Model-based analysis of Java EE web security misconfigurations

Salvador Martínez, Valerio Cosentino, Jordi Cabot

Model-based Analysis of Java EE Web Security Misconfigurations

Salvador Martínez[a,b,*], Valerio Cosentino[a,d], Jordi Cabot[c,d]

[a] *AtlanMod team (Inria, Mines Nantes, LINA), Nantes, France*
[b] *CEA, LIST, Laboratory of Model Driven Engineering for Embedded Systems, Gif-sur-Yvette, France*
[c] *ICREA, Barcelona, Spain*
[d] *UOC, Barcelona, Spain*

Abstract

The Java EE framework, a popular technology of choice for the development of web applications, provides developers with the means to define access-control policies to protect application resources from unauthorized disclosures and manipulations. Unfortunately, the definition and manipulation of such security policies remains a complex and error prone task, requiring expert-level knowledge on the syntax and semantics of the Java EE access-control mechanisms. Thus, misconfigurations that may lead to unintentional security and/or availability problems can be easily introduced. In response to this problem, we present a (model-based) reverse engineering approach that automatically evaluates a set of security properties on reverse engineered Java EE security configurations, helping to detect the presence of anomalies. We evaluate the efficacy and pertinence of our approach by applying our prototype tool on a sample of real Java EE applications extracted from GitHub.

Keywords: Model-Driven Engineering, Security, Reverse-engineering

1. Introduction

Java EE is a popular technology of choice for the development of dynamic web applications (serving also as the basis for other less general purpose frameworks) that expose distributed information and services to remote users. In this scenario, security is a main concern [1], as the web resources that constitute the web application can be potentially accessed by many users over untrusted networks. As a consequence, the Java EE framework provides developers with the means to specify access-control policies in order to assure the confidentiality and integrity properties of the resources exposed by web applications.

Unfortunately, despite the availability of these security mechanisms, implementing security configurations remains a complex and error prone activity where high expertise

* Corresponding author
Email address: salva.martinez@mines-nantes.fr (Salvador Martínez)

is needed to avoid misconfiguration issues, that could inflict critical business damages. In fact, the Open Web Application Security Project (OWASP) document ranks web application misconfigurations in 5th position on the top ten of most critical security flaws [2], since they are easy to exploit and can have strong business impact.

For the concrete case of access-control and Java EE applications, and disregarding ad-hoc security implementation mechanisms tangled in the application code, role-based access-control (RBAC) [3] policies are specified by writing constraints using a low-level rule-based language with two different textual concrete syntaxes and with relatively complex execution semantics. Concretely, the user can either write constraints in the XML web descriptor file by using a set of predefined tag elements, directly write annotations (with a different syntax and organization w.r.t. the XML tag elements) on the Java Servlet components or combine both mechanisms. Then, combination rules between constraints and the corresponding execution semantics must be taken into account in order to understand what policy is being effectively enforced.

This complexity may lead to the introduction of anomalies and misconfiguration problems (e.g., unexpected rule outcomes, unexpected interactions between access-control rules, etc.) with effects varying from simply increasing unnecessarily the complexity of the specified policies to the introduction of unexpected behaviors such as granting access to resources to unauthorized parties or precluding it to the authorized ones, as confirmed as well by the participants in the online survey reported in Section 3.

In order to tackle this problem, we introduce a reverse engineering approach to automatically detect inconsistencies and misconfigurations in Java EE web applications. First, we define a list of properties a web application must satisfy in order to be free from important anomalies, such as redundancy (i.e., specification of unneeded constraints that overcomplicate the policy) and shadowing (i.e., specification of constraints that are never enforced). Secondly, we present an extraction method to parse the security configuration of a given web application (taking into account both, the web descriptor configuration and the Java annotations) and represent it as a Platform Specific security Model (PSM) specific to Java EE web access-control policies. Then, OCL queries and model transformation operations are implemented on top of that model in order to enable the automatic evaluation of the defined properties on any given Java EE web application. Additionally, our tool produces diagnosis reports that help to identify the source configuration elements responsible of the property violations, thus, helping developers to fix them.

We evaluate the efficacy of our approach by exercising our tool on a battery of publicly available Java EE web applications extracted from GitHub, a web-based Git repository hosting platform. This evaluation has shown that a relevant number of security configurations do violate our recommended properties and that our tool is able to successfully detect those violations.

The rest of the paper is organized as follows. Section 2 describes the access-control mechanisms of Java EE. Section 3 presents a motivation survey about the use of security in Java EE projects. Section 4 describes a number of security properties. Section 5 shows how to extract access-control models from Java EE web applications while Section 6 details our automatic approach to evaluate our properties on them. Section 7 presents a number of additional applications to our approach. Section 8 shows evaluation results and Section 9 gives details about the tool implementation. Related work

is discussed in Section 10. Finally, we conclude the paper in Section 11 by presenting conclusions and future work.

2. Java EE Web Security

Roughly speaking, in the Java EE realm, when a web client makes a HTTP request, the web server translates the request into HTTP Servlet calls to web components (Servlets and Java Server Pages) to perform some business-logic operations.

In this schema, a very important requirement is to ensure the confidentiality and integrity of the resources managed by the web application as they can be accessed by many users and traverse unprotected networks. In that sense, the Java EE framework provides ready-to-use access-control facilities. In the following we will briefly describe the mechanism offered by Java EE for the implementation of access-control policies in web applications.

As introduced before, Java EE applications are typically constituted of JSPs and Servlets (JSPs are in turn translated to Servlet). The access-control mechanism in place in this tier is in charge of controlling the access to these elements along with any other accessible artifact (pure HTML pages, multimedia documents, etc.) [1]. These access-control policies can be specified using two different mechanisms: declarative security and programmatic security, the latter being provided for the cases where fine access-control, requiring user context evaluations, is needed. Nevertheless, the Java EE specification recommends a preferential use of declarative security whenever possible.

Regarding declarative access-control policies, two alternatives are available: 1) writing security constraints in a *Portable Deployment Descriptor (web.xml)* and 2) writing security annotations as part of the Servlets Java code (note however that not all security configurations can be specified by means of annotations).

Listing 1 shows a security constraint defined in a *web.xml* descriptor. It contains three main elements: a *web-resource-collection* specifying the path of the resources affected by the security constraint and the HTTP method used for that access (in this case the */restricted/employee/** path and the GET method); an *auth-constraint* declaring which roles, if any, are allowed to access the resources (only the role *Employee* in the example) and a *user-data-constraint* that determines how the user data must travel from and to the web application (set to *None* in the example, i.e., any kind of transport is accepted). Additionally, although it is not mandatory, the web descriptor may contain role declarations (see Listing 2).

Listing 1: Security constraint in web.xml

```
<security-constraint>
  <display-name>GET To Employees</display-name>
  <web-resource-collection>
    <web-resource-name>Restricted</web-resource-name>
    <url-pattern>/restricted/employee/*</url-pattern>
    <http-method>GET</http-method>
  </web-resource-collection>
```

[1] http://download.oracle.com/otndocs/jcp/servlet-3.0-fr-oth-JSpec/

```
    <auth-constraint>
        <role-name>Employee</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>NONE</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

Listing 2: Role Declaration in web.xml

```
<security-role>
    <role-name>Employee</role-name>
</security-role>
```

The equivalent security constraint, defined by means of annotations is shown in Listing 3. The *@WebServlet* annotation identifies the Servlet and the resource path in the web container. Then, the main security annotation is *@ServletSecurity* that has two attributes: value, that corresponds to a nested annotation *@HttpConstraint* and httpMethodConstraint that contains a list of nested *@HttpMethodConstraint* annotations. The *@HttpConstraint* is used to represent a security constraint to be applied to all HTTP methods while the second is used to define per-HTTP method constraints. Both, *@HttpConstraint* and *@HttpMethodConstraint* contain as attributes a list of allowed roles (*allowedRoles*), the data protection requirements (equivalent to the *user-data-constraint* element in the *web.xml*) and the behavior when the list of allowed roles is empty. Finally, as in the web descriptor, security roles may be declared with annotations by using the annotation @DeclareRoles. E.g., *@DeclareRoles("employee")*.

Listing 3: Annotated Servlet

```
1 @WebServlet(name = "RestrictedServlet", urlPatterns ={"/restricted/employee/*"})
2 @ServletSecurity((httpMethodConstraints = {
3 @HttpMethodConstraint(value = "GET", rolesAllowed = "Employee")
4 transportGuarantee = TransportGuarantee.None)})
5 public class RestrictedServlet extends HttpServlet { ... }
```

Both alternatives can be used at the same time and the final security policy is the result of combining the security constraints specified with both mechanisms. When in conflict, the constraints specified in the *web.xml* file take precedence and moreover, constraints defined by using annotations may be completely ignored if so is established in the *web.xml* descriptor (the *metadata-complete* parameter set as true) which may clearly create confusing situations to non-experts.

Besides, access-control policies defined with the mechanisms described above may contain inconsistencies (rules stating different access actions for the same resource). These inconsistencies are resolved by execution semantics such as rule precedence and combination algorithms as defined in the Java EE Servlet specification. Unfortunately, while this process eliminates inconsistencies in the policy, it may introduce typical access-control anomalies such as shadowing and redundancy ([4, 5, 6]), along with other misconfigurations particular to the Java EE access-control (e.g., not declaring all HTTP methods in a given access constraint). As these anomalies may lead to unexpected security and/or availability problems, the automatic detection of their presence in a given web security policy appears as a critical requirement for the security assessment.

Note that, as mentioned above, fine-grained access-control constraints requiring context information may also be defined in the Java EE framework.These constraints cannot be declaratively defined and require the use of programmatic security. Examples would be constraints checking that a user holds two roles or that a connection may only be accepted in specific time slots. We leave the analysis of these fine-grained programmatic constraints as a future work.

3. Motivation

In order to assess the relevance of security concerns in Java EE web development, we have elaborated an online survey addressed to Java EE web developers. From the set of answers, we have identified a number of important security properties developers find critical. We describe the details of our online survey in the reminder of this section.

3.1. Survey settings

Our survey has been directly sent to a pool of Java EE developers. We have obtained this pool by 1) searching projects using the Java EE technology in GitHub, and, 2) looking for the contributors (owner, collaborators, reporters) involved with those projects. Additionally, we have also openly published it in programming discussion forums and blogs. From this process, we have obtained a total of 41 answers, 87% of them corresponding to developers with at least 2 years of experience in Java EE web development.

3.2. Survey results

In the following, we will present a summary of the survey contents and obtained answers. The complete set of questions and answers can be found online [2]

> Q: In your opinion, how critical are security aspects in the development of web application(s)?

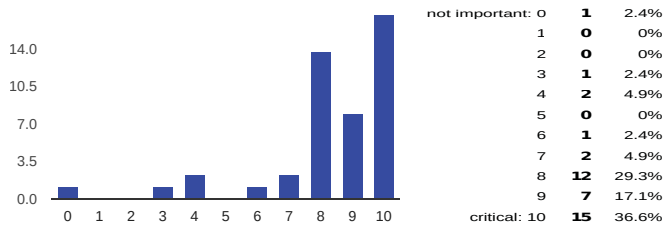

Figure 1: how critical are security aspects in the development of web application(s)?

Developers were asked to rate the importance of security properties between 0, unimportant and 10, very critical. 88% of developers consider security aspects critical during the development of web applications, rating its importance between 8 and 10 (see Figure 1).

[2] https://docs.google.com/forms/d/1K3hK4rY5yMkI3HbXOxRNfTHAQqR_tvLbf1s5RbvwaIA/viewanalytics

Q: Do you normally define access-control policies in your web application(s)?

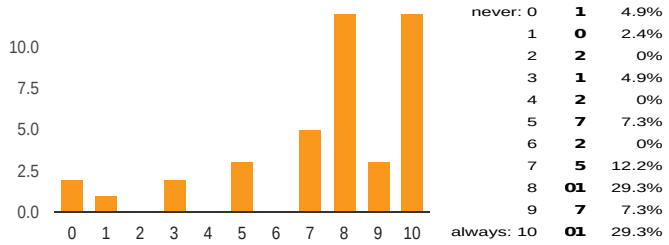

Figure 2: Definition of access-control policies

As can be seen in Figure 2, 78% of developers frequently (values between 7 and 10) define access-control policies as part of the creation of a web application.

How difficult do you find the definition of access-control policies in Java EE application(s)?

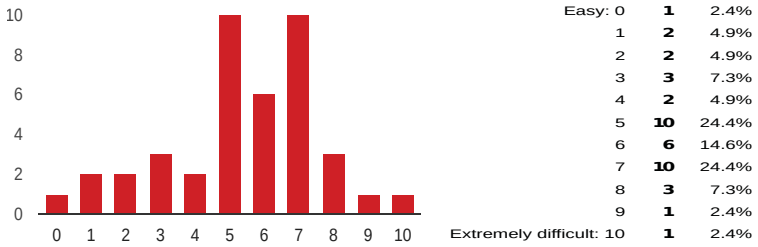

Figure 3: Difficulty of defining access-control policies

Developers were asked about the degree of difficulty of defining access-control policies, 0 being easy and 10 extremely difficult. 75% of developers feel that defining access-control policies, while not extremely difficult, is a challenging activity with most answer grouped between 5 and 8.

From the previously presented set of questions we can conclude that security concerns are considered critical for the majority of participant Java EE developers. Moreover, defining an access-control policy for their projects is both, a difficult and common activity. After these generic questions, developers were asked to evaluate the prevalence and severity of some common security errors.

We asked Java EE developers about the importance of having complete access-control policies, i.e., policies that completely cover the access space for a given resource. We also asked them to evaluate how likely it is to produce incomplete policies. From Figure 4 we can see that having complete policies is seen as important for almost all developers (most answer above 5). Moreover, 29 out of 41 consider that producing incomplete policies is very likely.

W.r.t. redundancy in security policies (i.e., policy elements that can be removed without affecting the set of access decisions and that make the policy over-complex), from

Q: In your view, how important is to make sure security policies are complete?

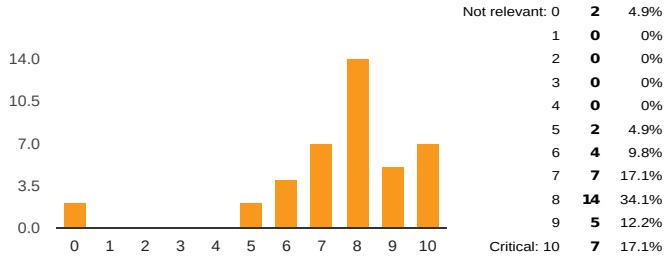

Figure 4: How dangerous are incomplete policies

Q: How important is to avoid redundancy in security policies?

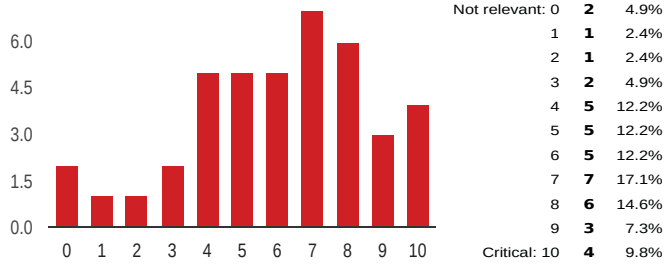

Figure 5: How bad is to have redundancies in policies

Figure 5 we can conclude that Java EE developers consider, in general, redundancy important but less harmful that incomplete policies.

Q: How important is to make sure all element references in a security rule are properly declared?

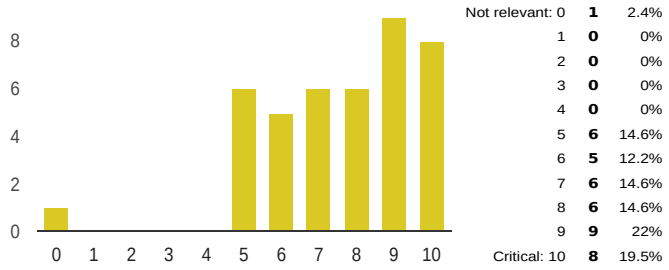

Figure 6: How important is to declare all the elements referenced in a policy

It is not uncommon in security policies to rely on implicit or external information in their contained rules while also allowing for a formal declaration of these elements (e.g., roles may often be taken from the environment whereas they may be directly declared in the security policy as well). We have asked Java EE developers about the

importance of declaring all the elements used in a security policy obtaining the results shown in Figure 6. As we can see, most developers consider explicitly declaring all used information a good practice.

Q: How dangerous is to have inconsistent rules?

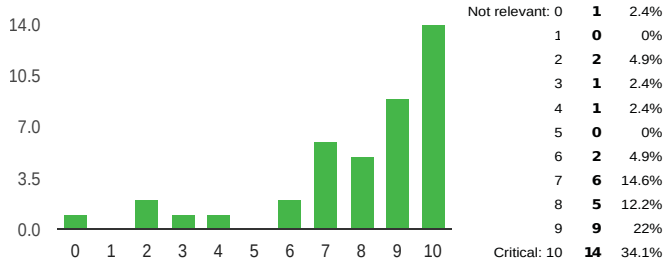

Figure 7: How dangerous is to have inconsistent rules

Inconsistent policies are policies containing rules that, for the same access-request, yield different decisions. The questioned Java EE developers consider this eventuality as very dangerous. 36 out of 41 give to this security problem a value of six or more (see Figure 7) in an scale ranging from zero (irrelevant) to ten (critical).

Q: How dangerous is to have private resources involuntarily exposed?

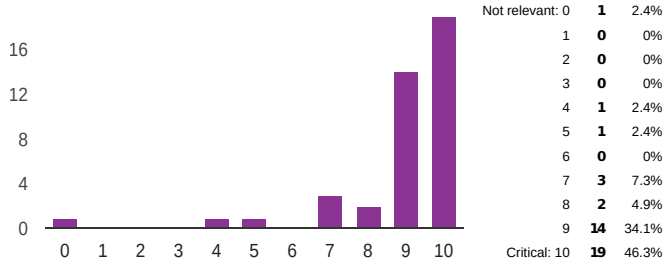

Figure 8: How dangerous is to have private resources involuntarily exposed

To conclude the questions related to common security anomalies related with access-control configurations, we asked the survey participants about the danger of involuntarily leave private resources (i.e., resources not intended to be publicly available) exposed to unintended parties. Not surprisingly, and as it can be see in Figure 8, Java EE developers consider this as extremely dangerous (with most answer grouped around 9 and 10).

Finally, we asked the survey participants whether they would add other types of security errors to the ones listed above. More than 87% would not and the other 13% did not provide a new description.

We ended the survey by asking the participants about the usefulness of having a tool helping to detect and eventually correct the aforementioned security problems.

Q: Would you add other security problems to the ones discussed above?

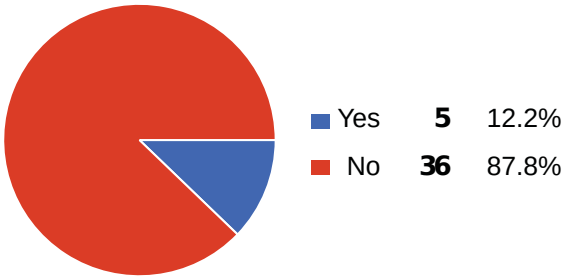

Figure 9: Would you add other security concerns to the ones discussed above

Q: Would you find useful an automatic tool for detecting (and fixing) security problems related with access-control configurations?

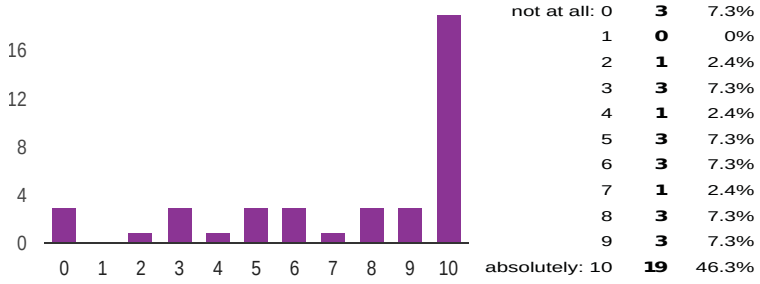

Figure 10: Usefulness of having a tool to help analysing policies

For the majority (29 out of 41 See Figure 10) of questioned developers, having an automatic tool for evaluating security properties related with access-control configurations would be useful (values between 6 and 10).

3.3. Survey evaluation

Based on these answers, we can conclude that, in general, security is a critical concern. Moreover, the relevance of each of the security problems associated with access-control configurations is evaluated as very high, with most answers grouped around the higher values.

Inspired by these results, in the following section we will propose a set of security properties tailored to the Java EE framework that allow the detection of the aforementioned security problems on declarative access-control policies.

4. Identification of Relevant Web Security Properties

Inspired by our survey, we propose a number of properties, that, when violated, lead to the introduction of policy anomalies regarding the semantics of Java EE access-control. Before getting into the actual properties, we introduce some required definitions.

4.1. Definitions

The following definitions aim to characterize the semantics of the Java EE declarative access-control.

Definition 1: Web application's security domain. *Let R be the set of resources the application uses and makes available for remote access (e.g., web pages and stored documents), S the set of subjects intended to access those resources (subjects are roles in the Java EE access-control realm) and, A the set of actions (e.g., accesses to the resource by means of a given HTTP method) that can be performed on them, we define the domain of web application's security as the $tuple < R, S, A >$. Resources are identified by its URL pattern (or path), Subjects are identified by their name and Actions are identified by the name of the HTTP method.*

A very simple example of domain is the tuple $< R, S, A >$ where R={employee.jsp, private.jsp, index.html}, S={anonymous, manager, employee}, and A={GET, POST}.

Definition 2: Resource pattern. *Resources can be grouped by using resource patterns instead of the concrete URL (or path). This way the notation .extension will denote all the files of a given extension, the * symbol will denote any resource and the symbol container_name resources contained in container_name. Note that a resource pattern could also be a concrete URL path corresponding to a single element.*

As an example '.pdf' will denote all the files with pdf extension, '/foo/' will denote all elements contained in foo (such as '/foo/index.html') and '/*' will denote any resource pattern contained in the root of the web application (including, for example, '/foo/index.html').

Definition 3: Security constraint. *For a given application's security domain $< R, S, A >$, and being $R_{pattern}$ the set of all the resource patterns, a security constraint c is an ordered pair (d, t) of the function $f : D \rightarrow T$ where $D = \{\mathcal{P}(R_{pattern}) \times \mathcal{P}(S) \times \mathcal{P}(A)\}$ and $T = \{allow, deny\}$. We use $c.S, c.R$ and $c.A$ to refer to the respective sets of* Subject*,* Resource *and* Action *of the constraint.*

Access to resources is by default allowed to any user. Security constraints restrict this access by mapping tuples composed of the power sets of *Subject*, *Resource* patterns and *Action* to the set $\{allow, deny\}$ where *allow* means the access to a resource is granted and *deny* that it is precluded. As an example, see Listing 1 in Section 2 were a constraint restricting the access to the resources referred by '/restricted/employee/*' using the GET HTTP method to users holding the Employee role is presented.

Definition 4: Security Policy. *A security policy P for a given application domain $< R, S, A >$ is a set of constraints and of declared subjects. We refer to this set of declared subjects as S_d where $S_d \subseteq S$.*

As an example, see Section 2, Listing 2, where the role Employee is declared.

Definition 5: Rule combination. *Given a security policy P, and two contained constraints c_i and c_j with identical URL pattern ($c.R$) and set of Actions ($c.A$), we define the function $combine(c_i, c_j) \rightarrow c_g$ that combines them in a global constraint c_g. It works as follows:*

- *All named subjects (explicit or implicitly mentioned through the use of the wildcards *, implying all declared roles, or ** implying any authenticated user, as defined in the Java EE specification) become permitted subjects in the global constraint. Thus, the global set of subjects with access to the resource is:* $c_g.S = c_i.S + c_j.S$.

- *If a security constraint authorizes all possible subjects, unauthenticated access is allowed in the global constraint.*

- *A constraint* c_i *that denies access to all subjects shall combine with any other constraint* c_j *by overriding the effect of* c_j *and cause the accesses to the resource to be precluded in the global constraint. Thus* $c_g.S = \emptyset$.

As an example, if a security constraint with an URL pattern equal to *employees/private* grants access to a set of subjects and another security constraint with identical URL pattern precludes access to every subject (e.g., by using an empty auth-constraint tag), the final global constraint will preclude all access to the given URL pattern (deny overrides allow). Note that the process works iteratively, combining pairs of constraints until no more combinations are possible.

Definition 6: Access Request. *For a given application's security domain* $< R, S, A >$, *an access-request* ar *is a triple of the set formed by the cartesian product* $\{R \times S \times A\}$. *We use* $ar.S, ar.R$ *and* $ar.A$ *to refer to the respective* Subject, Action *and* Resource *of the access-request.*

Note that we do not use power set (nor resource patterns), as an access-request can only refer to one *Subject*, one *Resource* and one *Action*.

Definition 7: Best match. *In an application's security domain* $< R, S, A >$, *given an access request ar with URL request path* $ar.R$ *and Action* $ar.A$ *and a policy P, the operation* $bestMatch(P, ar.R, ar.A) \rightarrow c$ *returns one and only one constraint* c *by first, applying in order the following rules for matching the URL path identifying* $ar.R$ *with the Resource patterns* $c.R$ *defined in the constraints:*

1. *Exact match of* $ar.R$ *and one of the elements of* $c.R$.
2. *Longest prefix match between* $ar.R$ *and one of the elements of* $c.R$ *(e.g., the resource pattern with the higher number of path segments in common with* $ar.R$ *will match).*
3. *If* $ar.R$ *contains a file extension (e.g., .pdf), a* $c.R$ *containing the same file extension.*
4. *If no other constraint matches and a constraint is defined for the resource identified with '/*', this constraint is matched as default.*

and second, in case of this resulting in several constraints c *with the same* $c.R$ *but different* $c.A$, *selecting the constraint containing* $ar.A$.

The *bestMatch* algorithm is used to match access-request to 1) responding servlets and 2) security constraints. As an example, supposing we have a policy with three constraints having '/foo/bar/*', '.doc' and '/*' as respective URL patterns, a request for the

URL pattern '/foo/bar/index.html' will match first with the constraint with '/foo/bar/*' as URL pattern. A request for '/foobar/list.doc' will match first with a constraint with URL pattern .doc. Finally, if there is no constraint with '.pdf' (or 'list.pdf'), a request for 'list.pdf' would match to a constraint defined for '/*' if it exist.

4.2. Properties

Once we have introduced the definitions characterizing the semantics of the Java EE declarative access-control mechanism, we can proceed to introduce a number of properties that, when violated, lead to the introduction of policy anomalies.

If an Action (HTTP method) is named in a security constraint c_i, all other standard Actions (HTTP methods) must be specified in the same or other security constraint matching the same set of requests. Otherwise non-named Actions will be uncovered giving unconstrained access to them. This happens because, as per Definition 7, the Action is only used for disambiguation in the process of matching security constraints.

Property 1: Completeness. *If a constraint $c_i.A$ contains an Action, it must contain all other available Actions (HTTP methods) or there should be at least another constraint c_j where, $c_j.R = c_i.R$ and $c_j.A$ contains the other Actions.*

An example of a security constraint violating this property is shown in Listing 4 (reduced version of Listing 1). In the example, a constraint is defined for the GET method, stating that only users holding the *Employee* role are allowed. However, nothing is said about any other HTTP methods. In the absence of other constraints with identical URL pattern, from *Definition 7* it follows that this constraint will be matched (but not applied), and thus, unconstrained access will be granted to any HTTP method different from *GET*.

Listing 4: uncovered methods

```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/restricted/employee/*</url-pattern>
    <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
     <role-name>Employee</role-name>
  </auth-constraint>
</security-constraint>
```

A security constraint is redundant when it can be removed from the security policy without modifying its set of accepted or denied requests (see [4], [7] and [5].

Property 2: Redundancy. *There should not exist two different constraints c_i, c_j that for the same set of requests produce the same effect (i.e., the constraint function yields the same result). Otherwise one of the constraints is redundant. Summarizing, let c_i, c_j be two security constraints, c_j is redundant if $c_j.R \subset c_i.R$, $c_j.S = c_i.S$, $c_j.A = c_i.A$ and the result of applying c_i is equal to the result of applying c_j.*

Listing 5 shows a policy containing a redundant constraint. Effectively, as the path of the second constraint is contained in the more general path defined for the first

constraint (and being the rest equal), the second constraint, while still executable, can be removed from the policy without changing the global behaviour. Note that we only consider here fully overlapping constraints.

Listing 5: Redundant constraints

```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/restricted/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
     <role-name>Employee</role-name>
  </auth-constraint>
</security-constraint>
<security-constraint>
  <web-resource-collection>
    <url-pattern>/restricted/employee/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
     <role-name>Employee</role-name>
  </auth-constraint>
</security-constraint>
```

A non declared subject must not be used in a security constraint. Not declaring the roles (subjects) used by the security constraints is not considered as an error in the Java EE framework (as roles are string-mapped from the environment). However, we consider that a policy declaring them is easier to understand and less prone to errors as the mappings between environment roles and application roles are centralized. Moreover, the framework provides the special role name '*' that, when used in a security constraint (in the web descriptor web.xml) maps to the list of all declared roles. Therefore, using '*' without explicitly declared roles will preclude the access to the resource.

Property 3: Syntactical *Let S_d be the set of declared subjects and S_u the set of used subjects in security constraints, for each subject in S_u, there must exist a corresponding declared subject in S_d. Summarizing, $S_u \subseteq S_d$ must be true.*

As an example, a policy containing the constraints showed in Listing 5 will only be correct if it also contains a declaration for the role Employee as showed in the Listing 2 in Section 2.

If all the requests that could be matched by a constraint are matched by another constraint with a different effect, one of them is shadowed (see again [4, 7] and [5]). This may happen for instance when an authorization constraint names no subjects (existing a similar constraint naming specific subjects) or when a security constraint uses the special names '*' or '**' (existing again a similar constraint naming specific subjects). Note that this case is different from the redundancy case described in *Property 2*, as the shadowed constraints would change the global policy if they were enforceable.

Property 4: Shadowing *let c_i, c_j be two security constraints, c_i is shadowed by c_j (c_j having higher precedence as for the combination rules in* Definition 5*) if $c_j.R = c_i.R$, $c_j.A = c_i.A$ and the intersection of their subjects is empty, $c_j \cap c_i = \emptyset$.*

Listing 6: Shadowed constraints

```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/restricted/employee/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
     <role-name>Employee</role-name>
  </auth-constraint>
</security-constraint>
<security-constraint>
  <web-resource-collection>
    <url-pattern>/restricted/employee/*</url-pattern>
  </web-resource-collection>
  </auth-constraint>
</security-constraint>
```

Listing 6 shows an example of a shadowed rule. While the first constraint restricts the access to the resource so that it can only be accessed by users holding the *Employee* role, the second, by using an empty authorization constraint, completely precludes any access. As from *Definition 5* it follows that the second constraint takes precedence in the combination process and given that the set of roles that get granted access to the resource differ, the first rule is shadowed.

All resources named in the constraints of a web security policy should allow at least one role to access them. We adapt here the accessibility property defined in [8] for the deployment of network policies. We consider any resource exposed by the web application is supposed to be accessible (although possibly constrained). Otherwise, the resources should not be deployed with the web application as 1) this increases the complexity of the security policy configuration 2) errors may cause the resources to end up being exposed.

Property 5: Reachability. *All resources named in the constraints of a web security policy should allow at least one role to access them. Summarizing,* $c.R \neq \emptyset \implies c.S \neq \emptyset$ *should hold.*

The violation of any of these security properties may cause a given Java EE web application to present security and/or availability problems. In order to prevent these problems to arise in newly developed applications or to correct them in already deployed ones, we describe in the next section a model-based approach that automatically evaluates Java EE web applications to assess the absence of security property violations.

5. Policy Extraction

Our approach for the analysis of Java EE security configurations is depicted in Figure 11. It is composed of four steps, an *extraction process* that collects the security information contained in the Java EE application and represents it as part of platform-specific software models, an *integration* process that combines the extracted models into a single global security model, a *property evaluation process* that performs model

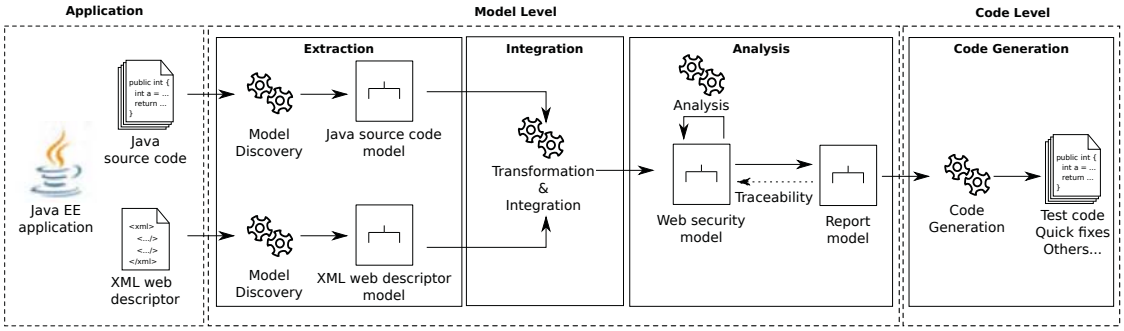

Figure 11: Java EE web application analysis approach

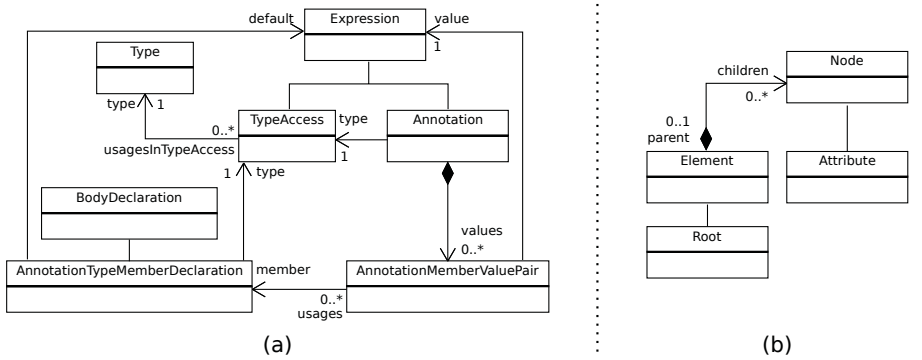

Figure 12: Excerpt of the Java (a) and XML (b) metamodels used in the extraction phase

transformations and query operations over the generated PSM in order to evaluate security properties and produce diagnosis reports and a *code generation* step that generates code artifacts (e.g., test cases and quick fixes). The *extraction process* is based on text-to-model techniques, the *integration* and *property evaluation* processes are based on OCL queries and model-to-model transformations and the *code generation* on model-to-text transformations. We choose model transformations for the sake of flexibility and reusability as they permit us to easily reuse the invariant evaluation in a wider range of operations.

In this section we will describe in detail the first two steps. The evaluation of properties is discussed in Section 6 whereas generation (among other applications) is presented in Section 7. An initial sketch of this process was discussed in [9] where the *extraction* and *integration* processes along with some of the applications presented in Section 7 were discussed.

5.1. Extraction Process

The first step of our approach is the extraction process. It parses the Java source code and XML web descriptor to obtain corresponding model representations, namely, a Java model and an XML model (see Figure 12 where excerpts of the Java annotations and XML metamodels are depicted). By doing so we move from the technical space of source code annotations and XML files to that of the modelware realm. For this stem we rely in MoDisco[10], a model discovery.

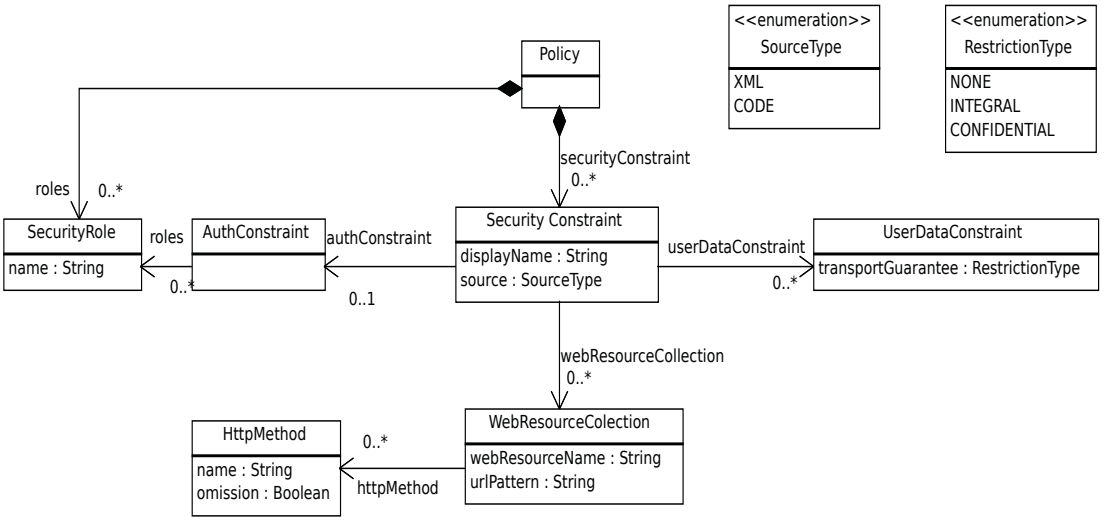

Figure 13: Servlet Security metamodel

5.2. Integration

The models obtained in the previous step are manipulated and the security information they contain mixed in an integration process aimed to obtain a model (see Section 5.2) representing the global security policy configured in the Java EE application.

The integration process allows us to ease the manipulation operations required to analyze the policies in comparison with the effort of directly manipulating the constraints separately in the XML and source code annotations with more basic techniques (like text manipulation or XSLT transformations). Concretely, it will allow us to 1) use well-known model-driven tools and frameworks and 2) treat security information in an uniform way, disregarding whether it was specified by using XML or annotations. Moreover, as this global model lays at the same abstraction level with respect to the original configurations, no information-loss is produced in the extraction process.

5.2.1. Metamodel

The definition of metamodels able to accurately represent the information contained in the configuration source files is key to our approach and to this integration step. We define a Servlet Access-Control Security metamodel (hereinafter referred to as Servlet Security metamodel) for that purpose. It is depicted in Figure 13. Both annotations and XML security definitions share similar semantics and therefore can be mapped to this metamodel.

The Servlet Security metamodel allows to handle security roles (*SecurityRole* class) and security constraints (*SecurityConstraint* class) of a given web application. The former specifies the security roles defined by the security policy. The latter is used to represent access control constraints, coming from XML web descriptors or Java annotations (*source* attribute), on a collection of web resources (*WebResourceCollection* class). Each web resource collection identifies the resources (*urlPattern* attribute) and HTTP methods (*HttpMethod* class) to which a security constraint applies. It is important to note that if a HTTP method is defined as omitted (*omission* attribute), the

security constraint applies to all methods except the omitted method. The set of roles participating in the security constraint for a collection of web resources are grouped under an authorization constraint (*AuthConstraint* class). The absence of an *AuthConstraint* element in a given constraint represents public access whereas total preclusion is represented by its presence with no role association. A security constraint can be also associated with a given data protection requirement (*UserDataConstraint* class). This data protection requirement can be defined as: NONE indicating that the container must accept requests when received on any connection, INTEGRAL establishing a requirement for content integrity and CONFIDENTIAL, establishing a requirement for communication confidentiality.

5.2.2. Integration transformation

Once we have a metamodel able to represent the access-control configurations of Java EE application ready, we can proceed to integrate the models obtained in the *extraction* step. For that purpose, we use the ATL [11] model transformation language, but a similar approach would be followed when using other relational transformation languages.

ATL is a declarative transformation language that produces output models from input ones. Transformations written using this language are composed, basically, by two main elements: Transformation Helpers and Transformation (declarative) Rules. Helpers play a similar role as functions do in general purpose programming languages. They are used to factorize OCL[12] queries in order to improve reusability and to simplify the transformation rules. Declarative rules use a pattern-based approach (similar to how graph transformation do) to match source input elements and produce and initialize target model elements with them.

The code of the full integration transformation is available on GitHub. It is composed of 32 helpers and 20 rules, in particular 6 helpers and 11 rules are used to extract security information from the XML web descriptor, while the remaining helpers and rules target the annotations embedded in the Java code.

Listing 7 shows an ATL rule, part of the transformation to combine the XML and Java Annotations models, that maps annotated Servlets to *SecurityConstraint* entities. It takes a *Java Annotation* model element (line 3) and creates a *Security Constraint* element in a model conform to our Servlet Security metamodel (line 8). Then, helpers are used to initialize this *Security Constraint* with the corresponding values from the *Java Annotation* values (e.g., URLPattern, etc.). Finally, an *AuthConstraint* element with the corresponding list of authorized roles is created (lines 14-15) and assigned to the *Security Constraint* (line 12).

6. Analysis of Security Properties

Once we have the security configuration as a model conforming to the Servlet Security metamodel presented before, we can start analyzing it in order to evaluate possible violations of the properties defined in Section 4. This evaluation is fully automated.

In particular, we have specified our properties as OCL queries. The execution of the OCL queries over the model (using any of the available OCL execution environments

Listing 7: ATL rule to map security annotated servlets to security-constraints

```
lazy rule createSecurityConstraint {
  from
    s : JAVA!Annotation
    using {
      servletAnnotation : JAVA!Annotation =
      s.getContainerAnnotation('ServletSecurity');}
  to
    t: SEC!SecurityConstraint (
      webResourceCollection <-
         thisModule.createWebResourceCollection(
         servletAnnotation.getWebServlet, s),
      authConstraint <- ac,
      source <- #CODE)
    ac: SEC!AuthConstraint
         roleName <- s.getRolesAllowed
}
```

like USE[13] or the EMF OCL[14]) returns the model elements that are violating a property, if any. These same OCL queries have been integrated in model transformations to convey the same information in the form of a Report model (see Section 6.1). Thus, we have implemented an ATL transformation for each of the security properties. The transformations are available on GitHub and in total they include 12 rules and 30 helpers.

In the following, we will give details about the calculation of each property over our Servlet Security metamodel. Note that for brevity, we only show the full formal specification in OCL of the Completeness and Syntactical properties, while we limit the specification of the rest of properties to a descriptive level. However, the full specification (in the form of ATL Helpers) of all the properties is available at the web site of the project.

Completeness query:
In Listing 8 we show the query that calculates the completeness property.

The OCL code defines two main variables, *HTTP_METHODS*, listing all the standard HTTP methods and *USED_HTTP_METHODS*, that collects all the elements in the input model whose class is *HttpMethod*. These collections of elements are used then to check if each *SecurityConstraint* element, containing a given instance of *HttpMethod*, includes all the other standard HTTP methods or if they are listed in another *SecurityConstraint* element with same *URLpattern* element value (directly or by the use of the method_omission tag).

Redundancy query:
In order to detect redundant constraints we proceed as follows: for each security constraint, we look for other security constraints that, not having a contrary effect, match an equal or contained set of requests and subject authorizations. Concretely, for each *SecurityConstraint* c_i in a given extracted model we look for other rules c_j fulfilling the following requirements: 1) c_j contains a *WebResourceCollection* with an *UrlPattern* element that is more general or equal to the one declared in c_i. and 2) the set of *RoleName* elements named in c_j is equal or contained in the set of *RoleName* elements contained in c_i (this includes the case of empty sets of named roles with or without the

Listing 8: OCL query to calculate the completeness property

```
context HttpMethod inv:
let HTTP_METHODS : Sequence(OclAny) =
  Sequence{'OPTIONS','GET','HEAD','POST',
  'PUT','DELETE','TRACE','CONNECT'} in
let USED_HTTP_METHODS : Sequence(PSM!HttpMethod) =
  PSM!HttpMethod.allInstances() in
let httpMethodsToCheck : Sequence(String) =
  if self.omission then
    USED_METHODS->select(m | m = self.name)
  else
    USED_METHODS->reject(m | m = self.name)
  endif in
  let selfUrlPatterns : Sequence(PSM!UrlPattern) =
  self.refImmediateComposite().urlPattern in
  selfUrlPatterns->iterate(sup ; output : Boolean = true |
    let declaredHttpMethods : Sequence(PSM!HttpMethod) =
        USED_HTTP_METHODS->reject(hm | hm = self)
        ->select(hm | hm.refImmediateComposite().urlPattern
        ->exists(up | sup.value = up.value)) in
    if declaredHttpMethods->isEmpty() then
      false
    else
      output and httpMethodsToCheck
      ->forAll(m | declaredHttpMethods
      ->exists(dhm | dhm.name = m))
    endif
```

use of an *AuthConstraint* element).

Syntactical query:
The syntactical query (see Listing 9) is calculated as follows. For each *AuthConstraint* containing *RoleName* elements, all named *RoleName* elements are retrieved. For each *RoleName*, we check if it is contained in a *SecurityRole* role definition.

Shadowing query:
Shadowed security constraints are detected as follows. For each *SecurityConstraint* c_i in a given model, we look for *SecurityConstraints* c_j that having the same set (or contained subset) of *UrlPattern* elements, declare contrary effects. In our model, contrary effects on two rules c_i and c_j matching the same access requests occur in two cases: 1) when the constraint c_i contains an *AuthConstraint* element without *RoleName* and c_j names roles or does not contain an *AuthConstraint* element (and viceversa) 2) when the c_i constraint contains an *AuthConstraint* element naming a list of roles and the c_j names the special role name '*' (meaning the set of all declared roles S_d) or the special name '**' (meaning all authenticated users). Note that we also check if the list of roles is equal to S_d. In that case, we classify the faulty security constraint as redundant and not as shadowed.

Reachability query: The reachability query is calculated by finding 1) all the *SecurityConstraint* elements using an empty *AuthConstraint* element, as this, in the JEE framework, precludes all access to the resource named in the constraint 2) if no roles are defined in the given security configuration, all *SecurityConstraint* elements in the web descriptor defining an *AuthConstraint* using the special '*' role name, as this special role name provides access only to declared roles. Note that, as our approach relies

Listing 9: OCL query to calculate the Syntactical property

```
let ALL_CONSTRAINTS : Sequence(PSM!SecurityConstraint) =
PSM!SecurityConstraint.allInstances() in

ALL_CONSTRAINTS->iterate(it; Undeclared : Sequence(String)
= Sequence{} |
  if self.authConstraint.oclIsUndefined() then
    Undeclared
  else
    Undeclared->including(
    self.authConstraint.roleNameTxt
      ->iterate(name; output:Sequence(String)
      = Sequence{} |
       if self.authConstraint.roleName
       ->exists(ref | ref.name = name) then
         output
       else
         if name <> '*' then
           output->including(name)
         else
           output
         endif
       endif))
  endif)->flatten()->asSet()->asSequence()
```

on the static analysis of the security policy, this property can only be partially checked (i.e., we can only check whether the declarative policy fully precludes access to a given resource or not. However, resources not constrained in the policy may be precluded by other mechanisms or misconfigurations). In order to fully check the reachability of web resources, dynamic analysis approaches, as the one proposed in [15], will be needed.

6.1. Security Report and Policy Repair

As mentioned in the previous section, the OCL queries used to evaluate our security properties can be integrated in a model transformation scenario in order to obtain a report model with the detected violations that could be used for feedback and further analysis. This model will conform to the Report metamodel depicted in Figure 14.

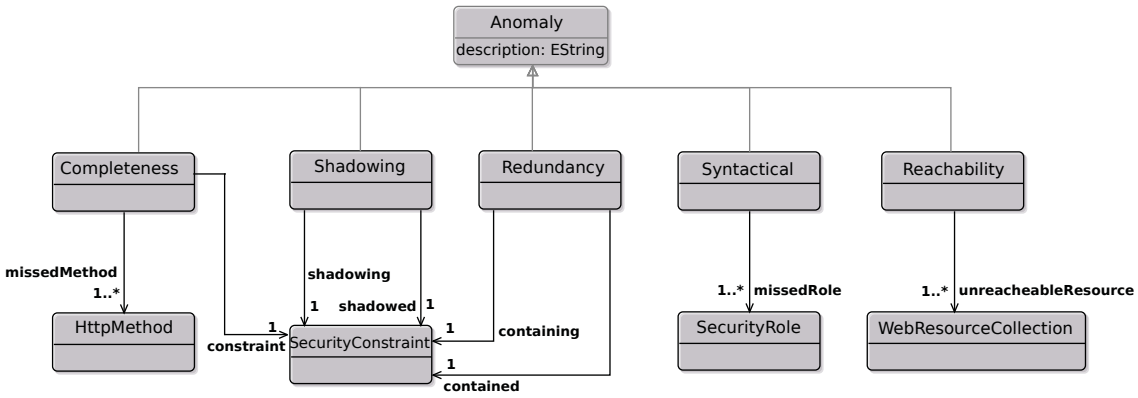

Figure 14: Report metamodel

The Report metamodel consists in an *Anomaly* metaclass extended by five meta-classes intended to represent each of our security properties. Semantically, these report

models should be read in the following way: each instance of any of the subclasses of the anomaly metaclass is an anomaly present in the security configuration of the web application. Note that, in order to help in further analysis and manipulation tasks, each concrete anomaly metaclass contains a reference to the element(s) in our Servlet Security Metamodel responsible from the anomaly. Concretely, the *Shadowing* metaclass has a link to the shadowed constraint as well as a link to the shadowing constraint. Similarly, the *Redundancy* metaclass has links pointing to the constraint that is contained and thus redundant and a link to the constraint whose resource path contains the other one. The *Completeness* metaclass links to the collection of missed HTTP methods and to the original constraint whereas the *Syntactical* metaclass has links to the set of undeclared roles used in constraints. Finally, the *Reachability* metaclass has links to the web resources that are unreachable.

Model Transformations for Reporting

Once we have specified all our properties in the form of OCL queries, we can integrate them in the pattern-matching phase of model transformations to produce the property violation reports. Using our OCL queries in ATL basically implies wrapping the contents of *let* expressions into helper definitions and leaving the rest of the code body untouched (if it is not by modifying the call to the variable defined by the *let* expression). Then, the only thing left to be done is to create a transformation rule matching the corresponding element on which the OCL query works (context), call the helper in the guard (to just generate anomaly reports in faulty elements) and produce as a rule output element the corresponding Anomaly model element conforming to our Report metamodel.

Listing 10 shows an ATL rule integrating the OCL query for calculating the completeness property into ATL for the generation of a report model. As it can be seen, the Completeness query described above is integrated in the transformation module in the form of a helper (lines 6 to 12) to be called in the guard of a transformation rule matching elements of type *HttpMethod* (line 17). When the guard holds (i.e., a security constraint is not complete) an *Completeness* model element containing the list of missing HTTP methods and a link to the faulty constraint is generated (lines 19 to 22).

Model Transformations for Fixing Configurations

The report model generated in the previous step can also be used in order to provide quick-fixes to the developers, so that they can easily correct some of the anomalies present in their configurations. For simplicity, we will limit ourselves here to the demonstration of the generation of fixes (in a textual form) that the developer would have to manually include in the access-control configuration in a semi-automatic workflow.

Table 1 shows for each of the security problems described in Section 4 a possible solution. Note however that in some cases several solutions may fit and thus, human interaction would be needed in order to decide which solution to apply.

As en example, in order to generate a fix for the *completeness* anomaly we would have to create an ATL helper able to produce a textual representation of a new security

Listing 10: ATL rule to calculate completeness property

```
-- @path ANO=anomalies.ecore
-- @path PSM=ServletSecurity.ecore
module checkingCompleteness;
create OUT : ANO from IN : PSM;

helper def : HTTP_METHODS : Sequence(OclAny) =
    ...

helper def : ALL_HTTP_METHODS : Sequence(PSM!HttpMethod) =
    ...

helper context PSM!HttpMethod def : isComplete : Boolean =
    ...

rule HttpMethod2Completeness {
  from
    s: PSM!HttpMethod (not s.isComplete)
  to
    t: ANO!Completeness (
      description <- 'constraint does not name all HTTP methods',
      t.httpMethod <- s.getUncompleteMethods(),
      t.constraint <- s.refImmediateCompisite().refImmediateComposite()
    )
}
```

Error	Solution
Completeness	Generate a security constraint that: 1) has the same URL pattern 2) explicitly uses the missed HTTP methods 3) provides an empty set of roles so that all access is precluded.
Redundancy	Given two security constraints that are mutually redundant, eliminate the less general of them i.e., the one with the more concrete URL pattern (see Definition 4 in Section 4) and/or set of subjects.
Syntactical	Add an explicit declaration for a non-declared role used in a security constraint.
Shadowing	Given two security constraints that are mutually inconsistent (i.e., they have contradictory effects), one of them needs to be eliminated. User input is required in this case to determine which of the two constraints should be eliminated as this depends of the desired behaviour.
Reachability	The non-accessible resources should be removed from the web application exposed directories and the security constraint removed from the policy.

Table 1: Security error fixes

constraint (by using the XML or the annotations concrete syntax, depending on the original location of the faulty security constraint). This security constraint must contain the same URL pattern of the original security constraint where an HTTP method was named and name all the unnamed methods. As an access decision, the new security constraint will preclude access to the now included, previously unnamed methods (alternatively, unconstrained access to unnamed methods could be granted). Listing 11 shows the code of a Helper implementing the aforementioned completeness quick-fix. It takes as context a HTTP method and generates a security constraint that, for the same URL pattern, precludes all access for all the previously unnamed HTTP meth-

Listing 11: ATL Helper to generate completeness quick-fix for XML

```
1  helper context PSM!HttpMethod def : quickFix(source : String) : String =
2     let unnamedMethods : Sequence(String) =  s.getUncompleteMethodsNames() in
3     if source = 'XML' then
4      return '<security-constraint>
5            <web-resource-collection>
6              <url-pattern>' + self.getUrlPattern() + '<url-pattern>
7                <http-method>' + unnamedMethods + '</http-method>
8            </web-resource-collection>
9            </auth-constraint>
10         <security-constraint>'
11    else
12     return '@HttpMethodConstraint(value="'+ unnamedMethods +'",
13         emptyRoleSemantic = EmptyRoleSemantic.DENY))'
14    endif
15 ;
```

ods. Note that it produces a different textual output depending on the source of the original security constraint (XML or Java annotation).

7. Other Applications of the MDE Extraction and Property Analysis processes

Having all the access-control information of a Java EE web application in a single model facilitates the development and reusability of a wide range of model-driven tools and techniques to derive additional interesting analysis applications. In this section we discuss a few of them.

7.1. Queries & Metrics

One of the most immediate applications is, as seen with the evaluation of properties, to use query languages such as OCL to perform queries and calculate security metrics via the information in our model. The range of possible queries and metrics that can be performed goes beyond the properties we have previously discussed. From simple queries such as listing all the resources reachable by a given subject to more complex operations as detecting equivalent roles (a bad smell for a possible break in the *least privilege* strategy), our model representation reduces the complexity of performing any analysis or metric calculation. Instead, working with the original (scattered) policy representation would require the developer to use different tools for each concrete mechanism and a further integration step.

As an example, we show the definition of a query that counts the freely accessible resources. Resources in Java EE applications can be either visible to a reduced set of users having certain roles or being freely accessible to everybody, no matter their roles. A manual inspection of the application to discover the open-access resources can be a tedious task, however leveraging on the Servlet Security metamodel, we can easily define a query to detect and count this kind of resources.

Listing 12 shows a possible OCL query able to detect open-access resources. All the security constraints that do not define an authorization constraint (*authConstraint*) are selected, since they do not enforce any access restriction on the resources declared within the constraint (see lines 2-3). Then, the URLs within each resource contained in the selected constraints are collected (see lines 4-6) and finally counted (see line 7).

Listing 12: metric query to count the open-access resources

```
query reachableResources =
  SEC!SecurityConstraint.allInstances()
  ->select(sc | sc.authConstraint.oclIsUndefined())
  ->collect(sc | sc.webResourceCollection)->flatten()
  ->collect(wrc | wrc.urlPattern)->flatten()
  ->collect(up | up.value)->flatten()
  ->asSet()->asSequence()->size();
```

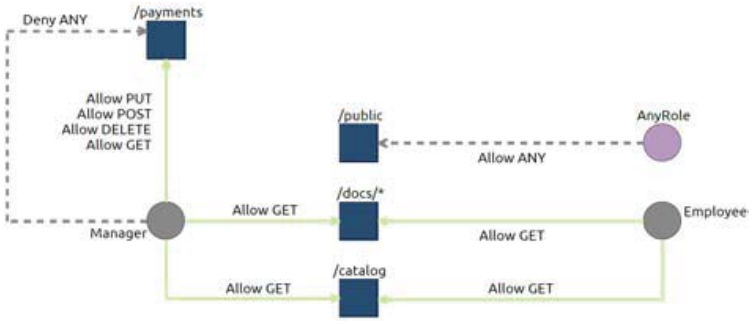

Figure 15: Sirius Policy Visualization Viewpoint

7.2. Visualization

Graphical information is often easier to grasp at a glance than textual one. In this sense model-driven workbench creation tools such as Sirius[3] allow the definition of different viewpoints for a given domain-specific language. Different graphical representations can be obtained without the need of manipulating the source model to create the view.

This way, we can predefine several visualizations for our security data. We could provide a general representation, summarizing the access-control policy of a given application, or more detailed representations showing, for example, the detailed security information related to a given resource or role. As an example, Figure 15 shows the visualization obtained for a simple web application where web resources, roles and the defined constraints are visible (note that, for readability, some relations are omitted).

7.3. Interoperability

Our metamodel and extraction process can be also used as a pivot interoperability mechanism in order to build bridges towards other (model-based) representations. (Partial) translations from our metamodel to more generic access-control languages like SecureUML or XACML [16] would enable the reuse of the vast amount of research performed in the general field of access-control analysis while keeping the specificities of Java EE when required.

Notably, formal validation and verification techniques[17], along with change impact analysis[18] for access-control policies would become available.

[3] http://eclipse.org/sirius/

Listing 13: Acceleo template for test generation

```
[comment encoding = UTF-8 /]
[module generate('http://www.xtext.org/servletSecurity/ServletSecurity')]

[template public generateElement(s : SecurityConstraint)]
    [comment @main /]
    [for (it : DisplayName | s.displayName)]
        [file (it.value.concat('.java'), false, 'UTF-8')]
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
public Class IntegrationTests {

    @Test
    public void testAccessTo[it.value/] {
       //Create a new instance of Firefox Browser
       WebDriver driver = new FirefoxDriver();
       //Open the URL in firefox browser
       driver.get("[s.webResourceCollection.urlPattern.value/]");
       //Check whether the WebElement containing the login form is present
       boolean isPresent = driver.findElements(By.id("id-of-the-form")) > 0;
       [if (s.authConstraint.oclIsUndefined())]
        Assert.assertFalse("NoLoginFormPresent", isPresent);
        [else]
            Assert.assertTrue("NoLoginFormPresent", isPresent);
       [/if]
    }

}
        [/file]
    [/for]
[/template]
```

7.4. Forward Engineering: Test Generation

As part of the forward engineering of web applications, our Servlet security models could also be used in order to generate integration tests aimed to verify the function and reliability of the web application w.r.t. a given security policy. In this sense we can generate tests evaluating if the access to certain resources is as established in the declarative policy. This would ease performing regressions testing in case modifications are added to the policy and would also help to detect possible inconsistencies between the declarative rules and other access rules that may have been added programmatically.

As an example, an Acceleo[19] model-to-text transformation that generates tests using Selenium [4], a web testing tool, can be integrated so that analysis and verification of the security policy becomes a part in the application testing workflow.

Listing 14 shows a JUnit test that verifies if the access to a resource that is constrained in the security policy configuration actually requires the remote users to authenticate by prompting a login and password form (note that we could also use Selenium to actually log into the application and thus, simulate the navigation with different roles). It is automatically generated by using the Acceleo template excerpt shown in Listing 13. The template uses as input a model conforming to our Servlet security Metamodel to generate a JUnit test for each *Security Constraint*.

[4] http://www.seleniumhq.org/

Listing 14: Selenium access test

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

public class IntegrationTests {
    private static FirefoxDriver driver;
    WebElement element;

    @BeforeClass
    public static void openBrowser(){
        driver = new FirefoxDriver();
        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
    }

    @Test
    public void testRestrictedAccess {
       //Create a new instance of Firefox Browser
       WebDriver driver = new FirefoxDriver();
       //Open the URL in firefox browser
       driver.get("https://restricted/employee/");
       //Check whether the WebElement containing the login form is present
       boolean isPresent = driver.findElements(By.id("id-of-the-form")) > 0;
       Assert.assertTrue("NoLoginFormPresent", isPresent);
    }

    @AfterClass
    public static void closeBrowser(){
        driver.quit();
    }
}
```

Other similar tests can be generated to validate other properties, e.g., tests verifying HTTP methods, etc. Moreover, by directly manipulating our Servlet Security model we can create an objective access control policy and test if the deployed access-control policy conforms to the rules in the model by providing the same set of access to web resources.

8. Evaluation

In order to evaluate the efficacy and relevance of our approach we have addressed three research questions:

- *R.Q.1.* Do the anomalies described in Section 4 occur in existing Java EE projects?
- *R.Q.2.* Is our approach capable of automatically evaluating these properties over existing projects in a correct manner?
- *R.Q.3.* Is our approach efficient w.r.t. time?

To answer them we have conducted an evaluation based in the analysis of a set of projects embedding Java EE web security information randomly sampled from GitHub.

8.1. R.Q.1 Analysis

We have used the GitHub Search API[5] to collect a random sample consisting in 500 GitHub repositories of Java Web EE projects using declarative security. Concretely, we looked for repositories containing the string *javax.servlet.annotation.ServletSecurity* in Java files (required when using servlet security annotations) or the string *<web-resource-collection>* in XML files (the string web-resource-collection is used in security constraints to identify the protected resources).

Due to the limitations of the GitHub Search API, this initial sample may include a high number of meaningless and toy example projects. Thus, we have developed a semi-automatic refinement process aimed to obtain a subset of projects fulfilling two more sophisticated filtering rules:

- A project must not include in either its path or name the following words or variations of them: test, sample, demo, example, tutorial, training, exercise, lesson, helloWorld, practice, template, solution, gettingStarted.
- A project must contain at least five security constraints[6] defined either in the web descriptor (identified by the XML element <web-resource-collection>) or through the annotations (identified by the annotation *@ServletSecurity*).

We applied this process to the initial sample of 500 projects and we obtained 60 projects satisfying the aforementioned constraints [7]. We have then proceeded to analyse these selected projects with our tool (see Appendix A for more information). The results of the analysis are shown in Fig. 16 and commented below:

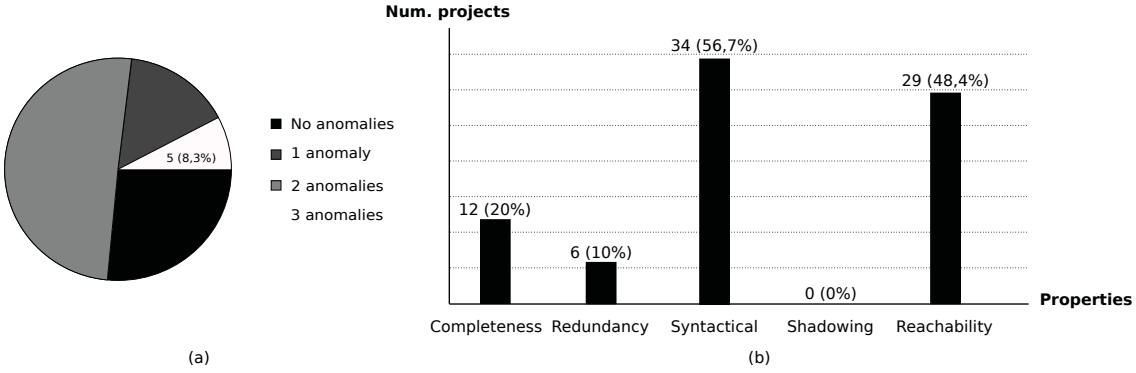

Figure 16: Analysis of the security properties for the selected projects. Subfigure (a) shows the percentage of projects violating properties, from 0 (no property violated) to 3. Subfigure (b) shows the number of projects that violate a given property.

[5] https://developer.github.com/v3/search/

[6] We believe that five security constraints is a fair trade-off between the number of security configurations found and their complexity

[7] The dataset of the collected projects is available at https://drive.google.com/file/d/0B_5Zf1cHxMFXR3hrWXdJa1BlYkE/view

- Around 26% of projects do not present any anomaly, while the remaining projects contain no more than than 3 anomalies (see (a) in the figure). In particular, 15% of the projects contain one anomaly, 50% two and around 8% three. Moreover, the anomalies concerning undeclared roles (Property 3) and resource reachability (Property 5) appear to be the most common ones in our sample (see (b) in the figure). They happened respectively on 56,7% and 48,4% of the projects in the sample.

- None of the projects is affected by shadowing. Shadowing being a very severe but normally quick to notice anomaly (as it modifies the accessibility of a given resource) our intuitions is that this anomaly is one frequently corrected in the initial testing phases. Nevertheless, due to the important consequence of this anomaly, even if it is unlikely to introduce such an anomaly without noticing it, we consider its detection important and thus, keep it in our approach.

- 20% of the projects present a violation of the completeness property (Property 1 in the table) while 10% present redundancy in rules (Property 2).

- Reachability problems are detected in many projects of the sample, however the manual analysis allowed us to discover that the anomaly was actually not taking place in some of the projects. Effectively, those projects were meant to be deployed in the Google cloud services container, where the semantics of the symbol '*' are redefined so that it implies all authenticated users. Nevertheless the detection of this anomaly by our tool remains useful as it provide a hint for possible portability issues.

From these results we can conclude that the answer to our first research question is yes, as indeed, we found a relevant number of anomalies in Java EE projects.

8.2. R.Q.2. Analysis

We have manually inspected 20 of the projects' security configurations (randomly selected from the sample) in order to verify the absence of false positive or negatives.

Table 2 shows, for each of the 20 Java EE projects, the number of rules composing its security configuration, and for each security property, if an anomaly (i.e., property violation) is detected manually and/or automatically. As it can be seen no discrepancies between the automatic and manual analysis are detected and thus we can positively answer to *R.Q.2* w.r.t. the accuracy of our automatic approach.

8.3. R.Q.3. Analysis

Table 2 (and Appendix A) also shows the time taken by our tool to extract models out of the corresponding Java EE web projects (gen.) and to evaluate (eval.) our properties over them. The time for model-extraction depends linearly on the number of Java files whereas the time for calculating the properties, that is much lower does not seem to present a great variability. Table 3 presents some descriptive statistics concerning the extraction and evaluation times, calculated over the 60 projects composing our sample. As can be seen, both times are arguably low for an offline analysis and concretely, the sum of the maximum values for extracting and evaluating the security

Project Name	Rules	Property					Time (seconds)	
		1	2	3	4	5	gen.	eval.
reviewer	8			■□			7.56	0.06
dar	11	■□	■□				4.91	0.14
shop	8	■□					2.18	0.06
Itrust	13						14.64	0.06
jersey	6		■□				0.54	0.1
FanTalk	5			■□		■□	2.7	0.06
wellmia	5			■□		■□	0.12	0.14
adware	8			■□		■□	5.85	0.1
avicena	8			■□		■□	3.08	0.06
BridgeMonitoring	8	■□		■□			1.63	0.09
SkillMaps	7			■□		■□	2.18	0.1
pki	11						0.11	0.08
IrssiNotifierWP7S	11			■□		■□	0.3	0.06
smsr	10						13.98	0.05
tims	7	■□					1.92	0.06
TeBES-war	11			■□		■□	0.48	0.06
KMS	5			■□		■□	0.7	0.08
libreria-project	7	■□					0.03	0.09
planets-suite	5						4.03	0.06
plucial-blog	6			■□		■□	1.63	0.11

■ automatic evaluation
□ manual evaluation

Table 2: Evaluation Results

properties is below one minute (see Max. column in the table). Thus, we can positively answer to *R.Q.3*.

8.4. Threats to Validity

Internal validity: The validity of the conclusions obtained by our experiments may be affected by 1) the degree of relevance of the tested projects and 2) a bias in the selection of the projects. We tackle the first point by establishing a set of conditions to be fulfilled by the projects in order to become part of the experiments. Although this set of conditions may not be complete or may leave out relevant projects, by reducing the number of toy examples, it augments the relevance of the obtained random set. Then, the possible bias in the selection of the projects is avoided by using an automatic tool to obtain the random samples from GitHub.

External validity: Our evaluation evidences that security anomalies do occur in existing Java EE projects. However, as our study uses only projects extracted from GitHub, a threat to the external validity of our results may have been introduced. Although GitHub is currently the most popular coding platform, we can not directly extend our

Table 3: Summary for the extraction and evaluation times (in seconds).

Time	Min	1^{st} Qu.	Median	Mean	3^{rd} Qu.	Max.	SD (σ)
Extraction	0.01	0.58	1.45	4.15	3.11	35.04	7.30
Evaluation	0.05	0.06	0.1	0.99	0.62	14.78	2.45

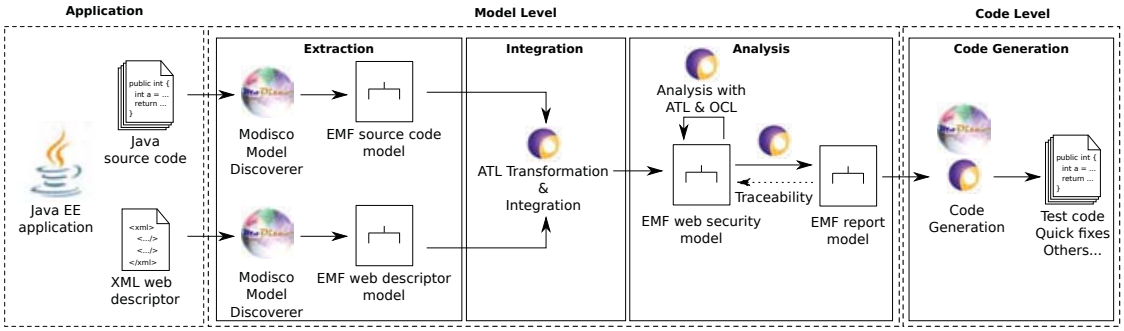

Figure 17: Architecture of our Java EE web application analysis approach

results to the full domain of Java EE web applications, as we can not assure that the subset of projects stored in GitHub is representative of it. In the same sense, there is no guarantee that the selected Github projects have ever been deployed in industrial scenarios.

9. Tool Support

Our tool support covers two different aspects of the work. On the one hand, we have created a tool to randomly sample GitHub projects (to be then analyzed for security issues). On the other hand, a tool has been developed as plugin under the Eclipse[20] platform with the purpose of automatically evaluate the security properties described in Section 4. Both tools are available on GitHub[8].

9.1. Sampling GitHub Projects

The component that creates random samples of projects in GitHub is written in Java and relies on the GitHub Search API [21] to retrieve relevant projects. The tool first parametrizes (based on file size, extension and text string to search) the search operation to retrieve Java EE projects. In a second step, a random sample is derived from the former result set. Repositories of the projects in the sample are automatically downloaded. A manual filtering (according to the set of rules discussed in Section 8) finishes the preparation for the subsequent analysis.

9.2. Evaluating Security Properties

The main building blocks of the tool that can be used to evaluate the security properties of a given Java EE application are depicted in Figure 17 as an instantiation with

[8] `https://github.com/atlanmod/web-application-security`

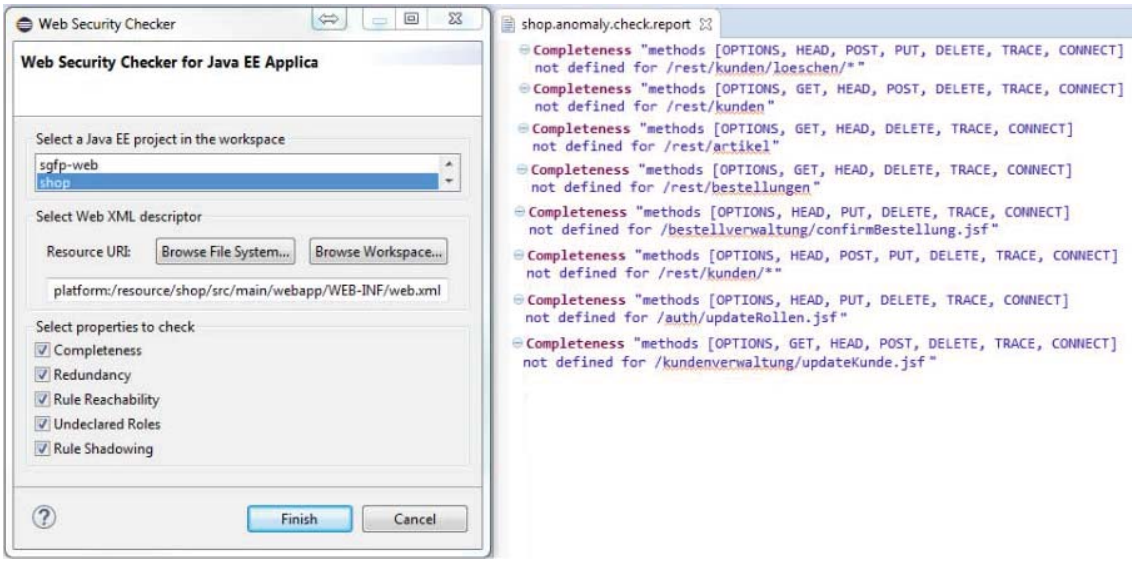

Figure 18: Screenshot of the tool

concrete technologies of our general approach initially presented in Figure 11. It relies on MoDisco [10] (for the model extraction), EMF and ATL (the transformation language used to implement all model manipulation operations required by our approach). The tool comes with a GUI that allows the user to select the Java EE project, its web descriptor and the security properties to check. In Figure 18, the GUI of the tool and the corresponding diagnosis report are shown.

10. Related Work

10.1. Policy Analysis

Analysis of security policies is a research subject widely addressed in different research communities. To give a few examples, [7] presents a general taxonomy of potential anomalies for the domain of network security. These anomalies are formalized for network packet-filtering rules in [4]. [6] tackles security anomalies in database access control policies, concretely, for MySQL grant tables. Similar approaches, specially tailored to the analysis of anomalies in environments where several policies coexist and interact have also been contributed. Concretely, in [22] an approach is presented for policies belonging to different authorization parties, in [23] for policies in different abstractions levels and [24] for policies in different architecture layers. Closer to our web domain, [25] and [5] provide algorithms to detect anomalies in XACML[16] web services policies.

Nevertheless, none of these works can be directly used to validate Java EE security configurations since they would need to be refined to accurately match the execution semantics and combination algorithms required for conflict resolution in Java EE security specifications. Moreover, our approach includes other properties related with the implicit semantics of the Java EE Servlet security or with syntactical misconfigurations (e.g., the absence of declared roles), not part of previous works.

10.2. Policy Modeling

Regarding the modeling of the policies themselves, several mechanisms have been used: internal logical representations, BDDs, XACML (and extensions) and, as our own approach, different kinds of models. Among this last group we have EMF-based platform-specific metamodels [26], SecureUML[27] to model Java EE security policies for code generations purposes and UmlSEC [28] , extension of the UML language to integrate security concepts which is used for instance in [29] to model high-level security properties (requirements) for mobile communication. While some of those languages could have been reused in our approach, we believe that coming up with a Java EE specific metamodel significantly simplifies the manipulation and understanding of the extracted policy. Moreover, being a direct representation of the original configuration policies, forward engineering tasks (i.e. to regenerate code once anomalies are corrected) are also simpler.

10.3. Policy Reverse Engineering

W.r.t. the extraction of access-control policies from deployed components, no specific method is proposed in the previously mentioned works. The few approaches devoted to the extraction of security policies employ dynamic techniques to infer the security policy. In [30] authors use dynamic analysis techniques to extract SecureUML models from PHP web applications. Also for PHP applications, in [31] inter-procedural privilege violations are detected by analyzing automatas extracted from the source code. In [32] a similar approach is applied to the Moodle Web Learning Platform. More recently, in [15], the authors use dynamic analysis and machine learning techniques in order to infer access rules for a given set of users and roles and to test the reachability of web resources (other security properties are not analyzed but left to human expert validation). Finally, not focused in web applications but on Java code, there is an approach to perform interprocedural privilege analysis for Java applications [33]. Instead, we extract the policies using static analysis. In a different line of work, we could combine our approach with some of these other approaches as a way to detect deviations between the running system and the specified security configuration.

11. Conclusion

We have presented an approach to automatically detect security misconfiguration in reverse-engineered Java EE web applications. Our approach is based in the automatic modeling of the declarative security configurations of Java EE web applications and in the definition of a number of security properties whose violation are evidence of the presence of anomalies. We have evaluated the feasibility and pertinence of our approach by using our proof of concept tool on a set of projects retrieved from GitHub. This evaluation has shown that a relevant number of security configurations in actual projects do violate the properties we defined.

We believe that our approach and tool can be useful for the Java EE development community. Besides, as it is fully automated it could be easily adopted and integrated in both, the development phase of Java EE applications, to incrementally asses that security rules are integrated correctly and in a testing phase to assure that already developed applications are free from anomalies.

As future work, we envisage several possible extensions. We would like to extend our framework to include:

1. Programmatic security constraints. We plan to leverage on previous work that focus on extracting business logic embedded in Java code [34]. In particular, we plan to extract Java models using Model-driven reverse engineering techniques and then apply static slicing techniques [35] to identify the security rules in the Java model.
2. Other sources of security information, beyond the Java code itself (e.g. security access constraints from an underlying database system) to provide a more complete analysis of the security of the software system as a whole. We intend to integrate to the present approach database reverse engineering results [36, 37] that explore the extraction of security constraints from working database systems.

Finally, we also plan to adapt our approach to more complex Java frameworks built on top of Java EE, as the Spring framework. These frameworks typically provide their own security mechanisms and therefore may require additional support and/or the extension of several of the artifacts presented here (e.g., the Servlet Security Metamodel may need to be extended to be able to represent constraints with a higher level of granularity).

Regarding the applications, we would like to enhance the semi-automatic integration test generation application we have discussed here. We intend to investigate the possibility of incorporating reasoning mechanisms able to automatically generate a more comprehensive battery of integration test for web applications.

[1] X. Li, Y. Xue, A survey on server-side approaches to securing web applications, CSUR 46 (4) (2014) 54.

[2] OWASP Foundation, OWASP top 10 - 2013, Tech. rep., OWASP Foundation (2013).

[3] R. Sandhu, D. Ferraiolo, R. Kuhn, The NIST model for role-based access control: towards a unified standard, in: RBAC'00, ACM, 2000, pp. 47–63.

[4] J. G. Alfaro, N. Boulahia-Cuppens, F. Cuppens, Complete analysis of configuration rules to guarantee reliable network security policies, JIS 7 (2) (2008) 103–122.

[5] H. Hu, G.-J. Ahn, K. Kulkarni, Anomaly discovery and resolution in web access control policies, in: SACMAT'11, ACM, 2011, pp. 165–174.

[6] M. Shehab, S. Al-Haj, S. Bhagurkar, E. Al-Shaer, Anomaly discovery and resolution in MySQL access control policies, in: DEXA'12, Springer, 2012, pp. 514–522.

[7] H. Hamed, E. Al-Shaer, Taxonomy of conflicts in network security policies, Communications Magazine, IEEE 44 (3) (2006) 134–141.

[8] S. Preda, N. Cuppens-Boulahia, F. Cuppens, J. García-Alfaro, L. Toutain, Model-Driven Security Policy Deployment: Property Oriented approach, in: ESSoS'10, 2010, pp. 123–139.

[9] S. Martínez, V. Cosentino, J. Cabot, Model-based Analysis of Java EE Web Security Configurations, in: MISE workshop, IEEE Press, 2016, To Appear.

[10] H. Bruneliere, J. Cabot, G. Dupé, F. Madiot, Modisco: A model driven reverse engineering framework, IST 56 (8) (2014) 1012–1032.

[11] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: a model transformation tool, SCP 72 (2008) 31–39.

[12] OMG, OCL, version 2.0, Object Management Group (June 2005).

[13] M. Richters, M. Gogolla, Validating UML models and OCL constraints, in: UML'00, Springer, 2000, pp. 265–277.

[14] E. Framework, Emf: Ocl plugin for the eclipse modeling framework (2011), URL http://www. eclipse. org/emf.

[15] H.-T. LE, D. C. Nguyen, L. Briand, B. Hourte, Automated inference of access control policies for web applications, in: SACMAT'20, 2015.

[16] H. Lockhart, B. Parducci, A. Anderson, OASIS XACML TC (2013).

[17] Y. Ledru, N. Qamar, A. Idani, J.-L. Richier, M.-A. Labiadh, Validation of security policies by the animation of z specifications, in: SACMAT'11, ACM, 2011, pp. 155–164.

[18] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, M. C. Tschantz, Verification and change-impact analysis of access-control policies, in: ICSE'27, ACM, 2005, pp. 196–205.

[19] J. Musset, É. Juliot, S. Lacrampe, W. Piers, C. Brun, L. Goubet, Y. Lussaud, F. Allilaire, Acceleo user guide, See also http://acceleo. org/doc/obeo/en/acceleo-2.6-user-guide. pdf 2.

[20] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework 2.0, 2nd Edition, Addison-Wesley Professional, 2009.

[21] GitHub Search API. https://developer.github.com/v3/search/.

[22] P. Mazzoleni, B. Crispo, S. Sivasubramanian, E. Bertino, XACML Policy Integration Algorithms, TISSEC 11 (1) (2008) 4.

[23] S. Davy, B. Jennings, J. Strassner, The Policy Continuum–Policy Authoring and Conflict Analysis, Computer Communications 31 (13) (2008) 2981–2995.

[24] S. Martínez, J. García-Alfaro, F. Cuppens, N. Cuppens-Boulahia, J. Cabot, Model-driven integration and analysis of access-control policies in multi-layer information systems, in: IFIP SEC'15, Springer, 2015, pp. 218–233.

[25] V. C. Hu, E. Martin, J. Hwang, T. Xie, Conformance checking of access control policies specified in XACML, in: COMPSAC'07, Vol. 2, IEEE, 2007, pp. 275–280.

[26] D. Basin, J. Doser, T. Lodderstedt, Model driven security: From uml models to access control infrastructures, TOSEM 15 (1) (2006) 39–91.

[27] T. Lodderstedt, D. Basin, J. Doser, Secureuml: A uml-based modeling language for model-driven security, in: UML'02, Springer, 2002, pp. 426–441.

[28] J. Jürjens, UMLsec: Extending UML for secure systems development, in: UML'02, Springer, 2002, pp. 412–425.

[29] J. Jürjens, J. Schreck, P. Bartmann, Model-based security analysis for mobile communications, in: ICSE'08, ACM, 2008, pp. 683–692.

[30] M. H. Alalfi, J. R. Cordy, T. R. Dean, Recovering role-based access control security models from dynamic web applications, in: ICWE'12, Springer, 2012, pp. 121–136.

[31] D. Letarte, E. Merlo, Extraction of inter-procedural simple role privilege models from php code, in: WCRE'09., IEEE, 2009, pp. 187–191.

[32] F. Gauthier, D. Letarte, T. Lavoie, E. Merlo, Extraction and comprehension of moodle's access control model: A case study, in: PST'11, IEEE, 2011, pp. 44–51.

[33] L. Koved, M. Pistoia, A. Kershenbaum, Access rights analysis for java, in: ACM SIGPLAN Notices, Vol. 37, ACM, 2002, pp. 359–372.

[34] V. Cosentino, J. Cabot, P. Albert, P. Bauquel, J. Perronnet, A model driven reverse engineering framework for extracting business rules out of a Java application, in: International Workshop on Rules and Rule Markup Languages for the Semantic Web, Springer, 2012, pp. 17–31.

[35] M. Weiser, Program slicing, in: Proceedings of the 5th international conference on Software engineering, 1981, pp. 439–449.

[36] S. Martinez, V. Cosentino, J. Cabot, F. Cuppens, Reverse engineering of database security policies, in: International Conference on Database and Expert Systems Applications, Springer, 2013, pp. 442–449.

[37] V. Cosentino, S. Martinez, Extracting UML/OCL integrity constraints and derived types from relational databases, in: 13th International Workshop on OCL, Model Constraint and Query Languages, 2013.

Appendix A. Automatically analysed projects

GitHub Owner/Repository	Project	Constraints	Property					Time (sec.)	
			1	2	3	4	5	gen.	eval.
sinaptico/reviewer	reviewer	8			■			7.56	0.06
servetechgit/ZCoreDar	dar	11	■	■				4.91	0.14
memo1012/shop	shop	8	■					2.18	0.06
ekirkilevics/itrust	Itrust	13						14.64	0.06
richeso/jersey	jersey	6		■				0.54	0.1
cndoublehero/FanTalk	FanTalk	5			■		■	2.7	0.06
rilkeanheart/wellmia	wellmia	5			■		■	0.12	0.14
mfaqir/adware	adware	8			■		■	5.85	0.1
mfaqir/avicena	avicena	8			■		■	3.08	0.06
andrea-bottoli/DSD	BridgeMonitoring	8	■		■		■	1.63	0.09
YusukeKokubo/SkillMaps	SkillMaps	7			■		■	2.18	0.1
tiran/pki	pki	11						0.11	0.08
Tol1/IrssiNotifierWP7	IrssiNotifierWP7Server	11			■		■	0.3	0.06
chromial/Echo	smsr	10						13.98	0.05
batbold0514/OnlinePos	tims	7	■					1.92	0.062
cristianonovelli/tebes3	TeBES-war	11			■		■	0.48	0.06
AbeHaruhiko/KMS	KMS	5			■		■	0.7	0.08
adricouci/libreria-project	libreria-project	7	■					0.03	0.09
opf-labs/planets-suite	planets-suite	5						4.03	0.06
PluCial/plucial-blog	plucial-blog	6			■		■	1.63	0.11
gilsonsilvati/cid	cid	8						1.27	1.91
sachdeva-vivek/grouper-i2	grouper-i2	5						0.01	0.09
gpanic/rest-shop	rest-shop	16	■	■				0.67	0.08
matheusbersot/siga-uff	siga-uff	8						0.03	0.05
ualerts-org/ualerts-server	ualerts-server	5						0.42	0.17
abhishekmunie/btech-project-lms	btech-project-lms	5			■		■	3.22	1.91
lukewen427/DeploymentTool	api-server	7						0.66	0.06
AurionProject/Aurion_4.1	AurionAdapterWeb	28			■		■	1.7	0.07
AurionProject/Aurion_4.1	AurionGatewayWeb	21			■		■	1.38	0.07
AurionProject/Aurion_4.1	AurionNhinServicesWeb	18			■		■	0.77	0.84
batbold0514/OnlinePos	callcenter	5	■		■			2.15	0.06
Hellek1/viaja-facil	Colectivos GBA	5			■		■	3	0.06
tristanrichard/OOAD—Database	GameWorld	7						1.13	0.06
wiraqutra/my-doctor	my-doctor	5			■		■	0.59	0.08
CCATObservatory/alma-ot	ProjectRepository	8						2.73	0.08
sborpo/sborpoandsolegr7	sborpoandsolegr7	24	■					1.56	0.16
appbas/openshif-sgfpweb	sgfpweb	6						2.08	0.05
carlosthe19916/sistema-financiero	sistema-financiero	13	■					0.03	0.06
skiley22/Telegenda	Telegenda	7			■		■	0.26	0.06
rodericj/TopDish	TopDish	5		■	■		■	1.84	0.05
sysdocearth/MoguMoguRecipe	MoguMoguRecipe	5			■		■	1.14	0.79
fikriauliya/car-rental-management	car-rental-management	9			■			1.06	0.66
TEAMMATES/teammates	teammates	6			■		■	21.57	7.15
ClausPolanka/GAE-Projets	connectr	9			■			0.97	0.35
lrkirven/docked_gae	docked_gae	9			■		■	1.2	0.47
jameswhite/dogtag-pki	dogtag-pki	9			■		■	0.16	0.94
LauraMJ/mscEricsson	group-project	32		■				0.69	0.4
louipark/class	MP2	13						10.32	2.83
google-code-export/partychapp	partychapp	8			■		■	1.52	0.59
aurelios/questionario	questionario	6						0.2	0.13
BillyZafack/repo1	repo1	6			■		■	15.47	4.58
bwgz/safetysim	SafetySim-AppEngine	35			■		■	1.06	0.61
awadTeam/smacrs	smacrs	5						11.25	2.1
kamdjouduplex/tech-biz	tech-biz	16			■		■	0.09	0.11
getse/vgatisiaccmxproject	vinculacion	8	■					4.66	1.78
smhoekstra/iaf	iaf	5						12.48	3.11
renatodanielss/projetoredesocialilto	PrjRedeSocial	11	■		■		■	35.04	14.78
douglasmen/ge	ge	5			■			1.17	0.81
camptocamp/secureOWS	secureOWS	7	■		■		■	34.55	9.13
e-admin/allocalgis	LOCALGIS-SERVER-Geopista	5		■	■		■	0.42	0.14