JOT
THE JOURNAL OF
OBJECT TECHNOLOGY

Mitigating Injection-Based Weaknesses: A Flexible Template Architecture

Onyeka Ezenwoye
Augusta University, USA

ABSTRACT

Addressing vulnerabilities in software requires consideration of architectural issues. This paper presents a flexible architecture using design patterns to address software vulnerabilities. The architecture employs an extensible security template for mitigating injection-based weaknesses in authentication, authorization, and data validation. The paper presents the architecture's associated algorithms, demonstrates the mapping of weaknesses to use cases and the utilization of architectural, behavioral, and security patterns to mitigate them. A case study application is used to show the prevention of use-case specific weaknesses.

KEYWORDS Design Pattern, Software Vulnerability, Software Architecture

1. Introduction

Software security remains a persistent challenge in the field of software engineering, despite significant advancements in vulnerability identification and mitigation techniques (Leveson 2020). While efforts have been made to combat security issues through secure coding practices, addressing vulnerabilities at the architectural level remains a challenging aspect of software security (Jøsang et al. 2015; Ryoo et al. 2015; Cervantes et al. 2016; Amoroso 2018). Many vulnerabilities are rooted in the design and implementation flaws, making it essential to integrate security considerations into the architecture design process (da Silva Santos, Tarrit, & Mirakhorli 2017).

Neglecting architectural issues can undermine even the most comprehensive coding efforts, leading to insecure systems. One such vulnerability is client-side authentication, where authentication is performed solely within the client code of a client-server product, rather than on the server code. This approach enables attackers to circumvent the authentication processes by using a modified client (da Silva Santos, Tarrit, & Mirakhorli 2017).

It is important to develop effective security measures that integrate security considerations into the software architecture, using appropriate security design patterns (Halkidis et al. 2006; A. V. Uzunov et al. 2012; Villagrán-Velasco et al. 2020). Existing works have focused on software design patterns and security methodologies in isolation, overlooking the integration of these elements to create a unified approach for adaptively mitigating common software weaknesses. These existing works undoubtedly contribute to the understanding of software architecture and security and propose valuable frameworks, methodologies, and patterns. However, they lack a holistic and practical approach to address the evolving security challenges of web-based software.

In contrast to the existing related work, this research focuses on a novel approach that offers a flexible MVC-based architecture for mitigating common contemporary injection-based weaknesses across multiple use cases. This research does not rely on pre-existing architectures such as Hexagonal and Onion due to their inherent complexity and the absence of official reference implementations, which can impose significant adoption challenges (Nunkesser 2022; Khalil et al. 2016). This work, as detailed in the following sections, introduces a comprehensive framework that bridges this gap by providing adaptable and practical security solutions that can be tailored to diverse web application scenarios. It proposes a template architecture that leverages software design patterns to mitigate injection-based weaknesses in authentication, authorization, and data validation.

The research question guiding this study is: "How does an extensible security template enhance the prevention of common

JOT reference format:
Onyeka Ezenwoye. *Mitigating Injection-Based Weaknesses: A Flexible Template Architecture*. Journal of Object Technology. Vol. 23, No. 1, 2024.

http://dx.doi.org/10.5381/jot.2024.23.1.a3

injection-based weaknesses across multiple use cases?" The primary contribution of this research is the development and evaluation of an extensible security template and its associated algorithms, that can be applied to various use cases to effectively mitigate injection-based weaknesses. By mapping security requirements to use cases and utilizing architectural, behavioral, and security patterns, a repeatable solution for enhancing application security is provided.

To assess the effectiveness of the proposed architecture, a case study is conducted using a web application with known vulnerabilities. Through this evaluation, the mitigation of various security weaknesses in each use case is demonstrated, providing empirical evidence of the architecture's efficacy. This solution can be generalized to other similar applications or systems.

For the rest of this paper, an overview of the research methodology presented in Section 2, Section 3 describes the background work that supports this research, and Section 4 discusses the process for determining the requirements for the flexible architecture. The architecture is presented in Section 5. Section 6 outlines the evaluation methodology and presents the results of the case study. Section 7 provides some lessons learned and suggests areas for further research. Related work can be found in Section 8. Finally, Section 9 summarizes the work and contribution.

2. Methodology

This section presents the methodology used for this work. The methodology focuses on understanding the requirements, analyzing the use cases of the application, establishing security requirements, implementing the architecture, and systematically evaluating its effectiveness and coverage.

The first step is to analyze each use case of the case study application to identify their weaknesses. This involves an examination of the known weaknesses of the system. The analysis will identify the cause of these weaknesses and the impact they have on the security of the system. Additionally, security requirements for each use case are established to mitigate all potential weaknesses.

The proposed architecture along with the associated algorithms, is then designed to address the requirements for security that have previously been established. The architecture is implemented along with the functional aspects of the original case study application. The implementation will create a new secure version of the original case study application.

Given that the implementation involves integrating security patterns into the proposed architecture, a systematic evaluation is necessary (Halkidis et al. 2006; Villagrán-Velasco et al. 2020). The evaluation will assess the effectiveness and coverage of the architecture in mitigating the identified weaknesses. The system is tested against various injection-based attacks and measuring its ability to prevent them. The coverage of the architecture is assessed by analyzing its capability to address the known weaknesses specific to the original implementation of the case study.

3. Background

This section offers background information on the software vulnerability landscape addressed in this research. It introduces the case study web application and also the process of capturing security requirements for use cases.

3.1. Vulnerability Landscape

A vulnerability refers to a defect in a software artifact that can be exploited to compromise the security of a system. Vulnerabilities are commonly categorized based on their types, also known as *weaknesses* (Bojanova & Galhardo 2023). Examples of weaknesses include *buffer overflow* and *race condition*. The vulnerability landscape of modern software is extensive, owing to its complex nature, interconnectivity, and reliance on existing software components (Ponta et al. 2021; Banga 2020). The Common Weakness Enumeration (CWE), a community-maintained dictionary, currently lists over 900 known weaknesses (MITRE 2023). In this article, the terms vulnerability and weakness will be used interchangeably, acknowledging their technical distinction but recognizing their relationship.

Software vulnerabilities can exist in any software artifact produced at any stage of the software development process, not solely in source code. Mitigating all weaknesses in a system with a single measure is practically impossible. Therefore, efforts to minimize weaknesses must span all phases of the software lifecycle. This research focuses on mitigating specific weaknesses through architectural design.

Addressing software weaknesses requires a comprehensive understanding of the potential security threats faced by the software. In previous work, vulnerability data was analyzed to determine the susceptibility of weaknesses across various software types (Ezenwoye et al. 2020; Ezenwoye & Liu 2022a). The results indicated variations in weaknesses among software types, including operating systems, browsers, utilities, middleware, servers, and web applications. This research concentrates on a group of weaknesses identified as common in web applications.

CWE	Name	Rank
79	Cross-Site Scripting	1
89	SQL Injection	2
200	Exposure of Sensitive Information	3
352	Cross-Site Request Forgery	4
284	Improper Access Control	5
20	Improper Input Validation	6
22	Path Traversal	7
119	Buffer Overflow	8
434	Unrestricted Upload of File	9
94	Code Injection	10
287	Improper Authentication	11

Table 1 The most common web application weaknesses, which make up 70% of reported vulnerabilities (2011-2020)

Table 1 presents the most prevalent weaknesses for web applications based on an analysis of vulnerability data from the National Vulnerability Database (Ezenwoye & Liu 2022a). The weaknesses are ranked by their frequency of occurrence, rather than employing other ranking methodologies that consider addi-

tional metrics such as potential impact and severity (Mell et al. 2006; Wu et al. 2015; Galhardo et al. 2020). The results demonstrate that Cross-Site Scripting (CWE-79) is the most frequently reported weakness among web application vulnerabilities. The data covers over thirteen thousand vulnerabilities in a ten-year period (2011 to 2020). While the total number of reported weaknesses exceeds 100, these 11 weaknesses accounted for 70.4% of all vulnerabilities, making them representative of the most common weaknesses in this software type.

3.2. Insecure Web Application

Previously, a web application called *SecureEd* was developed to demonstrate typical vulnerabilities mentioned in Table 1. SecureEd is an educational application within the domain of a course management system and contains ten real-world use cases. Its purpose is to allow students to explore common weaknesses found in web applications (Lee et al. 2021). This research aims to analyze and fix these vulnerabilities for educational purposes, and hence it is introduced here.

SecureEd has a total of 11 known vulnerabilities, including 10 of the web application weaknesses specified in Table 1. It is worth noting that there are no vulnerabilities related to Buffer Overflow (CWE-119) in this system. Further details regarding these vulnerabilities and their characteristics are provided in Section 4.

To mitigate vulnerabilities within the application's architecture, it is necessary to define appropriate security requirements. These security requirements, in turn, rely on a comprehensive understanding and analysis of the vulnerability risks associated with the application's various use cases. The following section will delve into previous research on the process of specifying security requirements by carefully examining the vulnerability risks associated with the system.

3.3. Specifying Security Requirements

In the process of threat modeling, an important step involves determining the necessary actions to address vulnerability risks once they have been identified and prioritized (Shostack 2014). In most cases, these actions take the form of security requirements that complement the functional requirements of the system. Security requirements are objectives that must be met by the system to prevent specific vulnerabilities. They can apply to the entire system or specific functional requirements (use cases) (Firesmith 2003, 2004; Bruegge & Dutoit 2009).

To illustrate, in a healthcare application, the security requirement to encrypt private information storage would be applicable to the entire system, regardless of individual use cases (Thapa & Camtepe 2021). However, user authentication requirements may vary depending on the specific use case. For instance, an e-commerce application may not require user authentication for adding items to the shopping cart, but it may require authentication for completing the payment process. Therefore, the security requirement to mitigate Improper Authentication (CWE-287) would only be specified for the checkout use case and not for adding items to the shopping cart.

In previous work, the Risk-Based Security Requirements (RBSR) model was proposed (Ezenwoye & Liu 2022b). This

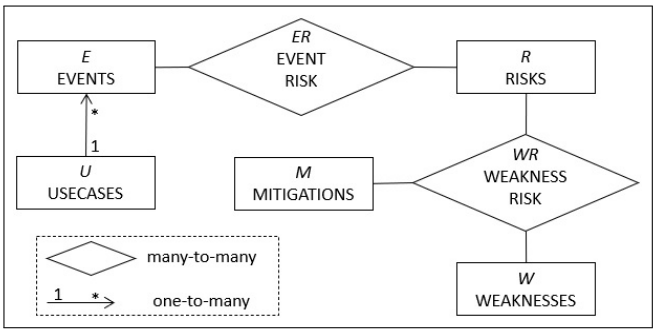

Figure 1 RBSR model: Use cases are associated with events, which have security risks. Weaknesses are associated with risks, which have mitigations.

model facilitates the process of linking security requirements to use cases based on their respective vulnerability risks. Figure 1 illustrates the relationships among the entities within the RBSR model, which include use cases (U), events (E), weaknesses (W), risks (R), and mitigations (M).

The RBSR model enables the linkage of each use case to essential events necessary for its completion. For instance, in an e-commerce application, a use case for paying for items in a shopping cart may involve an essential event where the user submits their credit card information using a designated form. Each event associated with a use case is then linked to potential vulnerability risks. For example, entering credit card details may pose the risk of malicious data input, leading to SQL Injection (CWE-89) or Buffer Overflow (CWE-119). Each identified risk is associated with potential weaknesses that could be exploited. To mitigate the risk of malicious input resulting in SQL Injection, a security requirement may specify the validation of input data to detect any SQL Injection-specific scripts. By considering the functional requirements of each individual use case, the RBSR model allows for comprehensive and coherent integration of security requirements across multiple use cases, reducing the possibility of security vulnerabilities in the system.

Table 2 provides an example of a use case description enhanced with security requirements. Use case descriptions offer a structured representation of use cases, outlining specific interactions and scenarios within an application. They may include non-functional requirements, such as relevant security considerations. In this example, Table 2 describes a scenario where the login feature is utilized within SecureEd. The use case is analyzed using the RBSR model to identify security requirements. The description includes events, actors, pre/post-conditions, along with the associated security requirements and their corresponding weaknesses (CWEs). This example demonstrates the inclusion of use case-specific security requirements. Further discussion on these security requirements for SecureEd is provided in Section 4.2.

While some web application vulnerabilities require system-level security measures, this project focuses on addressing use-case-specific security requirements rather than system-level ones. For instance, the weakness *Exposure of Sensitive Information* (CWE-200), which may involve the lack of secure HTTP

Use case name	Login
Participating actors	Initiated by Admin, Faculty, or Student
Pre-condition	
Flow of events	1. The user enters their username, password, and submits the form. 2. System displays user dashboard or error message for incorrect credentials.
Post-condition	Correct dashboard or login form is displayed
Security requirements	1. Ensure input matches its specification (CWE-20) 1. Ensure input is free of SQL injection (CWE-89) 1. Ensure input is free of XSS (CWE-79) 1. Ensure user is authenticated (CWE-287) 1. Ensure user is authorized (CWE-284) 2. Prevent Information Exposure (CWE-200)

Table 2 Description for login function with security requirements for specific events and weaknesses.

usage, poses a risk of a man-in-the-middle attack (Patni et al. 2017). To mitigate this, a system-level security requirement is necessary, which specifies the use of a secure application layer protocol (Rescorla & Schiffman 1999). It's important to note that this requirement is a server configuration solution rather than an architectural solution.

4. Vulnerability Analysis

Prior to designing the architecture for weakness mitigation, there is need to understand the requirements. This was done by first analysing each use case of the application to identify their weaknesses. The next step was to establish the security requirements for each use case to mitigate all potential weaknesses. This section discusses those two steps.

4.1. Weakness Enumeration

The vulnerabilities in SecureEd that pertain to its use cases , of which there are 9, where identified and analyzed. An overview of these vulnerabilities is presented in Table 3. The table provides a description of each vulnerability, the corresponding use case affected by it, and the associated weakness. As mentioned earlier in Section 3.2, SecureEd has ten use cases, but only the five use cases listed in Table 3 are known to have weaknesses. The application defines three user roles: *Admin*, *Student*, and *Faculty*. Below is a brief description of the use cases identified in Table 3.

- **CreateAccount**: The Admin has the ability to create accounts for all types of actors.
- **EditAccount**: The Admin can modify the profile of any existing account.
- **EnterGrade**: The Faculty can upload a grade report in CSV format to enter grades for students assigned to their course.
- **ForgotPassword**: In case of a forgotten password, all actors can change their password.
- **Login**: Access to the system requires all actors to log in using a username and password.

Table 3 lists the nine vulnerabilities specific to use cases in SecureEd, which can be addressed through architectural measures. Each vulnerability in the table is associated with a corresponding weakness. For example, the first vulnerability is related to a flaw in the Login use case, enabling an attacker to bypass the login mechanism through SQL script insertion. This vulnerability highlights the presence of the Improper Input Validation weakness (CWE-20). Understanding the cause-and-effect relationship among weaknesses is crucial in mitigating them (Bojanova et al. 2016; Zhu et al. 2019).

Upon analyzing the vulnerabilities in Table 3, it became clear that all vulnerabilities can be attributed to three main weaknesses; CWE-20 Improper Input Validation, CWE-287 Improper Authentication, and CWE-284 Improper Access Control. For instance, in the case of the first vulnerability (#1), the lack of proper input validation facilitates SQL Injection (CWE-89), which in turn leads to the bypass of the authentication procedure (CWE-287). This cause-and-effect relationship between weaknesses can be visualized using a causal model (Zhu et al. 2019; Hitchcock 2023), as shown in Figure 2.

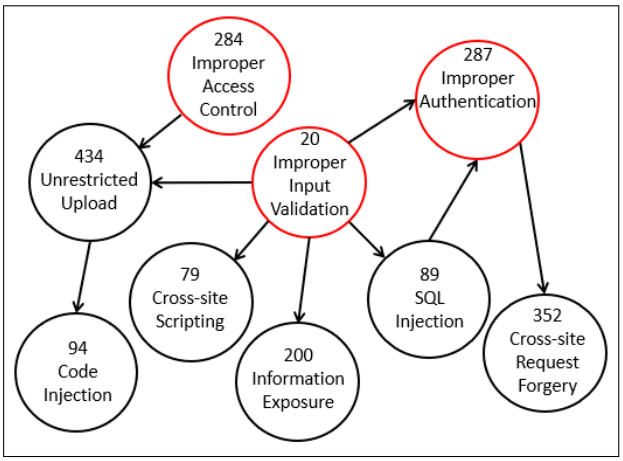

Figure 2 The causal model of the weaknesses, with three highlighted weaknesses as root causes of the vulnerabilities.

Figure 2 is a directed acyclic graph illustrating the causal relationship among the associated weaknesses of the nine vulnerabilities in SecureEd, including (Table 3). Each of the nine weaknesses is depicted as a node, and the outgoing edges indicate the resulting effects of these weaknesses. The highlighted weaknesses signify the three root causes for all nine vulnerabilities. The model shows that the most prominent root cause is *Improper Input Validation* (CWE-20), which is responsible for the majority of the vulnerabilities. Additional information on this cause and effect relationship is provided in Section 6.

The vulnerability analysis also reveals that the attack method for the three causal weaknesses was *injection*. Injection attacks are among the most common software weaknesses, where an attacker can insert malicious data or commands into the system to exploit deficiencies in data validation and access control mechanisms (Bojanova et al. 2021). All the web application

No.	Vulnerability Description	Use Case	Weakness
1	The absence of validation for user name and password fields enables a boolean attack to bypass the login mechanism.	Login	CWE-20 - Improper Input Validation
2	Malicious script in the form a video link is inserted into a form field (first name) which is subsequently executed by the application's client.	EditAccount	CWE-79 - Cross-Site Scripting
3	When an SQL script is injected into a form field (new password), it causes data to be deleted from the database.	ForgotPassword	CWE-89 - SQL Injection
4	Leaving the user name and password input fields blank triggers the application to expose debugging information on the client view.	Login	CWE-200 - Exposure of Sensitive Info.
5	The system has a vulnerability that enables an authorized user (Admin) to upload a file of any type, which can then be accessed afterwards.	EnterGrade	CWE-434 - Unrestricted upload of file
6	A Student can access functionality meant for Faculty by directly navigating to the page.	EnterGrade	CWE-284 - Improper Access Control
7	Instead of using their actual password, a user can log in using a hashed password.	Login	CWE-287 - Improper Authentication
8	An attacker exploits a user's session privilege to create an account by tricking the user into clicking on a link that initiates the creation request.	CreateAccount	CWE-352 - Cross-Site Request Forgery
9	Malicious code can be uploaded to the system via file upload by an authenticated user, and later used to execute PHP commands.	EnterGrade	CWE-94 - Code Injection

Table 3 List of vulnerability instances in SecureEd, along with the impacted use cases and their corresponding weaknesses.

vulnerability types listed in Table 1 are susceptible to some form of injection attack (Bojanova et al. 2021; Ezenwoye & Liu 2022a,c).

The findings of the analysis align with existing research that identifies inadequate *input data validation* and *access control* tactics as the primary source of architectural flaws in software (da Silva Santos, Peruma, et al. 2017). Thus, the proposed architecture's ability to mitigate injection attacks for these root causes is crucial and should serve as the first line of defense in mitigating the resulting weaknesses.

4.2. Security Requirements

The RBSR model, explained in Section 3.3, was utilized to define security requirements for each use case, aiming to address potential weaknesses and known vulnerabilities. As an example, Table 4 presents the description of the grade entry use case, along with its specified security requirements that covers seven possible weaknesses across six events. These weaknesses include Improper Authentication (CWE-287), Improper Access Control (CWE-284), Improper Input Validation (CWE-20), SQL Injection (CWE-89), Cross-site Scripting (CWE-79), Code Injection (CWE-94), and Unrestricted Upload of File (CWE-434).

Certain security requirements apply to multiple events. For instance, authentication is required for both events 1 and 5, given the use case's precondition. Table 3 shows the three vulnerabilities in SecureEd that are associated with the grade entry use case, along with their corresponding weaknesses. The specified security requirements in Table 4 effectively specifies mitigation for these weaknesses.

Due to space constraints, only the descriptions for *Login* and *EnterGrade* use cases from Table 3 are provided here (Tables 2 and 4). However, the remaining use cases share similar scenarios with the two described here.

Use case name	EnterGrade
Participating actors	Initiated by Faculty
Pre-condition	Faculty is logged in
Flow of events	1. Faculty selects the enter grade function from the faculty dashboard. 2. System displays the enter grades form. 3. Faculty enters a course identifier, and selects the upload file function. 4. System displays a file chooser for user to select a CSV file containing data for Student and Grade. 5. Faculty selects a CSV file to upload, then submits the form. 6. System updates the grades for the selected course using the data in the file and displays the faculty dashboard or error message if data is invalid.
Post-condition	Faculty dashboard is displayed AND Data is correctly stored.
Security requirements	1,5. Ensure user is authenticated (CWE-287) 1,5. Ensure user is authorized (CWE-284) 3,5. Ensure input matches its specification (CWE-20) 3. Ensure input is free of SQL injection (CWE-89) 3. Ensure input is free of XSS (CWE-79) 5. Prevent code injection (CWE-94) 5. Prevent unrestricted upload of file (CWE-434)

Table 4 Use case description: Security requirements determined by event-specific vulnerability risk in grade entry

5. A Flexible Architecture

A software architecture outlines the components and subsystems within a system, as well as the connections between them (Buschmann et al. 1996). This section presents such de-

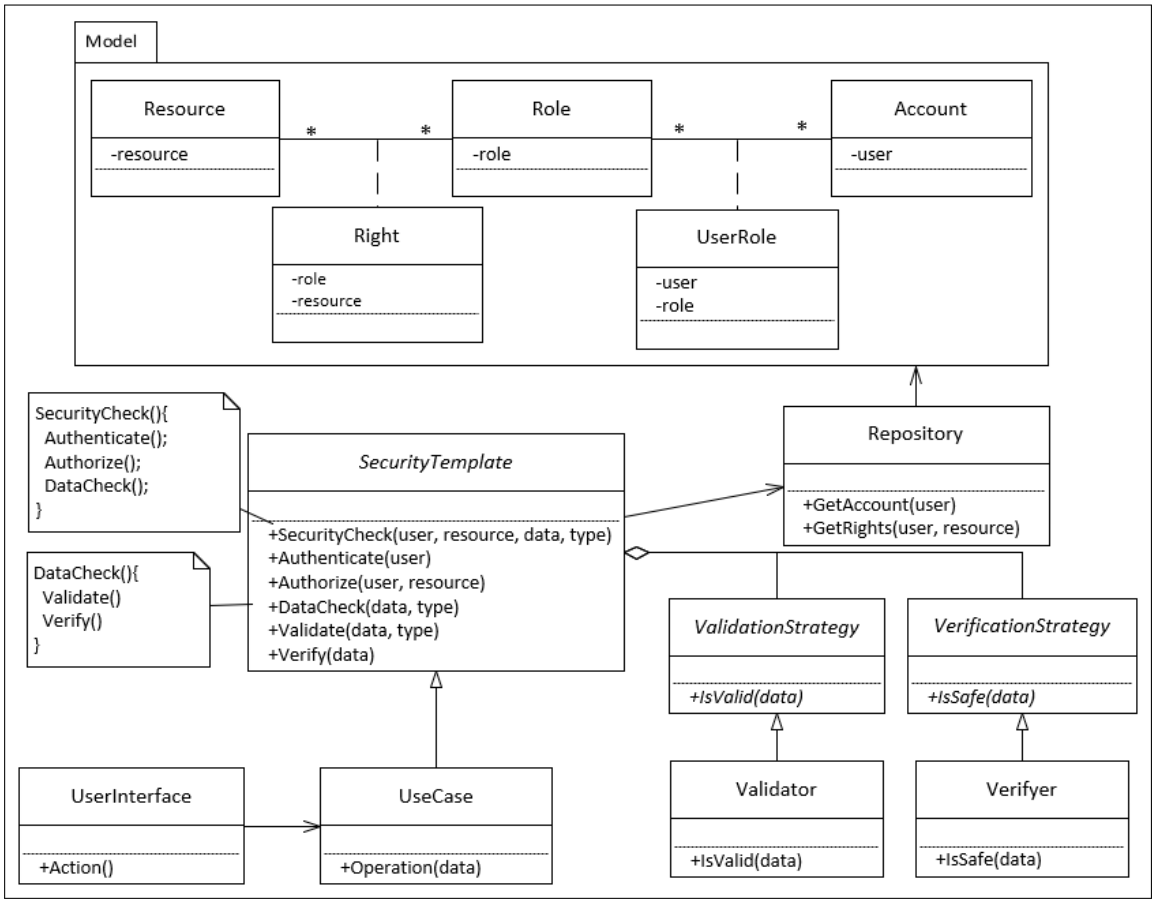

Figure 3 Object model showing the associations between the components of the proposed architecture.

tail for the proposed architecture. The architecture has been designed to incorporate security measures that can safeguard against use-case specific weaknesses as outlined by the requirements. A flexible architecture is needed to accommodate the variations in the functional and security needs of the use cases.

An objective of this project is to develop the flexible architecture by employing well-established design patterns. The design patterns leveraged by the proposed architecture can be classified into three categories: *Architectural*, *Behavioral*, and *Security* patterns.

- **Architectural patterns** refer to the styles that specify how a system is divided into subsystems. This subsystem decomposition is a crucial step in managing the complexity of the system (Bruegge & Dutoit 2009). Two architectural styles are applied: *Model-View-Controller* and *Repository*.
- **Behavioral patterns** are used to provide solutions for flexible interaction between objects. These patterns specify the actions of objects, their responsibilities, and the communication flow between them. Utilizing these patterns promotes flexibility and reuse (Gamma et al. 1995). Two behavioral patterns used in the proposed architecture are *Template Method* and *Strategy*.
- **Security patterns** are standard solutions to recurring security problems. These solutions can be composed into concrete design structures (Schumacher et al. 2013; Villagrán-Velasco et al. 2020). The implemented security patterns in this project include *Authenticator*, *Role-Based Access Control*, and *Input Validation*. Their implementation details are discussed in Section 6.

Figure 3 is class diagram that highlights the key components of the architecture and the relationships that exist between them. The following sections will elaborate on the architect by focusing on the design patterns. Each specific design pattern is explained, taking a pattern-centric approach to justify its inclusion in the architecture. Although certain patterns may depend on each other, presenting them individually provides a logical structure to comprehend their role in the architecture.

5.1. Model-View-Controller

The Model-View-Controller style (MVC) decomposes a system into three subsystems, Model, View and Controller. The Model captures domain knowledge in the form of data structures for the system. The View is responsible for displaying data, it represents the user interface subsystem. The Control subsystem manages interactions with the user. Controllers manage the flow of data between the Model and View (Buschmann et al. 1996; Bruegge & Dutoit 2009). This separation of concerns is an important advantage of MVC, which is used in various types of applications including web applications (Leff & Rayfield 2001).

The model subsystem consists of `Resource`, `Role`,

Account, Right, and UserRole entity objects, which are responsible for maintaining system information required for implementing Role-Based Access Control (RBAC). RBAC is a security model and pattern that restricts system access to users based on their roles (Sandhu et al. 1996; E. B. Fernandez & Pernul 2006; E. B. Fernandez 2007). Permissions are thus assigned to protected resources, such as system's data or functions. The user's access rights are determined by their assigned role. The Account entity object stores user information, Role encapsulates different roles, and UserRole maps each user to their roles. Resource represents all protected resources, while Right captures the relationship between resources and roles as their access rights. It is important to note that other access control models such as Attributed Based Access Control (ABAC) (Yuan & Tong 2005) could be applied here.

The UserInterface class in the view subsystem serves as a boundary object enabling user interaction with the system. The control subsystem contains the remaining components, classified as control objects. The MVC architecture provides adaptability, allowing subsystems to grow and develop independently while dividing responsibilities. The following sections will detail the responsibilities of each component in each subsystem.

5.2. Repository

The Repository design pattern recommends that subsystems should access application data structures by using a centralized component, commonly referred to as the repository. This approach allows for the separation of the logic for data retrieval and update from the rest of the application, which can greatly enhance modularity. This design pattern is frequently used in applications that involve managing databases or other large data sets. One of the key advantages of the Repository pattern is that it provides a mechanism for facilitating access to shared data across multiple subsystems through a standardized interface.

By centralizing access to data, it becomes easier to manage and maintain application-wide data consistency, while minimizing the risk of data inconsistencies and conflicts (Bruegge & Dutoit 2009; Lalanda 1998). In the proposed architecture, the Repository object encapsulates the logic necessary to access entity objects within the model subsystem. This approach enables Repository to serve as a bridge between the application and the underlying data storage mechanisms, providing a layer of abstraction that simplifies the management of data access operations. Repository offers a unified interface that allows the model subsystem to be shared across multiple use cases. It exposes two operations as part of this interface: GetAccount and GetRights.

```
GetAccount(user)
  pre: user ≠ null
  post: (result = null) ∨ (result ∈ Accounts) ∧
    (result.user = user)
  result := null
  FORALL a ∈ Accounts | a.user = user DO
    result := a
  END
  RETURN result
```

Listing 1 The algorithm for GetAccount.

Listing 1 outlines the algorithm for GetAccount. The algorithm is a function that takes in a single parameter, user, which represents the user whose account information is being sought. The algorithm specifies a precondition that the user parameter must not be null. For the postcondition, the returned result can be null or a valid Account object for the user. The Account object returned will have a user attribute that matches the input user parameter. The body of the function iterates through all the Account objects in the Accounts collection and checks if the user attribute of each object matches the user parameter passed into the function. If a match is found, the corresponding Account object is returned. Otherwise, the function returns null. This algorithm is useful in situations where it is necessary to retrieve the account information for a specific user, such as when implementing user authentication or authorization in a use case like that described for the login function (Table 2).

Listing 2 outlines the algorithm for GetRights. The algorithm represents a function GetRights that takes in two parameters, user and resource. The function returns an object (Right) if the user has the rights to access the resource. The algorithm specifies a precondition that user and resource cannot be null. The postcondition states that the function will return either a Right object that belongs to the user or null if the user does not have access to the resource. The body of the function iterates through all the Right objects and checks if the user attribute of each object matches the user parameter passed into the function. If a match is found, the corresponding Right object is returned. Otherwise, the function returns null. This algorithm is useful in situations where access control needs to be implemented, as it allows for the retrieval of user permissions for a particular resource.

```
GetRights(user, resource)
  pre: user ≠ null ∧ resource ≠ null
  post: (result = null) ∨ (result ∈ Rights) ∧ (
    user = result.user) ∧ (resource = result.
    resource)
  result := null
  FORALL r ∈ Rights | user = r.user ∧ resource =
     r.resource DO
    result := r
  END
  RETURN result
```

Listing 2 The algorithm for GetRights.

5.3. Template Method

With the Template Method, algorithmic steps can be defined in abstract terms. These steps may utilize operations of the object. Subclasses may extend some steps of the algorithm without modifying the structure of the original algorithm (Gamma et al. 1995). This pattern is useful for a variety of applications where it is necessary to define the algorithm for a sequence of operations such as the *Connect-Send-Receive-Disconnect* sequence of a network Socket API (Kalita 2014). Subclasses may extend the operations but not the sequence.

Figure 4 illustrates the Template Method pattern from overall architecture (Figure 3). The class SecurityTemplate contains two template methods, SecurityCheck and DataCheck. The UseCase class is an abstraction of an application various use

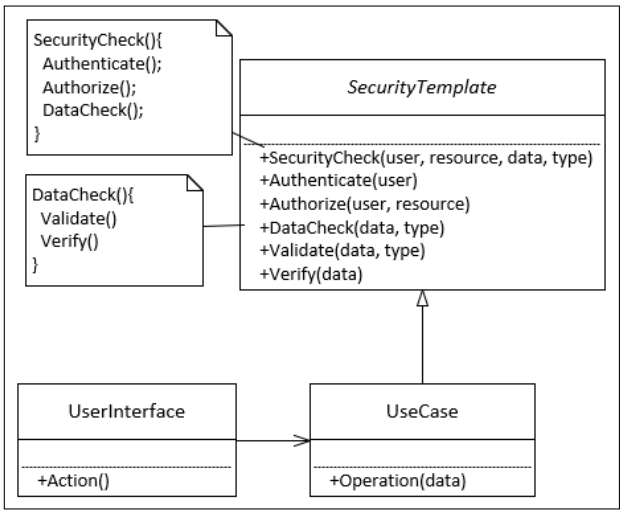

Figure 4 The template methods `SecurityCheck` and `DataCheck` as members of `SecurityTemplate`.

cases. These use cases consist of operations that could potentially involve certain data, such as user credentials in the login function (Table 2). `SecurityTemplate` is a generalization of `UseCase`, thus every `UseCase` *is-type-of* `SecurityTemplate`. The template methods `SecurityCheck` and `DataCheck` define the skeleton of algorithms in the base class, allowing the subclasses (use cases) to provide specific implementations of certain steps of the algorithm. The subclasses can override the methods in the `SecurityTemplate` class to implement their security checks `SecurityCheck` includes three necessary steps: *Authenticate-Authorize-DataCheck*. Listing 3 outlines the skeleton of the algorithm.

```
SecurityCheck(user, resource, data, type)
 pre: user ≠ null ∧ resource ≠ null
 post: result = true ∨ result = false
 result := false
 IF Authenticate(user) THEN
    IF Authorize(user, resource) THEN
       result := true
       IF (data ≠ null) AND (type ≠ null) THEN
          result := DataCheck(data, type)
       END
    END
 END
 RETURN result
```

Listing 3 The algorithm for `SecurityCheck`.

The `SecurityCheck` algorithm is a function that takes in several parameters: a `user` parameter which represents the user attempting to access a resource, a resource parameter which represents the resource the user is trying to access, a `data` parameter which contains input data from the user, and a type parameter which represents the desired specification of data. The algorithm specifies a precondition that both the `user` and `resource` parameters must not be null. The postcondition of the function states that the function will return either a `true` or `false` value. The purpose of this function is to ensure that the input data is safe, and only authorized users with the right access rights can access specific resources.

The body of the function first checks if the user is authenticated using the `Authenticate` method. If authentication is successful, the function proceeds to check if the user is authorized to access the resource using the `Authorize` method. `Authenticate` and `Authorize` go together, as merely possessing knowledge of a user's identity may not be adequate to determine whether to permit or deny them from performing specific actions. Therefore, authorization must follow authentication (Arce et al. 2014; Schumacher et al. 2013; Shiroma et al. 2010). If authorization is successful, the algorithm sets the `result` variable to `true` and proceeds to check the data and type parameters using the `DataCheck` function. If both the data and type parameters are not `null`, the algorithm sets the `result` variable to the value returned by the `DataCheck` function. Finally, the algorithm returns the value of the `result` variable.

For use cases that require strict control over resource access and validation of incoming data, this algorithm is necessary. The template method `SecurityCheck` is useful as it specifies a common set of steps that need to be performed, however the details of each step may vary depending on the specific use case. For use cases that may not require either authentication, authorization or data check, these can be skipped by specifying `null` values for the `user`, `resource` or `data` parameters of `SecurityCheck`. This affords each uses the needed flexibility in its implementation. Implementation of the template ensures that only authorized users gain access to specific resources, and data undergoes validation before the system uses it. The algorithm plays a crucial role in improving system security by enforcing these security checks and thwarting unauthorized access or data tampering attempts.

The upcoming sections will provide an explanation of the `DataCheck` template method. It's essential to understand that this data validation process occurs in the `UseCase` component, not the `UserInterface` component. As previously mentioned (Section 5.1), the `UserInterface` is a part of the view subsystem. In a client-server setup, the client hosts the view subsystem, while the model and controller subsystems are located on the server (Leff & Rayfield 2001). As a result, the `UserInterface` is within the reach of users and beyond the system's trust bound-

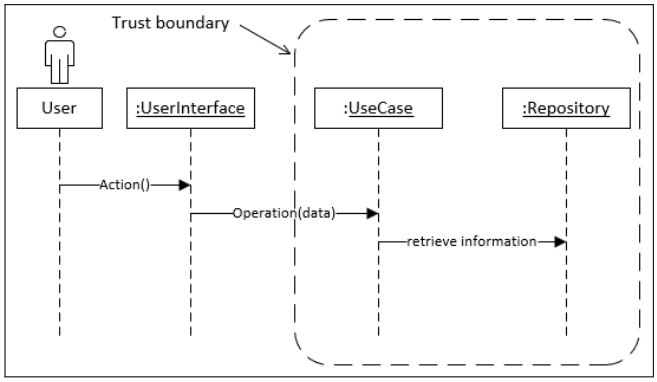

Figure 5 Interaction between architecture's components. `UserInterface` and incoming data are outside of system's trust boundary.

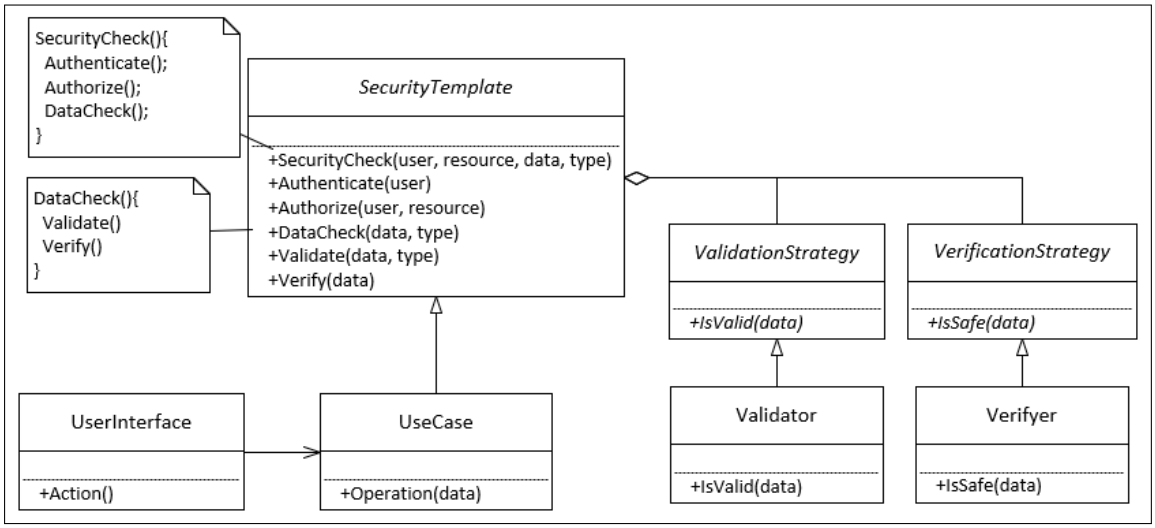

Figure 6 Class diagram illustrating the aggregation association between `SecurityTemplate` and its validation and verification strategies.

ary, posing some vulnerability risks.

The sequence diagram in Figure 5 demonstrates that `UserInterface` triggers an operation in `UseCase` when the user initiates an action. This operation may include data originating from beyond the trust boundary, which could be harmful. Despite initial validation of this data on the client-side, it should not be trusted (Shostack 2014; Ezenwoye & Liu 2022a). To mitigate such risks, it's recommended to incorporate centralized validation mechanisms. This would help ensure that any incoming data to the system or significant components from other components within the same system are subjected to appropriate validation (Arce et al. 2014; da Silva Santos, Tarrit, & Mirakhorli 2017). Consequently, the `DataCheck` template method forms part of `SecurityTemplate` for this purpose.

5.4. Strategy Pattern

The Strategy pattern allows for families of related algorithms to be defined as objects (strategies). For instance, a set of sorting algorithms could be encapsulated as a set of interchangeable concrete strategies. An application can then extend its behavior at runtime by selecting a suitable strategy. This pattern makes it possible to switch algorithms dynamically when multiple algorithms can be applied to solve a particular problem, and the specific algorithm to use may vary depending on the circumstances (Gamma et al. 1995).

The proposed architecture employs the strategy pattern to perform data checks. In Figure 6, the relationship between `SecurityTemplate` and the classes `ValidationStrategy` and `VerificationStrategy` is illustrated through the aggregation (*is-part-of*) association. `ValidationStrategy` and `VerificationStrategy` represent the strategy patterns for data validation and verification, respectively, with `Verifyer` and `Validator` being the concrete strategies. This allows for the creation of multiple versions of `Verifyer` and `Validator`, each encapsulating a unique algorithm for data validation and verification.

By integrating the `DataCheck` template method with the validation and verification strategies, `SecurityTemplate` is able to combine the strengths of both patterns. This results in a highly modular and flexible application that can be easily adapted and extended to meet changing requirements. The template method provides a consistent algorithmic structure, while the strategy pattern offers a variety of interchangeable algorithms that fit within that structure (Buschmann et al. 1996).

List 4 describes the algorithm for `DataCheck` template method. The `DataCheck` algorithm is a function that takes two parameters: `data` (some form of incoming data) and `type`, which specifies the required format of the data. The precondition specifies that both `data` and `type` must not be null. The postcondition of the function is that it will return a `true` or `false` value. The function will return `true` if the data is valid according to its specified format and contains no extraneous or malicious content. Otherwise, it will return `false`.

```
1  DataCheck(data, type)
2    pre: data ≠ null ∧ type ≠ null
3    post: result = true ∨ result = false
4    result := false
5    IF Validate(data, type) THEN
6      IF Verify(data) THEN
7        result := true
8      END
9    END
10   RETURN result
```

Listing 4 The algorithm for `DataCheck`.

The body of the function first calls the `Validate` method to check if the input data matches its specified format. If the input data is valid, the function then calls the `Verify` method to check for any extraneous or malicious content. If both checks pass, the function returns `true`. Otherwise, it returns `false`.

The data check algorithm proves to be advantageous when there is a need to verify that the input data is both in the proper format and without any potential security risks. This algorithm adheres to the Bugs Framework (BF) model which separates

data semantics check as data verification. The data check process in the BF model comprises data validation, which assesses the proper structure of the data syntax, and data verification, which confirms that the data semantics are accurate, making it suitable for the intended purpose (Bojanova et al. 2021).

The `DataCheck` template method employs two methods, `Validate` and `Verify`, to ensure that the data is in the desired format and free from tampering. These methods rely on the `ValidationStrategy` and `VerificationStrategy`, respectively, to provide the necessary algorithms depending on the type of data. Data input, such as the content found in form fields can have varying formats, requiring distinct algorithms to assess their correctness during runtime. To address this need, particular `Validator` and `Verifyer` instances are chosen to execute the necessary algorithms for each input field `Validator` and `Verifyer` instances are selected to implement the necessary algorithms for each input field.

Concrete `Verifyer` instances check for malicious inputs, such as SQLi or XSS, while concrete `Validator` instances ensure that the input adheres to the specified format. Both validation and verification are necessary, as the data can be well-formed but still be malicious (Jan et al. 2015; Bojanova et al. 2021). The combination of data verification and validation forms a defense-in-depth strategy, providing a layered approach to robust vulnerability mitigation (Stytz 2004). Using concrete implementations of various validation and verification algorithms ensures consistent application of data checks for different instances of the same type of data. Validation should use a whitelisting approach, while verification utilizes blacklisting (Arce et al. 2014).

6. Evaluation

The objective of this section is to evaluate how effectively the architecture deals with injection-based security vulnerabilities related to authentication, authorization, and data validation. The primary aim of the evaluation is to demonstrate the adaptability of the architecture in diverse scenarios, covering different use cases and weaknesses present in SecureEd. The evaluation process consists of the following two steps, by which the rest of this section is organized:

1. Implement both the architecture and the use cases.
2. Assess the effectiveness and coverage of the architecture.

6.1. Implementation

The source code for SecureEd is readily available and is of a reasonable scope (Lee & Steed 2021). The application is written mostly in PHP, which is an object-oriented language (*PHP* 2023). The source code underwent refactoring to integrate the proposed architecture. Through this refactoring process, the functionality of SecureEd was retained while simultaneously addressing its security requirements.

The `SecurityTemplate` class of the architecture (Figure 4) was implemented as an abstract class that cannot be instantiated. Rather, it serves as a template for implementing specific security checks in child classes that extend it. Listing 5 shows a snippet of the implementation that includes the template method `SecurityCheck`, which performs various security checks on input data, user sessions, and access rights. The method checks the user's session using the `Authenticate` method, the user's access rights using the `Authorize` method, and the input data using the `DataCheck` method. If any of these checks fail, the method returns `false`, indicating that the security checks did not pass.

```
<?php
abstract class SecurityTemplate{
 public static function SecurityCheck($user, $res
    , $data, $type){
   $secFlag = false;

   if($user != null){ //check session
    $secFlag = self::Authenticate($user);
    if($secFlag == false)
       return false;
   }

   if($res != null){ //check access rights
    $secFlag = self::Authorize($user[0], $res);
    if($secFlag == false)
       return false;
   }

   if($data != null && $type != null){ //check
    input data
    $secFlag = self::DataCheck($data, $type);
    if($secFlag == false)
       return false;
    }

    return $secFlag;
 }
 ...
}
?>
```

Listing 5 Implementation of the `SecurityCheck` algorithm from Listing 3.

As required by the architecture, every implementation of a use case is a child class of `SecurityTemplate` (Figure 4). For this application, every use case inherits the implementation of the authentication, authorizations, and data check procedures. As previously noted, the algorithm for `SecurityCheck` allows for these procedures to be skipped for use cases that do not require them.

The realization of the authentication, authorizations, and data check procedures leveraged the use of security patterns, which provide solutions to recurring security problems. The security patterns implemented here are *Authenticator*, *Role-Based Access Control*, and *Input Validation*. Their implementation details are provided in the following sections, within the context of the `Authenticate`, `Authorize`, and `DataCheck` methods.

6.1.1. Authenticate Method:

The implementation of the `Authenticate` method involved the use of the Authenticator security pattern. The Authenticator pattern is applied to verify the identity of system users. This is typically achieved by using authentication information like credentials, biometrics, or tokens. The primary goal of the Authenticator pattern is to guarantee that a user can establish their identity before gaining access to a secured system (Schumacher et al. 2013).

The Authenticator pattern was implemented in the `Authenticate` method of `SecurityTemplate`. The authentication information utilized are user credentials in the form of a username and password. Listing 6 shows the code for the `Authenticate` method. The method takes a parameter named `user` and its purpose is to authenticate the user's credentials and create a new session ID. The method first checks whether the user has valid session (line 4). And if so, a new session ID is created, indicating successful authentication. Otherwise, the method checks whether the user's password is set. If the password is set, the method retrieves the user's account from the database using the `GetAccount` method of the `DBConnector` class (line 13).

```
1  <?php
2  abstract class SecurityTemplate{
3    public static function Authenticate($user){
4      if(self::IsSetSession($user[0])==true){
5        //create new session ID, discard old one
6        session_regenerate_id();
7        return true;
8      }
9      else if($user[1] != null){ //password is set
10       $username = $user[0];
11       $password = $user[1];
12
13       $User = DBConnector::GetAccount($user);
14
15       if($User != null){//check password matches
16         $validPass = self::PasswordMatches($User,
                 $password);
17
18         if($validPass){//create user session
19           self::CreateSession($username, $User->
                  GetAccType());
20           return true;
21         }
22       }
23     }
24     return false;
25   }
26   ...
27 }
28 ?>
```

Listing 6 Implementation of Authenticator pattern in `Authenticate` method.

The `DBConnector` plays the role of the Repository pattern described in Section 5.2. The associated `GetAccount` algorithm is captured in Listing 1. If the account is found, a check is performed to see whether the password matches the one in the database (line 16). If the password matches, a new user session is created (line 19). If any of the authentication checks fail, the method returns `false`, indicating that the user's credentials could not be authenticated. This method can be extended and customized to support various authentication methods such as two-factor authentication.

6.1.2. Authorize Method: The Role-Based Access Control pattern implements the role-based access control model (RBAC), which is used to manage and regulate access to protected resources based on a user's designated role. Protected resources can include a system, specific features, or data. RBAC involves defining a collection of roles and assigning permissions to each role. This enables resource access to be granted or denied based on the user's role (Schumacher et al. 2013; Sandhu et al. 1996).

In this implementation, RBAC is utilized as the Authorization pattern. Security patterns have interdependencies, such as the one that exists between the Authenticator and Authorization patterns. The Authenticator pattern must be implemented before the Authorization pattern, with Authenticator preceding RBAC (Shiroma et al. 2010). This implementation accomplishes this requirement.

```
1  <?php
2  abstract class SecurityTemplate{
3    public static function Authorize($username,
       $res){
4      $auth_flag = false;
5
6      if($username != null && $res != null){
7        //check user rights to requested resource
8        if(DBConnector::GetRights($username, $res
          ) != null)
9          $auth_flag = true;
10     }
11     return $auth_flag;
12   }
13   ...
14 }
15 ?>
```

Listing 7 Implementation of RBAC pattern within the `Authorize` method.

Listing 7 presents the code for the `Authorize` method within the `SecurityTemplate` class. This method takes in two parameters, a username and a resource. The method is used to verify if the user specified by the username has the necessary rights to access the requested resource. The method invokes the `GetRights` method of the `DBConnector` class to verify if the user has the required access rights. The `GetRights` method implements the algorithm described in Listing 2.

```
1  <?php
2  abstract class SecurityTemplate{
3    public static function DataCheck($data, $type){
4      $correctData = false;
5
6      //check data format
7      $validFormat = self::Validate($data, $type);
8
9      if($validFormat == true){
10       //check data for taint
11       $correctData = self::Verify($data);
12     }
13     return $correctData;
14   }
15   ...
16 }
17 ?>
```

Listing 8 Implementation of `DataCheck` algorithm described in Listing 4.

6.1.3. DataCheck Method: The Input Validation pattern verifies the accuracy of incoming data, such as from web forms, to ensure it is both semantically and syntactically correct. This validation is done prior to processing the data in downstream system components. A validator component enforces data rules during input validation, which may involve the use of whitelists

and blacklists. Data that does not conform to the expected format, range, or is contaminated with harmful content is rejected. This pattern assists in reducing injection-based vulnerabilities such as SQLi and XSS (Netland et al. 2007; Bojanova et al. 2021).

Input Validation pattern is implemented as part of the `DataCheck` method within the `SecurityTemplate` class. The code block in Listing 8 presents the implementation. The `DataCheck` method has two parameters, `data` and `type`. The method is used to verify the integrity of input data by first checking if the data format is valid based on the specified type, this done using the Validate method (line 7). If the data format is valid, the method then proceeds to check if the input data is free of taint using the `Verify` method (line 11). If the `Verify` method returns `true`, then the `correctData` flag is set to `true`, indicating that the input data is valid. The implementation of the `DataCheck` method provides a security mechanism that helps prevent attacks such as SQLi and XSS by ensuring that input data is verified for format and taint before processing. This enhances the overall security of the system by reducing the risk of attacks that exploit vulnerabilities in input data.

```
<?php
abstract class SecurityTemplate{
  public static function Validate($data, $type){
      $validationStrategy = null;
      //select validation starategy
      switch($type){
        case Constants::$CHAR_STRING_TYPE:
          $validationStrategy = new
              CharStringTypeValidator();
          break;
        case Constants::$INT_TYPE:
          $validationStrategy = new
              IntTypeValidator();
          break;
        case Constants::$USERNAME_TYPE:
          $validationStrategy = new
              UsernameTypeValidator();
          break;
        case Constants::$PASSWORD_TYPE:
          $validationStrategy = new
              PasswordTypeValidator();
          break;
        ...
        default:
      }
      //use selected strategy
      if($validationStrategy != null)
         return $validationStrategy->
                      IsValid($data);
      else
         return false;
  }
  ...
}
?>
```

Listing 9 Utilizing concrete validation strategies to perform data format checks in the `Validate` method implementation.

The validator components for enforcing data rules where implemented as Strategy patterns (Section 5.4). Listing 9 shows the implementation of data format checks in the `Validate` method using various concrete validation strategies. The `Validate` method takes in two parameters: `data`, which is the input data to be validated, and `type`, which specifies the type of validation strategy to be used. The code then selects the appropriate validation strategy based on the value of `type` using a switch statement (line 6).

Each concrete validation strategy is defined in a separate class, such as `CharStringTypeValidator` or `IntTypeValidator`, and implements the `IsValid` method, which takes in the input data and returns a boolean value indicating whether the data is valid according to the specified format. Once the appropriate validation strategy is selected based on the `type` parameter, the `IsValid` method of that strategy is called with the input data as its parameter. The `IsValid` method then performs the necessary checks on the input data and returns a boolean value indicating whether the data is valid or not. If no valid validation strategy is found, the `Validate` method returns `false`.

```
<?php
abstract class SecurityTemplate{
 public static function Verify($data){
   //check for SQL injection
   $VerificationStrategy = new SQLiVerifier();
   $sqlSafe = $VerificationStrategy->IsSafe($data
    );

   if($sqlSafe == true){ //check for xss
     $VerificationStrategy = new XssVerifier();
     $xssSafe=$VerificationStrategy->IsSafe($data
    );

     if($xssSafe == true)
       return true;

     return false;
   }
  }
  ...
}
?>
```

Listing 10 Utilization of concrete verification strategies to perform checks for tainted data.

Listing 10 shows the implementation of the `Verify` method that uses concrete verification strategies to check for data taint. In this case, it checks that the data isn't tainted with either SQLi or XSS. The code first creates a new instance of the `SQLiVerifier` class to check for SQL injection vulnerabilities in the input data (line 6). If the data is deemed safe from SQL injection, the code proceeds to create a new instance of the `XssVerifier` class to check for XSS vulnerabilities in the input data (line 10). Both strategies implement the `IsSafe` method, which takes in the input data and returns a boolean value based in the outcome of the check. If the data passes both checks, the `Verify` method return `true`, indicating that the input data is free from any malicious content.

Listing 11 shows the implementation of a concrete validation strategy that checks if a password data follows a set of rules for its format. The strategy is implemented as `PasswordTypeValidator` class, which implements the `ValidationStrategy` interface. The `IsValid` method of the class receives the password data to be validated and checks if its length is between 8 to 10 characters (line 7). Then, it applies

a regular expression pattern to verify if the password contains at least one uppercase letter, one lowercase letter, one number, and one special character from a whitelist that includes some special characters. If the password data passes all these checks, the method returns `true`, indicating that the data is valid. The implementation of this concrete validation strategy helps ensure that password data in a system is correctly formatted.

```
<?php
class PasswordTypeValidator implements
    ValidationStrategy{
   public function IsValid($data){
      $password = $data[0];
      $length = strlen($password);

      if($length >= 8 && $length <= 10){
        $pattern = "/(?=.*[A-Z])(?=.*[a-z])
    (?=.*[0-9])(?=.*[!@#$%^&*])/"; //whitelist

       if(preg_match($pattern, $password)==true)
         return true;
      }
      return false;
  }
}
?>
```

Listing 11 Implementation of a validation strategy that uses a whitelist to check if the input complies with password requirements.

```
<?php
class SQLiVerifier implements
    VerificationStrategy{
    public function IsSafe($data){
       $harmfuls =  array("'", '"', "\x1a", ';',
    '=', '\\', "\0", "\n", "\r"); //blacklist

       foreach($harmfuls as $harmful){
          if(strpos($data[0], $harmful)!=false)
             return false;
       }
       return true;
    }
}
?>
```

Listing 12 Implementation of a verification strategy using a blacklist to detect SQL Injection in input data.

```
<?php
class DBConnector{
    public static function GetRights($uname,
    $resource){
      $query = "SELECT *
                FROM User
                INNER JOIN UserRole ON User.
    UserID = UserRole.uid
                WHERE Email=:un";
      $stmt = $GLOBALS['db']->prepare($query);
      $stmt->bindParam(':un', $uname,
    SQLITE3_TEXT); //parameterized query
      $result = $stmt->execute();
      $userinfo = array();
      ...
    }
    ...
}
?>
```

Listing 13 Implementation of `GetRights` showing use of parametrized query to mitigate SQL injection.

The code snippet at Listing 12 shows the implementation of the concrete verification strategy to ensure that the input data is safe from SQLi attacks. The `SQLiVerifier` class checks the input data for specific characters that are commonly used in SQLi attacks, such as single quotes, double quotes, and semicolons. These characters are included in a blacklist of harmful values (line 4). The code uses a `foreach` loop to iterate through the list of harmful characters and checks whether any of them are present in the input data. If any harmful character is found, the code returns `false`, indicating that the data is not safe from SQLi attacks. If none of the harmful characters are found, the code returns `true`, indicating that the input data is safe.

It is important to note that there are several approaches to mitigating SQLi (Singh 2016; OWASP 2023). Listing 12 could be extended with other strategies. The implementation of the `Repository` component uses parameterized queries as to prevent SQLi. Listing 13 shows the implementation of the `GetRights` method where a parameter is used to bind the username (line 9).

The code at Listing 14 shows the implementation of the Login use case in PHP. The code is contained within the `LoginController` class, which extends the `SecurityTemplate` class. The `Login` method within the `LoginController` class takes in user input data and an optional data type parameter, which is set to a default value of 1. The `SecurityCheck` method is called within `Login` to validate the input data, perform authentication, and authorization after validating the data (line 5). If the input data is valid, the `SecurityCheck` method is called again to authenticate the user (line 10). If the user is authenticated successfully, the user is redirected to the appropriate dashboard page based on their account type.

```
<?php
class LoginController extends SecurityTemplate{
 public static function Login($data,$dataType=1){
   //validate input
   $validData=self::SecurityCheck(null,null,$data
        ,$dataType);

   if($validData == true){
     //authenticate user
     $validUSer=self::SecurityCheck($data,null,
         null,null);

     if($validUSer == true){
       if(self::IsAccountType(
                   Constants::$FACULTY_TYPE))
         header("Location: ../FacultyDash.php");
       elseif(self::IsAccountType(
                   Constants::$ADMIN_TYPE))
         header("Location: ../AdminDash.php");
       elseif (self::IsAccountType(
                   Constants::$STUDENT_TYPE))
         header("Location: ../StudentDash.php");
     }
     else
       LoginForm::Error(
```

```
26                 Constants::$INVALID_CREDENTIALS);
27     }
28     else
29       LoginForm::Error(
30              Constants::$INVALID_INPUT);
31   }
32   ...
33 }
34 ?>
```

Listing 14 Implementation of the Login use case as a subclass of `SecurityTemplate`.

The `IsAccountType` method is used to check the account type of the authenticated user against the predefined account types. If the user account type matches any of the predefined types, then the user is redirected to the appropriate dashboard page. If the user account type does not match any of the predefined types, the user is redirected to an appropriate error page. In case the user input data is invalid, the `LoginForm::Error` method is called to display an appropriate error message to the user (line 25). Similarly, if the user account credentials are invalid, the `LoginForm::Error` method is called to display a user not found error message to the user.

6.2. Effectiveness and Coverage

After implementation, the use cases were tested for all the vulnerabilities listed in Table 3, all of which were effectively resolved. Further tests were performed to verify that all security requirements were met for each use case. Table 5 illustrates the results of the evaluation. The table highlights the security requirements and weaknesses that were targeted by the methods employed in `SecurityTemplate`. It further provides an overview of the SecureEd vulnerabilities that were directly resolved by addressing the root cause (C) and consequently mitigating the resulting effect (E).

Every use case within SecureEd was implemented as a child (*typeOf*) SecurityTemplate, functioning as controller components. These implementations effectively addressed the vulnerabilities in SecureEd by directly mitigating the root cause (C) and subsequently eliminating the resulting effect (E):

1. **Improper Input Validation (C) -> SQL Injection (E) -> Improper Authentication (E)**: In the implementation of the `Login` controller, the `SecurityCheck` component is invoked when users submit their credentials. The `DataCheck` method within `SecurityCheck` is utilized to validate that the provided username and password align with the specified criteria. The validation process incorporates two concrete Strategy patterns (`UsernameTypeValidator` and `PasswordTypeValidator`), effectively preventing the SQLi.

2. **Improper Input Validation (C) -> Cross-Site Scripting (E)**: Within the `EditAccount` controller implementation, when a user submits the account information form, the `SecurityCheck` component is triggered. The `DataCheck` method in `SecurityCheck` is employed to validate that all the submitted data fields adhere to their specified requirements. To accomplish this, the `UserTypeValidator` concrete Strategy pattern is utilized to validate all the form fields, effectively preventing the XSS.

3. **Improper Input Validation (C) -> SQL Injection (E)**: In the `ForgotPassword` controller implementation, when a user submits their new password, the `SecurityCheck` component is triggered. The `DataCheck` method within `SecurityCheck` is used to validate that the provided password meets the specified criteria. This validation process employs the `PasswordTypeValidator` concrete Strategy pattern, effectively preventing SQLi.

4. **Improper Input Validation (C) -> Exposure of Sensitive Information (E)**: The process utilized here is identical to that of vulnerability #1. The empty username and password fields did not cause a failure which prevented the exposure of sensitive information.

5. **Improper Input Validation (C) -> Unrestricted Upload of File (E)**: Within the implementation of the `EnterGrade` controller, when a user submits a form that includes a

Security Requirement	Weakness	SecurityTemplate			SecureEd Vulnerability								
		Authenticate	Authorize	DataCheck	1	2	3	4	5	6	7	8	9
Ensure input matches its specification	CWE-20			✓	C	C	C	C	C		C		
Ensure input is free of XSS	CWE-79			✓		E							
Ensure input is free of SQL injection	CWE-89			✓	E		E						
Prevent code injection	CWE-94			✓									E
Prevent Information Exposure	CWE-200			✓				E					
Ensure user is authorized	CWE-284		✓							C			C
Ensure user is authenticated	CWE-287	✓			E						E	C	
Prevent cross-site request forgery	CWE-352	✓										E	
Prevent unrestricted upload of file	CWE-434			✓					E				E

Table 5 Showing the security requirements and weaknesses that were targeted using `SecurityTemplate` methods. Also indicates the resolution of SecureEd vulnerabilities by addressing the underlying cause (C) and subsequent effect (E).

file, the `SecurityCheck` component is activated. The `DataCheck` method within `SecurityCheck` is employed to validate both the form and the file. This validation process incorporates the `EnterGradeTypeValidator` concrete Strategy pattern. `EnterGradeTypeValidator` further utilizes the `SectionIdTypeValidator` and `GradeFileTypeValidator` for this validation, effectively preventing unrestricted file uploads.

6. **Improper Access Control (C)**: In the `EnterGrade` controller implementation, when a user initiates a request, the `SecurityCheck` component is triggered. The `Authorize` method within `SecurityCheck` is utilized to verify the user's authorization to access the function. This authorization process efficiently prevents a Student from gaining access to Faculty functions.
7. **Improper Input Validation (C) -> Improper Authentication (E)**: The approach employed here mirrors that of vulnerabilities #1 and #4. The `DataCheck` mechanism identified the presence of incorrect input data in the password field, effectively preventing improper authentication.
8. **Improper Authentication (C) -> Cross-Site Request Forgery (E)**: Within the implementation of the `CreateAccount` controller, when the user loads the page that triggers the attacker's request, the `SecurityCheck` component is activated. The `Authenticate` method within `SecurityCheck` is employed to authenticate the user's session. This authentication process effectively safeguards against Cross-Site Request Forgery.
9. **Improper Access Control (C) -> Unrestricted Upload of File (E) -> Code Injection (E)**: This vulnerability bears

No.	Cause	Effect	Use Case	Sample Test Case	Mitigation
1	CWE-20 Improper Input Validation	CWE-89 SQL Injection	Login	Input User Name: *' OR 1=1;*	DataCheck
2	CWE-20 Improper Input Validation	CWE-79 Cross-Site Scripting	EditAccount	Input First Name: *<iframe height="166" src="http://www.youtube.com/ embed/oHg5SJYRHA0?autoplay=1"frameborder="0"> </iframe>*	DataCheck
3	CWE-20 Improper Input Validation	CWE-89 SQL Injection	ForgotPassword	Input New Password: *'; DROP TABLE User;*	DataCheck
4	CWE-20 Improper Input Validation	CWE-200 Information Exposure	Login	Input User Name: (blank), Password: (blank)	DataCheck
5	CWE-20 Improper Input Validation	CWE-434 Unrestricted File Upload	EnterGrade	Authenticated user in Faculty role uploads malicious Grade File (expected csv format - <student ID,letter grade>): Test Input 1: *non,always remember to sanitize your inputs'); DROP TABLE User;–* Test Input 2:*<?php eval ("echo ".$_REQUEST["parameter"].";"); ?>*	DataCheck
6	CWE-284 Improper Access Control		EnterGrade	Authenticated user in Student role attempts to access page for Faculty role via url: *https://localhost:44343/public/EnterGradeForm.php*	Authorize
7	CWE-20 Improper Input Validation	CWE-287 Improper Authentication	Login	Input valid user name and password hash: User Name: *scienceguy@email.com* Password: *1e031774109ee2e6ac244e778ca579d5199 e94fa3753848a3180e9d2e27e8ff7*	DataCheck
8	CWE-287 Improper Authentication	CWE-352 Cross-Site Request Forgery	CreateAccount	Authenticated user access a web page and clicks on an image that submits a hidden from: *<form id="accform" method="post" action="https://localhost:44343/public/ CreateAccountForm.php"> .. </form>*	Authenticate
9	CWE-284 Improper Access Control	CWE-434 Unrestricted File Upload	EnterGrade	An authenticated user in either Admin or Student role attempts to access EnterGradeForm.php and is prevented. Authenticated user in Faculty role is unable to upload malicious file in vulnerability No. 5 (Test Input 2)	DataCheck

Table 6 Each vulnerability in the SecureEd and the Ssample test cases used to validate its mitigation.

resemblance to vulnerability #6. The authorization process in `EnterGrade` controller, exclusively grants access to the file upload function for Faculty members, effectively preventing the occurrence of unrestricted file uploads and subsequent code injection.

All security measures involving the `DataCheck` method employ the `Verify` method immediately after validating the data format using the Validate method (see Listing 8). The `Verify` method ensures that the data is free from SQLi or XSS by utilizing two specific Strategy patterns, namely `SQLVerifier` and `XssVerifier` (see Listing 10). Table 6 provides a comprehensive view of the test cases illustrating successful attacks prior to the implementation of the proposed architecture and their subsequent mitigation within this adaptable framework.

This evaluation included the implementation of all 10 SecureEd use cases as `SecurityTemplate` types, including the previously mentioned 5 use cases (Section 4.1). A total of 17 concrete Strategy patterns were employed to meet the diverse `DataCheck` requirements across all use cases (Hall et al. 2022).

7. Discussion

The security template approach proves to be highly effective in mitigating software vulnerabilities as it adeptly addresses various use-case specific security requirements and weaknesses, successfully resolving multiple vulnerabilities within SecureEd. SecureEd represents a real-world scenario that is applicable to the problem being addressed. This solution described here can be generalized to other similar applications or systems.

7.1. The Choice of Approach

Previous works have contributed to the understanding of software architecture and security, addressing various aspects such as distributed systems, IoT, SOA, and quality requirements. This research takes a novel approach by offering a flexible MVC-based architecture tailored to mitigating common contemporary injection-based weaknesses across multiple use cases.

While architectures such as Hexagonal and Onion have their merits, including adaptability, modularity and separation of concerns, they can also be complex and lack official reference implementations, which can pose significant challenges in terms of adoption for developers who desire a more focused solution for less complex systems. This work offers a comprehensive, practical security solution supported by an architecture that is both comprehensible and straightforward to implement, all while preserving key attributes such as adaptability, modularity, and the separation of concerns.

Structural properties of software, such as coupling, can contribute to cognitive complexity for developers (Emam et al. 2001). The simplicity of the proposed architecture promotes clarity while being effective. The incorporation of the repository pattern emphasizes decoupling which enhances flexibility, robustness, and maintainability (Lalanda 1998). Also, the coupling between object classes (CBO) is very limited and the number of children (NOC) of each class is at most 1. Undue CBO is harmful to modular design, reuse, and maintainability, while large NOC increases the likelihood of improper abstraction and misuse of subclassing (Chidamber & Kemerer 1994). Reducing complexity and enforcing decoupling can be linked to minimizing vulnerabilities and strengthening system security (Chowdhury & Zulkernine 2010).

7.2. Lessons Learned

One of the key lessons learned from this project is the importance of understanding the vulnerabilities specific to a system, including their root causes and consequences. This understanding is instrumental in identifying suitable architectural solutions. The project also emphasizes the importance of taking architectural aspects into account when addressing software vulnerabilities. It shows the benefits of integrating security patterns into the architecture and leveraging design patterns to enhance security measures. It highlights the significance of adaptability in addressing software vulnerabilities and the need for an architecture that can be extended to meet varying security needs.

7.3. Scope of Application

The proposed flexible template architecture is tailored to address injection-based weaknesses in web applications. Its primary focus is on enhancing security in the context of contemporary software challenges. Web applications with similar security requirements as those discussed here, stand to benefit most from the adaptability and modularity offered by this approach. The architecture could potentially extend to other software domains, such as mobile applications or distributed systems that utilize the request-response communication pattern, have multiple use cases and require input data validation, user authentication, and authorization.

A possible limitation is that the extensibility of the architecture could lead to increased complexity as additional security measures are integrated. Increased complexity could adversely impact other application attributes such as performance and maintainability. Also, the architecture of some existing systems may not align with that of the proposed MVC-based design. For such systems, integrating the proposed architecture might not be feasible or may require that substantial modifications are made.

7.4. Future Research Directions

There are several promising directions for future research. One avenue involves investigating the integration of additional patterns to enhance system-wide security. Also, it would be valuable to explore the applicability of the architecture in different domains and extend the evaluation to include larger-scale systems. Another potential area of work is the development of automated tools and frameworks that support the implementation and adoption of the proposed architecture. Such tools can assist developers in identifying vulnerabilities, seamlessly integrating security patterns, and evaluating the effectiveness of the architecture, thereby streamlining the mitigation process.

Future research might also investigate the feasibility of the architecture's principles in non-software contexts, such as network protocols. This would assess the broader applicability of the design pattern-based approach.

Study	Advantages	Disadvantages
Uzunov et al. (A. Uzunov et al. 2012; A. V. Uzunov et al. 2012)	– Comprehensive evaluation of security methodologies. – Systematic methodology comparison.	– Lacks specific security patterns for contemporary weaknesses. – Doesn't offer practical solutions for improving security or methodology applicability.
Khan et al. (Khan et al. 2021)	– Emphasis on integrating security throughout the software development lifecycle. – Presents a SWOT analysis of security approaches.	– Lacks a unified framework for addressing various vulnerabilities. – Doesn't offer specific security architecture.
Fernandez et al. (E. Fernandez et al. 2021)	– Catalog of security patterns for IoT systems.	– Primarily focused on IoT security. – Doesn't address injection-based weaknesses. – No proposed architecture leveraging patterns.
Realpe-Muñoz et al. (Realpe-Muñoz et al. 2017)	– Focuses on effective authentication interfaces. – Discusses evaluation for usability and security.	– Limited scope in overall software security. – Doesn't address a wide range of security issues.
Alkussayer et al. (Alkussayer & Allen 2010)	– Framework for integrating security patterns. – Combines secure software development best practices with patterns.	– Generalized framework, lacks focus on specific vulnerabilities and patterns. – No practical case study for demonstration.
Awaysheh et al. (Awaysheh et al. 2021)	– Considers confidentiality, integrity, availability, and privacy as concerns for Big Data operations. – Presents a framework for designing BigCloud solutions with security analysis patterns.	– Specialized for cloud-based Big Data, limited applicability. – Introduces a high level of complexity which might not fit smaller projects.
Dwivedi et al. (Dwivedi & Rath 2015)	– Focus on incorporating security features in Service-Oriented Architecture (SOA). – Integrates components for concerns such as confidentiality, authentication, and authorization.	– Doesn't cover contemporary injection-based vulnerabilities or broader architectural issues. – Doesn't cover specific implementation details or discuss comprehensive mitigation strategies.
Savić et al. (Savić et al. 2010)	– Provides a software architecture integrating authentication and authorization processes. – Uses Chain of Responsibility pattern to describe authentication and authorization processes.	– Doesn't cover a wide range of security concerns, including data validation. – Doesn't include a case study to demonstrate the effectiveness of proposed security architecture.
Alebrahim et al. (Alebrahim et al. 2011)	– Focuses on integrating security and performance requirements into software architecture. – Introduces a method for annotating UML with quality requirements within the architecture.	– Limited focus on security weaknesses and security patterns to address vulnerabilities. – Doesn't explicitly emphasize the need for architectural adaptability.
Harrison et al. (Harrison & Avgeriou 2007)	– Introduces the concept of consequences for architectural pattern reviews. – Provides an analysis of architecture patterns and their impact quality attributes.	– Doesn't address contemporary security challenges comprehensively. – Lacks focus on security and has limitations in practical application.
Ratnaparkhi et al. (Ratnaparkhi & Liu 2021)	– Describes use of Strategy Factory pattern for SQL injection and Cross-Site Scripting (XSS). – Uses case study to demonstrate how the pattern prevents SQL injection and XSS.	– Primarily focuses on SQL injection and XSS. – Doesn't offer a broader perspective on addressing injection-based weaknesses across various aspects of a comprehensive architecture.

Table 7 Comparative Analysis of Related Work

8. Related Work

In this section, we present existing works grouped into three main categories based on their application areas. This structured overview provides insights into contributions related to specific domains, with a focus on software weaknesses. We compare and contrast these works with our proposed flexible template architecture.

8.1. Security Methodologies and Lifecycle Integration

- Uzunov et al. (A. Uzunov et al. 2012; A. V. Uzunov et al. 2012) offer a comprehensive survey of security methodologies for distributed systems, emphasizing software lifecycles, modeling languages, and security pattern evaluation.
- Khan et al. (Khan et al. 2021) contribute by highlighting the importance of integrating security measures throughout the software development lifecycle. These works provide overarching security principles and holistic considerations.

8.2. Security Challenges in Specific Domains

- Fernandez et al. (E. Fernandez et al. 2021) address security challenges in IoT systems, focusing on the use of patterns to handle specific concerns.
- Awaysheh et al. (Awaysheh et al. 2021) present the BigCloud security-by-design framework for cloud-based Big Data operations.
- Dwivedi et al. (Dwivedi & Rath 2015) incorporate security features in Service-Oriented Architecture using software security patterns. These works provide domain-specific insights into securing complex systems.

8.3. Architectural Design and Integration

- Realpe-Muñoz et al. (Realpe-Muñoz et al. 2017) discuss the design of effective interfaces for security management systems, focusing on usability and security constraints.
- Savić et al. (Savić et al. 2010) propose a three-tiered software architecture that integrates authentication and authorization processes.
- Alebrahim et al. (Alebrahim et al. 2011) present a UML-based approach for designing software architectures that consider quality requirements and integrate patterns. These works contribute to understanding the practical aspects of integrating security measures into system architectures.

8.4. Design Patterns for Specific Vulnerabilities

- Ratnaparkhi et al. (Ratnaparkhi & Liu 2021) introduce the Secure Strategy Factory pattern to address SQLi and XSS vulnerabilities in web applications. Their case study with the OWASP Juice Shop application aligns with our emphasis on design patterns for addressing specific vulnerabilities.

9. Conclusion

This paper presents a Flexible Template Architecture, which effectively leverages the MVC pattern's structure while incorporating customized security design patterns tailored to injection-based weaknesses. By focusing on these specific challenges, the architecture offers a more streamlined and effective approach to addressing these weaknesses. This approach effectively bridges the gap between architectural patterns and security methodologies by tailoring security patterns to software weaknesses. The architecture's adaptability and modularity provide a cohesive solution for enhancing software security.

A case study of a web application is used to demonstrate the architecture's effectiveness in mitigating security weaknesses across various use cases. This integration of security considerations into architecture design , combined with the utilization of existing design patterns, underlines the significance of this research in promoting a holistic approach to software security.

This project highlights the importance of understanding vulnerability causality, considering architectural aspects, and promoting adaptability. Future research directions include exploring additional patterns, different application domains, larger-scale systems, and the development of automated tools to support a seamless weakness mitigation process.

Acknowledgments

I would like to acknowledge the contributions of A. Hall, J. Parker, A. Blackman, and B. McClammy, Computer Science students at Augusta University, who helped with the implementation the proposed software architecture as part of their Senior Capstone Project. Their efforts were instrumental in the completion of this work.

References

Alebrahim, A., Hatebur, D., & Heisel, M. (2011). Towards systematic integration of quality requirements into software architecture. In I. Crnkovic, V. Gruhn, & M. Book (Eds.), *Software architecture*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Alkussayer, A., & Allen, W. (2010, 03). The isdf framework: Towards secure software development. *Journal of Information Processing Systems*, *6*.

Amoroso, E. (2018). Recent progress in software security. *IEEE Software*, *35*(2).

Arce, I., Clark-Fisher, K., Daswani, N., DelGrosso, J., Dhillon, D., Kern, C., ... others (2014). Avoiding the top 10 software security design flaws. *IEEE Computer Society*.

Awaysheh, F. M., Aladwan, M. N., Alazab, M., Alawadi, S., Cabaleiro, J. C., & Pena, T. F. (2021). Security by design for big data frameworks over cloud computing. *IEEE Transactions on Engineering Management*, *69*(6).

Banga, G. (2020). Why is cybersecurity not a human-scale problem anymore? *Commun. ACM*, *63*(4).

Bojanova, I., Black, P. E., Yesha, Y., & Wu, Y. (2016). The bugs framework (BF): A structured approach to express bugs. *2016 IEEE International Conference on Software Quality, Reliability and Security*.

Bojanova, I., & Galhardo, C. E. (2023). Bug, fault, error, or weakness: Demystifying software security vulnerabilities. *IT Professional*, *25*(1).

Bojanova, I., Galhardo, C. E., & Moshtari, S. (2021). Input/output check bugs taxonomy: Injection errors in spotlight. In

IEEE international symposium on software reliability engineering workshops.

Bruegge, B., & Dutoit, A. H. (2009). *Object-oriented software engineering using UML, patterns, and java* (3rd ed.). Prentice Hall Press.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-oriented software architecture: A system of patterns*. John Wiley & Sons, Inc.

Cervantes, H., Kazman, R., Ryoo, J., Choi, D., & Jang, D. (2016). Architectural approaches to security: Four case studies. *Computer*, *49*(11), 60-67.

Chidamber, S., & Kemerer, C. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20*(6), 476-493. doi: 10.1109/32.295895

Chowdhury, I., & Zulkernine, M. (2010). Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In *Proceedings of the ACM symposium on applied computing.* New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1774088.1774504

da Silva Santos, J. C., Peruma, A., Mirakhorli, M., Galstery, M., Vidal, J., & Sejfia, A. (2017). Understanding software vulnerabilities related to architectural security tactics: An empirical investigation of chromium, php and thunderbird. In *2017 IEEE international conference on software architecture.*

da Silva Santos, J. C., Tarrit, K., & Mirakhorli, M. (2017). A catalog of security architecture weaknesses. In *Proceedings of the 2017 IEEE international conference on software architecture workshops.*

Dwivedi, A. K., & Rath, S. K. (2015, feb). Incorporating security features in service-oriented architecture using security patterns. *SIGSOFT Softw. Eng. Notes*, *40*(1).

Emam, K. E., Melo, W., & Machado, J. C. (2001, feb). The prediction of faulty classes using object-oriented design metrics. *J. Syst. Softw.*, *56*(1). doi: 10.1016/S0164-1212(00)00086-8

Ezenwoye, O., & Liu, Y. (2022a). Integrating vulnerability risk into the software process. In *Proceedings of the ACM southeast conference.* Association for Computing Machinery.

Ezenwoye, O., & Liu, Y. (2022b). Risk-based security requirements model for web software. In *IEEE 30th international requirements engineering conference workshops.* IEEE Computer Society.

Ezenwoye, O., & Liu, Y. (2022c). Web application weakness ontology based on vulnerability data. *arXiv preprint arXiv:2209.08067*.

Ezenwoye, O., Liu, Y., & Patten, W. (2020). Classifying common security vulnerabilities by software type. In *International conference on software engineering and knowledge engineering.*

Fernandez, E., Washizaki, H., Yoshioka, N., & Okubo, T. (2021, September). The design of secure iot applications using patterns: State of the art and directions for research. *Internet of Things (Netherlands)*, *15*.

Fernandez, E. B. (2007). Security patterns and secure systems design. In *Proceedings of the third latin-american conference on dependable computing.* Springer-Verlag.

Fernandez, E. B., & Pernul, G. (2006). Patterns for session-based access control. Association for Computing Machinery.

Firesmith, D. (2003). Security use cases. *J. Object Technol.*, *2*, 53-64.

Firesmith, D. (2004). Specifying reusable security requirements. *J. Object Technol.*, *3*, 61-75.

Galhardo, C. C., Mell, P., Bojanova, I., & Gueye, A. (2020). Measurements of the most significant software security weaknesses. In *Annual computer security applications conference.* Association for Computing Machinery.

Gamma, E., Johnson, R., Helm, R., Johnson, R. E., & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH.

Halkidis, S. T., Chatzigeorgiou, A., & Stephanides, G. (2006). A qualitative analysis of software security patterns. *Computers & Security*, *25*(5).

Hall, A., Parker, J., McClammy, B., & Blackman, A. (2022). *Secure web application.* https://github.com/onniegit/Secure-Web-Application. GitHub.

Harrison, N. B., & Avgeriou, P. (2007). Leveraging architecture patterns to satisfy quality attributes. In F. Oquendo (Ed.), *Software architecture.* Berlin, Heidelberg: Springer Berlin Heidelberg.

Hitchcock, C. (2023). Causal Models. In E. N. Zalta & U. Nodelman (Eds.), *The Stanford encyclopedia of philosophy* (Spring 2023 ed.). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/spr2023/entries/causal-models/.

Jan, S., Nguyen, C. D., & Briand, L. (2015). Known XML vulnerabilities are still a threat to popular parsers and open source systems. In *IEEE international conference on software quality, reliability and security* (p. 233-241).

Jøsang, A., Ødegaard, M., & Oftedal, E. (2015). Cybersecurity through secure software development. In *Information security education across the curriculum.* Springer International Publishing.

Kalita, L. (2014). Socket programming. *International Journal of Computer Science and Information Technologies*, *5*(3).

Khalil, M. E., Ghani, K., & Khalil, W. (2016). Onion architecture: a new approach for xaas (every-thing-as-a service) based virtual collaborations. In *2016 13th learning and technology conference (l&t).*

Khan, R. A., Khan, S. U., Khan, H. U., & Ilyas, M. (2021). Systematic mapping study on security approaches in secure software engineering. *IEEE Access*, *9*, 19139-19160.

Lalanda, P. (1998). Shared repository pattern. In *5th annual conference on the pattern languages of programs.*

Lee, D., & Steed, B. (2021). *SecureEd 1.0.* https://github.com/onniegit/SecureEd-1.0. GitHub.

Lee, D., Steed, B., Liu, Y., & Ezenwoye, O. (2021, October). Tutorial: A lightweight web application for software vulnerability demonstration. In *2021 IEEE secure development conference.* IEEE Computer Society.

Leff, A., & Rayfield, J. (2001). Web-application development using the model/view/controller design pattern. In *Proceedings fifth IEEE international enterprise distributed object computing conference.*

Leveson, N. (2020). Are you sure your software will not kill anyone? *Commun. ACM*, *63*(2).

Mell, P., Scarfone, K., & Romanosky, S. (2006). Common vulnerability scoring system. *IEEE Security & Privacy*, *4*(6). doi: 10.1109/MSP.2006.145

MITRE. (2023). *Comprehensive weakness dictionary.* (https://cwe.mitre.org/data/definitions/2000.html, Retrieved February, 2023)

Netland, L.-H., Espelid, Y., & Mughal, K. A. (2007). Security pattern for input validation. In *Fifth nordic conference on pattern languages of programs.*

Nunkesser, R. (2022). Using hexagonal architecture for mobile applications. In *17th international conference on software technologies (icsoft 2022).*

OWASP. (2023). *OWASP cheat sheet series.* (https://cheatsheetseries.owasp.org/, Retrieved October, 2023)

Patni, P., Iyer, K., Sarode, R., Mali, A., & Nimkar, A. (2017). Man-in-the-middle attack in http/2. In *2017 international conference on intelligent computing and control (i2c2).*

PHP. (2023). (https://www.php.net/manual/en/oop5.intro.php, Retrieved May, 2023)

Ponta, S., Fischer, W., Plate, H., & Sabetta, A. (2021). The used, the bloated, and the vulnerable: Reducing the attack surface of an industrial application. In *IEEE international conference on software maintenance and evolution.* IEEE Computer Society.

Ratnaparkhi, A. C., & Liu, Y. (2021). Towards tackling common web application vulnerabilities using secure design patterns. In *2021 ieee international conference on electro information technology (eit).*

Realpe-Muñoz, P., Collazos, C. A., Granollers, T., Muñoz Arteaga, J., & Fernandez, E. B. (2017). Design process for usable security and authentication using a user-centered approach. In *Proceedings of the xviii international conference on human computer interaction.* New York, NY, USA: Association for Computing Machinery.

Rescorla, E., & Schiffman, A. (1999). *The secure hypertext transfer protocol* (Tech. Rep.). Network Working Group.

Ryoo, J., Kazman, R., & Anand, P. (2015). Architectural analysis for security. *IEEE Security & Privacy*, *13*(6), 52-59.

Sandhu, R. S., Coyne, E. J., Feinstein, H. L., & Youman, C. E. (1996, February). Role-based access control models. *IEEE Computer*, *29*(2).

Savić, D., Simić, D., & Vlajić, S. (2010). Extended software architecture based on security patterns. *Informatica*, *21*(2).

Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., & Sommerlad, P. (2013). *Security patterns: Integrating security and systems engineering*. Wiley.

Shiroma, Y., Washizaki, H., Fukazawa, Y., Kubo, A., Yoshioka, N., & Fernández, E. (2010). Model-driven security patterns application and validation. In *Proceedings of the 17th conference on pattern languages of programs.*

Shostack, A. (2014). *Threat modeling: Designing for security*. Wiley.

Singh, J. P. (2016). Analysis of sql injection detection techniques. *arXiv preprint arXiv:1605.02796*.

Stytz, M. R. (2004). Considering defense in depth for software applications. *IEEE Security & Privacy*, *2*(1), 72–75.

Thapa, C., & Camtepe, S. (2021). Precision health data: Requirements, challenges and existing techniques for data security and privacy. *Computers in Biology and Medicine*, *129*, 104130.

Uzunov, A., Fernandez, E., & Falkner, K. (2012). Engineering security into distributed systems: A survey of methodologies. *Journale of Universal Computer Science*, *18*(20).

Uzunov, A. V., Fernandez, E. B., & Falkner, K. (2012, jul). Securing distributed systems using patterns: A survey. *Comput. Secur.*, *31*(5).

Villagrán-Velasco, O., Fernández, E. B., & Ortega-Arjona, J. (2020). Refining the evaluation of the degree of security of a system built using security patterns. In *Proceedings of the 15th international conference on availability, reliability and security.* Association for Computing Machinery.

Wu, Y., Bojanova, I., & Yesha, Y. (2015). They know your weaknesses–do you?: Reintroducing common weakness enumeration. *CrossTalk*, *45*.

Yuan, E., & Tong, J. (2005). Attributed based access control (abac) for web services. In *Ieee international conference on web services (icws'05).*

Zhu, L., Zhang, Z., Xia, G., & Jiang, C. (2019). Research on vulnerability ontology model. In *2019 IEEE 8th joint international information technology and artificial intelligence conference.*

About the author

Onyeka Ezenwoye is an associate professor at the School of Computer and Cyber Sciences in Augusta University (USA). He holds a bachelors degree in Software Engineering and a doctorate degree in Computer Science.