Detection of Configuration Vulnerabilities in Distributed (Web) Environments*

Matteo Maria Casalino, Michele Mangili, Henrik Plate, and Serena Elisa Ponta

SAP Research Sophia-Antipolis, 805 Avenue Dr M. Donat, 06250 Mougins, France
{matteo.maria.casalino,henrik.plate,serena.ponta}@sap.com

Abstract. Many tools and libraries are readily available to build and operate distributed Web applications. While the setup of operational environments is comparatively easy, practice shows that their continuous secure operation is more difficult to achieve, many times resulting in vulnerable systems exposed to the Internet. Authenticated vulnerability scanners and validation tools represent a means to detect security vulnerabilities caused by missing patches or misconfiguration, but current approaches center much around the concepts of hosts and operating systems. This paper presents a language and an approach for the declarative specification and execution of machine-readable security checks for sets of more fine-granular system components depending on each other in a distributed environment. Such a language, building on existing standards, fosters the creation and sharing of security content among security stakeholders. Our approach is exemplified by vulnerabilities of and corresponding checks for Open Source Software commonly used in today's Internet applications.

Keywords: configuration validation, detection of misconfiguration, web security, distributed environments.

1 Introduction

The importance of security is nowadays well recognized and mechanisms to enforce it are being developed and adopted within enterprises. However, this is not sufficient to ensure that security requirements are met, as such mechanisms have to be correctly configured and maintained at operations time. In fact, a significant share of vulnerabilities results from security misconfiguration, as shown by data breach reports such as [1], [2] and projects such as the OWASP Top 10 [3]. The reason is that activities targeting the creation and maintenance of a secure setup, such as patch or configuration management, are labor-intense and error-prone. Software vendors, for instance, issue an increasing number of security advisories, while users, on the other hand, struggle to understand if a given vulnerability is exploitable under their particular conditions and requires immediate patching. As another example, configuration best-practice provided

* This work was partially supported by the FP7-ICT-2009.1.4 Project PoSecCo (no. 257129, www.posecco.eu)

A.D. Keromytis and R. Di Pietro (Eds.): SecureComm 2012, LNICST 106, pp. 131–148, 2013.
© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2013

as prose documentation and supposingly supporting system admininistrators, is often very broad and ambiguous.

Due to such difficulties, configuration validation is needed to gain assurance about system security, but again, often requires manual intervention, and thus is time-consuming and limited to samples. New trends focus on providing standards for security automation, e.g., the Security Content Automation Protocol (SCAP, [4]), provided by the National Institute of Standard and Technology (NIST), whose specifications receive a lot of attention in the scope of the configuration baseline for IT products used in US federal agencies [4]. SCAP comprises a language that allows the specification of machine-readable security checks to facilitate the detection of vulnerabilities caused by misconfiguration. While this represents an important step towards the standardization and exchange of security knowledge, SCAP focus on the granularity of hosts and operating systems, and as such cannot be easily applied to fine-granular and distributed system components[1] independent from their environment, e.g., a Java Web Application (JWA). Furthermore, SCAP does not leverage standards and technologies in the area of system and configuration management, in order to, for instance, separate check logic and information about configuration retrieval.

To address these limitations and make the advantages of SCAP available to Web security experts, we propose a SCAP-based language and approach for the declarative specification and execution of checks for sets of fine-granular components depending on each other in a distributed environment. Moreover we separate the check logic from the retrieval of the configuration values for which we rely on existing system management procedures and technologies, e.g., Configuration Management Databases (CMDB) as defined in the IT Infrastructure Library (ITIL). Each check is essentially a set of tests over software component properties - such as the release and patch level - and configuration settings that determine a system component's behavior. Though this is not a limitation of the language, we focus on security checks, i.e., one of the most important usages is the detection of security vulnerabilities. As an example, the language allows the specification of a check to express that the deployment descriptor of any JWA deployed in a Servlet container supporting a Servlet specification version of at least 3.0 must have the `http-only` flag enabled, to prevent the access of client-side scripts to session cookies.

This paper is structured as follows. Sect. 2 introduces a sample system based on common Open Source Software (OSS), introduces a set of scenarios for configuration validation, and derives requirements for a configuration validation language. Sect. 3 presents state-of-the-art with regard to the specification of security checks for software and configuration vulnerabilities. Sect. 4 presents the configuration validation language, while Sect. 5 describes our approach. The paper concludes with an outlook on future work in Sect. 6.

[1] A system component hereby represents a single installation of a software component (or product) in a specific system, such as a given deployment of a Java Web Application in a Servlet container.

2 Use Case and Requirements

This section outlines an example landscape composed of a custom application on top of common OSS, and herewith prototypic for many real-life systems. An overview about network topology and installed software components is shown in Fig. 1. The service provider ACME operates this landscape for its application service "eInvoice", which allows customers to manage electronic invoices, and to make them available to their business partners through the Internet. The application front-end for managing and accessing invoices is implemented as a JWA. Instances of the application, each dedicated to one customer, are deployed in Tomcat, in customer-specific context roots. Tomcat instances run inside an internal subnet, and are proxied by the Apache HTTP Server installed on a physical machine connected to the DMZ. Requests for a customer-dedicated sub-domain of acme.com are forwarded by the reverse-proxy to the respective, customer-dedicated instance of the JWA via the Apache JServ Protocol (AJP).

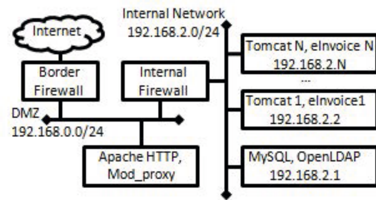

Fig. 1. ACME landscape

Another machine running in the internal network hosts a LDAP server for the management of user accounts, as well as a MySQL database used for persistency.

As the system is prototypic, so are the tasks related to configuration management and validation. In the following, we will describe different scenarios for configuration validation, different in terms of periodicity, urgency (response time), validation scope, and authorship of configuration checks.

Vulnerability Assessment (S1). This scenario focuses on the detection of known vulnerabilities. Upon disclosure of a new security vulnerability of off-the-shelf applications or software libraries, system administrators need to investigate the susceptibility of their system. First, they need to check for the presence of affected release and patch levels. This can be difficult in case of software libraries embedded into off-the-shelf applications as their presence is often unknown. Second, they need to check whether additional conditions for a successful exploitation are met. Such conditions often concern specific configuration settings of the affected software, as well as the specific usage context and system environment. The automation of both activities with help of machine-readable vulnerability checks decreases time and effort required to discover a system vulnerability, and at the same time increases the precision with which the presence of vulnerabilities can be detected. Precision is important as organizations are typically reluctant to apply patches or other measures in a productive environment unless absolutely necessary. Such checks would represent a valuable complement to textual descriptions published by security researchers or software vendors in

vulnerability databases such as the NVD [4]. As an example, CVE-2011-3190[2] reports a vulnerability in the AJP connector implementation of several Tomcat releases [5], which, however, only applies under certain conditions, e.g., if certain connector classes are used, and reverse proxy and Tomcat do not use a shared secret. A machine check looking at the Tomcat release level and related configuration settings could be easily provided by the application vendor (Apache Software Foundation). An example for a critical security bug in a software library is CVE-2012-0392 which describes a vulnerability in Apache Struts, a common framework to support the Model-View-Controller paradigm in JWAs. The detection of this vulnerability is made more problematic by the fact that end-users typically do not know if applications installed in their environment make use of such library, and they cannot rely on the presence of a well-established security response process at each of their application vendors. Thus security bugs may be dormant in libraries without the service operator being aware.

Configuration Best-Practice (S2). This scenario focuses on establishing if best practices are followed. During operations time, system administrators need to periodically check whether the system configurations follow best-practices, for single and distributed system components. Today, these are often described in prose and evolve over time thus requiring continuous human intervention. Examples of best-practices are the Tomcat security guide from OWASP [6], and the SANS recommendations for securing Java deployment descriptors [7].

Example 1 (SANS recommendation on cookie-based session handling). SANS recommends to configure the cookie-based session handling for JWAs (`<cookie-config>` section of the deployment descriptor), i.e., *(i)* preventing the access to session cookies (`<http-only>` set to `true`), and *(ii)* transmitting cookies securely (`<secure>` set to `true`). In particular the `http-only` flag is an example of recommendation that only applies after the release 3.0 of the Servlet specification.

Configuration best-practices may also cover a set of distributed components, e.g., the how-to about Apache HTTP server as a reverse proxy for Apache Tomcat [8]. A language supporting the specification of such best-practice checks should support the flexible adoption to a specific environment. A recommendation related to the session timeout, for instance, may be refined by an organization to reflect its particular policy.

Compliance with Configuration Policy (S3). This scenario focuses on the periodic validation of landscape specific configuration implementing the designed policy. Such a configuration includes a set of mandated configuration settings that an organization expects to be active in its system. As an example, the configuration that enforce the ACME's access control policy embraces configuration settings of several distributed system components, e.g., the realm definition of each Tomcat instance, as well as the deployment descriptor of each Java application instance. In particular the deployment descriptor has to allow the role `admin-role` to access to the URL path `/manager/*`. Moreover the realm of

[2] CVE entries are maintained in vulnerability databases, e.g., NVD.

Tomcat has to refer to the LDAP server located at 192.168.2.1. This example illustrates that configuration checks aiming to assess compliance with a given configuration policy strongly reflect a particular system and environment, and are therefore authored internally to the organization rather than by externals, as in the previous scenarios.

A language for supporting the above scenarios have to fullfill the following requirements.

(RL1) The language must support the definition of configuration checks for diverse software components (e.g., network-level firewalls or application-level access control systems) and diverse technologies.
(RL2) The language must be expressive enough to cover new technologies or configuration formats without requiring extensions. This would avoid the need to update the language interpreter every time a new extension is published.
(RL3) It must be possible to specify target components by defining conditions over properties such as name, release, and supported specification, or over the existence of relationships between components. This is necessary in cases where externally provided checks must be applied to all instances of the affected software components (scenarios S1 and S2).
(RL4) Motivated by scenario S3, it must be possible to specify target components by referring to specific instances of a software component.
(RL5) It must be possible to validate the configurations of different, potentially distributed system components within one check.
(RL6) Checks must be uniquely identifiable, declarative, standardized and certifiable, to support trusted knowledge exchange among security tools and stakeholders, e.g., software vendors, experts, auditors, or operations staff.
(RL7) The language must support parametrization in order to adopt externally provided checks to a specific configuration policy.
(RL8) The specification of checks must be separated from the collection of the involved configuration settings from a given managed domain.

3 State of the Art

Prior art for the definition of the configuration validation language comprises several specifications out of the Security Content Automation Protocol (SCAP), as well as proprietary languages supported by vulnerability and patch scanners.

SCAP [11] is a suite of specifications that support automated configuration, vulnerability and patch checking, as well as security measurement. Some of the specifications are widely applied in industry, e.g., the Common Vulnerabilities and Exposures (CVE, `http://cve.mitre.org`), and those related to configuration validation will be discussed with regard to above-described requirements. Note that several approaches assess a system's overall security level by analyzing and reasoning about the potential combination of individual vulnerabilities (exploits) by an adversary [9], [10]. Though referring to SCAP specifications, these approaches do not look into the vulnerability specification itself, but use the language and related tools merely for the discovery of individual vulnerabilities.

Common Platform Enumeration (CPE, `http://cpe.mitre.org`) is a XML-based standard for the specification of structured names for information technology systems, software, platforms, and packages. It allows the definition of names representing classes of platforms which can be compared in order to establish if, e.g., two names are equal or if one of the names represents a subset of the systems represented by the other. CPE 2.3, the latest version, consists of four modular specifications which work together in layers: *(i)* CPE Naming providing a formal name format, *(ii)* CPE Language allowing the description of complex platforms, *(iii)* CPE Matching providing a method for checking names against a system, and *(iv)* CPE Dictionary binding text and tests to a name.

While the specifications CPE Naming and CPE Matching allow the definition and comparison of single software components according to properties such as vendor or product name, the CPE Language specification does not meet (RL3) with regard to component relations. It supports the specification of a complex platform through a logical condition over several CPE Names, but the semantics of their relationship is not explicitly defined. The typical interpretation used in many CVE entries is that a complex platform condition is met as soon as all software components are installed on the same machine. This interpretation, however, is in many cases not sufficient to state that a vulnerability exists. CVE-2003-0042, for instance, is only exploitable if Tomcat actually uses a given JDK version, the mere presence of both components on the same system is not sufficient. This interpretation is even more misleading if vulnerabilities are caused by combinations of client-side and server-side components, e.g., CVE-2012-0287. A special kind of relationship is the composition of software components, e.g., in the case of Java libraries. Today, each vendor of an application that embeds a vulnerable library needs to issue a dedicated CVE, as CPE insufficient to detect the use of a given library (in an application).

Open Vulnerability Assessment Language. (OVAL, [12]) defines a language for the definition of security tests detecting the presence of vulnerabilities or configuration issues on a computer system (machine). It defines several XML schemas: *(i)* OVAL System Characteristics represent system configuration information that is subject to testing, *(ii)* OVAL Definitions specify conditions for the presence of a specified machine state (vulnerability, configuration, patch state, etc.), *(iii)* OVAL Results report the assessment result, i.e., the comparison of OVAL Definitions and OVAL System Characteristics.

Since OVAL already fulfills some of the before-mentioned requirements, the language proposed in Sect. 4 is to a good extent based on OVAL concepts. According to SCAP design goals, the language supports standardized, unambiguous, and exchangeable representations of configuration checks (RL6) as well as variables for parametrization (RL7). However, a significant limitation is that OVAL checks (like CPE) work on the granularity of machines (computer systems). This impacts several other requirements. With regard to (RL1), it is difficult, sometimes impossible, to write configuration checks for fine-granular system components independently from their software computing environment (container), e.g., JWAs. The reason is that generic OVAL objects from the

`independent schema` (e.g., `textfilecontent54_object`) are relative to the machine's file system, which varies from one Servlet container to the other The definition of container-specific objects (e.g., `spwebapplication_object` for Microsoft Sharepoint), on the other hand, restricts the use of checks to dedicated environments. Requirement (RL2) is not fulfilled as OVAL requires the extension of several schemas to address new software components. This either requires tool vendors to constantly update the language interpreter, or leads to a fragmented market where tools only support a subset of the language. We believe that the broad adoption of OVAL could be reached more easily by the use of generic types (RL2), e.g., on the basis of XML, herewith leveraging the fact that it is used for many application-level configuration formats. With regard to (RL3), (RL4) and (RL5) it is impossible in OVAL to specify a target for checks that look at distributed components, since the execution of a set of OVAL definitions and their tests are meant to be executed on a single machine. Furthermore, OVAL does not clearly separate check logic from the retrieval of the actual configuration values (RL8), herewith missing to leverage industry efforts in the area of IT Service and Application Management (ITSAM). The deployment descriptor of a JWA, for instance, can be retrieved by several means and potentially from different sources (the actual component, or a configuration store with copies). The mixture of these concerns makes the work of check authors difficult and error prone, as they cannot focus on the check logic (e.g., the session configuration of a deployment descriptor), but also care for the retrieval of values, e.g., the identification of a file path depending on installation directories and environment variables. To allow the separation of these concerns, the check language itself must be agnostic to potential configuration sources, the latter being cared for by administrators.

As representative vulnerability and patch scanner, we consider Nessus (`http://www.tenable.com/products/nessus`), which is a widely adopted tool and comes with a proprietary syntax for the definition of so-called audit checks. Organizations can either write custom checks according to this language, or subscribe to a commercial feed to receive compliance checks tailored for a variety of standards and regulations, e.g., PCI DSS (`https://www.pcisecuritystandards.org`). Having comparable expressivity, checks written in this proprietary language can be transformed into SCAP content, which is why Nessus and similar tools were SCAP-validated by the MITRE. SCAP and Nessus' proprietary language also have in common that they focus on operating systems, which makes it difficult to specify checks on a more fine-granular level, i.e., for objects which cannot be easily identified relative to the OS: *custom items* for Windows and Unix require, for instance, the specification of file paths which is not necessarily possible for JWA or Web services; *built-in* checks for Unix hide the configuration source from the check author, but instead of making the source customizable, it is hard-coded (RL9). Checks considering distributed system components are not supported at all (RL5). Nessus does also not allow to condition the applicability of the check

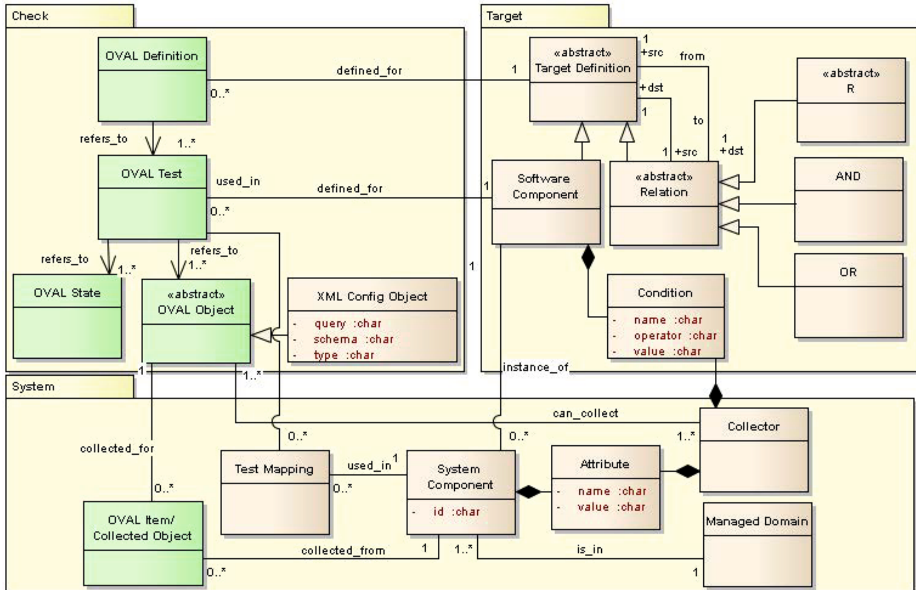

Fig. 2. Configuration validation language class diagram

on the basis of component properties (e.g., release level) or component relationships (RL3) but only on the basis of hard-coded keywords such as `Unix`. As a proprietary language, processed only by Nessus, it is not extensible by 3rd parties (RL3), nor standardized (RL4).

4 Configuration Validation Language

The configuration validation language allows the definition of checks for selected software components and addresses the use cases presented in Sect. 2. It includes the definition of the checks as well as of their results. This section introduces all the concepts used within the language, and defines the extensions we carried out over the OVAL standard. We formally define the semantics of the language without binding to a specific syntax. Notice that in the definitions we only consider the parts of the OVAL standard which are extended by our language. As OVAL is XML-based, a straightforward implementation of our formalism is an XML serialization.

Fig. 2 shows the main concepts of the configuration validation language. The concepts are organized into three main areas. The Check and Target areas concern the definition of the configuration checks and of the affected software components, resp., the System area contains elements corresponding to actual configurations and components of a managed domain.

The *Check* area (top left of Fig. 2) concerns the definition of checks in the form of tests comparing an expected and an actual value. This area relies on the OVAL standard [12]. The concepts we borrow and extend are shown in Fig. 2 and prefixed

with "OVAL". In a nutshell, a definition is characterized by an arbitrary complex boolean combination of tests and a test defines an evaluation involving an object (possibly containing a set of other objects) and zero or more states. As described in Sect. 3, the existing OVAL objects do not fulfill requirements (RL2), and (RL8). To fulfill them, we defined a new test, object, and state, generic enough to apply to multiple configurations of multiple software components and independent from the collection mechanisms. The test and state we defined are not shown in Fig. 2 as it is the object, `XML Config Object`, that contains the major contributions. The `XML Config Object` is characterized by three attributes: the `type` denoting a type of configuration relevant for a software component, the `schema` denoting the format in which the configurations are represented, and `query` expressing how to identify the object within the configuration. Such object also overcomes the OVAL drawbacks about (RL1) discussed in Sect. 3.

Example 2 (Object, state, and test for http-only flag). The `XML Config Object` can be used to specify the recommendation described in Ex. 1. In the excerpt below, type (line 2) indicates that the configuration considered is a deployment descriptor (computing environment independent), schema (line 3) refers to the location of the schema for the deployment descriptor of J2EE web application and the Xpath query (line 4) points to the `http-only` configuration.

```
1  <xmlconfiguration_object id="oval:sans.security:obj:1">
2    <type>deployment descriptor</type>
3    <schema>http://java.sun.com/xml/ns/j2ee</schema>
4    <query>//*session-config/*cookie-config/*http-only/text()</query>
5  </xmlconfiguration_object>
```

By modifying only the query element, all the other recommendation of Ex. 1 can be specified. Moreover, by modifying also the type and schema, our object can be used for any other XML based configuration. The expected value for the configuration is defined through a `xmlconfiguration_state` defining `true` as expected value for the `http-only` tag. Finally, the OVAL test, `xmlconfiguration_test`, contains the object and state above which are used to evaluate the configuration.

Definition 1 (OVAL Definition). *An OVAL Definition $OD \subseteq \mathcal{T}$ is a set of OVAL Tests.*

Example 3 (OVAL Definition for SANS). The OVAL definition checking for the SANS recommendations described in Ex. 1 is a set of tests, one for each recommendation, i.e, $OD_{sans} = \{t_{http-only}, t_{secure-flag}\}$.

According to OVAL, a definition is a boolean combination of tests. As SANS requires all recommendations to be followed, all the tests involved are characterized by an OR boolean relation in order to raise an alarm whenever one of the recommendation is not followed. $t_{http-only}$ (line 3) is described in Ex. 2. All other tests can be analogously defined.

```
1  <definition id="oval:sans.security:def:1">
2    <criteria operator="OR">
3     <criterion test_ref="oval:sans.security:tst:1" comment="HttpOnly flag"/>
4     <criterion test_ref="oval:sans.security:tst:2" comment="Secure flag"/>
5    </criteria>
6  </definition>
```

Table 1. Properties description

`product`	Product name, e.g., Struts	`unc_path`	UNC path for shared location
`vendor`	Product vendor, e.g., Apache	`ctx_root`	JWA context root
`release`	Product release, e.g., 2.3.1.1	`ip_jmx`	IP address of JMX endpoint
`sup_spec`, `req_spec`	Supported/Required specification	`port_jmx`	Port number of JMX endpoint

Table 2. Relations description

`depl_in`	deployed in, models a component installed in another	`comp_of`	composed of, represents the internal structure of applications (e.g. linked libraries)
`comm_with`	communicates with, represents network communication	`instr_set`	instruction set, for either compiled (x86, x64) or interpreted (Java Runtime) binaries

The *Target* area (top right of Fig. 2) allows the definition of targets for the checks. A *target definition* is an abstract concept representing either a software component or a relation which can be defined over software components or relations themselves. A *software component* is characterized by a set of conditions on specific properties such as those in Tab. 1 (left side). A *relation* defines the relationship between software components. We distinguish three kinds of relations. A static relation, i.e., "composed of", which allows to represent the internal structure of a software. Run-time relations, i.e., "deployed in" and "communicates with", which allow to define relations among software components running in a landscape. Finally, boolean relations (AND, OR) combine either static or dynamic relations. Dynamic and boolean relations can be nested whereas the static relation can only be applied to software components. These types of relations, combined with the possibility to nest them, allow to define a set of software components satisfying an arbitrary complex expression.

Definition 2 (Software Component). *A software component $SC \subseteq \mathcal{C}$ is a set of conditions. A condition $C \in \mathcal{C}$ is a tuple $C = \langle P, \theta, V \rangle$, where*

- *$P \in \mathcal{P}$ is a property name,*
- *$\theta \in \{=, <, >, \geq, \leq\}$ is an operator,*
- *$V \in \mathrm{dom}(P)$ is a value for the property.*

We define $\mathcal{R}$ as a set of relations. Examples are listed in Tab. 2. We define $\hat{\mathcal{R}} = \mathcal{R} \times \mathbb{N}$ as the set of numbered relations where any relation can occur an arbitrary number of times and is uniquely identified by a natural number. In the examples we omit the natural number when no ambiguity arises.

Definition 3 (Target Definition). *A target definition is a tuple $TD = \langle SCS, RS, \rho \rangle$ where*

- *SCS is a set of software components (cf. Def. 2),*
- *$RS \subset \hat{\mathcal{R}}$ is a set of numbered relations,*

– *$\rho : RS \rightarrow (SCS \cup RS) \times (SCS \cup RS)$ is a total and acyclic function mapping a relation into the pair of elements, denoted as ρ_1 and ρ_2, sharing the relation (either software components or relations).*

A target definition $TD = \langle SCS, RS, \rho \rangle$ is valid *iff $|SCS| = 1$ when $RS = \emptyset$.*

Example 4 (Software Component and Target Definition for SANS). SANS applies to JWAs developed according to one of the releases of the Servlet specification and deployed in a web application container supporting such specification. In particular the recommendations in Ex. 1 refer to the release 3.0. According to Def. 2, a software component for the web application container can be defined as the set containing a single condition referring to the supported specification, $SC_{webappcont} = \{\langle \texttt{sup_spec}, \geq, \mathsf{Java_Servlet_3.0} \rangle\}$. As the recommendation applies to all JWAs therein deployed, the software component for the web application can be specified as an empty set $SC_{webapp} = \emptyset$. Finally, the target definition, according to Def. 3, can be expressed as $TD_{sans} = \langle SCS_{sans}, RS_{sans}, \rho_{sans} \rangle$ where $SCS_{sans} = \{SC_{webapp}, SC_{webappcont}\}$, $RS_{sans} = \{\texttt{depl_in}\}$, and $\rho 1_{sans}(\texttt{depl_in}) = \{SC_{webapp}\}$, $\rho 2_{sans}(\texttt{depl_in}) = \{SC_{webappcont}\}$.

We extend the OVAL standard by referring each OVAL definition to a target definition, i.e., to a set of related software components, and referring each OVAL test contained in the definition to a software component of the target definition. Thus we fulfill requirements (RL3) and (RL5). We name the resulting new artifact *check definition*. Note that this artifact is not represented by a single class in Fig. 2 but it involves several of the concepts therein presented and formalized above. Def. 1 and 3 provide the building blocks for the check definition.

Definition 4 (Check Definition). *A check definition is a tuple $CD = \langle OD, TD, \tau \rangle$ where*

– *$OD \subseteq \mathcal{T}$ is an OVAL definition,*
– *$TD = \langle SCS, RS, \rho \rangle$ is a target definition,*
– *$\tau : OD \rightarrow SCS$ is a total function that maps an OVAL test included in the definition OD into the software component to which it applies defined for the target definition TD.*

Example 5 (Check Definition for SANS). Given OD_{sans} and TD_{sans} defined in Ex. 3 and Ex. 4 resp., a check definition for SANS recommendations on cookies is $CD_{sans} = \langle OD_{sans}, TD_{sans}, \tau_{sans} \rangle$ where $\tau(t) = SC_{webapp}$ for all $t \in OD$.

The *System* area (bottom of Fig. 2) contains the concepts characterizing systems in a landscape and their configurations. A *system component* represents a single installation of a software component in a specific domain. As the purpose is to identify its configurations, the system component is defined as a set of attributes denoting how the configurations can be retrieved. The configurations required are given by the OVAL tests which are defined for software components. To evaluate the tests, the objects they contain have to be retrieved for each installation of the software component, i.e., for each system component. The tests to be

performed on system components are defined through the test mapping. The set of attributes necessary to collect a configuration is given by the collector (more details about how system components are derived starting from the target definition and the collector can be found in Sect. 5). By allowing the separation of the check logic from the attributes needed for the collection, our language fulfills requirement (RL8).

Definition 5 (Collector). *A collector is a tuple $K = \langle SC_K, PS, O_K \rangle$ where SC_K is a set of conditions, $PS \subseteq \mathcal{P}$ is a set of properties, and O_K is a query over OVAL objects.*

Example 6 (Collector for Web Applications deployment descriptor). A collector for web applications deployment descriptor has to define the set of attributes for retrieving the deployment descriptor of the web application installed in the landscape. Several alternatives are viable, e.g., accessing a shared file system via the Universal Naming Convention (UNC) or relying on the JMX interface of Tomcat. These alternatives can be defined as two collectors, $K_{unc} = \langle SC_{K_{webapp}}, \{\texttt{unc_path}\}, O_{K_{webapp}} \rangle$ $K_{jmx} = \langle SC_{K_{webapp}}, \{\texttt{ctx_root}, \texttt{ip_jmx}, \texttt{port_jmx}\}, O_{K_{webapp}} \rangle$ where $SC_{K_{webapp}} = \{\langle \texttt{req_spec}, =, \textsf{Java_Servlet_3.0} \rangle\}$ is the same for both as they apply to the same software component, and $O_{K_{webapp}}$ is an Xpath query over the XML serialization of the object (omitted for the sake of brevity).

Definition 6 (System Component). *A system component $SI \subseteq \mathcal{A}$ is a set of attributes. An attribute is a tuple $A = \langle P, V \rangle$, where $P \in \mathcal{P}$ and $V \in \mathrm{dom}(P)$ are properties and values, resp.*

Example 7 (System Component for SANS). The check definition for SANS in Ex. 5 includes the software component $SC_{webapp} = \emptyset$ defined in Ex. 4 which is referred to by an `XML Config Test`. Moreover the web application installed in the managed domain of Fig. 1 are characterized by the property of supporting the Servlet specification 3.0. Thus the collector defined in Ex. 6 can be used for establishing the set of attributes of the resulting system components. By using K_{unc}, the resulting system component for one installation of the eInvoice web application sold by ACME is $SI_{unc} = \{\langle \texttt{unc_path}, \textsf{\textbackslash\textbackslash 192.168.2.3\textbackslash path\textbackslash to\textbackslash web.xml} \rangle\}$. By using K_{jmx}, the resulting system component is $SI_{jmx} = \{\langle \texttt{ctx_root}, \textsf{/manager/}* \rangle, \langle \texttt{ip_jmx}, \textsf{192.168.2.2} \rangle, \langle \texttt{port_jmx}, \textsf{8059} \rangle\}$.

Definition 7 (System Test). *A system test is $ST = \langle SIS, OD, TM \rangle$ where*

- *SIS is a set of system components,*
- *$OD \subseteq \mathcal{T}$ is an OVAL definition, i.e., a set of tests,*
- *$TM \subseteq OD \times SIS$ is a set of test mappings defining which test of the definition applies to which system component.*

Example 8 (System Test for SANS). The check definition $CD_{sans} = \langle OD_{sans}, TD_{sans}, \tau_{sans} \rangle$ defined in Ex. 5 originates several system tests, one for each set of software components installed in the managed domain fulfilling the

target definition TD_{sans}. Given, $OD_{sans} = \{t_{http-only}, t_{secure-flag}\}$, $TD_{sans} = \langle SCS_{sans}, RS_{sans}, \rho_{sans} \rangle$, and $\tau(t) = SC_{webapp}$, a system test defining the tests to be performed for one possible installation of the software components is $ST_{sans} = \langle SIS_{sans}, OD_{sans}, TM_{sans} \rangle$ where $SIS_{sans} = \{SI_{jmx}\}$, and $TM_{sans} = \{(t_{http-only}, SI_{jmx}), (t_{secure-flag}, SI_{jmx})\}$. Notice that no system component for $SC_{webappcont}$ is included in SIS_{sans} as no tests apply to it.

The system test refers a test to specific system, thus (RL4) is met.

Finally, the *OVAL Item* in Fig. 2 represents the configuration collected from a system component for the OVAL object defined in the OVAL test. By evaluating such items according to the test, a boolean result for the test is produced. Based on the test results, the boolean result of the definition is also evaluated. Differently from OVAL, our OVAL Items may derive from different system, however this does not affect the evaluation algorithm defined in [12], which we rely on. A check definition originates several system tests, each one originating a check result.

Definition 8 (Check Result). *A check result is a tuple* $CR = \langle ST, \omega \rangle$ *where*

- $ST = \langle SIS, OD, TM \rangle$ *is a system test,*
- $\omega : TM \rightarrow \{\top, \bot\}$ *is a function that maps test mappings into its result, i.e., the boolean values true* ($\top$) *or false* ($\bot$).

5 Approach

The language presented in Sect. 4 separates the checks' logic from the systems to which they apply. In this section we establish the link between these two aspects, thereby describing how the checks can be instantiated and executed in a concrete landscape.

The overall approach is outlined in Fig. 3. External and internal authors (from the perspective of an organization) can define, independently from the landscape, checks *CD* (Def. 4) for known vulnerabilities affecting software components (cf. (S1)), and for best practices of single or multiple software components sharing relations (cf. (S2)). An additional input is the set of collector definitions $\mathcal{K}$, that has to be provided by system administrators as creates the link between the software components used in the checks and the attributes of system components which allow the collection of the configurations. The `TD Evaluator` module has in input the above artifacts and is responsible for producing all the system tests ST defining which test has to be executed on which system component. To produce the System Test artifact, the `TD Evaluator` relies on a `Data Source`, an authoritative source of information about the software components installed in a managed domain. We assume a single `Data Source` to provide information about several aspects of the managed domain, ranging from the properties of installed software (e.g. product names and vendors), or the internal structure of applications (e.g. linked libraries), up to architectural details on the deployment or the network interaction among different pieces of software. Since such information is often scattered over several repositories within an organization (e.g., CMDBs), the `Data Source` is a federated set of

views over these repositories, which constitute the interface to our language. Although strong, this assumption is not unrealistic. Indeed, several theoretical formulations of this problem are tackled in literature on data integration [13][14]. Furthermore the increasing adoption of standards such as DMTF's CMDBf [15] demonstrates the practical feasibility of configuration data federation.

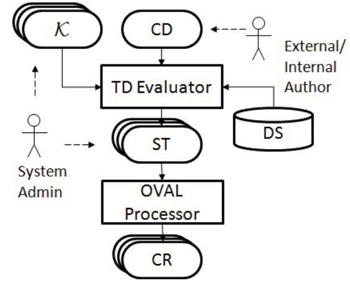

Fig. 3. Detection of vulnerabilities approach

The system test can also be manually provided by system administrators in case of checks for selected system components (cf. scenario (S3)). System tests are then processed by the `OVAL Processor` module that interprets the OVAL content and collects the objects defined for each system component within ST. The configurations collected from distributed systems are then evaluated and check results CR are produced, highlighting existing misconfiguration issues (if any).

A key step of the approach is the generation of the system tests based on the data source. In the following we formally define the interpretation of target definitions w.r.t. a data source, which provides information about the properties of software components deployed within a managed domain. We then describe how this leads to the generation of system tests.

Informally a data source can be seen as a particular instantiation of software component properties (cf. Def. 2) and target definition relations (cf. Def. 3) for a managed domain. Let $\mathcal{I}$ be the domain of instances of software components, namely software component identifiers, containing one unique symbol for each software component installed in a given managed domain. The data source then maps every software component identifier to the actual values of its properties and links it to the other software component identifiers it is related to.

Definition 9 (Data Source). *A data source is the pair of sets $DS = \langle \Pi, \Gamma \rangle$. Π contains a partial function $\pi_P : \mathcal{I} \rightarrow \mathrm{dom}(P)$ for each property $P \in \mathcal{P}$, while Γ includes a relation $\gamma_R \subseteq \mathcal{I} \times \mathcal{I}$ for each symbol $R \in \mathcal{R}$.*

Example 9 (Data Source). Figure 4 depicts a tabular representation of the data source DS_1 for the example landscape of Fig. 1. Due to space limitations, only a subset of the properties listed in Tab. 1 and relations of Tab. 2 are considered.

A software component can be seen as a simple conjunctive query ranging over properties of software deployed within a managed domain. The data source provides the necessary views on the managed domain to answer such a query. The answer consists of the set of software component identifiers matching to all the conditions within the software component. If it contains no conditions, the answer is the entire domain of software component identifiers $\mathcal{I}$. This evaluation

i	$\pi_{\texttt{vendor}}(i)$
a	Apache
l	OpenLDAP
t_1	Apache
t_2	Apache
w_a	ACME
w_b	ACME
w_c	ACME

(a) $\pi_{\texttt{vendor}}$

i	$\pi_{\texttt{release}}(i)$
a	2.2
l	2.4.30
t_1	7.0.18
t_2	7.0.18
w_a	1.0
w_b	1.0
w_c	1.0

(b) $\pi_{\texttt{release}}$

i	$\pi_{\texttt{product}}(i)$
a	Apache HTTPd
l	OpenLDAP
t_1	Tomcat
t_2	Tomcat
w_a	Web eInvoice
w_b	Web eInvoice
w_c	Web eInvoice

(c) $\pi_{\texttt{product}}$

i	$\pi_{\texttt{sup_spec}}(i)$
t_1	Java_Servlet_2.5
t_1	Java_Servlet_3.0
t_2	Java_Servlet_2.5
t_2	Java_Servlet_3.0

(d) $\pi_{\texttt{sup_spec}}$

i_1	i_2
w_a	t_1
w_b	t_2
w_c	t_2

(e) $\gamma_{\texttt{depl_in}}$

Fig. 4. Example of data source instance

is performed by the data source interpretation of software components, given by the mapping $\lceil\cdot\rceil_{DS} : \mathcal{SC} \to 2^{\mathcal{I}}$:

$$\begin{aligned} \lceil\emptyset\rceil_{DS} &= \mathcal{I} \\ \lceil\langle P, \theta, v\rangle \cup SC\rceil_{DS} &= \{i \in \mathcal{I} \mid \pi_P(i)\ \theta\ v\} \cap \lceil SC\rceil_{DS}. \end{aligned} \tag{1}$$

A target definition $TD = \langle SCS, RS, \tau\rangle$ is instead a more complex selection predicate (cf. Def. 3) and there can be several sets of software component identifiers which satisfy it. The interpretation of TD over a data source DS, $[\![TD]\!]_{DS}$, provides all such sets. This is done by relying on two interpretation functions, one providing the sets of software component identifiers, and one providing a function that maps each software component identifier to the corresponding software component.

The interpretation function $\lceil\!\lceil\cdot\rceil\!\rceil_{DS,\rho} : (\mathcal{SC} \cup \hat{\mathcal{R}}) \to 2^{2^{\mathcal{I}}}$ associates every $SC \in \mathcal{SC}$ and $R \in \hat{\mathcal{R}}$ to a powerset of software component identifiers, as defined in (2) and (3), respectively. Notice that this function depends both on the data source DS and the function ρ that carries the structure of target definition expressions.

$$\lceil\!\lceil SC\rceil\!\rceil_{DS,\rho} = \left\{\lceil SC\rceil_{DS}\right\} \tag{2}$$

$$\lceil\!\lceil R\rceil\!\rceil_{DS,\rho} = \begin{cases} \lceil\!\lceil\rho_1(R)\rceil\!\rceil_{DS,\rho} \times \lceil\!\lceil\rho_2(R)\rceil\!\rceil_{DS,\rho} & \text{if } R = \wedge \\ \lceil\!\lceil\rho_1(R)\rceil\!\rceil_{DS,\rho} \cup \lceil\!\lceil\rho_2(x)\rceil\!\rceil_{DS,\rho} & \text{if } R = \vee \\ \big\{\{v_1, \ldots, v_n, w_1, \ldots, w_m\} \mid \{v_1, \ldots, v_n\} \in \lceil\!\lceil\rho_1(R)\rceil\!\rceil_{DS,\rho}, & \\ \{w_1, \ldots, w_m\} \in \lceil\!\lceil\rho_2(R)\rceil\!\rceil_{DS,\rho}, \langle v_{1\le i\le n}, w_{1\le j\le m}\rangle \in \gamma_R\big\} & \text{otherwise} \end{cases} \tag{3}$$

Similarly, the interpretation function $\lfloor\!\lfloor\cdot\rfloor\!\rfloor_{DS,\rho} : (\mathcal{SC} \cup \hat{\mathcal{R}}) \to (\mathcal{I} \to \mathcal{SC})$ maps every $SC \in \mathcal{SC}$ and $R \in \hat{\mathcal{R}}$ to a function σ associating each software component identifier to the corresponding software component, according to (4) and (5).

$$\lfloor\!\lfloor SC\rfloor\!\rfloor_{DS,\rho} = \sigma,\ \text{where } \sigma(i) = SC, \forall i \in \lceil SC\rceil_{DS} \tag{4}$$

$$\begin{aligned} \lfloor\!\lfloor R\rfloor\!\rfloor_{DS,\rho} = \sigma,\ &\text{where } \sigma(i) = \lfloor\!\lfloor\rho_1(R)\rfloor\!\rfloor_{DS,\rho}(i), \forall i \in \mathrm{dom}(\lfloor\!\lfloor\rho_1(R)\rfloor\!\rfloor_{DS,\rho}) \\ &\text{and } \sigma(j) = \lfloor\!\lfloor\rho_2(R)\rfloor\!\rfloor_{DS,\rho}(j), \forall j \in \mathrm{dom}(\lfloor\!\lfloor\rho_2(R)\rfloor\!\rfloor_{DS,\rho}). \end{aligned} \tag{5}$$

Finally, the evaluation function for a valid target definition $TD = \langle SCS, RS, \rho\rangle$ over the data source DS, $[\![\cdot]\!]_{DS} : \mathcal{TD} \to \left(2^{2^{\mathcal{I}}} \times (\mathcal{I} \to \mathcal{SC})\right)$, associates a TD to

the pair $\langle I^*, \sigma\rangle$, where I^* is a powerset of software component identifiers and σ a function mapping every $i \in I \in I^*$ to a $SC \in SCS$. As expressed in (6), the definition of $[\![\cdot]\!]_{DS}$ relies on the aforementioned recursive interpretation functions of all the elements within the target definition expression, starting, in the general case, from the only relation R_0 which never appears in the ρ co-domain. In case $RS = \emptyset$, we know from Def. 3 that $\exists!\ SC_0 \in SCS$ and therefore SC_0 is the only element being interpreted.

$$[\![TD]\!]_{DS} = \begin{cases} \langle \lceil\!\lceil R_0 \rceil\!\rceil_{DS,\rho}, \lfloor\!\lfloor R_0 \rfloor\!\rfloor_{DS,\rho}\rangle, \text{with } \{R_0\} = \text{dom}(\rho) \setminus \text{cod}(\rho) & \text{if } RS \neq \emptyset \\ \langle \lceil\!\lceil SC_0 \rceil\!\rceil_{DS,\rho}, \lfloor\!\lfloor SC_0 \rfloor\!\rfloor_{DS,\rho}\rangle, \text{with } \{SC_0\} = SCS & \text{otherwise.} \end{cases} \quad (6)$$

Example 10. We hereby compute the interpretation of the target definition TD_{sans}, introduced in Ex. 4, w.r.t. the data source DS_1, shown in Ex. 9.

First, we recognize (Eq. (6)) that $[\![TD_{sans}]\!]_{DS_1} = \langle I^*_{sans}, \sigma_{sans}\rangle = \langle \lceil\!\lceil \texttt{depl_in} \rceil\!\rceil_{DS_1,\rho}, \lfloor\!\lfloor \texttt{depl_in} \rfloor\!\rfloor_{DS_1,\rho}\rangle$, since $\texttt{depl_in} \in \text{dom}(\rho) \setminus \text{cod}(\rho)$.

In order to obtain $\lceil\!\lceil \texttt{depl_in} \rceil\!\rceil_{DS_1,\rho}$, according to (3), we now need to compute the two following terms:

(i) $\lceil\!\lceil \rho_1(\texttt{depl_in}) \rceil\!\rceil_{DS_1,\rho} = \{\lceil SC_{webapp} \rceil_{DS_1,\rho}\} = \{\lceil \emptyset \rceil_{DS_1}\} = \{\mathcal{I}\}$;

(ii) $\lceil\!\lceil \rho_2(\texttt{depl_in}) \rceil\!\rceil_{DS_1,\rho} = \{\lceil SC_{webappcont} \rceil_{DS_1,\rho}\} =$
$= \lceil\{\langle \textsf{sup_spec}, \geq, \textsf{Java_Servlet_3.0}\rangle\}\rceil_{DS_1,\rho} =$
$= \{\{i \in \mathcal{I} \mid \pi_{\textsf{sup_spec}}(i) \geq \textsf{Java_Servlet_3.0}\}\} = \{\{t_1, t_2\}\}$.

We then have $I^*_{sans} = \lceil\!\lceil \texttt{depl_in} \rceil\!\rceil_{DS_1,\rho} = \big\{\{v, w\} \mid v \in \mathcal{I}, w \in \{t_1, t_2\}, \langle v, w\rangle \in \{\langle w_a, t_1\rangle, \langle w_b, t_2\rangle, \langle w_c, t_2\rangle\}\big\} = \big\{\{w_a, t_1\}, \{w_b, t_2\}, \{w_c, t_2\}\big\}$.

Analogously, by applying (5), we obtain $\sigma_{sans} = \lfloor\!\lfloor \texttt{depl_in} \rfloor\!\rfloor_{DS_1,\rho} = \{w_a : SC_{webapp}, w_b : SC_{webapp}, w_c : SC_{webapp}, t_1 : SC_{webappcont}, t_2 : SC_{webappcont}\}$.

As last step, the `TD Evaluator` needs to identify one or more system tests, mapping each OVAL test to the system component carrying the information about how to collect the object.

A check definition $CD = \langle OD, TD, \tau\rangle$ is defined for the target definition TD, being interpreted over a data source resulting in a pair $[\![TD]\!]_{DS} = \langle I^*, \sigma\rangle$. Every $I \in I^*$ is a set of software component identifiers satisfying the TD expression. Therefore one system test has to be created for every such set I.

When the `TD Evaluator` processes a check definition, it must identify a *matching collector* K, among the set $\mathcal{K}$ of all the ones defined for a given managed domain. This has to be done for every software component identifier $i \in I$, and provides the set of properties PS necessary to collect the to-be-checked configurations for specific OVAL Objects from i. For this reason, every $K \in \mathcal{K}$ (cf. Def. 5) references a software component SC_K and contains a Xpath query O_K, matching to the XML serialization of the OVAL Objects it applies to. We write $t \models O_K$ whenever the XML serialization of all the OVAL Objects referenced within an OVAL Test t satisfy the Xpath query O_K.

Given a collector property set PS and a software component identifier i, Eq. (7) defines how to retrieve the corresponding system component from a data source DS, through the interpretation function $\|\cdot\|_{DS}(i) : 2^{\mathcal{P}} \rightarrow \mathcal{SI}$.

$$\|PS\|_{DS}(i) = \|\{P_1, \ldots, P_n\}\|_{DS,i} = \{\langle P_1, \pi_{P_1}(i)\rangle, \ldots, \langle P_n, \pi_{P_n}(i)\rangle\}. \quad (7)$$

The conditions required to determine whether a collector matches to a software component identifier are now formalized by the following definition.

Definition 10 (Matching Collector). *For a $CD = \langle OD, TD, \tau\rangle$, where $TD = \langle SCS, RS, \rho\rangle$, let $[\![TD]\!]_{\mathcal{DS}} = \langle I^*, \sigma\rangle$ be an interpretation of TD over DS and $\tau^{-1} : SCS \to 2^{OD}$ be the inverse of τ, mapping every SC to the set $\{t \in OD \mid \tau(t) = SC\}$. We then say that $K = \langle\ SC_K,\ PS,\ O_K\rangle$ matches to $i \in I \in I^*$, iff*

$$i \in \lceil SC_K \rceil_{DS} \text{ and } P \in PS \Rightarrow \exists\langle P, \cdot\rangle \in \|PS\|_{DS}(i) \text{ and } t \in \tau^{-1}(\sigma(i)) \Rightarrow t \models O_K.$$

Given the interpretation $[\![TD]\!]_{DS} = \langle I^*, \sigma\rangle$ of a target definition within a check definition $CD = \langle OD, TD, \tau\rangle$, we are now in a position to associate each $I \in I^*$ to a system test $ST_I = \langle SIS_I, OD, TM_I\rangle$, constructed as follows. (i) OD is the same OVAL Definition contained in CD. (ii) Every element $SI \in SIS_I$ is a system component, i.e. a collection of attributes associated to properties of the software component identifier which allows to collect configuration information from it. For every $i \in I$ we first need to find a matching collector K carrying such set of properties PS, and we then retrieve the system component SI, i.e. the attributes corresponding to the properties in PS, from the data source DS. (iii) TM_I maps every test $t \in OD$ to a system component $SI \in SIS_I$.

Eq. (8) finally specifies how the system test's components SIS_I and TM_I, informally described above, are built by the `TD Evaluator`.

$$\begin{gathered}\forall i \in I \text{ if } \exists K \in \mathcal{K} \text{ s.t. } K \text{ matches to } i, \text{ then} \\ \|PS\|_{DS}(i) \in SIS_I \text{ and } \langle t, \|PS\|_{DS,i}\rangle \in TM_I\ \forall t \in \tau^{-1}(\sigma(i)).\end{gathered} \quad (8)$$

Example 11. Let us consider the check definition $CD_{sans} = \langle OD_{sans}, TD_{sans}, \tau_{sans}\rangle$, introduced in Ex. 5, and the data source interpretation of its target definition $[\![TD_{sans}]\!]_{DS_1} = \langle I^*_{sans}, \sigma_{sans}\rangle$, which has been derived in Ex. 10. Three sets of software component identifiers satisfy the target definition, namely $I^*_{sans} = \big\{\{w_a, t_1\}, \{w_b, t_2\}, \{w_c, t_2\}\big\} = \big\{I_a, I_b, I_c\big\}$, hence three system tests will be created. Among those, we shall only discuss, for brevity, the system tests ST_{I_a} and ST_{I_b}, related to I_a and I_b resp.

For the sake of this example we extend the data source $DS_1 = \langle \Pi_1, \Gamma_1\rangle$ such that it includes the properties required by the collectors (cf. Ex. 6). Let such an extended data source be $DS_1' = \langle \Pi_1 \cup \{\pi_{\texttt{ctx_root}}, \pi_{\texttt{ip_jmx}}, \pi_{\texttt{port_jmx}}, \pi_{\texttt{unc_path}}, \}, \Gamma\rangle$, where: $\pi_{\texttt{ctx_root}}(w_a) = \textsf{/manager/*}$, $\pi_{\texttt{ip_jmx}}(w_a) = \textsf{192.168.2.2}$, $\pi_{\texttt{port_jmx}}(w_a) = \textsf{8059}$, and $\pi_{\texttt{unc_path}}(w_b) = \textsf{\\\\192.168.2.3\\path\\to\\web.xml}$.

According to Def. 10 the collector K_{jmx} matches to the software component identifier w_a (and not to w_b), as (i) $w_a \in \lceil SC_{K_{webapp}} \rceil_{DS}$, (ii) $\pi_{\texttt{ctx_root}}(w_a)$, $\pi_{\texttt{port_jmx}}(w_a)$, $\pi_{\texttt{port_jmx}}(w_a)$ are all defined in DS (while this is not the case for w_b), and (iii) both $t_{http-only} \models O_{K_{webapp}}$ and $t_{secure-flag} \models O_{K_{webapp}}$ hold. From analogous reasoning it follows that K_{unc} matches to w_b (and not to w_a).

By applying (8) we finally derive that $ST_{I_a} = \langle\{SI_{jmx}\}, OD_{sans}, \{(t_{http-only}, SI_{jmx}), (t_{secure-flag}, SI_{jmx})\}\rangle = ST_{sans}$, as anticipated in Ex. 7 and 8. Analogously, we obtain $ST_{I_b} = \langle\{SI_{unc}\}, OD_{sans}, \{(t_{http-only}, SI_{unc}), (t_{secure-flag}, SI_{unc})\}\rangle$.

6 Conclusion and Future Work

This paper presents a formal approach to specify and execute declarative and unambiguous checks able to detect vulnerabilities caused by system misconfiguration. This paper extends the state of the art on configuration validation as security checks can be specified for fine-granular components in a distributed environment and separate the check logic from the configuration retrieval.

A proof of concept has been developed to explore the feasibility of our approach at the example of OWASP and SANS recommendations for JWA, using a CMDB as data source for resolving target definitions, and JMX for the collection of configuration settings. In future work, we will evaluate the prototype in near-world environments that comprise a greater numbers of system components. Furthermore, we plan to generate security checks and checklists in an automated fashion to facilitate scenario (S3), where checks are used for gaining assurance about compliance with system-specific configuration policies. This would allow to gain assurance without the need to manually author check on a low technical level. Lastly, we intent to investigate the usage in cloud scenarios, were cloud providers could use and offer a corresponding tool for ensuring the security of consumer-managed resources.

References

1. 7Safe, the University of Bedfordshire: Uk security breach investigations report 2010 (2010), http://www.7safe.com/breach_report/Breach_report_2010.pdf
2. Verizon: 2009 data breach investigations report. Verizon (2009), http://www.7safe.com/breach_report/Breach_report_2010.pdf
3. Williams, J., Wichers, D.: Top 10 most critical web application security risks. OWASP (2010), https://www.owasp.org/index.php/Top_10_2010-A6
4. http://{scap.usgcb.nvd}.nist.gov
5. http://tomcat.apache.org/security-6.html#Fixed_in_Apache_Tomcat_6.0.35
6. https://www.owasp.org/index.php/Securing_tomcat
7. http://software-security.sans.org/blog/2010/08/11/security-misconfigurations-java-webxml-files
8. http://tomcat.apache.org/connectors-doc/generic_howto/proxy.html
9. Chen, X., Zheng, Q., Guan, X.: An OVAL-based active vulnerability assessment system for enterprise computer networks. In: ISF, pp. 573–588 (2008)
10. Ou, X., Govindavajhala, S., Appel, A.W.: MulVal: a logic-based network security analyzer. In: USENIX Security Symposium (2005)
11. Waltermire, D., Quinn, S., Scarfone, K.: The technical specification for the Security Content Automation Protocol (SCAP): SCAP version 1.1. NIST (2011), http://csrc.nist.gov/publications/nistpubs/800-126-rev1/SP800-126r1.pdf
12. Baker, J., Hansbury, M., Haynes, D.: The OVAL language specification (version 5.10.1). MITRE Corporation (2012), http://oval.mitre.org/language/version5.10.1/OVAL_Language_Specification_01-20-2012.pdf
13. Ullman, J.D.: Information Integration Using Logical Views. In: Afrati, F.N., Kolaitis, P.G. (eds.) ICDT 1997. LNCS, vol. 1186, pp. 19–40. Springer, Heidelberg (1996)
14. Lenzerini, M.: Data integration: a theoretical perspective. In: PODS, pp. 233–246 (2002)
15. DMTF Distributed Management Task Force: Configuration Management Database (CMDB) Federation Specification. DMTF Technical Report DSP0252 (2010)