AuthCheck: Program-state Analysis for Access-control Vulnerabilities

Goran Piskachev[1], Tobias Petrasch[2], Johannes Späth[1], and Eric Bodden[1,3]

[1] Fraunhofer IEM, Germany
{goran.piskachev,johannes.spaeth}@iem.fraunhofer.de
[2] BCG Platinion, Germany
petrasch.tobias@bcgplatinion.com
[3] Paderborn University, Germany
eric.bodden@upb.de

Abstract. According to security rankings such as the SANS Top 25 and the OWASP Top 10, access-control vulnerabilities are still highly relevant. Even though developers use web frameworks such as Spring and Struts, which handle the entire access-control mechanism, their implementation can still be vulnerable because of misuses, errors, or inconsistent implementation from the design specification. We propose AuthCheck, a static analysis that tracks the program's state using a finite state machine to report illegal states caused by vulnerable implementation. We implemented AuthCheck for the Spring framework and identified four types of mistakes that developers can make when using Spring Security. With AuthCheck, we analyzed an existing open-source Spring application with inserted vulnerable code and detected detected the vulnerabilities.

Keywords: static analysis, access control, authentication, authorization, web systems, security

1 Introduction

With increasing popularity and amount of processed data, web applications are attractive targets for attackers. The access-control vulnerabilities are still ones of the most relevant as rankings show. For instance, five of the SANS Top 25[4] most dangerous vulerabilities are related to access-control. On the OWASP Top 10[5] ranking, on place two is *broken authentication* vulnerability and on place five is *broken authorization* vulnerability.

Nowadays, web frameworks are heavily used by software developers [19]. Modern frameworks, such as Spring [6] and Struts. [7] provide mechanism for access-control making developers' implementation effort smaller. At runtime, the actual

[4] https://cwe.mitre.org/top25/
[5] https://www.owasp.org/index.php/Top_10-2017_Top_10
[6] https://spring.io/
[7] https://struts.apache.org/

access-control checks of such mechanism are performed within frameworks' code, hereby software developers do not need to write customized access-control code and implementation bugs are avoided.

Instead of writing access-control code manually, frameworks allow software developer to specify the access rules via framework specific APIs. Spring, for instance, provides a fluent interface with specification language SpEL [3] combined with Java annotations to allow the specification of access rules.

However, implementing the access-control rules using the frameworks APIs according to a design specification, created by the software architect, remains a challenging task. The access-control is often a combination of annotations of methods, a specifications in a configuration class, and a set of permission groups for the resources of the system (i.e., URI). The resulting access control of the implementation easily diverges from the design specification and the application may accidentally grant an unauthorized user access to confidential data.

In this paper, we propose a typestate-inspired analysis for detecting three access-control vulnerabilities:

– *CWE-306* missing authentication [8] - The system does not perform an identity check on a request to a resource which by design should be accessed only be identified requests.
– *CWE-862* missing authorization [9] - The system does not perform a check whether an authenticated request has the correct rights to access a resource.
– *CWE-863* incorrect authorization [7] - The system performs an authorization check on the resources, but this check is wrong.

Our static analysis uses *finite state machines* (FSMs) of each vulnerability to track the authorization state of the program. The state changes are triggered by method calls that authorize the user or access a critical resource along the control flow paths.

The main contributions of this paper are:

– AuthCheck: a program-state analysis for access-control vulnerabilities,
– an implementation of AuthCheck for the Spring Security framework,
– a running example and four typical errors in Spring Security, and
– a case study demonstrating the applicability of the implementation.

The following section introduces our running example within the Java Spring framework. In Section 3, we provide background information and definitions for the AuthCheck approach, which is then introduced in Section 4. Implementation details are discussed in Section 5. A case study and limitations are discussed in Section 6.

2 Running Example

As running example consider a minimal web-application that helps a user to organize her tasks. An anonymous user browsing the web application must only see the web applications version number. A user that is authenticated can view tasks assigned to herself. An administrator (group *ADMIN*) can create new tasks for a particular user.

Table 1 details the design specification of the web-application's REST-API [10]. The specification maps the URI of an incoming request to the actual API method which shall be invoked to process the incoming request. Table 1 additionally details the permissions required for each request. A software architect specifies these requirements and hands them to a software developer.

Table 1: Specification resources and access rules in the running example

HTTP	URI	Resource	Description	Access rule
GET	/version	version()	Returns application's version.	No rule
GET	/profile	profile()	Returns user profile.	Authenticated
GET	/task	retrieveAll()	Returns list of all tasks.	*USER* or *ADMIN*
CREATE	/task	create()	Creates new task.	*ADMIN*

Spring-based Implementation The software developer uses the Spring framework[1] to implement the software as specified. Spring provides a security component [2] that ships with a mechanism for access control of resources. Spring handles requests from users via chain of filters (chain of responsibility design pattern [11]). The requests are matched and processed based on their URIs.

```
public class WebSecurityConfig extends
    WebSecurityConfigurerAdapter {
@Override
protected void configure(HttpSecurity http) throws
    Exception {
http.csrf().disable().sessionManagement()
.sessionCreationPolicy(SessionCreationPolicy.STATELESS)
.and().authorizeRequests()
.antMatchers(HttpMethod.GET, "/version").permitAll()
.antMatchers(HttpMethod.GET,
    "/task").access("hasAnyRole('USER', 'ADMIN')")
.antMatchers(HttpMethod.CREATE, "/task").hasRole("USER")
.antMatchers(HttpMethod.GET,
    "/profile").authenticated().and().httpBasic();
}}
```

Listing 1.1: Resource and access-control configuration of the running example implemented with Spring Security

Listing 1.1 shows the implementation of Table 1 using Spring Security. By the use of a fluent interface the developer can implement the chain of filters that is applied upon each incoming request at runtime. Each filter is created through the method `antMatcher(..)` defined by the HTTP method and the URI of the resources which that filter can process. The `permitAll()` method allows any request to access the resource. The `authenticated(..)` method creates a filter that restricts the incoming request to the one where the useer is authenticated. The `hasRole(..)` method allows access to the resource by any request that has the role of the specified group. The `access(...)` method evaluates the specified argument which has to be defined in the *Spring Expression Language (SpEL)* [3], and when evaluated to true, allows the corresponding request to access the resource.

The implementation has inconsistency with the specification. The software developer erroneously allowed basic users (*USER*) to create new tasks as opposed to restricting the action to *ADMIN*s only. AuthCheck detects the deviation from the specification automatically.

3 Background and Definitions

3.1 Typestate Analysis and Program-state Analysis

Typestate analysis [20] is a data-flow analysis that can detect invalid states of objects from the code being analyzed. The analysis uses specification of all possible states of the object, typically expressed as *final state machine (FSM)*. For example, using the *FSM* of the type *java.io.FileWriter*, in a given program, the analysis can report if any object of type *java.io.FileWriter* is not closed at the end of the program. Another example is CogniCrypt [13], a typestate analysis for detecting API misuses of cryptographic libraries.

To detect access-control vulnerabilities, such as *CWE-306* [8], *CWE-862* [9], and *CWE-863* [7], we designed a program-state [5] [12] analysis. Similar to the typestate analysis, the program-state analysis uses *FSM* to track the states, not of single objects, but the state of the program. Figure 1 shows the *FSM* that models the program states when detecting *CWE-306*. Based on our running example (Section 2), the *acm()* (authentication-critical method) is replaced by one of the resources, e.g. *profile()*. The legal states are *NA* (not authenticated) and *A* (authenticated). The *init()* transition models the entry point of the analysis, which in this case is the arrival of a request from a user. If the request is for the resource *profile()*, the application has to make sure that the call to the method *authenticate()* from Spring was successfully called before. This is modeled by the transition with label *authenticate()*. If this transition was fired, the state of the *FSM* will be changed from *NA* to *A*. In case, the implementation of the application does not contain a call to the method *authenticate()*, when the resource *profile()* is requested, the *FSM* will go to the state *CWE-306*, which models an illegal state and this can be reported.

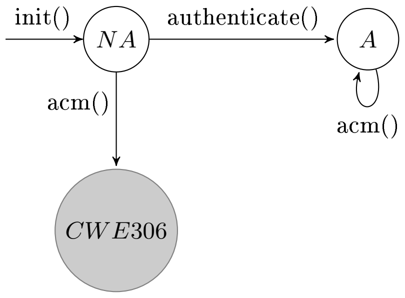


Fig. 1: *FSM* for missing authentication *CWE-306*

3.2 Definitions

Before we introduce the AuthCheck approach (Section 4), we define the term *web application*. In the following, we introduce the required terms. A user is a client program, e.g. web browser, that can send requests to the server.

Definition 1. *Authorization group*
Authorization group g is boolean characteristic of a user u with a unique name and access rights. A user can belong to more authorization groups. The set of all authorization groups is G:

$$G = \{g_i \mid g_i \textit{ is an authorization group}, 1 \le i \le m,\ m \in \mathbb{N}\}$$

The function *userGroups*: $U \rightarrow Pow(G)$ maps each user $u \in U$ to the authorization groups. $Pow(G)$ is the power set of G. We define the help function *hasRole*: $U \times G \rightarrow \mathbb{B}$, that expresses whether a user u belongs to an authorization group g: $hasRole(u, g) := g \in userGroups(u)$

Each user that is authenticated in the system belongs to the special authorization group ANONYMOUS.

Authorization formula is a boolean formula a, formed by the function *hasRole*, *true*, *false*, and the operators $\vee, \wedge, \neg$.

Definition 2. *Resource*
An authentication and authorization critical resource is a 4-tuple $r = (m, p, s, a)$, where m is HTTP method, p is URI*, s is a method signature, and a is an authorization formula that defines the access rule of the resource. Access to the resource is given when a is evaluated to true for a request of a user u. Users identify each resource with the* URI *p and the HTTP method m. The coresponding method in the system is identified by the signature s.*

Definition 3. *Web application*
A web application W, is 2-tuple $W = (R, G)$, where R is a set of resources and G is a set of authorization groups.

Example The web application from Section 2 has the authorization groups *ADMIN*, for administrators and *USER*, for basic users. By default, it also has the *ANONYMOUS* group. Thus, $G = \{ANONYMOUS, ADMIN, USER\}$. The set of resources has 4 elements (Table 1). The first resource is defined as $r_1 = (GET, /version, String\ version(), a_1)$, where $a_1(u) = true$.

We consider a user u with $userGroups(u) = \{ANONYMOUS, USER\}$. If this user requests the resource r_1, the access will be allowed because $a_1(u) = true$. However, a request to the resource r_4 will be denied because $a_4(u) = hasRole(u, ANONYMOUS) \wedge hasRole(u, ADMIN) = false$.

4 Approach

We present AuthCheck, a program-state analyis for detecting three access-control vulnerabilities, *CWE-306*, *CWE-862*, and *CWE-863*. The analysis uses a call graph of the program (deatailed in Subsection 5.2) and an *access-control specification model (ACSM)*, like the one in Table 1. *ACSM* is defined as a web application $S = W_S$, where $W_S = (R_S, G_S)$ (Definition 3). *ACSM* can be created manually by software architects or automated from requirements and design specifications. Either way, we assume that the following information is available: resource API, URI, and access rule, that is aware of the autorization groups in the system.

AuthCheck checks whether the call graph confirms the *ACSM* by checking each path from the call graph. To extract all paths, the depth first search *DFS* algorithm is used. AuthCheck uses *FSM* for each vulnerability, e.g. Figure 1. Algorithm 2 shows the tracking of each path with the *FSM*. The *FSM* starts in the initial state (e.g. *NA* in Figure 1) and for each node of the path a new state of the *FSM* is calculated (line 4 in Algorithm 2). If an error state is reached (e.g. *CWE-306* in Figure 1), a new vulnerability will be reported.

For each path, the function *DetectVuln* is called which is defined by Algorithm 2. *DetectVuln* uses the *FSM* to analyse the path.

Algorithm 1 Check the call graph against vulnerabilities

```
1: function CheckCallGraph($CallGraph, FSM$)
2:     $Paths \leftarrow DFS(CallGraph)$
3:     $Vul \leftarrow \emptyset$
4:     for each $p \in Paths$ do
5:         $Vul \leftarrow Vul \cup$ DetectVuln($p, FSM$)
6:     return $Vul$
```

The complexity of Algorithm 1 is $\mathcal{O}(|V| + |E| + |P| \cdot T(\text{DetectVuln}))$, where V is the number of nodes, E is the number of edges, and P is the number of paths in the call graph. In *DetectVuln*, every node of the path is analyzed, resulting in $\mathcal{O}(|P|)$. The worst case path is the one with all nodes from the call

Algorithm 2 Checking each path against vulnerabilities

```
1: function DetectVuln($Path, FSM$)
2:     $v \leftarrow FSM \rightarrow init()$
3:     for each $n \in Path$ do
4:         $v \leftarrow FSM \rightarrow nextState(n)$
5:         if $v \in FSM.ERROR\_STATES$ then
6:             return new $Vulnerability(v)$
7:     if $v \in FSM.NOT\_ERROR\_STATES$ then
8:         return new $Vulnerability(v)$
9:     return $\emptyset$
```

graph $|V|$. Additionally, the number of paths in the worst case is $|E|$. Thus, the total complexity of Algorithm 1 is

$$\mathcal{O}(|V| + |E| + |P| \cdot |V|) = \mathcal{O}(|V| + |E| + |V| \cdot |E|) = \mathcal{O}(|V| \cdot |E|)$$

In the following, we discuss the three the *FSM* used by AuthCheck.

Missing Authentication A program is vulnerable to *CWE-306* when an authentication-critical method (*acm()*) can be accessed by user that has not been authenticated before. AuthCheck models this vulnerability as shown in Figure 1. Authentication critical methods are all resources that in the *ACSM* have an access rule that requires authentication. The error state in Figure 1 is reached when an authentication-critical method is processed next in a given path and the current state of the *FSM* is *NA* (not authorized). In this case, the program-state analysis will create a vulnerability (Algorithm 2, line 8).

Missing Authorization and Incorrect Authorization *CWE-862* occurs in a given program when a non-authorized user u can request an authorization-critical method (*azcm()*). If the user is authorized but the belonging group g does not confirm the access rule for that authorization-critical method as specified in the *ACSM* ($hasRole(u, g) = false$), then incorrect authorization occurs (*CWE-863*). Figure 2 shows the *FSM* that AuthCheck uses to model *CWE-862* and *CWE-863*. The transitions with the label *azcm()* without an argument denote calls to an authorization-critical method when the user is not authorized. When there is an argument g, the user has been authorized and the belonging group is being checked. This happens in state *A2*. When the user's group evaluates to true the self transition of state *A2* is fired, otherwise the transition to state *CWE-863* is fired. AuthCheck performs a group hierarchy check.

Strategies for detecting critical methods The transitions *acm()* in Figure 1 and *azcm()* in Figure 2 denote an authentication-critical and authorization-critical method. These methods correspond to the resources defined in the *ACSM*. In the following, we discuss AuthCheck's strategies for detecting these methods in the call graph.

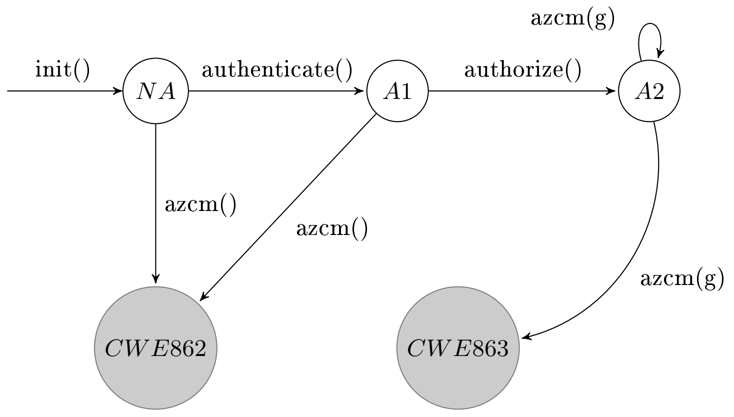


Fig. 2: *FSM* for missing authorization *CWE-862* and incorrect authorization *CWE-863*

Algorithm 3 Identifying methods as authentication-critical

```
1: function isMethodAuthenticationCritical($R, s'$)
2:     for each $r \in R$ do
3:         if $r_s = s'$ then
4:             return $true$
5:     return $false$
```

In the case of *CWE-306*, the authentication-critical methods are detected by iterating the set of all resources R from the *ACSM* for each method M that is currently processed in the path. The complexity for this strategy is $\mathcal{O}(|M| \cdot |R|)$.

Algorithm 4 shows the AuthCheck strategy to identify the authentication-critical methods in the call graph for *CWE-862*. When checking the *CWE-862*, each method M currently processed in the path is classified as authorization-critical if the method is contained in the *ACSM* as a resource and the access rule is not tautological. The evaluation of the authorization formula depends on the number of relevant authorization groups used in the authorization formula. For the calculation, all possible combinations $\forall\ g \in Pow(G')$ of relevant authorization groups G' must be evaluated. For a worst-case authorization formula with $|G|$ authorization groups, the resulting complexity is $\mathcal{O}(|M| \cdot |R| \cdot 2^{|G|})$.

Algorithm 5 shows the AuthCheck strategy to identify the authorization-critical methods in the call graph. For each resource in the *ACSM*, it checks whether its signature matches the signature of the method M currently processed in the call graph. In addition, the authorization formulas are checked. The runtime depends on the number of relevant authorization groups. For the calculation, all possible combinations $\forall\ g \in Pow(G')$ of relevant authorization groups G' must be evaluated. For a worst-case authorization formula with $|G|$ authorization groups, the resulting complexity is $\mathcal{O}(|M| \cdot |R| \cdot 2 \cdot 2^{|G|})$.

Algorithm 4 Identifying methods as authorization-critical

```
1: function isMethodAuthorizationCritical($R, s'$)
2:     for each $r \in R$ do
3:         if $r_s = s'$ and $r_a$ is not tautological then
4:             return $true$
5:     return $false$
```

Algorithm 5 Identifying methods as authorization-critical and group-belonging

```
1: function isMethodAuthorizationCritical($R, s', a'$)
2:     for each $r \in R$ do
3:         if $r_s = s'$ and $eval(r_a) = eval(a')$ then
4:             return $true$
5:     return $false$
```

5 Spring Security AuthCheck

We implemented the AuthCheck concept from Section 4, as a Java application that checks the implementation of a given Java Spring Security application and a given *ACSM*. We used the Soot framework [14] for the analysis. In the following, we discuss the architecture of our implementation, the insights of the call graph construction, and the four typical developer's mistakes with Spring Security that AuthCheck can detect. Our implementation is available on Github [18].

5.1 Architecture

The AuthCheck tool follows a pipeline architecture, since it consists of several sequential phases that work on shared artifacts. Our AuthCheck implementation consists of 3 phases:

1. *Call graph construction*: parses the code, the Spring Security configuration, and annotations, and constructs the call graph,
2. *Call graph update*: patches missing edges into the call graph based on Spring Security configuration,
3. *CWE analysis*: analyzes the call graph against *CWE-306*, *CWE-862*, and *CWE-863* based on Section 4.

The design an extendable architecure of the tool. Figure 3 shows the meta-model of the main components of the tool. The root class is the *Analysis* that contains all components. The *Phase* can process objects of type *Artifact*. In our implementation, the call graph instance, *FSMs*, and *ACSM* are defined as artifacts. The final results of the analysis are stored in a *Result* object which can be presented via *Presenter* object. Our tool has one presenter, that generates HTML pages (see Section 6 and Figure 4). In this architecture new phases can be added easily. Furthermore, new types of vulnerabilities can be created as *FSM* and added as artifacts in the analysis.

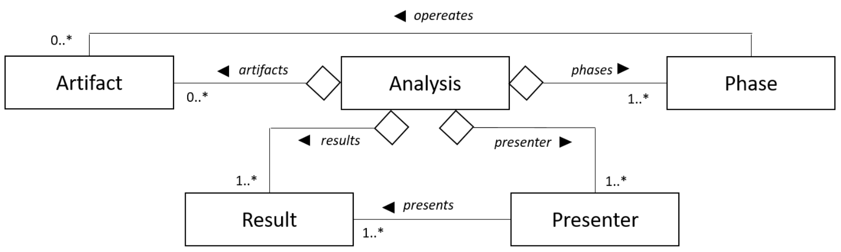


Fig. 3: UML class diagram of AuthCheck implementation for Spring Security

5.2 Call Graph Construction

Phase 1 constructs the call graph using the class hierarchy algorithm and extracts the Spring Security configuration needed in phase 2 to complete the missing edges in the call graph due to reflection. The extracted information is prepared according to Definition 3. Each critical method is annotated with its URI and HTTP method. This is transferred together with the signature of the method into a resource according to the Definition 2.

The Spring Security configuration is extracted from the program using an intraprocedural analysis. A special case is the method *access(a)*, which can take as an input a *SpEL* formula. For this, we use the Spring mechanism to evaluate the string values containg the *SpEL* formula.

An authorization formula is assigned to a resource when the defined filter matches the method and the URI. If multiple authorization formulas are applied to a resource, they are associated with a logical AND ($\wedge$).

The extracted information is stored as web application (Definition 3). Then, in phase 2, the missing edges are added to the call graph according to Algorithm 6. The algorithm gets the extracted web application W_J and generated call graph *CallGraph*. For each resource, it is checked whether the Spring Framework performs an authorization check, authentication check, or no access check. Accordingly, an edge is created to the critical method from the *authorize()*, *authenticate()*, or *init()* methods.

Algorithm 6 Adding missing edges in the call graph

```
1: function createMissingEdges($W_J = (R_J, G_J), CallGraph$)
2:     for each $r \in R_J$ do
3:         if isMethodAuthorizationCritical($R_J, r_{sig}$) then
4:             $CallGraph \rightarrow addEdgeFromAuthorize(r_{sig})$
5:         else if $isMethodAuthenticationCritical(R_J, r_{sig})$ then
6:             $CallGraph \rightarrow addEdgeFromAuthenticate(r_{sig})$
7:         else
8:             $CallGraph \rightarrow addEdgeFromInit(r_{sig})$
```

5.3 Developers' mistakes

As demonstrated in Listing 1.1, the access-control rules in Spring Security are specified with the SpEL fluent interface. With this approach, we foresee two factors that can lead to inconsistencies of the implementation and the intended design. First, the developer should be familiar with the domain specific language SpEL in order to specify the *antMatchers* correctly, i.e. in the correct order. Second, the string values of some of the arguments are not parsed and automatically checked. Based on that, we identified 4 mistakes that developers can make when using Java Spring Security.

Missing or wrong authentication rule: The developer forgets to include the authentication filter *authenticated()* for the URI of a particular resource in the configuration or uses the filter *permitAll()* to incorrectly allow access to all users. However, in the specification model, the resource requires valid authentication. If no filter is specified, this is equivalent to the filter *permitAll()*. As a result, any user without authentication is able to request this resource. The error causes the security vulnerability missing authentication *CWE-306*.

Missing authorization rule: The developer forgets to include one of the authorization filters *hasRole(role)* or *access(rule)* for the URI of a particular resource. However, according to the *ACSM*, the resource requires a valid authorization. The filter *authenticated()* leads to the same error because it only checks the authentication of the user. Depending on the filter used, either all users or only authenticated users are able to request this resource. The error causes the security vulnerability of missing authorization *CWE-862*.

Incorrect authorization rule: The developer incorporates an authorization filter *hasRole(role)* or *access(rule)* for the URI of a certain resource, but a wrong authorization formula is used. As a result, a user without the required access rights is able to request this resource. The error causes the security problem of incorrect authorization *CWE-863*.

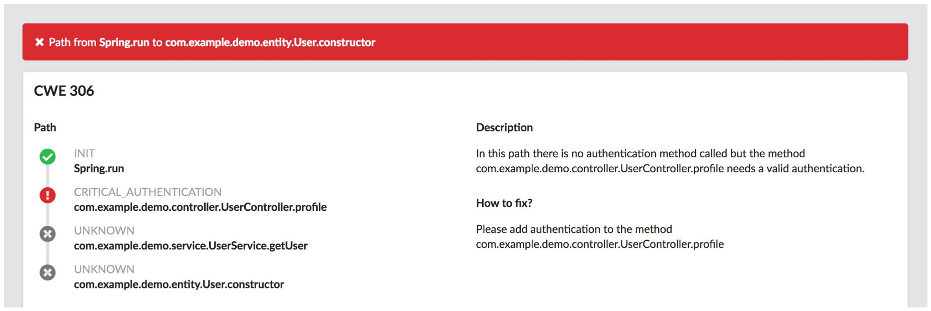
Method call with higher access rights: The developer creates a correct configuration for the resource, but in a deeper layer of the application, a call to a method is created that requires higher access rights and therefore should not be called by the user. The error causes the security problem of incorrect authorization *CWE-863*.

We implemented an extended version of the running example from Section 2 that includes the four mistakes and serves as a test scenario for our implementation. It is available under [18]. The tool generates a HTML page with all vulnerabilities detected. Figure 4 shows a detected *CWE-306* in our running example, including the path and description for solving the issue.

6 Case Study

We used the open-source project *FredBet*[8] to perform a case study that demonstrates the applicability of our analysis.

[8] https://github.com/fred4jupiter/fredbet



✖ Path from Spring.run to com.example.demo.entity.User.constructor

CWE 306

Path

✓ INIT
Spring.run

! CRITICAL_AUTHENTICATION
com.example.demo.controller.UserController.profile

✖ UNKNOWN
com.example.demo.service.UserService.getUser

✖ UNKNOWN
com.example.demo.entity.User.constructor

Description

In this path there is no authentication method called but the method com.example.demo.controller.UserController.profile needs a valid authentication.

How to fix?

Please add authentication to the method com.example.demo.controller.UserController.profile

Fig. 4: Screenshot from AuthCheck generated output with *CWE-306*

6.1 FredBet

The web application *FredBet* is a football betting system developed with Java Spring Boot and Spring Security. FredBet offers the possibility to initiate an online football bet with several users. In addition to the betting, the web application offers statistics about the matches, rankings, a profile management, and many other functions. The application is actively developed since 2015 and as of July 2019, it's repository has more than 1300 commits.

Since we have access only to the implementation and no design specification is available from which we can infer an *ACSM*, we decided to create the *ACSM* based on the implementation and insert the four types of mistakes discussed in Subsection 5.3. We focused on the *AdminController* from FredBet and made code modifications. AuthCheck analyzed the *AdminController* and detected the inserted vulnerabilities.

6.2 Limitations

When applying AuthCheck to FredBet, we realized that the specification scope in Spring Security is much broader than the available documentation. This means that there are many ways to specify the same configuration information when one develops an application. For example, a developer can specify an URI for a given class containing critical methods. This URI is then concatenated to the URIs of the critical methods it contains. Then, the annotations can have different formats or even some can be skipped, like the HTTP method, which in such case, a default value *GET* will be considered by the framework. The configuration of the *antMathers* (see Listing 1.1) can have different parameters. Such broad scope of specification options, is currently not supported by the AuthCheck parser. Even though, this is a technical disadvantage, in order to prepare AuthCheck for more complex web applications, the parser needs to be extended.

7 Related Work

Security vulnerabilities caused by the misuses of access-control mechanisms have been investigated by Dalton et al. [6]. The approach examines access-control problems by analyzing the flow of user credentials within the web application. In contrast to AuthCheck, their approach is dynamic and can not be used for early detection of the vulnerabilities.

Sun et al. [21] introduced a static analyis approach for the detection of access-control vulnerabilities. They assume that the source code contains implicit documentation of intended accesses. From this, sitemaps for different authorization groups are generated and checked whether forced browsing can happen. Another static analysis specific for access-control of XML documents was introduced by Murata et al. [16]. They use XPath representation for the access-control rules and XQuery for specifying the requests. The analysis checks all paths defined by the query against the XPath rules. Naumovich et al. [17] proposed a static analysis for Java EE applications where the resources are security fields from the Java Beans objects.

In the area of model checking, few approaches address the access-control protocols [15] [22]. In these approaches, the focus is to validate the message communication of the defined protocols. Similarly, Alexander et al. applied model checking to verify the authentication mechanism in the comminication of a set of interacting virtual machines [4].

8 Conclusion and Future Work

Even though sophisticated Java web frameworks, such as Spring, provide secure mechanism for access control of resources, for many developers using the APIs and the configuration specifications correctly, can be challenging. Thus, these misuses may cause access-control vulnerabilities in the code. In this paper, we presented AuthCheck, a static analysis, that tracks the program-state to detect the vulnerabilities *CWE-306*, *CWE-862*, and *CWE-863*. Based on *finite state machine* specification of each vulnerability, AuthCheck checks each path. We implemented the approach on top of the Soot framework and applied it to one open-source project on which we detected four types of errors that were previously inserted in the existing application.

We plan to evaluate the precision of AuthCheck in cooperation with industry to overcome the problem of the open-source projects of not having a design specification on which we can check the implementation against. Additionally, in future the choice of the call graph algorithm should be evaluated.

References

1. Spring framework, java spring. `https://spring.io/projects`, online; accessed 9 March 2019

2. Spring framework, java spring security. `https://spring.io/guides/topicals/spring-security-architecture`, online; accessed 9 March 2019
3. Spring framework, spring expression language. `https://docs.spring.io/spring/docs/5.0.5.RELEASE/spring-framework-reference/core.html`, online; accessed 12 March 2019
4. Alexander, P., Pike, L., Loscocco, P., Coker, G.: Model checking distributed mandatory access control policies. ACM Trans. Inf. Syst. Secur. **18**(2), 6:1–6:25 (Jul 2015)
5. Ball, T., Rajamani, S.K.: The slam project: Debugging system software via static analysis. In: Proceedings of the 29th ACM SIGPLAN POPL. pp. 1–3. POPL '02, ACM, New York, NY, USA (2002)
6. Dalton, M., Kozyrakis, C., Zeldovich, N.: Nemesis: Preventing authentication and access control vulnerabilities in web applications. In: Proceedings of USENIX. pp. 267–282. SSYM'09, USENIX Association, Berkeley, CA, USA (2009)
7. Enumeration, C.C.W.: Incorrect authorization. `https://cwe.mitre.org/data/definitions/863.html`, accessed 12 March 2019
8. Enumeration, C.C.W.: Missing authentication for critical function. `https://cwe.mitre.org/data/definitions/306.html`, accessed 12 March 2019
9. Enumeration, C.C.W.: Missing authorization. `https://cwe.mitre.org/data/definitions/862.html`, accessed 12 March 2019
10. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis (2000), university of California, Irvine
11. Gamma, E., Vlissides, J., Johnson, R., Helm, R.: Design Patterns CD: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1998)
12. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with blast. In: Proceedings of the 10th International Conference on Model Checking Software. pp. 235–239. SPIN'03, Springer-Verlag, Berlin, Heidelberg (2003)
13. Krüger, S., Späth, J., Ali, K., Bodden, E., Mezini, M.: CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In: ECOOP. pp. 10:1–10:27 (2018)
14. Lam, P., Bodden, E., Lhotak, O., Hendren, L.: The soot framework for java program analysis: a retrospective. In: Cetus Users and Compiler Infastructure Workshop (CETUS 2011) (Oktober 2011)
15. Marrero, W., Clarke, E., Jha, S.: A model checker for authentication protocols. In: Rutgers University (1997)
16. Murata, M., Tozawa, A., Kudo, M., Hada, S.: Xml access control using static analysis. ACM Trans. Inf. Syst. Secur. **9**(3), 292–324 (Aug 2006)
17. Naumovich, G., Centonze, P.: Static analysis of role-based access control in j2ee applications. SIGSOFT Softw. Eng. Notes **29**(5), 1–10 (Sep 2004)
18. Petrasch, T., Piskachev, G., Spaeth, J., Bodden, E.: Authcheck spring implementation. `https://github.com/secure-software-engineering/authcheck/`, online
19. del Pilar Salas-ZÃąrate, M., Alor-HernÃąndez, G., Valencia-GarcÃŋa, R., RodrÃŋguez-Mazahua, L., RodrÃŋguez-GonzÃąlez, A., Cuadrado, J.L.L.: Analyzing best practices on web development frameworks: The lift approach. Science of Computer Programming **102** (2015)
20. Strom, R.E.: Mechanisms for compile-time enforcement of security. In: Proceedings of the 10th ACM SIGPLAN POPL. pp. 276–284. ACM, New York, NY, USA (1983)
21. Sun, F., Xu, L., Su, Z.: Static detection of access control vulnerabilities in web applications. In: Proceedings of USENIX. USENIX Association, Berkeley, CA, USA (2011)

22. Xu, Y., Xie, X.: Modeling and analysis of authentication protocols using colored petri nets. In: Proceedings of the 3rd ASID. ASID'09, IEEE Press, Piscataway, NJ, USA (2009)