of static analysis tools focused on identifying security bugs.

The research experiments conducted in the domain of architectural risk analysis and Threat modeling has identified MS TMT as a well-established and industry accepted tool and it has been used in this proposed framework. The facts included in this section depicts that SonarQube is a code quality measuring tool which has been widely used in the software security domain. The proposed framework from this dissertation has used SonarQube for identifying software bugs and OWASP T10 categorization given for the identified software bugs is a major advantage of the tool for the proposed approach for the framework.

Architectural-level security flaws of a software project can be identified using the MS TMT as threats and correspondingly security bugs can be identified from SonarQube as vulnerabilities. There is a high possibility for a software project to be exposed to security attacks if the identified threats are not fixed during the implementation of the software. The main reason for this fact is that the root causes of the security bugs in a software are lie in the design phase. Despite the fact that security flaws and security bugs are identified for a particular software project, there is a lack of connecting these two results. The framework proposed in this dissertation is preliminarily focused on finding a connection between the identified security flaws and security bugs.

Chapter 3 : Design

3.1 Introduction

This chapter describes the proposed design approach to the aforementioned problem in this study with the methodologies used and considerations on designing the solution. Based on the critical review done in the background study, several design concerns were identified. Based on these design concerns and the identified requirements the system architecture was developed. The system architecture consists of five modules as Threat-based Processing Module, Security Bug-based Processing Module, Association Inference Module, Knowledge Base, and Output Builder. The detailed descriptions of problem analysis, design constraints, solution approach, and system architecture are explained under respective sections.

3.2 Problem Analysis

The primary aim of this project was to develop a framework to infer the association between source code and software design artifacts. To achieve this goal, an exploratory type of research was carried out by exploring relevant documents, dissertations, and tools. The main approach that was used to identify the requirements, functional details and system architecture was by analyzing the information gathered through the background study. The limitations that were identified in the current approaches, ideas and information gathered through concept papers, were incorporated in designing the system architecture.

Considering the fact that security bugs and architecture-level security flaws are the major causes for security issues as aforementioned in Chapter 1, code review and the architectural risk analysis were taken into account in inferring the association. The additional reason for selecting the preceding touchpoints was the order of effectiveness of seven touchpoints in the Secure SDLC [2].

Static code analysis tools [30, 31] were explored in order to elect a tool to identify the relevant code-level security bugs and Threat modeling tools [20, 21] were examined in order to elect a tool to identify the relevant architecture-level security flaws vulnerable to security issues as aforementioned in Chapter 2. Using the literature survey conducted, MS TMT [20] and SonarQube [31] were selected as the Threat modeling tool and Static code analysis tool respectively.

However, the inadequacy of a direct approach in inferring the association for the aforementioned problem definition in Chapter 1, the research component was based on discovering an approach to infer the association between security bugs and architecture-level security flaws.

3.3 Design Constraints and Assumptions

The analyzing software project from the Conexus Framework can be a complete software application or a component of a software application. According to the background study conducted in Chapter 2, the number of vulnerability types identified by SonarQube is maximum for the Java Web Application projects compared with the other supported languages. Hence, the analyzing project should be a Java Web Application which is compatible with the supported version of the Java language from SonarQube.

The intended users of the Conexus Framework are software developers who should have a basic knowledge on software security up to some extent in order to use the tool. The user should draw the Level-0 or Level-1 data flow diagram of the relevant analyzing software project or project component using the MS TMT. According to the background study conducted in Chapter 2, it is sufficient to draw the Level-0 or Level-1 DFD to identify the architecture-level security flaws. The Threat Model saved in TMT7 format should be used as the input to the Threat-based Processing Module of the Conexus Framework assuming that threats are categorized with respect to STRIDE threat categorization model [12].

Subsequently, the user should analyze the source code of the relevant analyzing software project or project component using SonarQube and identify the security bugs categorized with respect to OWASP T10 [13]. In SonarQube, security bugs are represented as Vulnerabilities. Thereafter, the identified vulnerabilities should be input into the Security Bug-based Processing Module of the Conexus Framework. In the case of SonarQube does not encounter any security bugs with respect to OWASP T10, the framework will not be able to generate the association report.

SonarQube has the capability of identifying different types of vulnerabilities. However, the rest of vulnerabilities which are not categorized into OWASP T10 are out of the scope of this framework.

The generated association report can be exported into CNX (supported by the Conexus Framework), XML and JSON format. The mapping between security bugs to architecture-level security flaws is limited to the level of security bug categorization model to threat categorization model due to the limited resources and the time.

3.4 Conexus Framework Approach

The main approach used to infer the association between security bugs and architecture-level security flaws was based on STRIDE threat categorization model [12] and OWASP T10 [13] respectively. The architecture-level security flaws of a specific software application are identified using MS TMT as threats with respect to STRIDE threat categorization model. The security bugs are identified using SonarQube as vulnerabilities with respect to OWASP T10. Subsequently, a method was discovered to identify a mapping between OWASP T10 and STRIDE in order to infer the association between security bugs and architecture-level security flaws.

The first method was based on identifying a direct mapping between OWASP T10 and STRIDE. The approach used for the aforementioned mapping was failed due to the inability of identifying a connection directly. The succeeding method used security controls violated by each STRIDE threat types and OWASP T10 vulnerabilities in identifying the relationship. The detailed description of the two methods as described in the following sub-modules.

3.4.1 Method 1: Direct mapping between OWASP T10 and STRIDE

The descriptions given for each vulnerability in OWASP T10 were mapped with the descriptions given for each threat category in STRIDE using a semantic similarity matching model. Despite that, the high-level description provided by the STRIDE was not adequate to attain a descriptive meaning to be used in the semantic similarity matching. Consequently, the details included in the descriptions of the OWASP T10 vulnerabilities were not supportive enough to get a semantic similarity. The aforementioned two facts made this method not applicable to finding the association between security bugs and architecture-level security flaws. Thus, a new method was identified.

3.4.2 Method 2: A mapping between OWASP T10 and STRIDE via Security Controls

An attacker can exploit a security breach of a software system due to a weakness of a security control. The security controls violated by each OWASP T10 vulnerabilities are represented by Proactive controls as illustrated in Table 2-2. Correspondingly, each threat type identified by STRIDE threat categorization is violating a specific security control as represented in Table 2-1.

ASF is a security control categorization mechanism which supports in identifying threats as weaknesses in security controls [15]. For an in-depth analysis of the threats affecting the software application data and functional assets, both the STRIDE attacker view and the ASF defensive view

for the enumeration of threats are considered as essential [15]. A relationship between STRIDE and ASF has been identified as illustrated in following Table 3-1: Mapping between STRIDE and ASF .

Table 3-1: Mapping between STRIDE and ASF [33]

ASF Type	**STRIDE Attack Type**
Auditing and Logging	Repudiation
Authentication	Spoofing
Authorization	Elevation of privileges
Configuration Management	Elevation of privileges
Data Protection in Storage and Transit	Tampering

The relationship depicts from the aforementioned Table 3-1 is not a complete association between ASF and STRIDE due to each category of STRIDE lacks an association to ASF type. Hence, this association was further improved by using the details given in the book Threat Modeling [34] as illustrated in following Table 3-2: Enhanced Mapping between ASF and STRIDE .

Table 3-2: Enhanced Mapping between ASF and STRIDE [33, 34]

ASF Type	**STRIDE Attack Type**
Authentication	Spoofing
Authorization	Tampering, Information Disclosure, Elevation of privileges
Configuration Management	Repudiation, Elevation of privileges
Data Protection in Storage and Transit	Tampering, Information Disclosure
Data Validation / Parameter Validation	Tampering
Error Handling and Exception Management	Information Disclosure
User and Session Management	Spoofing
Auditing and Logging	Repudiation

The security controls violated from both aspects of STRIDE and OWASP T10 was used to derive the association using a semantic text similarity matching model. The set of countermeasures of ASF and summarized Proactive control descriptions are used to get the semantic similarity between ASF and Proactive controls. The summarized view of this approach is illustrated by the Figure 3.1: High-level view of the method used to identify the association between STRIDE and OWASP T10.

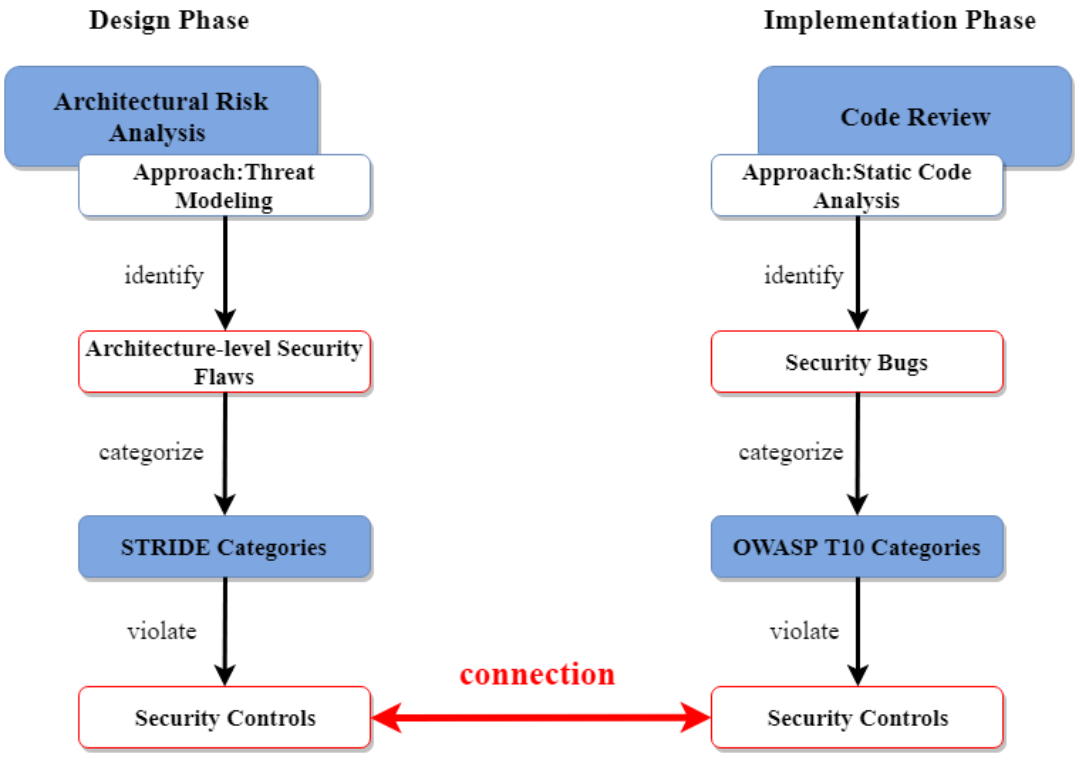


Figure 3.1: High-level view of the method used to identify the association between STRIDE and OWASP T10

3.5 Conexus Framework Architecture

This is a semi-automated software application developed as a proof-of-concept to represent the proposed solution given in this dissertation. The architecture proposed for the Conexus Framework consists of five main modules as Threat-based Processing Module, Security Bug-based Processing Module, Association Inference Module, Knowledge Base and Output Builder as illustrated in the following Figure 3.2: Conexus Framework Architecture.

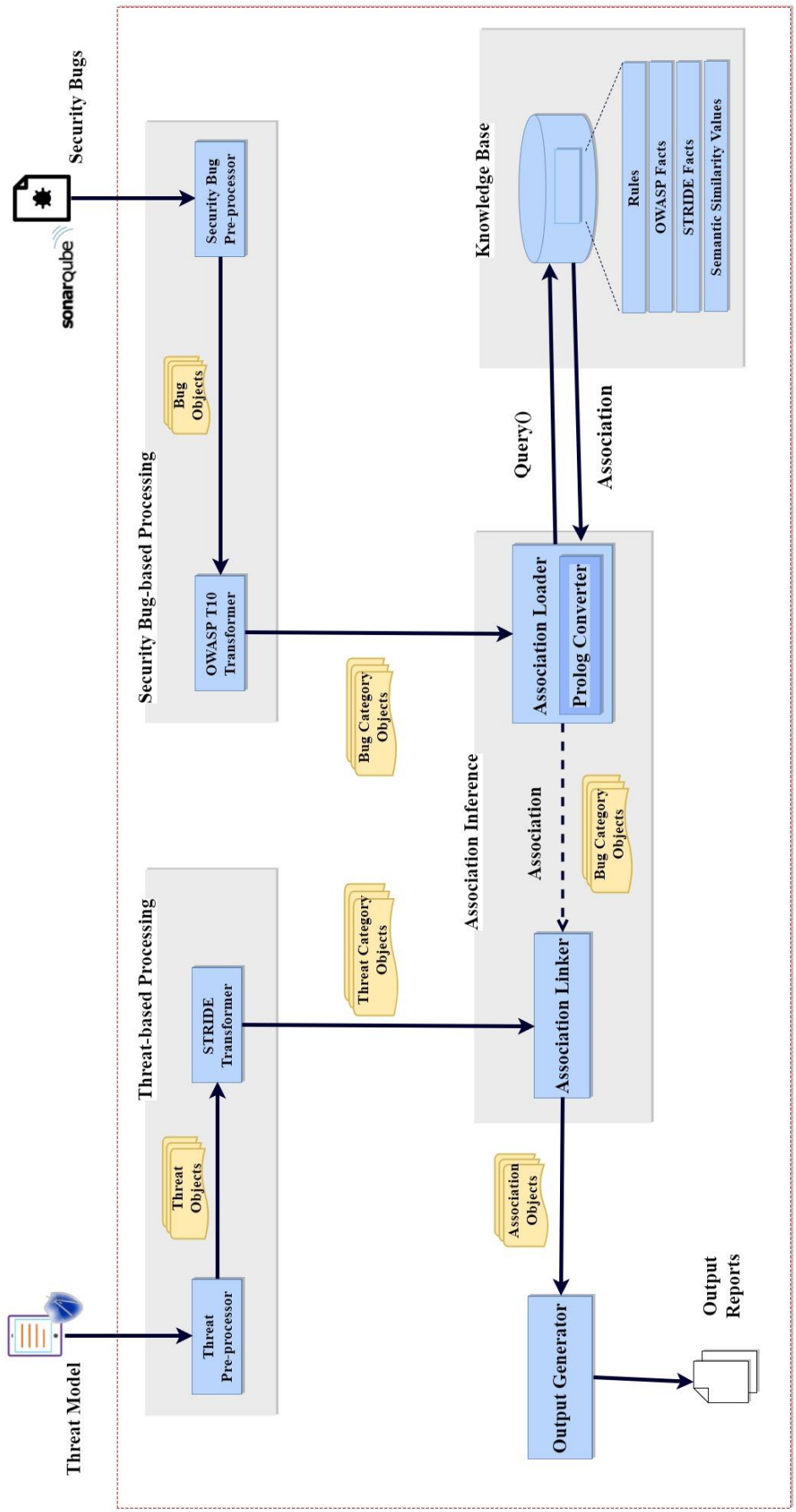


Figure 3.2: Conexus Framework Architecture

3.5.1 Threat-based Processing Module

This module is used to transform the threats contained in the Threat Model (generated from MS TMT) into Threat and Threat Category representations which are used as input to the Association Linker in the Association Inference Module.

3.5.2 Security Bug-based Processing Module

This module is used to transform the security bugs (identified using SonarQube) into Bug and Bug Category representation. This representation is used as input to the Association Loader in the Association Inference Module.

3.5.3 Association Inference Module

The Association Inference Module is used to infer the association between the Bug Category and Threat Category representations. Bug Categories are used to query the knowledge base and get the associated Threat Categories. Consequently, the associations and the Bug Category representations are linked with the Threat Categories. The Association representations are used to create the output using Output Builder.

3.5.4 Knowledge Base

A core part of the Conexus Framework architecture is the Knowledge Base. It contains the facts and rules related to the STRIDE, ASF, OWASP T10, Proactive Controls and Semantic Similarity Scores between ASF and Proactive controls. A Frame-based approach is used for knowledge representation of facts [35, 36]. The structure of the frames for the facts STRIDE, OWASP T10, and Similarity Matching are illustrated by the Figure 3.3: Frame Structure for STRIDE - I, Figure 3.4: Frame Structure for STRIDE - II, Figure 3.5: Frame Structure for OWASP T10 - I, Figure 3.6: Frame Structure for OWASP T10 - II and Figure 3.7: Frame Structure for Similarity Matching. Four additional frames structures were used for the ASF and Proactive controls.

```
frame
    (stride,
        [category_model - [val threat],
         types          - [val
                             [spoofing,
                              tampering,
                              repudiation,
                              information disclosure,
                             denial of service,
                             elevation of privileges]
                           ]
                  ]
    )
```

Figure 3.3: Frame Structure for STRIDE - I

```
frame
    (tampering,
        [ako              - [val stride],
         id               - [val t],
         security_control - [val  [
                                    s2,
                                    s4
                                ]
                           ]
        ]
    )
```

Figure 3.4: Frame Structure for STRIDE - II

```
frame
    (owasp_t10,
          [category_model - [val bug],
           types          - [val
                               [a1,
                                a2,
                                a3,
                                a4,
                               a5,
                               a6,
                               a7,
                               a8,
                               a9,
                               a10 ]
                             ]
                    ]
     )
```

Figure 3.5: Frame Structure for OWASP T10 - I

```
frame
    (a1,
        [ako                - [val owasp_t10],
         name               - [val injection],
         proactive_control - [val  [
                                   c2,
                                   c4
                                ]
                             ]
        ]
     )
```

Figure 3.6: Frame Structure for OWASP T10 - II

```
frame
    (semantic_similarity,
          [proactive_control   - [val c1],
           security_control    - [val s1],
           score               - [val  0.145]
           ]
     )
```

Figure 3.7: Frame Structure for Similarity Matching

In inferring the association between ASF and Proactive controls, the approach was based on computing semantic text similarity between the descriptions of each security control. A security specific semantic text similarity approach [37] was identified in order to be used in this process. The approach was unsuccessful due to the unavailability of a developed application. Hence, the approach was shifted towards a general semantic text similarity service [38].

The semantic text similarity was calculated for every single security control in ASF with every single Proactive Control. The descriptions of ASF security controls and Proactive Controls are not limited to a single phrase. Considering the above fact, the semantic text similarity of each phrase of the description of a particular ASF security control was calculated with respect to each phrase of the description of Proactive Control. Consequently, the average semantic similarity score between a particular ASF security control and Proactive control is calculated as follows.

S1 description of ASF has n phrases as ($s1phr_1$, $s1phr_2$, ..., $s1phr_n$)

C1 description of Proactive Control has m phrases as (c1phr1, c2phr2, ..., $c3phr_m$)

Table 3-3: Semantic Text Similarity scores between an ASF security control and a Proactive control

Semantic Text Similarity		*Description of S1*			
		$s1phr_1$	$s1phr_2$	.....	$s1phr_n$
Description of C1	$c1phr_1$	v_1	v_{m+1}		
	$c1phr_2$	v_2			
	. . . .				
	$c1phr_m$	v_m			v_{nm}

$$Semantic\ Text\ Similarity\ Score\ between\ S1\ and\ C1 = \left[\left(\sum_{i=0}^{nm} Vi\right) \div\ nm\right]$$

The aforementioned approach in calculating semantic similarity score is an automated approach. The calculated semantic text similarity score values are stored as facts in the knowledge base using Frames data structure. In order to infer the association between STRIDE and OWASP T10, Prolog rules were designed according to the aforementioned approach in section 3.4.2 as illustrated below.

Rule 1:

isCausedByThreatCategories(BugCategory , TList_Unique) :-
findall(T , isCausedByThreatCategory(BugCategory , T) , TList) ,
sort(TList, TList_Unique) .

Explanation:

Rule 1 is used to query the knowledge base. The list of unique threat categories can be discovered by querying the knowledge base using a bug category. Each threat category associated with bug category is revealed by the Rule 2. The Prolog built-in function *findall(Object, Goal, List)* is used to collect the threat categories which are identified using Rule 2. The list produced by *findall/3* is filtered to get the unique list by the built-in function *sort(List, SortedList).*

Rule 2:

isCausedByThreatCategory(BugCategory , T) :-
lacksProactive(BugCategory , P) ,
mapsToSecurityControl(P , S) ,
isWeakendByThreatCategory(S , T) .

Explanation:

Rule 2 is used to discover the associated threat category using the bug category. The threat category is revealed using the subsequent rules on the right-hand side of Rule 2. The *lacksProactive(BugCateogry, ProactiveControl)* is used to discover the proactive controls violated due to the given bug category.

The proactive control identified using the Rule 3, *lacksProactive/2* is used in Rule 5, *mapsToSecurityControl(ProactiveControl, SecurityControl)* to determine the security control through the semantic similarity score.

The security control revealed using the Rule 5, mapsToSecurityControl/2 is used to identify the mapping threat category through the Rule 12, *isWeakendByThreatCategory(AsfSecurityControl, ThreatCategory).*

Rule 3:

lacksProactive(BugCategory , C) :-
isProactiveListOf(CList , BugCategory) ,
member(C , CList) .

Rule 4:

isProactiveListOf(CList , BugCategory) :-
owasp_top10(BugCategory , _ , CList) .

Explanation:

Rule 3 is used to identify the proactive controls of the relevant bug categories in succession. The Rule 4, *isProactiveListOf(ProactiveControlList, BugCategory)* used to identify the proactive list of the given bug category using the *owasp_top10* frame.

Rule 5:

mapsToSecurityControl(Proactive , S) :-
isMappingSecurityControlListOf(SList , Proactive) ,
member(S , SList) .

Explanation:

Rule 5 is used to identify the mapping ASF security controls in succession by using the semantic text similarity score between ASF security controls and proactive controls. An ASF security control is mapping with a proactive control if it belongs to the top 03 semantic text similarity scores of the relevant proactive control. The rules Rule 6, Rule 7, Rule 8, Rule 9, Rule 10 and Rule 11 (see Appendix C) are used to identify the mapping security controls using the semantic text similarity scores.

Rule 12:

isWeakendByThreatCategory(SecurityControl , T) :-
stride(_ , T , _ , SecContList) ,
member(SecurityControl , SecContList) .

Explanation:

The ASF security controls discovered using the Rule 5 is used to identify the mapping threat category in succession. The *stride* facts stored in the knowledge base using the Frames data structure is used in Rule 12, *isWeakendByThreatCategory(SecurityControl, ThreatCateogry)* with the built-in *member/2* function in Prolog.

The knowledge base contains all the aforementioned facts and rules to infer the association between OWASP T10 and STRIDE. The component Association Loader in the Association

Inference module use the goal, *isCausedByThreatCategories(bugCategory, ?)* to query the Knowledge Base and results are sent back from the Knowledge Base to the Association Loader.

The facts regarding STRIDE and ASF security controls are static facts while OWASP T10 and Proactive Controls are dynamic facts. The reason for OWASP T10 and Proactive Controls to be dynamic is the revising of OWASP T10 and Proactive controls in a period of years. Hence, the knowledge base has the capability of renewing.

Considering the fact that OWASP T10 or Proactive controls are changed, the Conexus framework has the capability of updating the knowledge base in accordance with the new values. A particular user interface is given in the framework to input the new OWASP T10 details, new Proactive controls details and the updated mapping with respect to the changed data.

3.5.5 Output Builder

The output builder is used to create the association result output. The association results given by the Knowledge Base are used to create the Association representation of Bugs and the Threats representations. In designing the output builder, the Builder Design pattern was applied in order to create different types of report structures.

3.6 Summary

In this chapter, the facts related to the design of the entire framework is described under the major design modules. The design architecture presented in Figure 3.2 explained the design approach used to design the core functionality which is inferring the relationship between threats and security bugs. The identified threats using MS TMT are processed using the Threat-based processing module and security bugs determined though SonarQube is processed using the Security Bug-based processing module. The association is derived using the Association Inference module and the Knowledge Base. The Knowledge Base has the capability of revising according to the OWASP T10 and Proactive Controls. The Output Builder module is used to generate the association report and to export into to CNX, XML and JSON formats.

Chapter 4 : Implementation

4.1 Introduction

This section explains the development approaches taken in the implementation of the proposed framework described in Chapter 3 with the tools and technologies used for the development. A detailed description of the implementation of each component in the Conexus Framework architecture is described under sub-modules with the issues and challenges occurred and the decisions taken during the development process. The reasons for the selection of technologies and tools used in each component is also described in this section.

4.2 Tools and Technologies

The framework was developed as a standalone application using Object oriented concepts. It consists of a Knowledge Base which represents the data model of the application. The Knowledge Base is built using Prolog programming language. The framework was built using Java 8 programming language with Java FX[1] and Maven technology. The main reason behind selecting Java for the development of framework is a rich set of libraries which can be used to communicate with Prolog[2] is available in Java programming language. The additional reasons for selecting Java programming language is, it is a popular object-oriented, robust, secure, and high-performance language used in the industry.

The developed software application Conexus is not a follow-on member of a product family, it depends on outputs from the MS TMT [19] and SonarQube [30]. The interaction of the aforementioned tools with the Conexus Framework to fulfill the core functionality is illustrated by the following Figure 4.1: High-level view of the Conexus Framework.

[1] https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm
[2] http://www.swi-prolog.org/

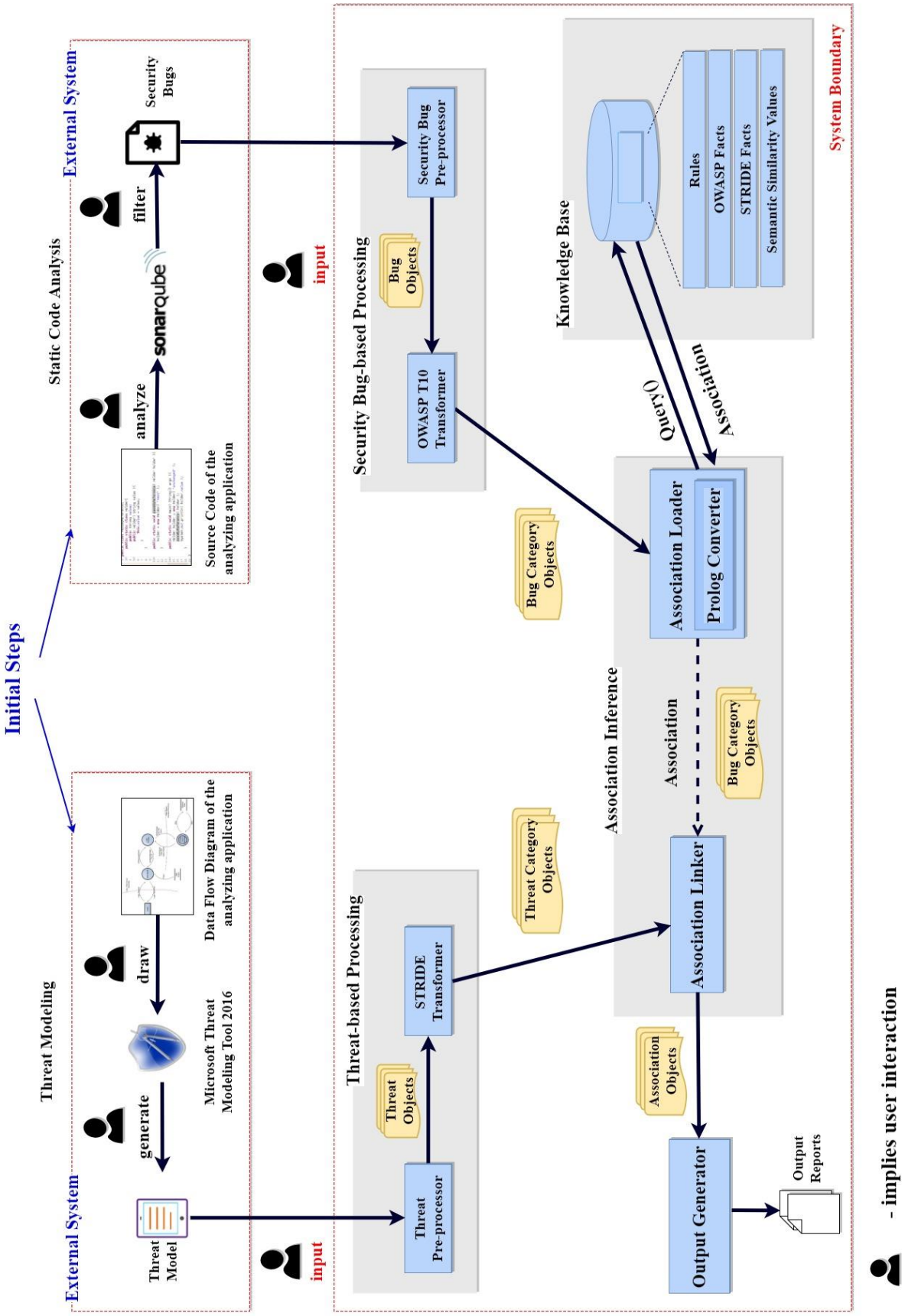


Figure 4.1: High-level view of the Conexus Framework

4.2.1 Threat Modeling Tool: MS TMT 2016

The Conexus Framework user needs to draw the Level-0 or Level-1 DFD of a particular software application to generate the Threat Model. MS TMT 2016 is used for this purpose and it will generate a Threat Model which include Threats categorized with respect to STRIDE [12] threat categorization model. The generated Threat Model of the given application must be input to the system.

The reason for selecting this tool is, it is a well-established and industry accepted tool as depicted in the background study performed in Chapter 2.

4.2.2 Static Code Analysis: SonarQube

In order to identify the Security bugs of a particular software application, the Conexus Framework user needs to analyze the source code using SonarQube. The vulnerabilities identified as security bugs are categorized with respect to OWASP T10 [13] by this tool. Thereafter, the user needs to input the identified security bugs and the relevant OWASP T10 categories into the system.

SonarQube is a static code analysis tool as well as a code quality measuring tool which has been widely used in the software security domain. Categorization of software bugs into OWASP T10 is the additional reason for selecting this tool for the proposed approach of the framework as identified in Chapter 2.

4.2.3 Threat Pre-processor

The Threat Model which is generated from MS TMT is an XML file. This XML file is processed using the dom4j[3] library to extract threats. The reason for using the dom4j library is its powerful navigation with XPath which allows complex navigation throughout the document with a single line of code. Thereafter, the extracted threats are converted into Threat Objects which contain the relevant details of threats sent out from this component.

4.2.4 STRIDE Transformer

The threat objects output from the Threat Pre-processor is transformed into Threat category objects. Six Threat category objects are created with respect to STRIDE threat categories which contains details of each Threat object belongs to a particular category.

[3] https://dom4j.github.io/

4.2.5 Security Bug Pre-processor

The Security bug inputs given into the system by the user are converted into Bug objects using Java FX technology. Each Bug object which contains the relevant details of bugs are sent out as an output from this component.

4.2.6 OWASP T10 Transformer

The Bug objects sent out by the Security Bug Pre-processor are transformed into Bug category objects. Ten Bug category objects are created with respect to OWASP T10 which contains details of each Bug object belongs to a particular category. Theses Bug category objects are sent to the Association Loader component.

4.2.7 Association Loader

Association Loader is used for querying the Knowledge Base using the Bug category objects in order to identify the associated Threat categories. A Prolog converter is built using JPL[4] library in SWI-Prolog to communicate Prolog with Java. The reason for selecting JPL is, it is a well-known and mature interface between Java and Prolog [39].

Each bug category will be used to query the Knowledge Base and the associated threat type results are held inside the Association Loader. The associated threat type results and the Bug Category Objects are sent to the Association Linker.

4.2.8 Knowledge Base

The Knowledge Base is built using the SWI-Prolog. All the facts and rules aforementioned in Chapter 3, are contained in the Knowledge Base. The reason behind the choice of SWI Prolog is based on popularity in the community, freely available and ultimately it works fine with JPL library.

The reason for selecting a knowledge base approach instead of a database approach is the complexity of handling constraints in a database in a complete update. The Knowledge Base has the capability of revising when the OWASP T10 or Proactive controls are revised. Hence, using a database will be a complex approach rather than using a Knowledge Base. The additional reason for selecting a Knowledge Base approach is the capability to expand the knowledge contained in the Knowledge Base using the security expert knowledge.

[4] http://www.swi-prolog.org/packages/jpl/java_api/index.html

4.2.9 Association Linker

Threat category objects from STRIDE Transformer and associated threat types and Bug objects from Association Loader will be the input to this component. Thereafter, the Association objects are generated. The Association objects are sent out from the Association Linker to the Output Builder.

4.2.10 Output Builder

The output builder is responsible for creating the Association report. This has the capability to export the report in CNX format (supported by the Conexus Framework), XML format and JSON format with the use of Builder Design pattern. FasterXML/Jackson[5] is used to create the report in XML or JSON format since it supports annotation based serialization and deserialization capability.

4.3 Summary

This chapter presented the implementation procedure of the entire system with the tools and technologies used in each submodule. The implementation of the design modules in Conexus Framework architecture explained in Chapter 3 were technically presented as an integration of them with the reasons behind in selecting relevant tools and technologies. The system workflow with respect to the core modules was also presented using the diagram in Figure 4.1. Implementation of the Conexus Framework followed the Object Oriented principles and the coding standards.

[5] https://github.com/FasterXML/jackson

Chapter 5 : Testing and Evaluation

5.1 Introduction

The evaluation of the proposed approach to achieve the goal of this dissertation is a challenging task. There are no standard evaluation measures for instance precision and recall (used in information retrieval) or accuracy measures (used in machine learning). Hence, an appropriate evaluation methodology was identified and a detailed description of the procedure is described in the sub-section 5.3. The evaluation was conducted using two case studies and the discussion of the evaluation results are further elaborated in section 5.4 and 5.5. In prior to the evaluation process, testing was conducted to identify whether the framework behaves according to the intended functionalities. The testing procedure followed is described in the sub-module 5.2.

5.2 Test Procedure

The testing procedure was conducted as a strategy to ensure that the product operates as intended in the specification. It can be realized under two main categories namely, functional testing and non-functional testing. Functional testing includes unit testing, integration testing, and system testing to verify that the implemented framework functions correctly and provides the results in accordance with the development constraints. Unit testing was performed using the TestNG[6] tool.

Performance testing was conducted under non-functional testing and the framework was tested for analysis of large-scale projects to check whether the system crashes or fails to produce expected outputs. The industry project used for the evaluation purpose in the section 5.3 was selected for this purpose. Due to the fact that the Conexus framework only processes the outputs generated from MS TMT and SonarQube, it is independent of the Threat Modeling processing and the Static Code Analysis. Hence, generating the association for large-scale projects can be conducted from the framework irrespective of the size of the project.

5.3 Evaluation Procedure

Concerning the goal of this dissertation, the identified evaluation methodology is focused on evaluating the framework for the concept of secure design will lead to a secure software system and whether the potential root causes of an identified security bug lie in the design phase of the software application as described in Chapter 1. A software application to be analyzed using the

[6] http://testng.org/doc/

Conexus Framework should include a Level-0 or Level-1 DFD of the application along with the source code. It was unable to find an open source project with a data flow diagram since most of the development groups does not follow a Secure SDLC. Hence, open source projects were not used for the evaluation purpose. In accordance with the aforementioned facts, a user authentication component of a web-based application and an industry project were chosen for the evaluation purpose as case studies 1 and 2 respectively. Mutation testing was carried out to the source code of the case study 1 in prior to the evaluation where OWASP T10 related security bugs were purposely introduced to the source code. The reason behind that fact was to identify whether the intended evaluation objectives of this dissertation is fulfilled.

In the evaluation process, threat modeling is conducted to identify the architectural-level security flaws of the selected case study as STRIDE categorized threats using MS TMT. The security bugs in the source code of the selected case study are identified from static code analysis using SonarQube as OWASP T10 related vulnerabilities. The identified threats and security bugs are given as inputs to the framework to derive the association. The association derived between security bugs and the threats is limited to the level of the security bug categories and threat categories. An association was derived for each bug category identified using the static code analysis process through the Conexus Framework. The association depicts that the derived threat categories can be the root causes of the encountered security bugs. Using a manual process, the highly relevant root causes (threat categories) of the security bug categories were identified and the relevant countermeasures were applied in order to remove the security bugs in the source code.

The countermeasures for a particular threat category are identified from the ASF security controls according to the aforementioned solution approach in section 3.4.2. The risks (identified as threats) in the design phase can be mitigated by implementing the countermeasures in the implementation-level of a case study (See Appendix D). The reason for this fact is the countermeasures derived by the ASF security controls are given for the implementation level. Hence, the countermeasures shown for the relevant threat categories are enforced in the source code. This fact leads to perform only the static code analysis for the modified source code to identify whether the previously encountered security bugs had been removed from the source code. The evaluation of the Conexus Framework will be succeeded if the security bugs were removed by the application of countermeasures for the relevant root causes.

5.4 Evaluation Results

5.4.1 Case Study 1: User Authentication component of a Web-Based Application

The data flow diagram drawn for the developed user authentication component of the web-based application is represented by the Figure 5.1: DFD of the User Authentication Component of the Web-Based Application.

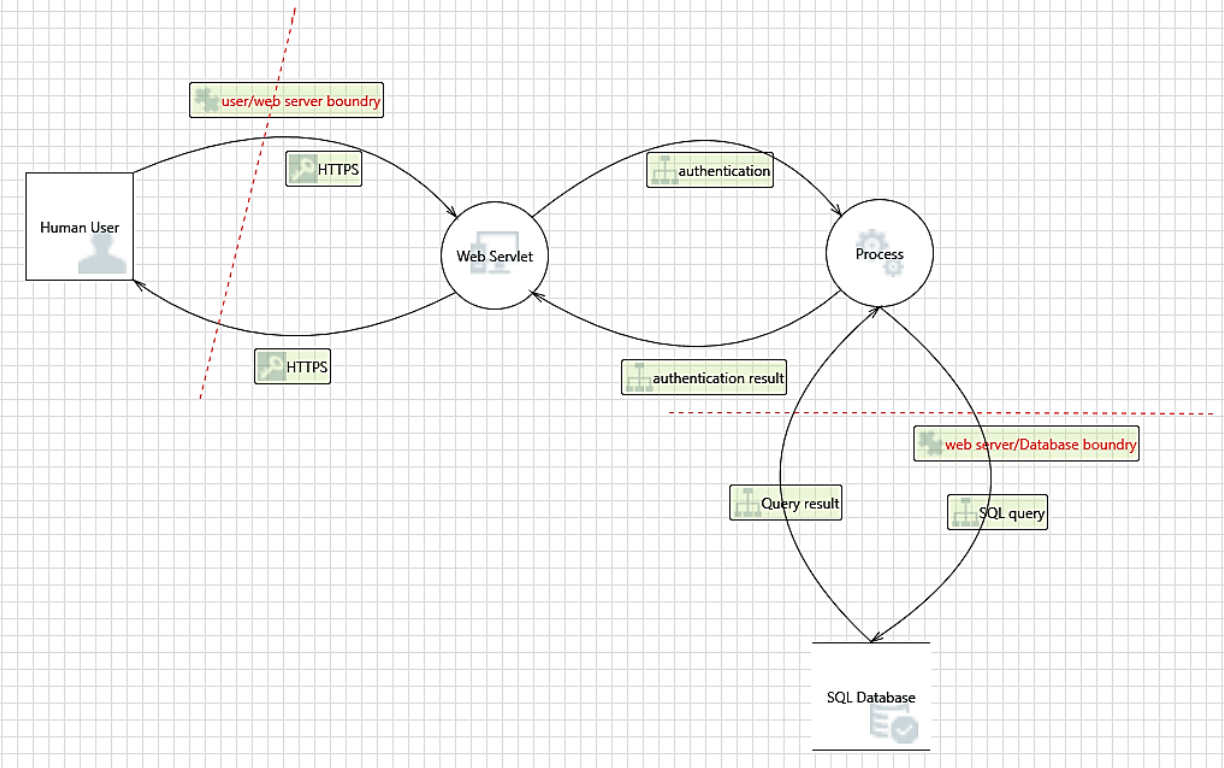


Figure 5.1: DFD of the User Authentication Component of the Web-Based Application

Threat modeling process was carried using MS TMT 2016 to identify the architecture-level security flaws in the design. A summary of the threats identified for each STRIDE threat category is described by the Table 5-1: Summary of threats identified in the User Authentication Component of the Web-Based Application.

Static code analysis was performed using the SonarQube and the summary of the results obtained is illustrated in Table 5-2: Summary of the Security-bugs identified in the User Authentication Component of the Web-Based Application.

Table 5-1: Summary of threats identified in the User Authentication Component of the Web-Based Application

STRIDE Threat Type	**No. of Threats**
Spoofing	6
Tampering	4
Repudiation	4
Information Disclosure	2
Denial of service	9
Elevation of privileges	8

Table 5-2: Summary of the Security-bugs identified in the User Authentication Component of the Web-Based Application

	OWASP T10 related vulnerability types caught by SonarQube	**OWASP T10 related vulnerability types uncaught by SonarQube**
Count	4	2
Types	A2, A5, A6	A1, A7

Thereafter, the results from the threat modeling process and the static code analysis were given as input to the Conexus framework to find the association. The derived association is illustrated in the following Figure 5.2, Figure 5.3 and rest of the association is included in Appendix F.

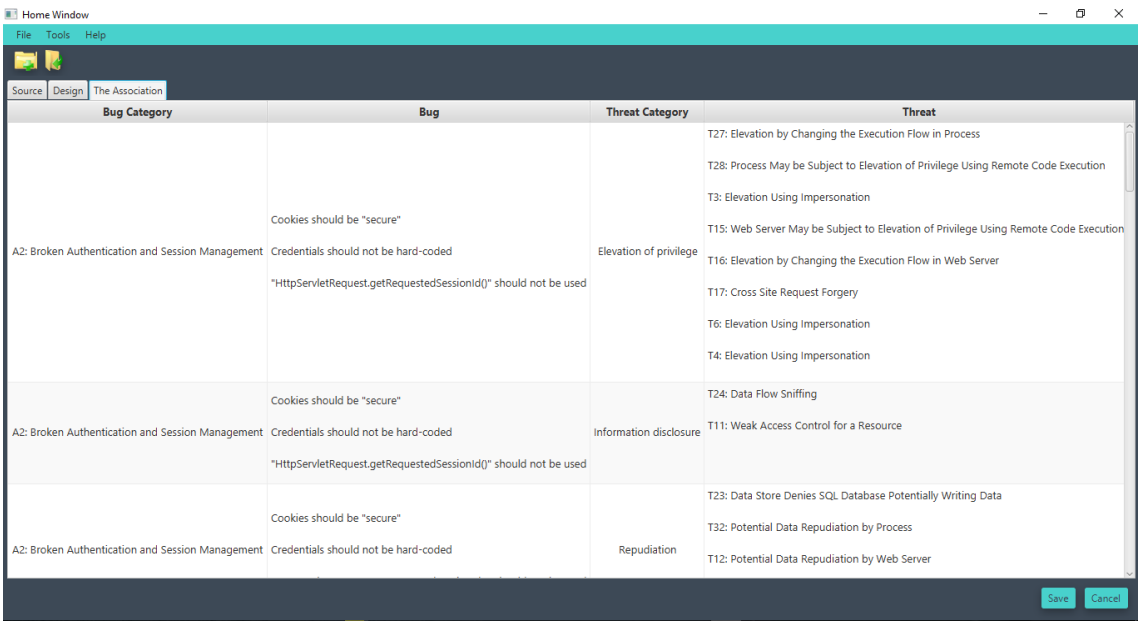


Figure 5.2: Part I - Association derived for the User authentication component using the Conexus Framework

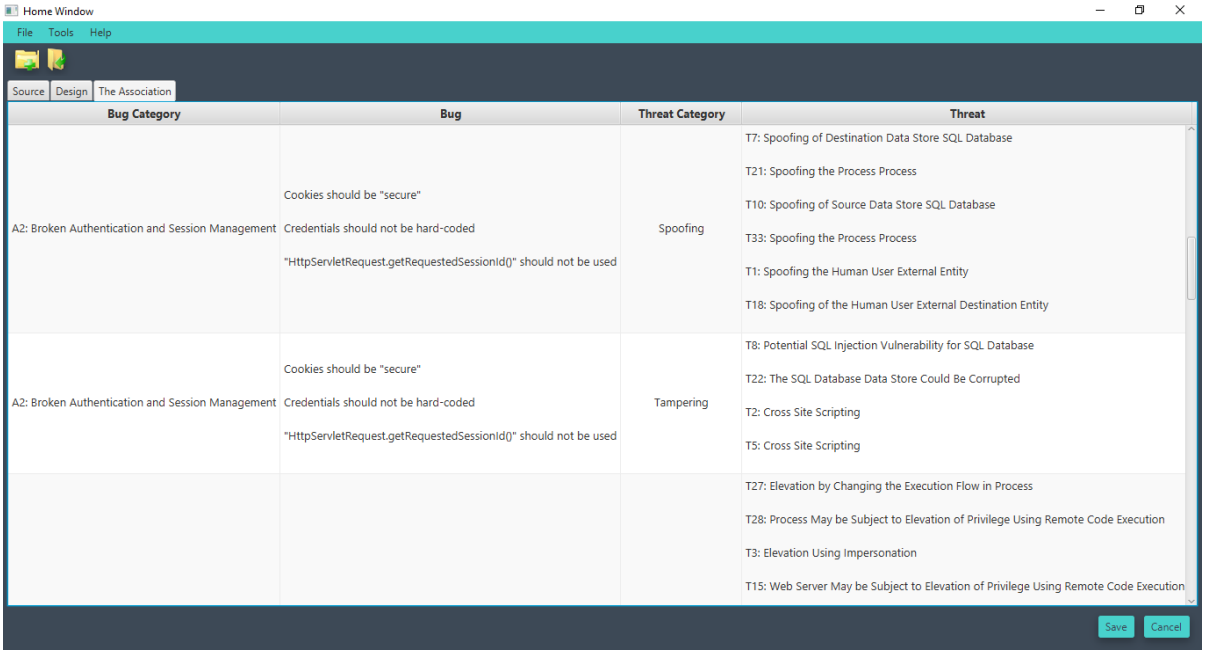


Figure 5.3: Part II - Association derived for the User authentication component using the Conexus Framework

According to the association derived and the security bug types, it depicts that the identified security bugs have a high relevancy towards the Spoofing and Tampering threat categories. The relevant countermeasures given for the Spoofing and Tampering attacks were applied to the implementation of the project and the static code analysis process was repeated. The previously identified security bugs (belongs to A2, A5, A6 OWASP T10 categories) by the SonarQube have been removed. In addition to that, SQL injection (belongs to A1 OWASP T10 category) which was not identified by the SonarQube has also been removed. It has been identified that the security bug related to access levels in this source code (A7) which has not been identified by the SonarQube can be removed by enforcing the countermeasures given for Elevation of privileges.

5.4.2 Case Study 2: Large-scale Web-Based Application

This is a large scale case study taken from the industry. The data flow diagram related to this case study is attached in the Appendix D. The summary of identified threats for this case study using threat modeling process is illustrated in Table 5-3: Summary of the threats identified in the Large-scale Application.

Static code analysis was performed on the case study and 26 security bugs related to OWASP T10 A2 and A6 categories were identified. However, the security bugs identified with respect to OWASP T10 A2 category were not real security bugs. The reason lies behind this fact is further explained in the section 5.5.

Table 5-3: Summary of the threats identified in the Large-scale Application

STRIDE Threat Type	**No. of Threats**
Spoofing	12
Tampering	0
Repudiation	0
Information Disclosure	5
Denial of service	5
Elevation of privileges	5

The identified threats and the security bugs were given as input into the Conexus Framework and the association was derived. According to the derived association, it depicts that there is high relevance towards the Information Disclosure threat category. By applying the countermeasures given for the Information Disclosure in the implementation, the security bugs (belongs to A6 OWASP T10 category) were removed.

5.5 Discussion on the Evaluation Results

The association results obtained from the Conexus framework depend on the analysis outputs given from SonarQube and MS TMT and the Semantic similarity scores obtained through UMBC Semantic Similarity Service. The accuracy of the results obtained from the Conexus Framework relies on the aforementioned tools and services. During the evaluation process, following issued were identified with respect to SonarQube.

SonarQube only supports 06 vulnerability types from OWASP T10 for the Java language as illustrated in Table 2-4 in Chapter 2. In spite of the fact that SonarQube supports for A1 vulnerability type of OWASP T10, the purposely introduced A1 type security bug for the source code in the Case Study 1 was unable to identify from the static code analysis. This is an issue in the SonarQube tool. Another issue identified from SonarQube was the incorrect detection of hard-coded passwords. During the analysis of the Case Study 2, a string variable name included the word phrase “password” was identified as a variable to store a hardcoded password. This was detected as an A2 type vulnerability of OWASP T10 by SonarQube.

Despite the fact that, if the source code of analyzing project contains OWASP T10 vulnerabilities, it is unable to detect all vulnerability types by SonarQube. Hence, Conexus Framework is unable to derive associations for each OWASP T10 vulnerability contained in the source code. Therefore a manual code review conducted to identify the rest of the vulnerability types of OWASP T10 in the Case Study 1. The manual process was succeeded since the vulnerabilities were introduced

purposely to the source code and the scale of this Case Study 1 was small. However, the manual code review was not successful for the Case Study 2 due to the scale and the complexity of this project.

The conducted analysis for the Case Study 2 resulted with a set of limitations in finding the association for large-scale projects. The association is derived from the Conexus Framework is used to find the possible root causes for a security bug. The possible root causes are the threats in the design. Although the association is found for the project, it is difficult to pinpoint the exact location of the source code even in manually to apply the countermeasures given for the possible threat categories. This is due to the fact that the generated DFDs are Level-0 or Level-1 diagrams. The information provided by these type of DFDs are not sufficient enough to find the relevant implementation of the source code. It has been identified from this evaluation that these type of issues can be prevented if the large-scale projects are divided into a set of small components and perform the analysis process for each component. The DFDs can be drawn for each component of the large-scale project. A component level analysis can be performed using the relevant implementation and the DFD of the component. This kind of analysis narrows down the identification of root causes to relevant data flow or process manually using the association derived.

The OWASP T10 related security bugs identified in the 02 case studies were removed by the application of countermeasures for the relevant root cause(s) (threat category) of the security bug category. The summary of the derived association and the relevant association (correct association) for the security bug categories in the 02 case studies is represented by the following Table 5-4: Summary of the Derived Association Results from the Conexus Framework and the Relevant Association results to remove the Security Bugs.

Table 5-4: Summary of the Derived Association Results from the Conexus Framework and the Relevant Association results to remove the Security Bugs

Case Study	Bug Category	Derived association from the Conexus Framework	Relevant association to remove the security bug
Case Study 1			
	A1	T, R, I, E	T
	A2	S, T, R, I, E	S, T
	A5	T, R, I, E	T
	A6	S, T, R, I, E	S, T
	A7	T, R, I, E	E
Case Study 2			
	A6	S, T, R, I, E	I

The probability of identifying correct association for a particular bug category is calculated as no. of correctly identified associations over the total no. of associations derived.

If C represents the identifying correct association for a bug category,

$$P(C) = \frac{no.of\ correctly\ identified\ associations}{total\ no.of\ identified\ associations}$$

Accuracy measures the degree of the system to identify associations correctly. Hence, the overall accuracy of the derived associations for the 02 case studies is calculated using the results in Table 5-4: Summary of the Derived Association Results from the Conexus Framework and the Relevant Association results to remove the Security Bugs.

If $C_1, C_2, \ldots, C_n$ represents the identification of correct associations for each result in Table 5-4,

$$Overall\ Accuracy = \frac{\sum_{i=1}^{n} \mathrm{P(Cn)}}{n} \times 100$$

$$Overall\ Accuracy = \left(\frac{\frac{1}{4} + \frac{2}{5} + \frac{1}{4} + \frac{2}{5} + \frac{1}{4} + \frac{1}{5}}{6}\right) \times 100 = 29.167$$

The overall accuracy is lower due to the noise included in deriving the association as shown in Table 5-4. However, for the above case studies, the relevant root causes (threats) were able to identify using the Conexus Framework. The derived association conveys that each threat category (STRIDE) is not associated with each security bug category (OWASP T10).

Chapter 6 : Conclusion

The particular area of a study conducted in this dissertation is aimed at finding an association between security bugs occur in the source code and architecture-level security flaws which can be found from software design artifacts by proposing a knowledge modeling based approach. The association is derived in accordance with finding the possible root causes (architecture-level security flaws) for a given security bug. A framework is built as a proof of concept for the proposed solution and it can be used in an SDLC as an aid for software developers. Static code analysis which is an approach to find security bugs in a source code is considered for this study and the results obtained from a static code analysis tool (SonarQube) are given as inputs to the framework. Correspondingly, Threat modeling which is an architectural risk analysis methodology to find security flaws in a software design is considered and the results obtained from a threat modeling tool (MS TMT 2016) are given as inputs to the framework.

The proposed approach to identify the association between security bugs and architecture-level security flaws is limited to the level of the security bug categories and threat categories. The results generated from threat modeling and static code analysis approaches are security flaws categorized into STRIDE and security bugs categorized into OWASP T10 respectively. STRIDE is identified as an attacker's view of the enumeration of threats and a specific security control is violated by each threat type. ASF is a security control categorization mechanism which supports in identifying threats as weaknesses in security controls by representing the defensive perspective of threats. A mapping between STRIDE and ASF is used to identify the security controls violated by each STRIDE threat type. Correspondingly, the security controls violated by each OWASP T10 vulnerabilities are represented by OWASP T10 Proactive Controls. The derivation of the association between STRIDE and OWASP T10 is created using a semantic text similarity matching model (UMBC Semantic Similarity Service). The semantic similarity has been generated by using the set of countermeasures given in ASF and summarized OWASP T10 Proactive control descriptions.

The results obtained by the evaluation process conveys that the derived association has an overall accuracy of 29.167 percent. However, the accuracy is relatively low due to noise, the derived association contained the relevant root cause(s) for a particular security bug. The given mapping between OWASP T10 and Proactive controls, maps all the OWASP T10 vulnerability types with proactive controls C8, C9 and C10. It has been identified from the evaluation that this mapping has resulted in repetition of the same set of STRIDE categories linked to each OWASP T10

vulnerability type as noise. The identified security bugs related to OWASP T10 vulnerability types were able to resolve by applying the countermeasures given for the relevant root cause of the security bugs using the derived association. The conducted evaluation leads to the justification that the framework evaluates the concept of secure design will lead to a secure software system. This dissertation concludes that some of the security bugs in the code level were caused due to the architecture-level security flaws.

6.1 Conexus Framework Applications

The framework consists of a knowledge base developed upon a set of Prolog rules and facts. In spite of finding the association between security flaws and security bugs based on STRIDE and OWASP T10 respectively, the framework provides countermeasures for the security flaws and prevention techniques for the security bugs given as inputs to the system.

The framework can be used conjointly with the re-engineering process of a previously developed software where the association between security flaws and security bugs can be identified and take necessary actions to implement security in the software. Implementation of a software application sometimes includes the use of legacy software components. Thus, the Conexus framework can be used to ensure the security of the legacy components which are a lack of software security.

In a security-focused agile development environment, the framework can be used to ensure the security of the working software in each product increment. The particular area of study proposed in this dissertation is exposed to a broad research space. The framework can be used as an aid for researchers in the security domain by enhancing and updating the knowledge base in consideration with different security aspects.

6.2 Future Work

The current approach of finding the association between STRIDE and OWASP T10 which has been proposed from this dissertation is based on semantic similarity values obtained from ASF countermeasures and OWASP T10 Proactive controls. In consideration of that fact, a different approach to obtain the association can be commenced as a future work. A suggested approach for this work is to use a security specific semantic similarity matching model which will enhance the accuracy of the similarity values obtained. Another approach is using attack trees to find the association between STRIDE to OWASP T10 in place of obtaining semantic similarities.

The proposed approach for the finding the association in this dissertation is limited to the level of security bug categories and threat categories. This approach can be further enhanced to map each

security bug to each threat as a future work. Use of a case-based reasoning model will be one possible approach for that.

It has been identified from the derived association that some of the identified threats in the design were not related to the implementation level problems. Hence, to ensure that each threat is mitigated in the final software product, security-specific test cases can be generated from the identified threats. This can be a further extension of this area of study.

References

[1] S. L. M. Howard, *The Security Development Lifecycle*, Redmond, WA: Microsoft Press, 2006.

[2] G. McGraw, *Software Security: Building Security In*, Upper Saddle River, NJ: Addison-Wesley, 2006.

[3] R. Kissel, *Glossary of Key Information Security Terms*, United States: National Institute of Standards and Technology (NIST), 2013.

[4] I. Arce, K. Clark-Fisher, N. Daswani, J. DelGrosso, D. Dhillon, C. Kern, T. Kohno, C. Landwehr, G. McGraw, B. Schoenfield, M. Seltzer, D. Spinellis, I. Tarandach and J. West, "Avoiding the top 10 software security design flaws," IEEE Computer Society Center for Secure Design (CSD), 2014.

[5] L. R. Vanciu, "Static Extraction of Dataflow Communication for Security," Ph.D. dissertation, Wayne State University, Michigan, 2014.

[6] R. Kazman, "A tool to address cybersecurity vulnerabilities through design," SEI Blog, Software Engineering Institute, Carnegie Mellon University, 29 February 2016. [Online]. Available: https://insights.sei.cmu.edu/sei_blog/2016/02/a-tool-to-address-cybersecurity-vulnerabilities-through-design.html. [Accessed 12 December 2017].

[7] R. S. Mark Sherman, "From Secure Coding to Secure Software," SEI Webinar Series, Software Engineering Institute, Carnegie Mellon University, 10 November 2016. [Online]. Available: https://www.sei.cmu.edu/webinars/view_webinar.cfm?webinarid=483646. [Accessed 12 December 2017].

[8] International Organization for Standardization, "ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models".

[9] CERT.org, "Cybersecurity Engineering," [Online]. Available: https://www.cert.org/cybersecurity-engineering/index.cfm#swamodel. [Accessed 12 December 2017].

[10] S. Lipner, "The trustworthy computing security development lifecycle," in *Computer Security Applications Conference, 20th Annual*, 2004.

[11] Microsoft, "Memo from Bill Gates | Stories," 2012. [Online]. Available: https://news.microsoft.com/2012/01/11/memo-from-bill-gates/. [Accessed 12 December 2017].

[12] Microsoft, "The STRIDE Threat Model," [Online]. Available: https://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx. [Accessed 12 December 2017].

[13] OWASP.org, "OWASP Top Ten Project," [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project. [Accessed 12 December 2017].

[14] G. McGraw, "Software security touchpoint: Architectural risk analysis," Technical report, Cigital, 2009.

[15] OWASP.org, "Application Threat Modeling," [Online]. Available: https://www.owasp.org/index.php/Application_Threat_Modeling. [Accessed 18 December 2017].

[16] M. Abi-Antoun, D. Wang and P. Torr, "Checking threat modeling data flow diagrams for implementation conformance and security," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ACM, 2007.

[17] M. Frydman, G. Ruiz, E. Heymann, E. César and B. Miller, "Automating risk analysis of software design models.," *The Scientific World Journal*, 2014.

[18] X. Yuan, E. Nuakoh, J. Beal and H. Yu, "Retrieving relevant CAPEC attack patterns for secure software development," in *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, 2014.

[19] B. Berger, K. Sohr and R. Koschke, "Automatically Extracting Threats from Extended Data Flow Diagrams," in *International Symposium on Engineering Secure Software and Systems*.

[20] Microsoft, "SDL Threat Modeling Tool," [Online]. Available: https://www.microsoft.com/en-us/sdl/adopt/threatmodeling.aspx. [Accessed 12 December 2017].

[21] The ThreatModeler, "Threat Modeling Tool," [Online]. Available: http://threatmodeler.com/threat-modeling-tool/. [Accessed 12 December 2017].

[22] I. Williams and X. Yuan, "Evaluating the effectiveness of Microsoft threat modeling tool," in *Proceedings of the 2015 Information Security Curriculum Development Conference*, 2015.

[23] H. Assal, "Collaborative security code review," in *Proceedings of the 14th International Conference on Mobile and Ubiquitous Multimedia*, 2015.

[24] T. Ishikawa and K. Sakurai, "Parameter manipulation attack prevention and detection by using web application deception proxy," in *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*, 2017.

[25] C. Heitzenrater, R. Böhme and A. Simpson, "The days before zero day: Investment models for secure software engineering," in *Proceedings of the 15th Workshop on the Economics of Information Security (WEIS)*, 2016.

[26] N. Kaur and P. Kaur, "Mitigation of SQL injection attacks using threat modeling," *ACM SIGSOFT Software Engineering Notes, 39(6),* pp. 1-6, 2014.

[27] OWASP.org, "OWASP Proactive Controls," [Online]. Available: https://www.owasp.org/index.php/OWASP_Proactive_Controls. [Accessed 18 December 2017].

[28] Coverity, "Coverity White Paper: Building Security into Your Software Development Lifecycle," 2012.

[29] S. Alqahtani, E. Eghan and J. Rilling, "SV-AF—A Security Vulnerability Analysis Framework," in *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*, 2016.

[30] Find-sec-bugs.github.io, "Find Security Bugs," [Online]. Available: http://find-sec-bugs.github.io/. [Accessed 12 December 2017].

[31] sonarqube.org, "SonarQube," [Online]. Available: https://www.sonarqube.org/. [Accessed 12 December 2017].

[32] H. Kaur, "Automating Static Code Analysis for Risk Assessment and Quality Assurance of Medical Record Software," M.Sc. dissertation, University of Victoria, 2017.

[33] T. UcedaVelez and M. Morana, *Risk Centric Threat Modeling: Process for Attack Simulation and Threat Analysis*, John Wiley & Sons, 2015.

[34] A. Shostack, *Threat Modeling: Designing for Security*, Indianapolis, Indiana: John Wiley & Sons, 2014.

[35] D. Merritt, *Building expert systems in Prolog*, Springer Science & Business Media, 2012.

[36] P. Rashid, "Semantic Network and Frame Knowledge Representation Formalisms in Artificial Intelligence," M.Sc. dissertation, Eastern Mediterranean University (EMU)-Doğu Akdeniz Üniversitesi (DAÜ), 2015.

[37] M. Chavan, "Developing a Cybersecurity Text Corpus and its Application for Augmenting Semantic Text Similarity," M.Sc. dissertation, University of Maryland, Baltimore County, 2014.

[38] L. Han, A. Kashyap, T. Finin, J. Mayfield and J. Weese, "UMBC_EBIQUITY-CORE: Semantic Textual Similarity Systems," * SEM@ NAACL-HLT, 2013.

[39] L. Ostermayer, F. Flederer and D. Seipel, "CAPJA-A Connector Architecture for Prolog and Java," in *10th Workshop on Knowledge Engineering and Software Engineering (KESE)*, 2014.

Appendices

Appendix A : Individual Contribution

In accordance to the problem definition in this dissertation, all the three team members were equally contributed in finding an approach to achieve the proposed goal. Following the identification of an approach, a knowledge base has to be created using a set of Prolog rules to store the relevant information to be used with the implemented framework. Creation of this set of rules included the contribution of all the members.

Contribution 1: K. A. I. Abeyrathna

A comparison of the existing threat modeling tools was conducted in order to identify the appropriate threat modeling tool to be used with the identified approach. The work related to generating the output reports from the selected threat modeling tool was carried out. The implementation of the Conexus framework related to processing the generated output from the threat modeling tool and output builder implementation was conducted.

Contribution 2: C. S. Samarage

A comparison of the existing static code analysis tools was conducted to select the appropriate static code analysis tool to be used with the identified approach. The work related to generating the output reports from the selected static code analysis tool was carried out. The implementation of the Conexus framework related to processing the generated output from the static code analysis tool and settings implementation was conducted.

Contribution 3: B. N. Dahanayake

A compatible prolog implementation and relevant libraries to be used with the Conexus framework was identified. The most appropriate data structure to store the information in the knowledge base was identified. The implementation of the Conexus framework related in Java to Prolog conversions with respect to the information retrieving and storing and association inference implementation were conducted.

Appendix B : Terminology

Bug	-	An unexpected and relatively small defect, fault, flaw, or imperfection in an information system or device.
Build Security In	-	A set of principles, practices, and tools to design, develop, and evolve information systems and software that enhance resistance to vulnerabilities, flaws, and attacks.
Defect	-	A problem that may lie dormant in software for years only to surface in a fielded system with major consequences.
Exploit	-	A script or plan that executes against a vulnerability, leading to a security compromise.
Risk	-	The potential for an unwanted or adverse outcome resulting from an incident, event, or occurrence, as determined by the likelihood that a particular threat will exploit a particular vulnerability, with the associated consequences.
Flaw	-	A design-level or architecture-level software defect.
Software Security	-	The idea of engineering software so that it continues to function correctly under malicious attacks.
Threat	-	A circumstance or event that has or indicates the potential to exploit vulnerabilities and to adversely impact organizational operations, organizational assets, individuals, other organizations, or society.
Touchpoint	-	A characteristic or specific weakness that renders an organization or asset open to exploitation by a given threat or susceptible to a given hazard.
Vulnerability	-	A characteristic or specific weakness that renders an organization or asset (such as information or an information system) open to exploitation by a given threat or susceptible to a given hazard.

Appendix C : Rules in the implemented Knowledge Base

The 12 rules in the implemented knowledge base are explained here.

Rule 1:

isCausedByThreatCategories(BugCategory , TList_Unique) :-
findall(T , isCausedByThreatCategory(BugCategory , T) , TList) ,
sort(TList, TList_Unique) .

Explanation:

Rule 1 is used to query the knowledge base. The list of unique threat categories can be discovered by querying the knowledge base using a bug category. Each threat category associated with bug category is revealed by the Rule 2. The Prolog built-in function *findall(Object, Goal, List)* is used to collect the threat categories which are identified using Rule 2. The list produced by *findall/3* is filtered to get the unique list by the built-in function *sort(List, SortedList)*.

Rule 2:

isCausedByThreatCategory(BugCategory , T) :-
lacksProactive(BugCategory , P) ,
mapsToSecurityControl(P , S) ,
isWeakendByThreatCategory(S , T) .

Explanation:

Rule 2 is used to discover the associated threat category using the bug category. The threat category is revealed using the subsequent rules on the right-hand side of Rule 2. The *lacksProactive(BugCateogry, ProactiveControl)* is used to discover the proactive controls violated due to the given bug category.

The proactive control identified using the Rule 3, *lacksProactive/2* is used in Rule 5, *mapsToSecurityControl(ProactiveControl, SecurityControl)* to determine the security control through the semantic similarity score.

The security control revealed using the Rule 5, mapsToSecurityControl/2 is used to identify the mapping threat category through the Rule 12, *isWeakendByThreatCategory(AsfSecurityControl, ThreatCategory)*.

Rule 3:

lacksProactive(BugCategory , C) :-
isProactiveListOf(CList , BugCategory) ,
member(C , CList) .

Rule 4:

isProactiveListOf(CList , BugCategory) :-
owasp_top10(BugCategory , _ , CList) .

Explanation:

Rule 3 is used to identify the proactive controls of the relevant bug categories in succession. The Rule 4, *isProactiveListOf(ProactiveControlList, BugCategory)* used to identify the proactive list of the given bug category using the *owasp_top10* frame.

Rule 5:

mapsToSecurityControl(Proactive , S) :-
isMappingSecurityControlListOf(SList , Proactive) ,
member(S , SList) .

Explanation:

The *mapsToSecurityControl/2* is used to identify the ASF security controls maps to the given proactive control in succession. In order to identify the list of mapping ASF security controls Rule 6, i*sMappingSecurityControlListOf(ASFSecurityControlList , ProactiveControl) is used.* The built-in function *member/2* of Prolog is used to iterate through the ASF security control list.

Rule 6:

isMappingSecurityControlListOf(SList , Proactive) :-
isMappingSimilarityValuesFor(SimList , Proactive) ,
isSecurityControlListOf(SList , Proactive , SimList) .

Rule 7:

isMappingSimilarityValuesFor(SimList , Proactive) :-
isMaxThreeOf(SimList , AllSimilarities) ,
isAllSimilarityValuesFor(AllSimilarities , Proactive) .

Explanation:

The Rule 7, *isMappingSimilarityValuesFor(SimilarityScoreList, ProactiveControl)* used in RHS of Rule 6 is used to get the list of maximum three semantic text similarity scores for a given proactive control from the full list of semantic scores.

Rule 8:

isMaxThreeOf([X , Y , Z] , [X , Y , Z | _]) .

Explanation:

The Rule 8, *isMaxThreeOf(MaxThreeList, SortedList)* is used to filter the maximum three values from a given sorted list in descending order.

Rule 9:

isAllSimilarityValuesFor(AllSimList_DESC, Proactive) :-
findall(Similarity ,
semanticAssociation(Proactive , _ , Similarity) , AllSimList) ,
sort(AllSimList , AllSimList_ASC) ,
reverse(AllSimList_ASC , AllSimList_DESC) .

Explanation:

Rule 9 is used to get the list of semantic text similarity scores for a given proactive control in descending order using the *semanticAssociation* frame. The findAll/3, sort/2, and reverse/2 are Prolog built-in functions used to collect all the semantic scores relevant to the given proactive control, sort the list of semantic scores in ascending order and reverse the order of semantic scores respectively.

Rule 10:

isSecurityControlListOf(SList, Proactive, SimList) :-
findall(S ,
isASecurityControlOf(S , Proactive , SimList) ,
SList) .

Rule 11:

isASecurityControlOf(S , Proactive , SimList) :-
semanticAssociation(Proactive , S , Similarity) ,
member(Similarity , SimList) .

Explanation:

In order to identify the relevant ASF security controls mapping for a given proactive control and list of top three semantic similarity scores, the Rule 10, *isSecurityControlListOf(SecurityControlList, ProactiveControl, TopThreeSimilarityScoreList)* is used.

Each mapping security control for the relevant proactive control is collected by Rule 11, *isASecurityControlOf(SecurityControl, ProactiveControl, TopThreeSimilarityScoreList)* using the *semanticAssociation* frame.

Rule 12:

isWeakendByThreatCategory(SecurityControl , T) :-
stride(_ , T , _ , SecContList) ,
member(SecurityControl , SecContList) .

Explanation:

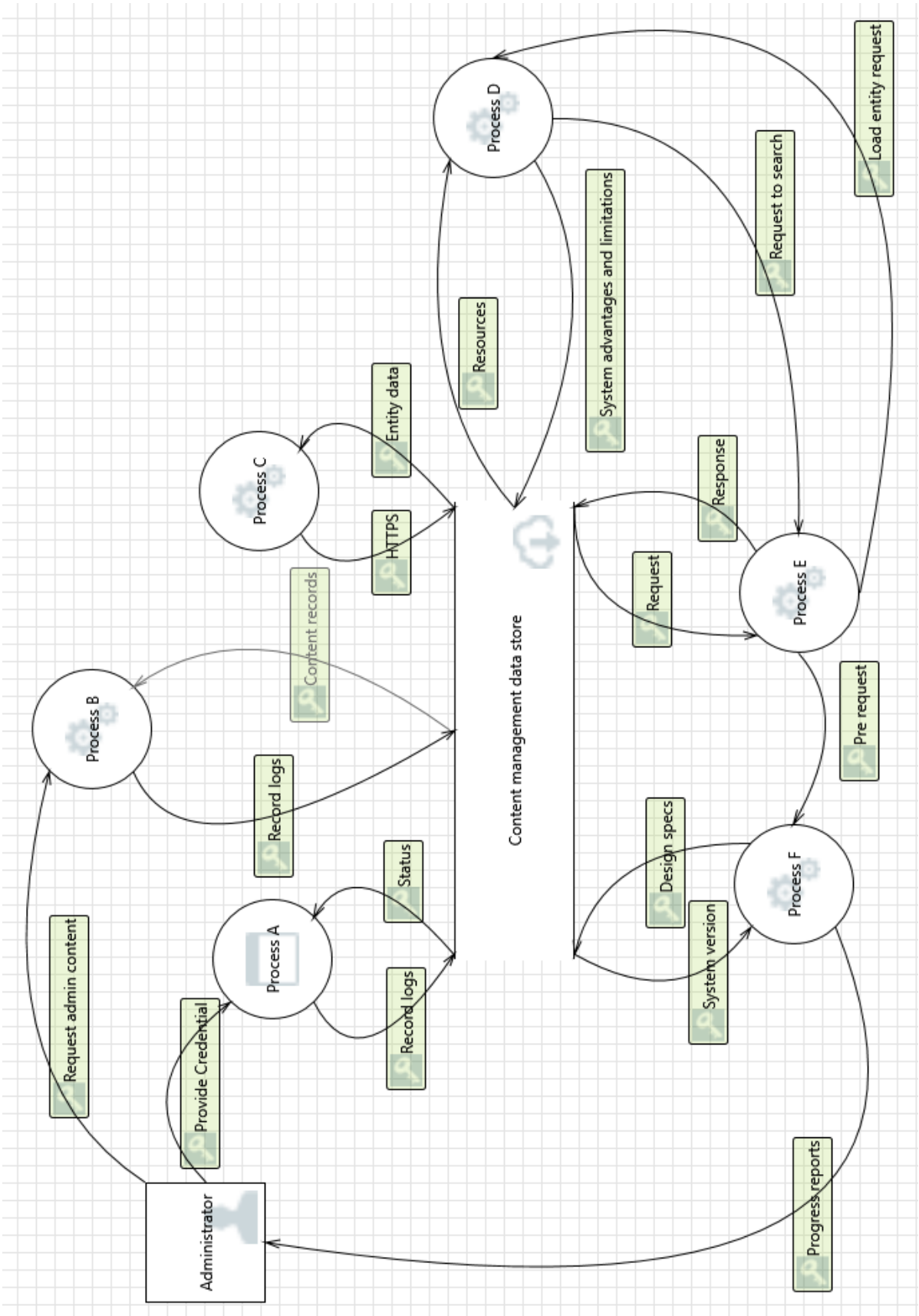
The ASF security controls discovered using the Rule 5 is used to identify the mapping threat category in succession. The *stride* facts stored in the knowledge base using the Frames data structure is used in Rule 12, *isWeakendByThreatCategory(SecurityControl, ThreatCateogry)* with the built-in *member/2* function in Prolog.

Appendix D : ASF Countermeasures

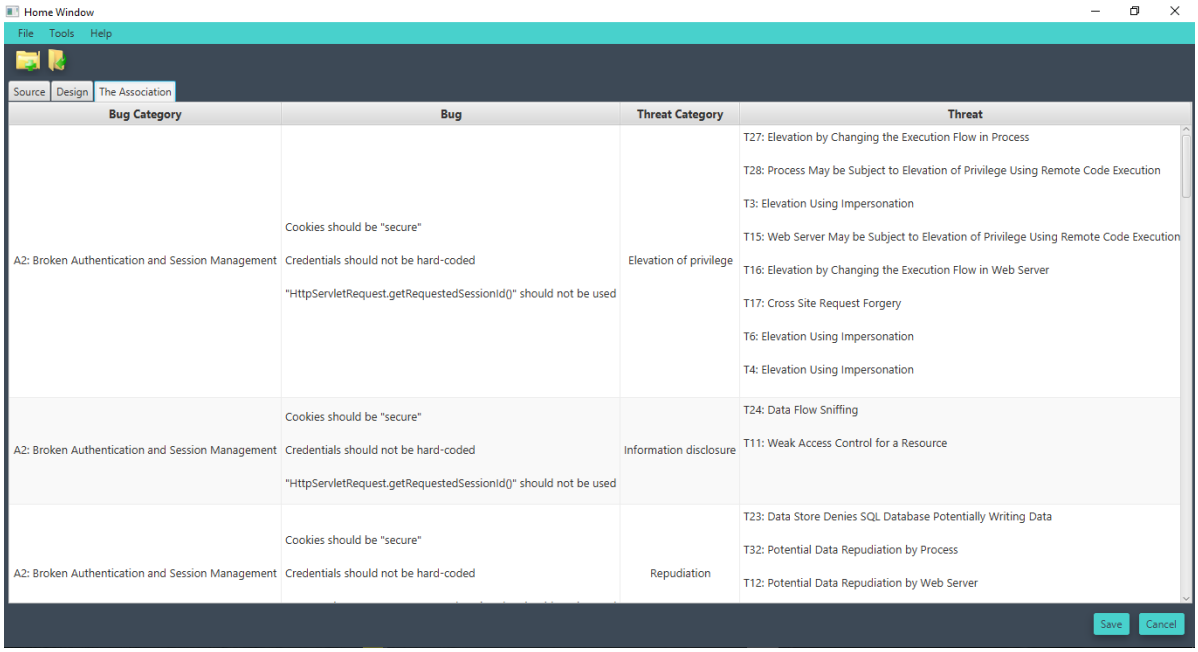
ASF Security Controls & Countermeasures List	
ASF Security Control Type	**Countermeasures**
Authentication	1. Credentials and authentication tokens are protected with encryption in storage and transit 2. Protocols are resistant to brute force, dictionary, and replay attacks 3. Strong password policies are enforced 4. Trusted server authentication is used instead of SQL authentication 5. Passwords are stored with salted hashes 6. Password resets do not reveal password hints and valid usernames 7. Account lockouts do not result in a denial of service attack
Authorization	1. Strong ACLs are used for enforcing authorized access to resources 2. Role-based access controls are used to restrict access to specific operations 3. The system follows the principle of least privilege for user and service accounts 4. Privilege separation is correctly configured within the presentation, business and data access layers
Configuration Management	1. Least privileged processes are used and service accounts with no administration capability 2. Auditing and logging of all administration activities is enabled 3. Access to configuration files and administrator interfaces is restricted to administrators
Data Protection in Storage and Transit	1. Standard encryption algorithms and correct key sizes are being used 2. Hashed message authentication codes (HMACs) are used to protect data integrity 3. Secrets (e.g. keys, confidential data) are cryptographically protected both in transport and in storage 4. Built-in secure storage is used for protecting keys 5. No credentials and sensitive data are sent in clear text over the wire
Data Validation / Parameter Validation	1. Data type, format, length, and range checks are enforced 2. All data sent from the client is validated 3. No security decision is based upon parameters (e.g. URL parameters) that can be manipulated 4. Input filtering via whitelist validation is used 5. Output encoding is used
Error Handling and Exception Management	1. All exceptions are handled in a structured manner 2. Privileges are restored to the appropriate level in case of errors and exceptions

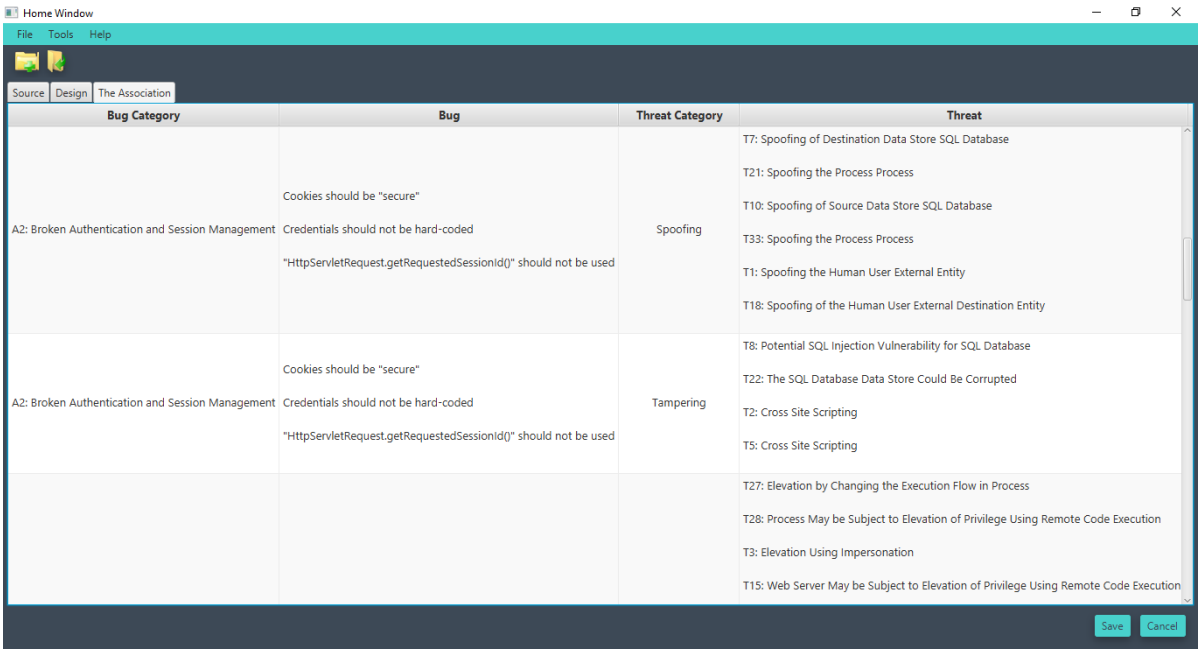
	3. Error messages are scrubbed so that no sensitive information is revealed to the attacker
User and Session Management	1. No sensitive information is stored in clear text in the cookie 2. The contents of the authentication cookies are encrypted 3. Cookies are configured to expire 4. Sessions are resistant to replay attacks 5. Secure communication channels are used to protect authentication cookies 6. The user is forced to re-authenticate when performing critical functions 7. Sessions are expired at logout
Auditing and Logging	1. Sensitive information (e.g. passwords, PII) is not logged 2. Access controls (e.g. ACLs) are enforced on log files to prevent unauthorized access 3. Integrity controls (e.g. signatures) are enforced on log files to provide non-repudiation 4. Log files provide for an audit trail for sensitive operations and logging of key events 5. Auditing and logging is enabled across the tiers on multiple servers

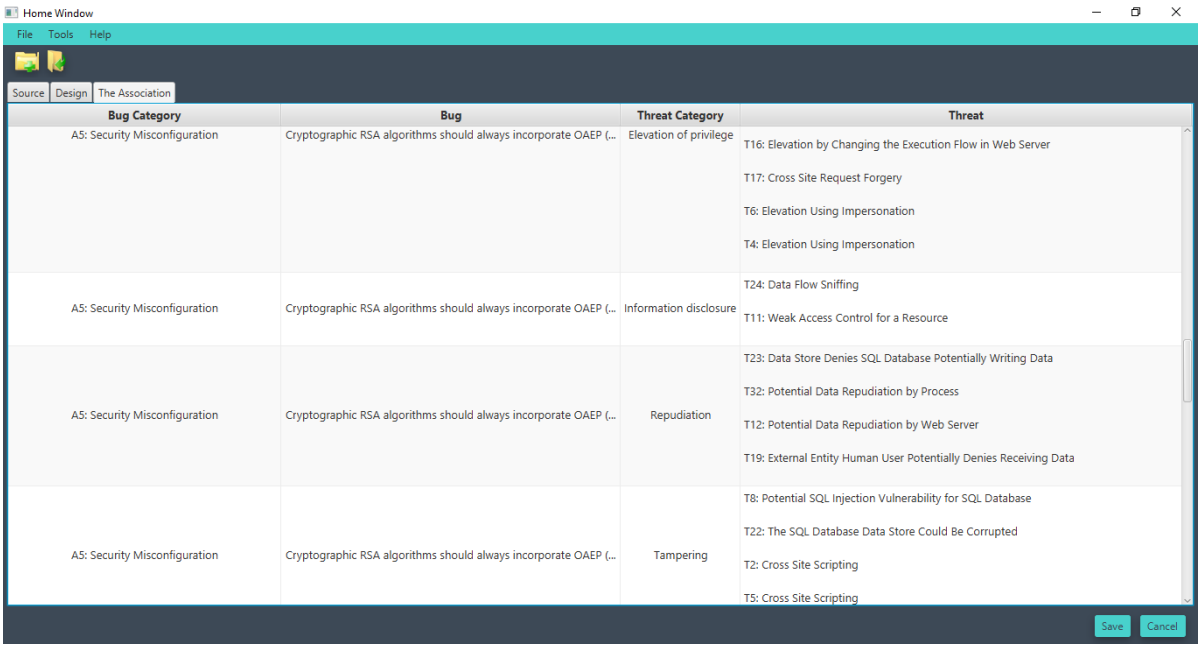
Appendix E : DFD of the Large-scale Analysis Project

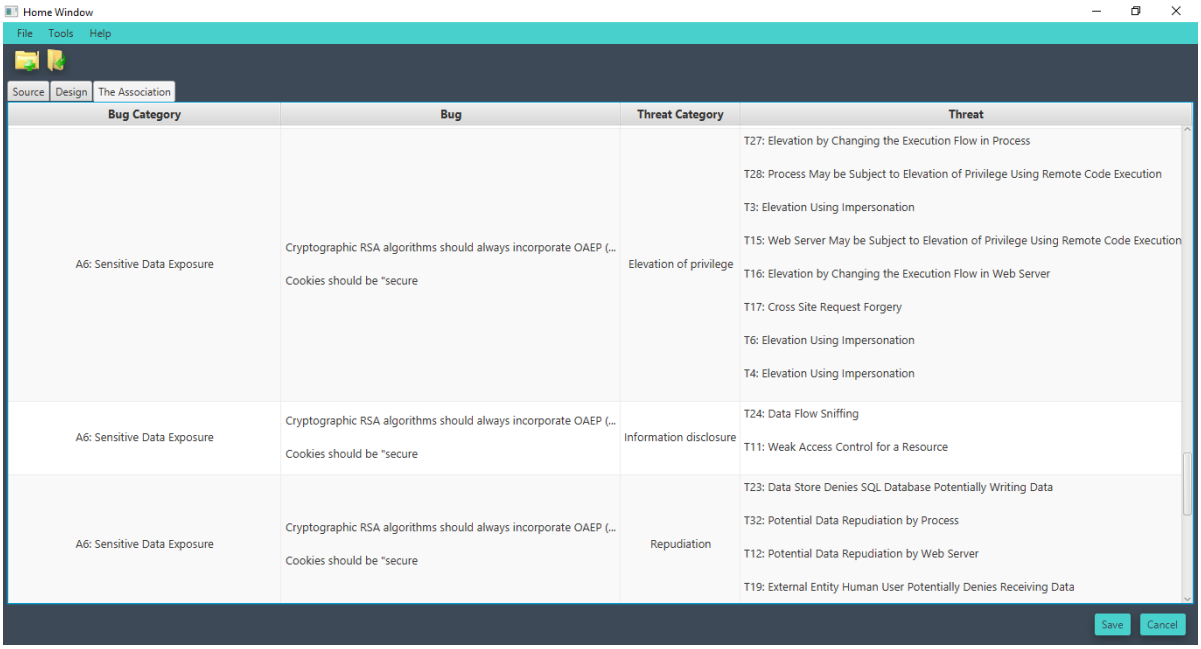


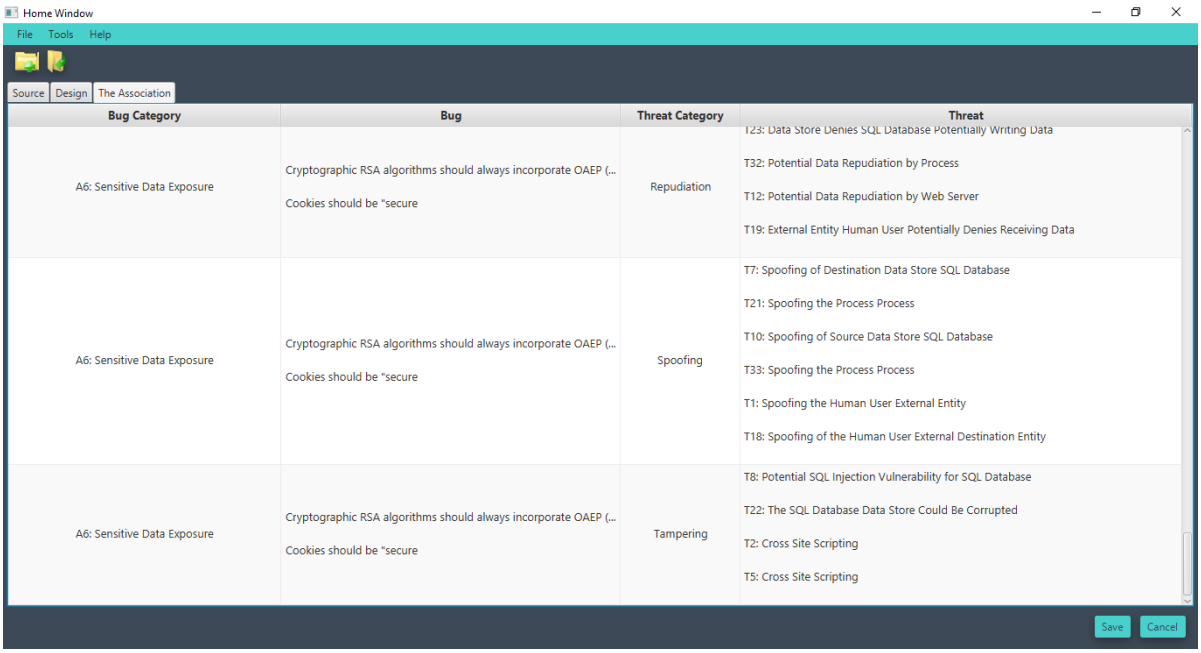
Appendix F : Derived Association for Case Study 1






Home Window
File
Tools
Help
Source
Design
The Association
Bug Category
Bug
Threat Category
Threat
A5: Security Misconfiguration
Cryptographic RSA algorithms should always incorporate OAEP (...
Elevation of privilege
T16: Elevation by Changing the Execution Flow in Web Server
T17: Cross Site Request Forgery
T6: Elevation Using Impersonation
T4: Elevation Using Impersonation
A5: Security Misconfiguration
Cryptographic RSA algorithms should always incorporate OAEP (...
Information disclosure
T24: Data Flow Sniffing
T11: Weak Access Control for a Resource
A5: Security Misconfiguration
Cryptographic RSA algorithms should always incorporate OAEP (...
Repudiation
T23: Data Store Denies SQL Database Potentially Writing Data
T32: Potential Data Repudiation by Process
T12: Potential Data Repudiation by Web Server
T19: External Entity Human User Potentially Denies Receiving Data
A5: Security Misconfiguration
Cryptographic RSA algorithms should always incorporate OAEP (...
Tampering
T8: Potential SQL Injection Vulnerability for SQL Database
T22: The SQL Database Data Store Could Be Corrupted
T2: Cross Site Scripting
T5: Cross Site Scripting
Save
Cancel


Home Window
File
Tools
Help
Source
Design
The Association
Bug Category
Bug
Threat Category
Threat
A6: Sensitive Data Exposure
Cryptographic RSA algorithms should always incorporate OAEP (...
Cookies should be "secure
Elevation of privilege
T27: Elevation by Changing the Execution Flow in Process
T28: Process May be Subject to Elevation of Privilege Using Remote Code Execution
T3: Elevation Using Impersonation
T15: Web Server May be Subject to Elevation of Privilege Using Remote Code Execution
T16: Elevation by Changing the Execution Flow in Web Server
T17: Cross Site Request Forgery
T6: Elevation Using Impersonation
T4: Elevation Using Impersonation
A6: Sensitive Data Exposure
Cryptographic RSA algorithms should always incorporate OAEP (...
Cookies should be "secure
Information disclosure
T24: Data Flow Sniffing
T11: Weak Access Control for a Resource
A6: Sensitive Data Exposure
Cryptographic RSA algorithms should always incorporate OAEP (...
Cookies should be "secure
Repudiation
T23: Data Store Denies SQL Database Potentially Writing Data
T32: Potential Data Repudiation by Process
T12: Potential Data Repudiation by Web Server
T19: External Entity Human User Potentially Denies Receiving Data
Save
Cancel


Home Window
File
Tools
Help
Source
Design
The Association
Bug Category
Bug
Threat Category
Threat
A6: Sensitive Data Exposure
Cryptographic RSA algorithms should always incorporate OAEP (...
Cookies should be "secure
Repudiation
T23: Data Store Denies SQL Database Potentially Writing Data
T32: Potential Data Repudiation by Process
T12: Potential Data Repudiation by Web Server
T19: External Entity Human User Potentially Denies Receiving Data
A6: Sensitive Data Exposure
Cryptographic RSA algorithms should always incorporate OAEP (...
Cookies should be "secure
Spoofing
T7: Spoofing of Destination Data Store SQL Database
T21: Spoofing the Process Process
T10: Spoofing of Source Data Store SQL Database
T33: Spoofing the Process Process
T1: Spoofing the Human User External Entity
T18: Spoofing of the Human User External Destination Entity
A6: Sensitive Data Exposure
Cryptographic RSA algorithms should always incorporate OAEP (...
Cookies should be "secure
Tampering
T8: Potential SQL Injection Vulnerability for SQL Database
T22: The SQL Database Data Store Could Be Corrupted
T2: Cross Site Scripting
T5: Cross Site Scripting
Save
Cancel