Access Control Configuration for J2EE Web Applications: A Formal Perspective Research Report

Matteo Maria Casalino[1,2], Romuald Thion[2], Mohand-Said Hacid[2]

[1] SAP Research Sophia-Antipolis, 805 Avenue Dr M. Donat, 06250 Mougins, France
matteo.maria.casalino@sap.com
[2] Université Claude Bernard Lyon 1, LIRIS CNRS UMR 5205, France
{romuald.thion, mohand-said.hacid}@liris.cnrs.fr

Abstract. Business services are increasingly dependent upon Web applications. Whereas URL-based access control is one of the most prominent and pervasive security mechanism in use, failure to restrict URL accesses is still a major security risk. We argue that this risk can be mitigated by providing formal analysis tools to evaluate access control policies as well as the impact of changes on configurations.
In order to tackle this issue, this paper gives a formal semantics for access control constraints standardized in the J2EE Java Servlet Specification, arguably one of the most common framework for web applications. Two different analysis tools are developed on top of this formal building block: a decision engine and a comparison algorithm for change impact of access control configurations. The formal semantics is compared against two major web application containers. The experiments reveal non-compliant access control decisions of these containers and validate our approach.

1 Introduction

The security of web applications has become increasingly important, since organizations have employed them more and more extensively as a lightweight front-end for business services. The J2EE Java Servlet Specification (JSS) [1] standardizes the interface between the J2EE web front-end components and the containers providing their execution environment.

The access control mechanisms described within the JSS belong to two categories: *programmatic security* and *declarative security*. Programmatic security describes functionalities which developers can use to implement security within their applications' code. Declarative security refers to HTTP-based access control specified not in the applications' code but in the container's deployment descriptor. In this case, the container is responsible for enforcement of access control at runtime. More details on declarative security are given in Sec. 2.

Failure to restrict URL accesses and security misconfigurations are considered as top ten Web application security risks by OWASP[3]. Unfortunately, declarative

[3] https://www.owasp.org/index.php/Top_10_2010

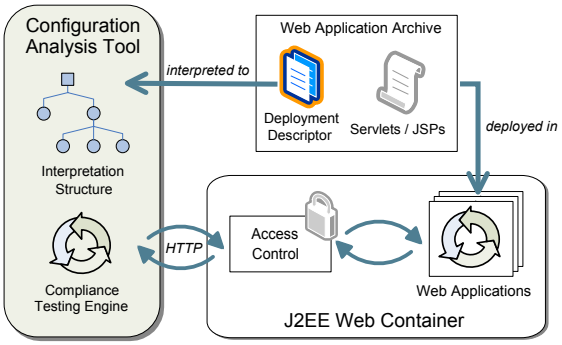

Fig. 1: J2EE Web framework and analysis tool

security semantics of the JSS is described in English prose throughout the document, which can lead to errors due to misinterpretation. Such errors can possibly lead to non-compliant containers' implementations or to security vulnerabilities in access control configurations. In fact, documented misconfiguration vulnerabilities such as [2, 3] prove that even small counter-intuitive fragments of the Servlet specification are among the causes of serious security breaches.

Significance of access control issues in web applications motivate the need for formal verification tools with which security developers can ensure correct behaviours of the policies they define. Interesting analysis tasks include determining whether a given access request is permitted or evaluating the impact of a change within a configuration without running the container. We contribute to solve these problems by defining a formal semantics for JSS declarative security (Sec. 3), from which we provide a query engine to evaluate access control requests and a comparison algorithm for configurations w.r.t. their permissiveness (Sec. 4). Figure 1 depicts the J2EE framework on the right and the related analysis tool, our contribution, on the left.

Together with a prototype implementation, Section 5 compares our semantics with existing web container implementations. Motivation for this experiment is twofold: on one hand we empirically verify that the formal semantics complies with the informal one in the JSS, on the other hand we are able to find cases where containers do not behave as expected. Experiments run on Tomcat and Glassfish application servers have led to the discovery of implementation errors.

Section 6 compares our contributions to related work on XACML, access control frameworks for web-services and other security analysis tools for J2EE compliant applications. We conclude with a summary of the results of our work, a discussion of related work and some future work.

2 Security Constraints

The security-related fragment of the deployment descriptor is composed by the *security constraints* XML tags. For the sake of brevity we provide in Fig. 2

⟨*ac*⟩ ::= '`*`' | '`<`' ⟨*rl*⟩ '`>`'

⟨*rl*⟩ ::= ⟨*empty*⟩ | `role` '`,`' ⟨*rl*⟩

⟨*up*⟩ ::= ⟨*empty*⟩ | `part` | '`*`' | `part` '`/`' ⟨*up*⟩

⟨*upl*⟩ ::= ⟨*up*⟩ | ⟨*up*⟩ '`,`' ⟨*upl*⟩

⟨*ml*⟩ ::= `method` | `method` '`,`' ⟨*ml*⟩

⟨*wrc*⟩ ::= '`{`' ⟨*upl*⟩ '`}`' '`[`' ⟨*empty*⟩ '`]`' | '`{`' ⟨*upl*⟩ '`}`' '`[`' ⟨*ml*⟩ '`]`'

⟨*wrcl*⟩ ::= ⟨*wrc*⟩ | ⟨*wrc*⟩ '`,`' ⟨*wrcl*⟩

⟨*sc*⟩ ::= ⟨*wrcl*⟩ | ⟨*wrcl*⟩ ⟨*ac*⟩

⟨*scl*⟩ ::= ⟨*sc*⟩ | ⟨*sc*⟩ '`\n`' ⟨*scl*⟩

Fig. 2: Shorthand syntax for security constraints [1, Sec. 12]

a BNF grammar modeled from the XML grammar given in the JSS. Access control is configured by associating web resource collections (⟨*wrcl*⟩) to at most one authorization constraint (⟨*ac*⟩), that is the set of roles allowed to access the mentioned resources. A web resource collection consists in a list of URL patterns (⟨*upl*⟩) followed by a (possibly empty) list of HTTP methods (⟨*ml*⟩). The special role name '`*`' is a shorthand for all the roles defined inside the deployment descriptor. Entire URL hierarchies can be specified with URL patterns ending with the '`/*`' wildcard. The initial non-terminal symbol ⟨*scl*⟩ represents a list of security constraints.

Example 1 (Sample security constraints). The following code snippet represents the example of security constraints specification included in [1, Sect. 12.8.2].

```
SC1 = {/*, /acme/wholesale/*, /acme/retail/*} [DELETE, PUT] <>
SC2 = {/acme/wholesale/*} [GET, PUT] <SALESCLERK>
SC3 = {/acme/wholesale/*} [GET, POST] <CONTRACTOR>
SC4 = {/acme/retail/*} [GET, POST] <CONTRACTOR, HOMEOWNER>
```

Constraint `SC1` denies any access to the URL patterns `/*`, `/acme/wholesale/*` and `/acme/retail/*` via the `DELETE` and `PUT` methods. `SC2` and `SC3` allow selected roles `SALESCLERK` and `CONTRACTOR` to access `/acme/wholesale/*`. Both roles can use the `GET` method and `CONTRACTOR` may access via `POST`. Though, `SALESCLERK` cannot access via `PUT` due to `SC1`. Last constraint `SC4` allows both `CONTRACTOR` and `HOMEOWNER` users to access to `/acme/retail/*`. This example will be used throughout this paper, with identifiers abbreviated by their first character (e.g., *a* for `acme`, *c* for `CONTRACTOR`, etc.).

According to the informal semantics from [1], in order to have access granted, a user must be member of *at least one of the roles* named in the security constraint (or implied by '`*`') that matches to her/his HTTP request. An empty authorization constraint means that *nobody* can access the resources, whereas

access is granted to *any* (possibly unauthenticated) user in case the authorization constraint is omitted. Unauthenticated access is also allowed by default to any unconstrained resources.

In case the same URL pattern and HTTP method occur in different security constraints, their authorization constraints have to be composed. If two non-empty authorization constraints are composed, the result is the *union* of the two sets of allowed roles. If one of the two allows unauthenticated access, the composition also does, conceptually resulting again in a *union*. In contrast, if one of the sets of roles is empty, their composition is empty; that is the *intersection* of the two sets is performed in this case. Constraints on more specific URL patterns (e.g. `/a/b`) always override more general ones (e.g. `/a/*`).

If some HTTP methods are explicitly mentioned in a web resource collection, all the other methods are unconstrained, whereas, if none is named, every method is implicitly constrained. Verb tampering attacks [2, 3] exploit this behaviour to bypass the access control check. Vulnerable web applications exhibit a deployment descriptor badly configured w.r.t. their implementation. Requests on unconstrained methods (such as `HEAD`) are in fact handled as ordinary ones (e.g., `GET`), instead of correctly returning an appropriate HTTP error to the client. The peculiar handling of unconstrained methods, combined with the fact that most specific constraints take precedence, leads to particularly counterintuitive behaviours, as illustrated by the following example.

Example 2 (Combination of security constraints). Let us consider that a web developer adds the new constraint `SC5 = {/a} [GET] <H>` to the set of constraints given in Example 1. With this new rule, `(/a, PUT)` and `(/a, DELETE)` accesses are granted to anyone, even unauthenticated users! This is the case because `/a` is more specific than `/*`, hence `SC1` is overridden by `SC5`. However the latter does not define behaviour for `PUT` and `DELETE` methods, so default allow policy is applied.

3 Formal Semantics

In this section, we provide a formal semantics for the language given in Fig. 2. In order to do so, a function $[\![\phi]\!]_{LIT}$ is defined for each non-terminal symbol $\langle lit \rangle$ in the grammar. These functions derive from case analysis on the structure of the language. Terminal symbols are interpreted within an associated domain. For instance, role literals in '`role`' are interpreted by the function $[\![\cdot]\!]_R : \text{`}\texttt{role}\text{'} \to \mathcal{R}$ in elements of the roles domain $\mathcal{R}$. Likewise $[\![\cdot]\!]_M$ maps '`method`' literals in the domain of HTTP methods $\mathcal{M}$, and $[\![\cdot]\!]_S$ interprets URL '`part`'s in an infinite domain of strings $\mathcal{S}$. The final interpretation function is $[\![\phi]\!]_{SCL}$, that is, the interpretation of initial symbol of the grammar.

Authorization Constraints. The interpretation function of authorization constraints is given by the function $[\![\cdot]\!]_{AC}$ defined in (1). This function maps every non terminal $\langle ac \rangle$ to an element of the powerset of the role domain. The function $[\![\cdot]\!]_{RL}$ defined in (2) *folds* roles into a set. Fold, also known as reduce or

accumulate, is a standard high-order functional operation on containers. It has an intuitive meaning: for instance, according to (2), the syntactic role list of `SC4` is turned into a subset of $\mathcal{R} = \{s, c, h\}$, $[\![\texttt{<C, H>}]\!]_{RL} = \{c, h\}$. Similar fold functions are used throughout this section, namely $[\![\cdot]\!]_{UPL}$, $[\![\phi]\!]_{ML}$ and $[\![\cdot]\!]_{WRCL}$ for URLs, methods and web resource collections respectively. Their definitions rest on the same principle and hence are not reported here.

$$[\![\phi]\!]_{AC} = \begin{cases} \mathcal{R} & \text{if } \phi = \text{`*'} \\ [\![\langle rl \rangle]\!]_{RL} & \text{if } \phi = \text{`<'}\langle rl \rangle\text{`>'} \end{cases} \tag{1}$$

$$[\![\phi]\!]_{RL} = \begin{cases} \emptyset & \text{if } \phi = \langle empty \rangle \\ \{[\![\texttt{role}]\!]_R\} \cup [\![\langle rl \rangle]\!]_{RL} & \text{if } \phi = \texttt{role`,'}\langle rl \rangle \end{cases} \tag{2}$$

In order to capture the semantics of authorization constraints, a partial order $\leq_R$ between sets of roles is defined. To take the case of unauthenticated users into account, the symbol $\top$ is added. The role lattice $\mathcal{R}^*$ is the complete lattice given by the powerset of the role domain, ordered by set inclusion, and containing the additional element $\top \notin \wp(\mathcal{R})$.

Definition 1 (Role Lattice). *The complete role lattice is* $\mathcal{R}^* = \langle \wp(\mathcal{R}) \cup \{\top\}, \leq_R \rangle$, *where* $R_A \leq_R R_B$ *iff* $R_B = \top$ *or* $R_A \subseteq R_B$.

The top element $\top$ semantically corresponds to the default *allow all* authorization constraint implicitly associated with any non-constrained web resource. In contrast, the bottom element $\emptyset$ represents a *deny all* authorization constraint. Definition (3) formally captures the composition rules of different authorization constraints mentioned in page 4. The operator $\otimes : \mathcal{R}^* \times \mathcal{R}^* \to \mathcal{R}^*$ performs composition by relying on the least upper bound lattice operator ($\bigsqcup$). For instance, $\{c, h\} \otimes \top = \top$, $\{c, h\} \otimes \{s, h\} = \{s, c, h\}$, $\{c, h\} \otimes \emptyset = \emptyset$ and $\top \otimes \emptyset = \emptyset$.

$$R_A \otimes R_B = \begin{cases} \emptyset & \text{if } R_A = \emptyset \text{ or } R_B = \emptyset \\ R_A \bigsqcup R_B & \text{otherwise.} \end{cases} \tag{3}$$

Web Resource Collections. The resources being subject to access control in a J2EE web application are URLs. The URL hierarchy must be taken into account while evaluating access control requests, since a URL pattern ending with a wildcard matches every URL sharing its prefix. We therefore interpret URL patterns as a tree, where each node is a prefix-ordered sequence of symbols.

Definition 2 (URL). *A URL* $u \in \mathcal{U}$ *is a (possibly empty) sequence of symbols each one belonging to* $\mathcal{S}$, *and ending with at most one symbol belonging to the set* $\mathcal{E} = \{\epsilon, *\}$[4], *where* $\mathcal{S} \cap \mathcal{E} = \emptyset$ *:*

(i) $u = \langle\rangle$ *is an (empty) URL;*
(ii) $u = \langle s_0, \ldots, s_n \rangle$, $n > 0, s_0, \ldots, s_n \in \mathcal{S}$ *is a URL;*
(iii) $u = \langle s_0, \ldots, s_n, s_e \rangle$, $n > 0, s_0, \ldots, s_n \in \mathcal{S}, s_e \in \mathcal{E} = \{\epsilon, *\}$ *is a URL.*

[4] Symbol ϵ is used to differentiate files from folders, e.g., between `/a/` and `/a`.

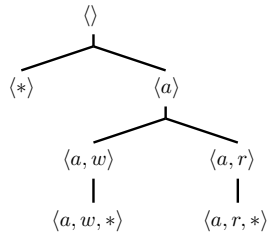

Fig. 3: URL tree of Example 1

For a given URL $u = \langle s_0, \ldots, s_n \rangle$ its *length*, written $|u|$, equals $n + 1$, the length of the empty URL being 0. The l-long *prefix* of u, written $u^{\leq l}$ is the sequence $\langle s_0, \ldots, s_{l-1} \rangle$, with $u^{\leq 0} = \langle\rangle$. The i^{th} symbol s_i of u is written u_i. Equality of URLs is defined in the traditional way. The URL concatenation operator $\oplus : \mathcal{U} \times \mathcal{U} \to \mathcal{U}$ is defined as follows:

$$u \oplus v = \begin{cases} \langle u_0, \ldots, u_{|u|}, v_0, \ldots, v_{|v|} \rangle & \text{if } u_{|u|} \in \mathcal{S} \\ \text{undefined} & \text{if } u_{|u|} \in \mathcal{E} \end{cases} \tag{4}$$

For instance, let $\mathcal{S} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and $u = \langle \mathtt{a}, \mathtt{b} \rangle$. The following equalities hold: $|u| = 2$, $u^{\leq 1} = \langle \mathtt{a} \rangle$, $v = u \oplus \langle \mathtt{c} \rangle = \langle \mathtt{a}, \mathtt{b}, \mathtt{c} \rangle$, $v_2 = \mathtt{c}$, $w = u \oplus \langle \epsilon \rangle = \langle \mathtt{a}, \mathtt{b}, \epsilon \rangle$. Finally $w \oplus \langle \mathtt{c} \rangle$ is not defined.

Definition 3 (URL Tree). *A URL tree $t \in \mathcal{U}^*$ is a non empty, finite, partially ordered set $t = \langle U, \prec \rangle$ with:*

(i) $U \subseteq \mathcal{U}$*;*
(ii) the empty URL always belongs to U*:* $\langle\rangle \in U$*;*
(iii) U *is prefix-closed:* $u \in U$ *and* $|u| > 0 \Rightarrow u^{\leq |u|-1} \in U$*;*
(iv) $u \prec v$ *iff* $|u| \leq |v|$ *and* $u = v^{\leq |u|}$*.*

Proposition 1. *Relation $\prec$ is indeed a partial order for $\mathcal{U}$. Moreover, for any URL tree $\langle U, \prec \rangle$ the set of predecessors of any of its elements $u{\downarrow} = \{p : p \prec u\}$ is well-ordered.*

Proposition 1 ensures that a URL tree is indeed a tree according to the set-theoretic definition. Figure 3 depicts the URL tree corresponding to the interpretation of all the URL patterns in Example 1.

Every URL pattern $\langle up \rangle$ is interpreted as a URL through the function $[\![\cdot]\!]_{UP}$ recursively defined in (5). Intuitively, a URL is simply a sequence of identifiers (‘`part`’s) separated by the ‘`/`’ character. For instance, concrete URL pattern `/a/r/*` from constraint `SC1` is turned into the sequence $[\![\mathtt{\{/a/r/*\}}]\!]_{UP} = \langle a, r, * \rangle$.

$$[\![\phi]\!]_{UP} = \begin{cases} \langle \epsilon \rangle & \text{if } \phi = \langle \mathit{empty} \rangle \\ \langle * \rangle & \text{if } \phi = \text{‘}\mathtt{*}\text{’} \\ \langle [\![\mathtt{part}]\!]_S \rangle & \text{if } \phi = \mathtt{part} \\ \langle [\![\mathtt{part}]\!]_S \rangle \oplus [\![\langle up \rangle]\!]_{UP} & \text{if } \phi = \mathtt{part}\text{‘/’}\langle up \rangle \end{cases} \tag{5}$$

Combination of URL patterns and HTTP methods into web resource collections is done by performing the cartesian product of the two sets by means of the function $[\![\cdot]\!]_{WRC}$ defined in (6). This definition is consistent with the JSS, since it states that naming no methods means that every method is constrained. For instance, if we consider `SC4` from Example 1, then $[\![$`{/a/r/*}, [GET, POST]`$]\!]_{WRC}$ is the set with two elements $\{\langle\langle a, r, *\rangle, GET\rangle, \langle\langle a, r, *\rangle, POST\rangle\}$.

$$[\![\phi]\!]_{WRC} = \begin{cases} [\![\langle upl\rangle]\!]_{UPL} \times \mathcal{M} & \text{if } \phi = \langle upl\rangle \\ [\![\langle upl\rangle]\!]_{UPL} \times [\![\langle ml\rangle]\!]_{ML} & \text{if } \phi = \langle upl\rangle\langle ml\rangle \end{cases} \tag{6}$$

Security Constraints. URL trees are to be mapped to the roles allowed to access each node in the tree. Such a mapping is given by the interpretation of security constraints within *Web application Access Control Trees.*

Definition 4 (Web application Access Control Tree). *A Web application Access Control Tree (WACT) is a pair $t = \langle U, \alpha\rangle$, where $U \in \mathcal{U}^*$ is a URL tree as defined in Def. 3 and $\alpha : U \times \mathcal{M} \to \mathcal{R}^*$ is a partial function giving the set of roles allowed to access a pair $\langle u, m\rangle$. The set of all WACTs is $\mathcal{T}$.*

A security constraint $\langle sc\rangle$ is interpreted as a WACT through the function $[\![\phi]\!]_{SC}$ which maps a constraint to a WACT $\langle U, \alpha\rangle$. Definition (7) computes the prefix-closure $U \in \mathcal{U}^*$ of every URL in the web resource collections, for instance $\{\langle\rangle, \langle a\rangle, \langle a, r\rangle, \langle a, r, *\rangle\}$ is the prefix closure of $\langle a, r, *\rangle$. The partial function α defined by (8) maps every web resource collection to the $\top$ element of the role lattice in case no authorization constraints are specified; otherwise it maps the web resource collection to the set of authorized roles $[\![\langle ac\rangle]\!]_{AC}$.

$$U = \{w \mid w \prec u, w \in \mathcal{U}, \langle u, \cdot\rangle \in [\![\langle wrcl\rangle]\!]_{WRCL}\} \tag{7}$$

$$\alpha(u, m) = \begin{cases} \top & \text{if } \phi = \langle wrcl\rangle \\ [\![\langle ac\rangle]\!]_{AC} & \text{if } \phi = \langle wrcl\rangle\langle ac\rangle \end{cases} \tag{8}$$

Trees obtained from $[\![\phi]\!]_{SC}$ have to be combined since a web applications' deployment descriptor may contain more than one security constraint. The union of two WACTs $\langle U_1, \alpha_1\rangle \dot\cup \langle U_2, \alpha_2\rangle$ is a tree $\langle U_1 \cup U_2, \alpha\rangle$ where α is defined by (9). In the case where both trees define a set of roles for a common pair $\langle u, m\rangle$, corresponding role sets are merged by using the operator $\otimes$ defined by (3).

$$\alpha(u, m) = \begin{cases} \alpha_1(u, m) \otimes \alpha_2(u, m) & \text{if } \langle u, m\rangle \in \mathrm{dom}(\alpha_1) \cap \mathrm{dom}(\alpha_2) \\ \alpha_1(u, m) & \text{if } \langle u, m\rangle \in \mathrm{dom}(\alpha_1) \setminus \mathrm{dom}(\alpha_2) \\ \alpha_2(u, m) & \text{if } \langle u, m\rangle \in \mathrm{dom}(\alpha_2) \setminus \mathrm{dom}(\alpha_1) \end{cases} \tag{9}$$

For example `SC2` and `SC3` turn into the WACTs $t_2 = \langle U_2, \alpha_2\rangle$ and $t_3 = \langle U_3, \alpha_3\rangle$ respectively, with $\alpha_2(\langle a, w, *\rangle, GET) = \{s\}$, $\alpha_2(\langle a, w, *\rangle, PUT) = \{s\}$, $\alpha_3(\langle a, w, *\rangle, GET) = \{c\}$ and $\alpha_3(\langle a, w, *\rangle, POST) = \{c\}$. Their union is the WACT $t_1 \dot\cup t_2 = \langle U, \alpha\rangle$, with $\alpha(\langle a, w, *\rangle, GET) = \{s, r\}$ and $\alpha(\langle a, w, *\rangle, PUT) = \{s\}$, $\alpha(\langle a, w, *\rangle, POST) = \{c\}$. Finally definition (10) folds all the security constraints from a deployment descriptor ($\langle scl\rangle$) and produces a single WACT.

$$[\![\phi]\!]_{SCL} = \begin{cases} [\![\langle sc\rangle]\!]_{SC} & \text{if } \phi = \langle sc\rangle \\ [\![\langle sc\rangle]\!]_{SC} \dot\cup [\![\langle scl\rangle]\!]_{SCL} & \text{if } \phi = \langle sc\rangle\langle scl\rangle. \end{cases} \tag{10}$$

4 Applications

4.1 Dealing with Access Control Requests

According to the JSS [1, Sec. 12.8.3], an access request is a triple $\langle u, m, R\rangle \in \mathcal{U} \times \mathcal{M} \times \mathcal{R}^*$ composed by a URL identifying the requested resource, a HTTP method and an element of the role lattice representing the set of roles assigned to the user who submitted the request. Access control decisions, i.e., $\{false, true\}$ answers to requests, are computed by means of two functions: ρ computes the set of roles needed to access URL u with method m, and Δ determines whether the roles associated with the user issuing the request are sufficient.

Such functions are beneficial to web application developers. For instance, Δ provides an oracle for the access control behaviour of their applications. This oracle can be leveraged to perform security assessment at either design or development time, prior to the deployment phase. Moreover, it can be used to run compliance testing on implementations of the JSS, as elaborated in Sec. 5.

For every URL tree U, we denote the set of $*$-predecessors of $u \in U$ by $u_*{\downarrow}$. The elements of this set are all the immediate successors of the ancestors of u, ending with the symbol $* \in \mathcal{E}$. Formally, $u_*{\downarrow} = \{w \oplus \langle * \rangle \mid w \prec u \wedge w \oplus \langle * \rangle \in U\}$. This behaviour captures the *best match* algorithm of [1, Sec. 12.8.3], which may be informally summarized by "*most specific URL pattern takes precedence*".

Definition 5 (Effective Roles). *Given a WACT $t = \langle U, \alpha\rangle$ the set of effective roles for each couple $\langle u, m\rangle \in \mathcal{U}\times\mathcal{M}$ is given by the function $\rho_{\langle U,\alpha\rangle} : \mathcal{U}\times\mathcal{M} \to \mathcal{R}^*$*

$$\rho_{\langle U,\alpha\rangle}(u, m) = \begin{cases} \alpha(u, m) & \text{if } \langle u, m\rangle \in \mathrm{dom}(\alpha) \\ \alpha(w, m) & \text{else if } \{w\} = \max(u_*{\downarrow}) \wedge \langle w, m\rangle \in \mathrm{dom}(\alpha) \\ \top & \text{otherwise} \end{cases} \tag{11}$$

Function max maps a set of URLs to the subset of them having maximum length. Equation (11) assumes that the set $\max(u_*{\downarrow})$ always contains at most one element. Proposition 2 ensures this property, thus (11) is well-defined.

Proposition 2. *Given a URL Tree $U \in \mathcal{U}^*$ and a URL $u \in U$, the set of $*$-predecessors of u has at most one maximum element.*

Example 3. Let us consider the WACT $t = \langle U, \alpha\rangle$ obtained from Example 1 with $U = \{\langle\rangle, \langle *\rangle, \langle a\rangle, \langle a, w\rangle, \langle a, r\rangle, \langle a, *\rangle, \langle a, w, *\rangle, \langle a, r, *\rangle\}$. On this example $|\,\mathrm{dom}(\alpha)| = 11$, $\alpha(\langle a, w, *\rangle, \texttt{GET}) = \{s, c\}$, $\alpha(\langle *\rangle, \texttt{PUT}) = \{\}$ and $\langle\langle a, w\rangle, \texttt{GET}\rangle \notin \mathrm{dom}(\alpha)$. Thus, we obtain the following values for ρ_t:

(i) $\rho_t(\langle a, w, b\rangle, \texttt{GET}) = \{s, c\}$ because `SC2` and `SC3` apply;
(ii) $\rho_t(\langle a, w\rangle, \texttt{PUT}) = \{\}$ because `SC1` applies;
(iii) $\rho_t(\langle b\rangle, \texttt{GET}) = \top$ because no security constraint applies.

The decision function Δ can be defined straightforwardly from the set of effective roles: access to $\langle u, m\rangle$ is granted either if the user is unauthenticated and the resource accessible to unauthenticated users or if the user endorses at least one role in the set of effective roles associated with $\langle u, m\rangle$.

Definition 6 (Decision Function). *For every $t = \langle U, \alpha \rangle \in \mathcal{T}$ the access control decision function $\Delta_t : \mathcal{U} \times \mathcal{M} \times \mathcal{R}^* \to \{false, true\}$ is defined as follows:*

$$\begin{aligned} \Delta_t(u, m, \top) &= true \quad \textit{iff} \quad \rho_t(u, m) = \top \\ \Delta_t(u, m, R) &= true \quad \textit{iff} \quad \rho_t(u, m) \bigsqcap R \neq \emptyset \end{aligned} \tag{12}$$

4.2 Comparison of authorization constraints

Developers might want to know the impact of changes into security constraints. For instance, one may wish to verify that a new constraint leads to a more restrictive policy. In order to tackle this problem, we define a comparison operator between WACTs according to their permissiveness and show that this order is compatible with access control decisions. A WACT t_1 is less permissive than t_2, written $t_1 \leq_T t_2$ if for any node in the tree and for any method, the set of effective roles of t_1 is included in that of t_2.

Definition 7 (Comparison of WACTs). *Let $t_1 = \langle U_1, \alpha_1 \rangle$ and $t_2 = \langle U_2, \alpha_2 \rangle$*

$$t_1 \leq_T t_2 \textit{ iff } \forall u \in U_1 \cup U_2, m \in \mathcal{M} : \rho_{t_1}(u, m) \leq_R \rho_{t_2}(u, m) \tag{13}$$

Proposition 3 gives an effective method to check whether a configuration is semantically more permissive than another: it is sufficient to verify if inclusion of roles holds for each node in the WACT. If $u \notin U_1 \cup U_2$, then $\rho_{t_1}(u, m) = \rho_{t_2}(u, m) = \top$ by (11), thus only a *finite* set of URLs have to be checked.

Proposition 3. $t_1 \leq_T t_2$ *iff* $\forall u, m, R : \Delta_{t_1}(u, m, R) \Rightarrow \Delta_{t_2}(u, m, R)$.

Example 4. Let us consider Example 2 where `SC5 = {/a} [GET] <H>` is added to the set of constraints of Example 1. We obtain $t = [\![\{\texttt{SC1}, \texttt{SC2}, \texttt{SC3}, \texttt{SC4}\}]\!]_{SCL}$ and $t' = [\![\{\texttt{SC1}, \texttt{SC2}, \texttt{SC3}, \texttt{SC4}, \texttt{SC5}\}]\!]_{SCL}$. On one hand $\rho_t(\langle a \rangle, \texttt{GET}) = \top$ because no constraint applies and $\rho_t(\langle a \rangle, \texttt{DEL}) = \emptyset$ because `SC1` applies, and on the other hand $\rho_{t'}(\langle a \rangle, \texttt{GET}) = \{h\}$ and $\rho_{t'}(\langle a \rangle, \texttt{DEL}) = \top$ because of the presence of `SC5`. Thus, neither $t \leq_T t'$ nor $t' \leq_T t$ holds, whereas intuitively, one expects that by adding `SC5` one will get a more restrictive configuration!

5 Experimental Evaluation

Written in Java language, our implementation of the WACT defined in Sec. 3 is based on a *trie* data structure, that is a prefix-ordered tree where the descendants of every node share a common prefix, which constitutes a natural representation of URLs. Our prototype contains an implementation of $[\![\cdot]\!]_{SCL}$ which compiles security constraints as well as the decision function Δ described in Sec. 4.

On top of this we have developed a *generate and test* methodology to compare the behaviour of existing J2EE application servers according to the formalized version of the JSS. First we generate an exhaustive set of different configurations of security constraints allowed by the grammar (cf. Fig. 2). Since such a set is

infinite we constrain the URL, method and role domains to a finite number of elements. We heuristically determine the value of such elements for some interesting corner cases of the language (e.g., overlapping URL patterns with or without wildcards). In the second phase – as shown in Fig. 1 – for every configuration *scl*, we instrument the application server under scrutiny by deploying a web application having *scl* within its deployment descriptor. At the same time, *scl* is interpreted according to the formal semantics. HTTP requests are then issued to the server and answers are compared to the results computed by Δ for every triple $\langle u, m, R \rangle$.

We conducted our experiments on Apache Tomcat version 6.0.35 and Oracle Glassfish version 3.1.2, two popular Java web application servers supporting the JSS v2.5, which we considered in this paper. The results provide evidence that the implementations did not comply with the specification for a number of tested configurations. In particular we noticed that the configurations producing a misbehaviour in Glassfish all share a common pattern, where one or more constraints apply to the context root (`/`) while different constraints are defined over `/*`. An example of one such a configuration is given by the two security constraints `SC6 = {/*} []` and `SC7 = {/} [] <>`. In fact, access to the URL `/` should always be denied, as `SC7` is more specific, in contrast Glassfish grants access to any client in this case.

Note that this faulty behaviour is not verified in Tomcat. However Tomcat is not fully compliant with the JSS. Further investigations revealed that its misbehaviours are not deterministic, but may disappear upon restarting the application server.

6 Related Work

Many proposals dealing with the formalization of industry standards can be found in literature. A prominent example, concerning access control, is given by XACML, a standard for specifying and enforcing access control policies. Its level of generality and expressivity is able to capture a broad class of access control requirements. However, XACML is quite a complex policy language with informal evaluation semantics, so the development of tools complementing testing with formal verification of XACML is difficult. To tackle this issue, different formal semantics have been given to core concepts of XACML using for instance process algebra [4], description logics [5], answer set programming [6], specific algebraic variety [7] or *ad hoc* compositional semantics [8].

It is tempting to translate J2EE security constraints into XACML and then rely on cited works to benefit from a formal semantics. Unfortunately, some of the selected subsets of the XACML language are incomparable and it seems there is no consensual agreement on its formal semantics, see related work of [8] for discussion and examples. Moreover, we argue that a direct semantics for J2EE security constraints from its specification without intermediate rewriting provides valuable insights to the policy developers.

Instead of working on a concrete language like XACML suffering from a lack of formal foundations, researchers have proposed access control languages with formal semantics. Several models have been proposed for specific domain of web services. For instance, the authors of [9] provide a model with identity attributes and service negotiation capabilities as key features. Attribute-based models remove the subject identification constraint in access control by allowing to specify *who* can access a resource by means of attributes the subject must have [10, 11]. Such an approach is particularly well suited to open environments where the set of all subjects cannot be known in advance. Those works are valuable as both sources of inspiration for new features and theoretical foundations for next versions of the J2EE standard.

In this paper we considered another challenge: in order to provide formal verification tools for concrete problems of querying and comparison, we do not design a language from scratch and give its formal semantics *a priori*, instead we analyse an existing language and give its semantics *a posteriori*. As the semantics of J2EE security constraints is quite specific, it is not clear whether the language can be translated into another one or not. For instance, the Malgrave System [12] is a powerful change impact assessment tool based on a restricted sub-language of XACML. However, hierarchical resources, which are the core of J2EE security constraints and very common in web oriented models, are not supported.

Related work on J2EE access control configurations analysis [13, 14] stems from premises analogous to ours. However these approaches rather focus on checking the consistency of programmatic access control configurations w.r.t. the implementation of J2EE components of the business tier [13] or both business and web tiers [14], in order, e.g., to detect accesses to EJB fields or methods inconsistent with the access control policy. Our work focusing on declarative security is complementary: our formalization supports other reasoning tasks, such as the comparison of different configurations.

7 Conclusion

In this paper we have proposed a formal role-based access control framework for hierarchical resources, able to effectively capture the semantics of the declarative security fragment of the J2EE Java Servlet Specification. This section discusses some of our choices and suggests further research directions.

Precisely, version 2.5 of the JSS specification has been considered. An interesting extension, left to future work, would be to cover version 3.0 of the specification. In this latest revision, configuration authors are allowed to explicitly *omit* selected HTTP methods. Intuitively, assuming the set of HTTP methods to be finite, there exists a sound rewriting for such more expressive configurations towards the ones considered in this paper. More generally, this extension suggests the possibility to deal with more explicit and expressive prohibitions in security constraints.

We highlighted several capabilities of the framework, namely answering to access control requests and comparing the permissiveness of security constraints.

Such tools can help web developers understand the security of their applications to prevent misconfiguration vulnerabilities. To this regard we plan to investigate on more intuitive visualisation techniques for policy design. We envision to provide an environment based on WACT from which configurations will be canonically generated. Such an environment can be complemented with algorithms to detect configurations errors which may possibly reflect authoring mistakes, for instance (non-)monotonicity of permissiveness along URLs paths.

Finally we conducted an evaluation of existing J2EE application server implementations and compared the results to our semantics. An interesting future research direction includes improving the test methodology and coverage, e.g., proving that all the interesting syntactic combinations of configurations are evaluated, and conducting larger experiments.

References

1. Coward, D., Yoshida, Y.: Java servlet specification, version 2.4. Technical report, Sun Microsystems, Inc (November 2003)
2. NIST: CVE-2010-0738. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-0738` (April 2010)
3. Polyakov, A.: A crushing blow at the heart of SAP J2EE engine. `http://dsecrg.com/files/pub/pdf/A%20crushing%20blow%20at%20the%20heart%20SAP%20J2EE%20engine_whitepaper.pdf` (March 2011) 27–29.
4. Bryans, J.: Reasoning about xacml policies using csp. In: SWS '05, New York, NY, USA, ACM (2005) 28–35
5. Kolovski, V., Hendler, J., Parsia, B.: Analyzing web access control policies. In: WWW '07, New York, NY, USA, ACM (2007) 677–686
6. Ahn, G.J., Hu, H., Lee, J., Meng, Y.: Representing and reasoning about web access control policies. In: COMPSAC '10, Washington, DC, USA, IEEE (2010) 137–146
7. Ni, Q., Bertino, E., Lobo, J.: D-algebra for composing access control policy decisions. In: ASIACCS '09, New York, NY, USA, ACM (2009) 298–309
8. Ramli, C.D.P.K., Nielson, H.R., Nielson, F.: The logic of xacml - extended. CoRR **abs/1110.3706** (2011)
9. Bertino, E., Squicciarini, A.C., Paloscia, I., Martino, L.: Ws-ac: A fine grained access control system for web services. World Wide Web **9** (June 2006) 143–171
10. Yuan, E., Tong, J.: Attributed based access control (abac) for web services. In: ICWS '05, Washington, DC, USA, IEEE Computer Society (2005) 561–569
11. Crampton, J., Morisset, C.: Ptacl: A language for attribute-based access control in open systems. In: POST. Volume 7215 of LNCS, Springer (2012) 390–409
12. Fisler, K., Krishnamurthi, S., Meyerovich, L.A., Tschantz, M.C.: Verification and change-impact analysis of access-control policies. In: ICSE, ACM (2005) 196–205
13. Naumovich, G., Centonze, P.: Static analysis of role-based access control in j2ee applications. SIGSOFT Softw. Eng. Notes **29** (September 2004) 1–10
14. Sun, L., Huang, G., Mei, H.: Validating access control configurations in j2ee applications. In: CBSE'08, Berlin, Heidelberg, Springer-Verlag (2008) 64–79

A Proofs of Propositions[5]

Proof (Proposition 1). We first show that $\prec$ is indeed a partial order for $\mathcal{U}$, as it is a reflexive, antisymmetric and transitive relation.

Reflexivity requires proving that $|u| \leq |u|$, which is trivial, and $u = u^{\leq |u|}$ which follows directly from the definition of URL prefix.

Antisymmetry holds since, assuming $u \prec v$ and $v \prec u$, it follows that $|u| = |v|$ and hence $u = v^{\leq |u|} = v^{\leq |v|} = v$.

For proving *transitivity* we first state a general property which follows from the definition of URL prefix:

$$\forall i, j : i < j \Rightarrow (u^{\leq j})^{\leq i} = u^{\leq i}. \tag{14}$$

In particular, given URLs u and v, s.t. $u = v^{\leq |u|}$, we can take the l-prefix of both sides of the equality $u^{\leq l} = (v^{\leq |u|})^{\leq l}$ and conclude, by (14):

$$l \leq |u| \Rightarrow u^{\leq l} = (v^{\leq |u|})^{\leq l} = v^{\leq l}. \tag{15}$$

Therefore, assuming $u \prec w$ and $w \prec v$, we directly have $|u| \leq |w| \leq |v|$ and $w = v^{\leq |w|} \Rightarrow w^{\leq |u|} = v^{\leq |u|} = u$, hence $u \prec v$.

To prove that $u\!\downarrow$ is well-ordered according to $\prec$, we shall prove that (i) $\prec$ is a total order for $u\!\downarrow$ and (ii) all its subsets have a least element.

For (i) we need to show that $\forall v, w \in u\!\downarrow$: $v \prec w \vee w \prec v$. Let $|w| \leq |v|$. Since $v \in u\!\downarrow$, then $v \prec u$, hence $v = u^{\leq |v|}$. As $|w| \leq |v|$, we can write $v^{\leq |w|} = u^{\leq |w|} = w$ (15), hence $w \prec v$. Assuming $|v| \leq |w|$ we would analogously obtain $v \prec w$.

For (ii) we want to prove that $\forall S \in \wp(u\!\downarrow)$ and $\forall v \in S$, $\exists w \in S : w \prec v$. It is easy to see that such element is the URL w having minimum length in S, because $\forall v \in S$ we have $w \prec u$, $v \prec u$ and $|w| \leq |v|$, hence $w \prec v$.

Proof (Proposition 2). We rewrite $u_*\!\downarrow$ in terms of the set of u predecessors $u\!\downarrow$, as $u_*\!\downarrow = \{w' \oplus \langle * \rangle \mid w' \in u\!\downarrow \wedge w' \oplus \langle * \rangle \in U\}$. We know from Proposition 1 that $u\!\downarrow$ is totally ordered w.r.t. $\prec$, therefore $\forall v, v' \in u\!\downarrow \Rightarrow v \prec v' \vee v' \prec v$. If we assume $v \neq v'$, it follows from Definition 3 that either $|v| < |v'|$ or $|v'| < |v|$ hold, hence:

$$v, v' \in u\!\downarrow \wedge\ v \neq v' \Rightarrow |v| \neq |v'|. \tag{16}$$

We now observe that all the URLs $v \in u\!\downarrow$ with $v \neq u$ can't end with the $*$ symbol by definition. In fact, if such a URL w could exist, then we would have $w = u^{\leq |w|} = \langle \ldots, * \rangle$, and therefore $u = \langle \ldots, *, \ldots \rangle$ which is not a URL according to Definition 2. We can then write $v \in u\!\downarrow \Rightarrow v_{|v|} \neq *$ leading, according to (4), to the following conclusion:

$$v \in u\!\downarrow \Rightarrow |v \oplus \langle * \rangle| = |v| + 1. \tag{17}$$

[5] Note for the reviewer. This appendix is only provided for the sake of completeness, but it is not meant to appear in the proceedings. This extended version of the paper will be made available on-line for interested readers.

From both (16) and (17) it follows that the URL length function, restricted to the domain of $*$-predecessors $|\cdot| : u_*\downarrow \to \mathbb{N}$, is injective. Indeed $\forall w = (v \oplus \langle * \rangle), \forall w' = (v' \oplus \langle * \rangle)$ with $v, v' \in u\downarrow$ (resp. $w, w' \in u_*\downarrow$), if $v \neq v'$ (equivalently $w \neq w'$) then $|w| \neq |w'|$; formally:

$$w, w' \in u_*\downarrow \wedge w \neq w' \Rightarrow |w| \neq |w'|. \tag{18}$$

Finally, recalling that $\max(u_*\downarrow) = \{v \in u_*\downarrow \mid \forall w \in u_*\downarrow, |w| \leq |v|\}$, we conclude:

(i) if $u_*\downarrow = \emptyset$ then $\max(u_*\downarrow) = \emptyset$, since $\nexists v \in u_*\downarrow$;
(ii) otherwise $\exists!\ w \in \max(u_*\downarrow)$ and w is the longest URL in $u_*\downarrow$. This follows since every distinct element of $u_*\downarrow$ is mapped through the injective function $|\cdot|$ (18) to a distinct element in a finite non-empty subset of $\mathbb{N}$, which is a totally-ordered set according to the natural ordering of integers, and hence it has exactly one maximum element.

Proof (Proposition 3). For the *only if* direction, we assume $t_1 \leq_T t_2$ and $\Delta_{t_1} = true$. According to definition of Δ we have two cases. First case, $r = \top$ and $\rho_{t_1}(u, m) = \top$, but as $t_1 \leq_T t_2$, $\rho_{t_2}(u, m)$ is $\top$ too and $\Delta_{t_2}(u, m, \top) = true$. Second case, $r \neq \top$, so $\rho_{t_1}(u, m) \bigsqcap r \neq \emptyset$, however, as $\mathcal{R}^*$ is a lattice, $\bigsqcap$ is monotonic with respect to $\leq_R$ and $\leq_R \rho_{t_2}(u, m) \bigsqcap r$ is not empty to. In both cases we conclude that $\Delta_{t_2}(u, m, r) = true$.

For the *if* direction, we use proof by contrapositive. Assume we have some u and m such that $r_1 = \rho_{t_1}(u, m)$ and $r_2 = \rho_{t_2}(u, m)$ with $r_2 \leq_R r_1$ and $r_2 \neq r_1$. We consider to cases. If $r_1 = \top$, then it suffices to look at the value for $r = \top$: $\Delta_{t_1}(u, m, \top) = true$ and r_2 is different from $\top$ so $\Delta_{t_2}(u, m, r) = false$ by definition of Δ. Otherwise $r_1 \neq \top$ thus r_1 is some subset of R and $r_2 \subseteq r_1$, so we consider an $x \in r_1 \backslash r_2$ which exists as the difference is not empty. $\Delta_{t_1}(u, m, \{x\}) = true$ and $\Delta_{t_2}(u, m, \{x\}) = false$ as x is in r_1 but not in r_2.