Research Article

An Approach for Verification of Secure Access Control Using Security Pattern

Charu Gupta (✉️), Rakesh Kumar Singh (✉️), and Amar Kumar Mohapatra

Department of Information Technology, Indira Gandhi Delhi Technical University for Women, Delhi 110006, India

Correspondence should be addressed to Charu Gupta; charugupta@igdtuw.ac.in

Received 15 June 2022; Accepted 12 August 2022; Published 6 September 2022

Academic Editor: Kapil Sharma

Copyright © 2022 Charu Gupta et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

According to OWASP-2021, more than 3,00,000 web applications have been detected for unauthenticated and unauthorised access leading to a breach of security trust. Security patterns are commonly used in web applications to address the problem of broken access. Web developers are not experts in implementing security patterns. Therefore, it is necessary to verify that the security pattern has been applied, specifying the original intent of the security pattern. In this paper, an approach has been proposed that analyses the behavioural aspect of security patterns to verify that it meets the security requirement of the web application. The proposed approach extracts the class diagram’s structural properties, relations, associations, and security-related constraints and verifies it using the first-order predicate logic. Experiments have been conducted using class diagrams of security patterns to detect instances of broken access control early in the design phase. The proposed approach will help minimise the risk of unauthenticated and unauthorised access to a web application.

1. Introduction

According to OWASP-2021, security vulnerabilities causing broken access control have moved to the first position from fifth in the last three years [1]. 94.5% of web applications have been detected with security weaknesses causing unauthorised disclosure, distortion, disruption, or data destruction by allowing users to perform actions, not within their respective limits [1]. Such weaknesses of broken authentication and unauthorised access have been reported in more than 318,000 web applications [1]. The percentage of vulnerabilities due to broken access control is increasing [2], as shown in Figure 1. These vulnerabilities exist in web applications due to inadequate design, hardcoding of access control and rights, and overlooking security best practices during the software development life cycle [3]. Moreover, frequent changes in software applications to patch security vulnerabilities bring new challenges for testing and removing the bugs [4–6].

Security patterns are commonly used in web applications and framework to address the problem of broken access [7]. In literature, the design and description of security patterns are commonly characterised by classes consisting of an interface class, abstract class, method, data members, and their respective implementations in concrete classes. The UML description, code examples, and design recovery regarding security patterns are not yet as mature as their counterparts in the software design patterns [8]. The organisation, structure, relationships, and dependencies of member data, methods, and classes in a security pattern represent a recurring design concept that can be utilised to verify and validate before implementation [9]. A sample code may not accompany the security pattern description for implementation. Therefore, during the software design phase, the class diagram of the security patterns requires a thorough examination to vouch for any unintentional security breach or hidden vulnerability. This vulnerability may be caused at run time due to the establishment of undesired connections among various objects of a class diagram. An attacker may exploit such a connection and access confidential data with time. The security patterns, thus applied, need to be verified for their consistency and usability. The class diagram in the design document provides a structural and behaviour view of member elements and functions. If not
applied appropriately, the properties of relationships and accessibility lead to unauthorised access to resources by the users. This leads to a hidden vulnerability that goes unnoticed until the attacker exploits the vulnerability. For example, consider an operation update op1 on a data element d1 is meant only for a role Rv. In contrast, other operation read op2 on the same data element d1 is meant for various Rv, Rp, and Re roles. The secure visitor pattern is applied to ensure role-based data access. However, suppose the accessibility (public, private, protected, and friend) of data elements in abstract and concrete classes is not defined appropriately. In that case, it will allow restricted roles to access the data elements and operations for which the role is not authorised.

Moreover, certain security constraints are not shown in the class diagram but are written in textual statements in security patterns. The textual statements that could not be depicted in the class diagram also go unnoticed during the implementation of the security pattern. These security issues leave vulnerability that goes unnoticed even during rigorous security testing. The attackers take advantage of these hidden security vulnerabilities and take over resource control.

This paper proposes an approach to verify the relationships and accessibility among various objects created during the execution of a security pattern. The proposed approach extracts the microarchitecture from the class diagram of the security pattern applied. The security constraints specified in the security pattern description are written in first-order predicate logic. The proposed approach has been applied to secure visitor, secure strategy factory, and authenticator patterns to check the consistency of relationships among various concrete elements of the abstract security pattern. The extracted microarchitecture and security constraints are analysed by generating their instances using Alloy. The experiments show that connections indicating unauthorised access and broken access control are detected in one or more instances. The detected instance is rectified using the proposed approach, and a metamodel is generated for the security pattern. The approach will facilitate the appropriate implementation of security-dependent logic in a web application and helps in the identification of hidden vulnerabilities at an early stage.

The rest of the paper is organised into four sections describing Related Work, Proposed Methodology, Experimental Results, Analysis, and Discussion, and Conclusion and Future Work.

2. Related Work

In this section, various approaches for verifying and validating security patterns have been discussed. Dong et al. [10] represented the composition of security patterns using Calculus of Communicating System- (CCS-) based model checker of sequence diagrams. The approach proposed by Dong et al. [10] verified the states and its transitions in the sequence diagram of a security pattern. Mourad et al. [11] proposed an aspect-oriented two-phased approach to verify integrated security patterns in an application. The approach defines security objects, methods, and events and manually verifies them with security requirements. Pedroza et al. [12] proposed a verification approach using block and state machine diagrams in the SysML environment for the safety and security of critical real-time embedded systems.


He and Fu [22] modelled and analysed six security patterns, namely, account lockout, authenticated session, client data storage, encrypted storage, password authentication, and password propagation, to check their completeness, consistency, and ambiguity in their textual descriptions. He and Fu [22] used high-level petri nets to ensure the correct implementation of these six security patterns. Near and Jackson [23] showed that previously unknown security bugs could be easily identified using their proposed formal approach SPACE (Security Pattern Checker), which finds implementation bugs in access control security patterns. In an approach to improve security pattern definition, Beherens [24] provided abstractions and their implementations using formalised notation. The constraints were represented as a finite state machine recognised by regular language for analysis and verification.

Berghe et al. [25] focused on defining security patterns using a modelling language and proposed four data-specific building blocks, namely, data types, data flows, data
creation, and data storage, to support security patterns. Security analysis of the web in terms of cache usage, temporal logic, and state transitions was presented by Shimamoto et al. [26]. Shimamoto et al. [26] also verified Web Deception attack, CSRF attack, and exact origin and cross-origin Browser Cache Poisoning (BCP) attack using Alloy and temporal logic syntax. Obeid and Dhaussy [27] presented a message, resource, and access-based approach for formalisation, verification, and composition of security patterns with increased complexity measures.

Gadouche et al. [28] used Event-B correct-by-construction methodology to specify declarative and behavioural aspects of Role-Based Access Control. Gabillon et al. [29] designed a model for representing dynamic and contextual authorisation rules using first-order predicate logic for security administration and policy in the Internet of Things. Gupta et al. [30] proposed a formal approach to represent security constraints of a security pattern using first-order predicate logic. The formal specification facilitated early detection of hidden security vulnerabilities in a software or a web application.

The approaches available in the literature for specification and verification of security patterns are based on transitions and a set of actions and cover few security patterns. The existing approaches have considered transactions, temporal logic, request, response, and resource messages to verify the security of an application. As the application grows in size bringing variations in code, the greater efforts are required to verify security properties and detect vulnerabilities [31]. However, the existing approaches grow exponentially as the number of transitions and states becomes larger in terms of complexity and execution time. The relationships and security-related constraints among various objects created in the execution of security patterns have not been analysed in existing approaches. The existing approaches have not verified the behavioural aspect of security patterns among various instances of concrete objects.

### 3. Proposed Approach

This paper proposes an approach to verify secure access control and detect unauthorised and unauthorised access arising from the inadequate implementation of security patterns. The class diagram of the security pattern available in literature has been used to extract its microarchitecture in the proposed approach. The microarchitecture of the class diagram contains structural properties such as interface class, abstract class, concrete class, methods, fields, and their respective relations and accessibility. An interface class is identified along with its member functions, data, and accessibility specifiers. The approach then identifies the abstract class and its concrete implementations. The private and protected data fields and member functions are identified for every concrete class. The private and protected member elements are the restricted elements to be verified for their non-accessibility from other elements. Subsequently, other member elements with public and friend accessibility are also identified and checked for the parameters passed and consistency. The interface, abstract, and concrete classes are analysed for the relationships of inheritance, association, composition, aggregation, and creation of objects.

Further, security constraints are written in first-order predicate logic and modelled with the microarchitecture. For example, suppose in a security pattern, it is defined that a concrete class should not have a public method. In that case, it is written as \( \exists \text{public}(\text{ConcreteClass}) \), in which \( \exists \) is a negative existential quantifier of predicate logic. Similarly, a class may have only one member function and the predicate logic for same is Operations \( (\text{ConcreteClass}X) = \{\text{OpX}\} \). The identified microarchitecture is then analysed and executed using Alloy Specification Language. The security pattern is then executed by creating multiple instances and objects. Each execution is verified for the existence of any counterexample instance in its respective instance. If a counterexample is detected, the constraints and structure are rectified suitably to present an accurate design of security pattern. All nodes in an instance are checked for reachability [32, 33] to verify the complete and unambiguous implementation of the security pattern. The proposed approach for verifying applied security patterns and identifying any hidden vulnerabilities is shown in Algorithm 1.

### 4. Experiments, Results, and Analysis

The proposed approach has been applied to three security patterns: authenticator, secure visitor, and secure strategy. The microarchitecture has been extracted and detailed in Section 4.1 for each applied security pattern in the first step. The subsequent modelling, execution of multiple instances, results, and analysis for each security pattern have been discussed.

#### 4.1. Extracting Microarchitecture of Security Patterns

##### 4.1.1. Authenticator Pattern

The class diagram of the authenticator pattern [34] is shown in Figure 2(a). The microarchitecture from the class diagram of the authenticator pattern is extracted by identifying its classes, member functions, security constraints, and relations and is shown in Figures 2(b) and 2(c).

##### 4.1.2. Secure Visitor Pattern

The secure visitor pattern class diagram [35] is shown in Figure 3(a). The secure visitor pattern separates conditional security logic from the business logic in hierarchical data nodes in a web application. It enables data nodes to authenticate the visitors requesting their access for certain operations. The usage of the secure visitor pattern helps prevent unauthorised access to data by implementing security logic in a separate code segment or class. The secure visitor pattern provides a solution to prevent such attacks by making data nodes lock themselves from being read by a visitor unless the visitor supplies the proper credentials to unlock the data node. The secure visitor pattern consists of an interface, abstract, and concrete classes representing secure visitors, unlocked and locked data nodes, various member data and functions, and their respective access specifiers. The abstract
function 'accept()' is implemented in concrete classes of locked data node type(s) that, in turn, unlock the respective data node after checking a user’s credentials. The microarchitecture of the secure visitor pattern extracted is shown in Figure 3(b). Security constraints such as for every locked data node, there should be an unlocked data node, restriction of access to parent and child data nodes, and restrictions on member functions of abstract and concrete classes are represented in predicate logic and shown in Figure 3(c). These security constraints are crucial for providing secure access and preventing broken access in web applications.
### 4.1.3. Secure Strategy Factory Pattern

The class diagram of the secure strategy factory pattern [35] is shown in Figure 4(a). The secure strategy factory pattern separates the security-dependent logic associated with each role from the basic functionality of object creation and selection.

Secure strategy factory implements the creation and selection of an object for executing an operation depending on a set of security credentials. The given security credentials are used to select and return the role-specific object. The different secure functions are implemented in various concrete classes:

- **Concrete classes: VisitorX, UnLockedDataNodeX, LockedDataNodeX**
  - Where X ranges from 1 to n.

#### Security constraints in first order predicate logic

- **a)** Association is from UnlockedObject to UnlockedDataNode
  - a. type (UnlockedObject) = Object ∧
  - b. type (UnlockedDataNode) = UnlockedDataNode ∧
  - c. multiplicity (UnlockedDataNode) = “**”

- **b)** Client depends only on SecureVisitor and UnlockedObject
  - a. Such that:
    - i. access (SecureVisitor, UnlockedObject),
    - ii. [UnlockedDataNode] ∪ subclasses (UnlockedDataNode) ∪ subclasses (SecureVisitor)

- **c)** Accessor is the only operation in UnlockedDataNode
  - i. Operations (UnlockedDataNode) = [accessor]

- **d)** For every subclass of UnlockedDataNode, there exists a subclass belonging to LockedDataNode and an operation VisitX belonging to the subclass of SecureVisitor. The function visit () takes the parameter UnlockedDataNode of typeX as its subclass such that each concrete node in the data hierarchy has both a locked and unlocked version. Only a locked node will be able to create an unlocked node.
  - ∀x ∈ DataHierarchy ∃! x (l, u) such that
    - ∃ u ∈ subclasses (UnlockedDataNode) ∧ ∃ l ∈ subclasses (LockedDataNode) ∧
    - create (l, u)

- **e)** No UnlockedDataNode should be able to access its parent or child data node.
  - ∀x ∈ subclasses (UnlockedDataNode)
    - ¬ access (x, x.child) ∧ ¬access (x, x.parent)

- **f)** LockedDataNodes has no public operations
  - ∄ ispublic (operations (LockedDataNode))

- **g)** To access UnLockedDataNode, visitor should call LockedDataNode, which in return create UnLockedDataNode after checking user credentials.
  - ∀x ∈ DataHierarchy
    - ∃ accept ∈ operations (LockedDataNodeX) ∧
    - ∃ checkCredentials ∈ operations (LockedDataNodeX)
    - such that
      - if return(checkCredentials)=true
      - create (l, u)

---

**Figure 3:** Extracting microarchitecture of secure visitor pattern.
implementations of abstract secure strategy factory for each set of roles defined by the security requirements. For example, in a web application, there are three roles having complete, little, or no trust, and then, three concrete implementations of secure strategy will be created, one for each level of trust. A concrete implementation of secure strategy factory will only contain functionality restricted to a secure role or trust level. The microarchitecture of the secure strategy factory pattern is extracted using the proposed approach as in Figure 4(b), and the security constraints written in first-order predicate logic are shown in Figure 4(c).

4.2. Detection of Broken Access Control. In this section, the class diagram of the security pattern is executed in the Alloy tool to detect any violation of security requirements. Each pattern has been executed for multiple instances to create objects, and each instance is analysed for the existence of any counterexample. On detecting a counterexample, the nodes in the graph are analysed for the inappropriate link. The security pattern is then rectified by correcting the definition of concrete classes and methods and implementing the security predicate to verify the security requirement.

4.2.1. Authenticator Pattern. The class diagram of authenticator pattern is executed to detect any hidden vulnerabilities. In Figure 5, while executing two or more concrete authenticators, it is found that concrete authenticator0 is creating an object for the concrete authenticator1. It is detected that the concrete object of concrete authenticator1 is inheriting the abstract method instead of overriding it. It is also detected that an object created by one authenticator can create an object of another authenticator by an inherited method of abstract authenticator. The security error is rectified by ensuring that pred creates $x : \text{ConcreteAuthenticator}$, $y : \text{ObjectA}$ and overriding the method create() in each concrete authenticator.

4.2.2. Secure Visitor Pattern. The secure visitor pattern class diagram is executed, and hidden vulnerabilities are detected using the proposed approach. The predicate logic for the security constraint is that only a locked data node can create its unlocked data node. However, on the execution of the predicate visit for two or more instances in secure visitor, an instance is found in Figure 6 that shows unlocked data node2 could unlock data node1. The instance creates unauthorised access to data node2 via data node1, against the security requirements. On analysing the microarchitecture, it is detected that the function visit() has been implemented inappropriately.

$$\text{pred visit} \{ v : \text{SecureVisitor}, l : \text{LockedDataNode} \} \{}.$$ (1)
The pattern is rectified by incorporating the predicate for accepting using credentials and creating an unlocked data node that the user will visit. Predicate create and assertion unique is added in the implementation of secure visitor pattern for specifying the constraint. One locked data node will create only one unlocked data node, and unlocked data node should not access are not or child data nodes.

\[ \text{pred accept} \forall v : \text{SecureVisitor}, u : \text{UserCredentials} \{ \}, \text{pred create} \forall l : \text{LockedDataNode}, u : \text{UnLockedDataNode} \{ \} \].

4.2.3. Secure Strategy Pattern. The class diagram of secure strategy factory is executed to detect any hidden broken access to operations. The secure strategy factory concrete classes are built for two trust levels. TrustLevel-1 can perform op11 and op12, and TrustLevel-2 can operate op21. On executing the predicate created for two or more instances in secure strategy factory, an instance is found such that TrustLevel-1 could access the operation op21 restricted for TrustLevel-2. The broken instance is detected using the proposed approach while implementing the secure strategy factory pattern and is shown in Figure 7, without checking user credentials. It is detected through the broken connection that op21 has been inappropriately defined in class meant for TrustLevel-1 and called by TrustLevel-2. The security pattern is rectified by implementing the predicate that \( \forall x \in \text{LevelAccess} \exists y(f, m) \) such that

\[ \exists f \in \text{subclasses}(\text{SecureStrategy}) \land \exists m \in \text{subclasses}(\text{secureMethod}) \land \text{define}(f, m, x) \].

Figure 5: An instance generated for the authenticator pattern showing the creation of multiple objects and broken authentication.

Figure 6: A broken instance detected through the proposed approach in the implementation of secure visitor.

Figure 7: A broken instance detected through the proposed approach in the implementation of the secure strategy factory.
4.3. Metamodel of Authenticator Pattern, Secure Visitor Pattern, and Secure Strategy Pattern. In this section, the extracted microarchitecture of the security pattern and its rectified predicate logic is executed in the Alloy tool to verify that it meets the security requirement. Each pattern has been executed for multiple instances to create objects, and each instance is analysed for the existence of any counterexample. The metamodel of the authenticator pattern is generated and shown in Figure 8(a). Consider two different authentication systems in a web application. Each of the two authentication systems is implemented using a separate concrete class. The concrete authenticator defines its concrete object and authentication mechanism. The separation of different authentication mechanisms ensures that the other authenticator does not create the object of one authenticator. The arrangement is easily extendible if a third or more authentication mechanisms are appended to the application in future versions. The class diagram of the authenticator pattern is verified using the security constraints written in first-order predicate logic as an assertion unique.

In the metamodel of secure visitor pattern shown in Figure 8(b), the client requests SecureVisitor Interface to visit UnlockedDataNode through its concrete implementations Visitor1 or Visitor2. SecureVisitor can implement any createA authA this/ConcreteAuthenticator this/authenticate extends extends extends extends this/create this/get this/ObjectA this/ObjectFactory

\[\text{requests: 1 unlock: 1 visit1: 1 visit2: 1} \]

Figure 8: The metamodel generated by the proposed approach: (a) authenticator; (b) secure visitor; (c) secure strategy factory.
Strategy Factory is verified. Operations op11 and op12 are performed by the secure visitor with TrustLevel-2. The user with TrustLevel-2 can perform actions op21 but not TrustLevel-2 after checking the credentials. Accordingly, a secure visitor with role1 can perform operations op11 and op12 but not operation op21.

Similarly, SecureStrategy ss2 creates an object for a user with TrustLevel-1 after checking the credentials. Accordingly, a user with role1 can perform actions op11 and op12. TrustLevel-1 can perform operations op11 and op12.

LockedDataNode accepts concrete visitor and user credentials, each role can perform the operations restricted to its role. The class RBAC contains policies for TrustLevel and operation mapping. It defines the operations that are restricted to a particular TrustLevel. In the present instance, TrustLevel-1 can perform operations op11 and op12. TrustLevel-2 can perform operation op21. SecureStrategy ss1 creates an object for a user with TrustLevel-1 after checking the credentials. Accordingly, a user with role1 can perform actions op11 and op12 but not operation op21. Similarly, SecureStrategy ss2 creates an object for a user with TrustLevel-2 after checking the credentials. Accordingly, a user with TrustLevel-2 can perform actions op21 but not operations op11 and op12. The security feature of the secure strategy factory is verified using the predicate logic \( \forall x \in LevelAccess : \exists f(x, m) \) such that

\[
\exists f \in \text{subclasses}(\text{SecureStrategy}) \land \\
\exists m \in \text{subclasses}(\text{secureMethod}) \land \\
\text{define}(f, m, x).
\]

### 4.5. Analysis and Discussion

The proposed approach represents the design, structure, dynamics, and other dependencies among various components of security patterns in the generated metamodel. The approach will enable web developers to implement code accurately and minimize threats to the web application. The proposed approach provides a general model that applies to every domain. It considers structural aspects and interrelationships among various structural elements and analyses behaviour based on possible relationships, states, transitions, and execution steps. The proposed approach verifies security patterns by defining the set of interface, abstract, and concrete classes and static methods and nonstatic method calls on the class and identifier and all dependencies at the design phase. It will be helpful in verifying applications. All security-related constraints, artefacts, and static and dynamic checks mentioned at different sections of a security pattern are represented and validated using the predicate modelling before implementation.

It provides precise semantics, verification and validation, and automated reasoning and is machine incomprehensible.
due to the usage of predicate logic. This specification facilitates automated verification and validation of security patterns applied in web applications. For example, in the secure strategy factory pattern, the constraint for every trust level, i.e., complete, partial, and none, should be a separate unique ConcreteObject implementing each trust level. This constraint is specified in predicate as $\forall x \in \text{TrustLevel} \exists y \in \text{ConcreteObject}$. Such representations in predicate logic are easily verified using automated tools and help identify any counterexample(s) simple and error-free.

The proposed specification and its metamodel of security pattern defined various structural elements, their relations, and how these elements will communicate using directed edges and appropriate markers. This form of representation enables developers to understand the complete structure. It helps implement an effective and productive design that is instantly easy to verify, validate, and correct at early stages to ensure security properties of confidentiality, integrity, availability, accountability, nonrepudiation, authentication, and authorisation. The proposed approach provides a verifiable specification of structural design and implementation of security patterns covering all security patterns. Moreover, security constraints such as accessibility among micro architecture elements and parent and child nodes that require extraordinary validation, verification, and testing are also analysed.

In the proposed approach, any flaw in the system’s design concerning security functionalities and role-based access to resources is found early through structural and behaviour validations, thereby preventing any consequent security breach.

5. Conclusions and Future Work

Security breach due to broken authentication and access control has been an essential concern for web developers. Though security patterns are applied in a web application, there is a need for a method that can verify the correctness of the class diagram built during the design phase. In this paper, the proposed approach verifies the relationships and accessibility among objects and classes of a security pattern applied in a web application. The approach extracts the structural properties, relations, associations, security-related constraints, artefacts, and static and dynamic checks of the class diagram of a security pattern. The extracted microarchitecture is executed using Alloy to identify unauthorised and broken access in the security pattern microarchitecture is executed using Alloy to identify unauthorised and broken access in the security pattern. The complex class diagram of the security pattern can be quickly evaluated and verified for secure access control during the design stage of the web application.

The authors intend to extend the approach to verifying the composition of security patterns in web applications in the future.

Data Availability

The proposed approach has been applied to security patterns: Authenticator, Secure Visitor, and Secure Strategy available at [34, 35].

Conflicts of Interest

The authors declare that they have no conflict of interest.

References


