Supporting Vulnerability Awareness in Autonomic Networks and Systems with OVAL

Martín Barrère, Rémi Badonnel and Olivier Festor

INRIA Nancy Grand Est - LORIA
Campus Scientifique
54506 Villers Les Nancy, France
Email: {barrere, badonnel, festor}@inria.fr

Abstract—**Changes that are operated by autonomic networks and systems may generate vulnerabilities and increase the exposure to security attacks. We present in this paper a new approach for increasing vulnerability awareness in such self-managed environments. Our objective is to enable autonomic networks to take advantage of the knowledge provided by vulnerability descriptions in order to maintain safe configurations. In that context, we propose a modeling and an architecture for automatically translating these descriptions into policy rules that are interpretable by an autonomic configuration system. We also describe an implementation prototype and evaluate its performance through an extensive set of experiments.**

I. Introduction

The continuous growth of networks, as well as the diversification of their services have considerably increased the complexity of their management. Traditional network management approaches are not suitable for supporting this sustained dynamics. Autonomic computing [16] provides new perspectives with respect to this issue, through the automation of the management task. Autonomic networks and systems are responsible for their own management. They have to adapt their configurations with respect to their environment, to protect themselves against security attacks, to repair their own failures, and to optimize their various parameters. When autonomic related operations are performed, the environment is modified in order to achieve specific objectives. However, such operations may lead to potential vulnerable states and increase the exposure to security threats.

Indeed, as systems and technologies evolve, new space for vulnerabilities comes into scene. Autonomic networks and systems should therefore integrate support mechanisms for preventing vulnerabilities. As happens in the real world, autonomic elements coexist within dynamic environments, interacting with others autonomic and non-autonomic elements. If an autonomic element is compromised, its functions and abilities become untrustworthy and eventually disabled; thus autonomic elements that use services of the former become compromised as well. This inevitably leads to distrust and failure of the autonomic environment. Thus, vulnerability awareness constitutes a fundamental property that must be present in self-governed entities. Autonomic elements unable to support this capability will age with time, becoming more vulnerable, insecure and useless.

Vulnerability management is a crucial activity for ensuring safe configurations and reducing the exposure of such autonomic systems. It consists in checking their configurations, identifying the presence of vulnerable states and performing the required maintenance operations (typically, modification of configuration parameters and/or application of security patches). While strong standardization efforts have been done for describing vulnerabilities, in particular with the OVAL[1] language, there is no fully integration of vulnerability management mechanisms within the framework of autonomic networks and systems. Such integration constitutes the target of our work. We consider that autonomic environments should dynamically capitalize the knowledge provided by vulnerability descriptions repositories in order to increase their security, stability and sustainability.

In that context, we propose a new approach for supporting vulnerability awareness in autonomic networks and systems using the OVAL language and the Cfengine tool [1], an autonomic maintenance system that provides support for automating the management of large-scale environments based on high-level policies. Our strategy consists in integrating OVAL vulnerability descriptions into the management plane, in order to enable the autonomic system to detect and prevent configuration vulnerabilities. For that purpose, the OVAL vulnerability descriptions are dynamically translated into policy rules directly interpretable by Cfengine. We have developed an implementation prototype, and have performed an extensive set of experiments in order to quantify the benefits and limits of our solution.

Finally, the remainder of this paper is organized as follows. Section II describes existing work and their limits in our area. Section III presents the proposed approach for increasing vulnerability awareness within autonomic networks and systems. It details the underlying architecture and the formalism for supporting the translation of vulnerability descriptions into Cfengine policy rules. Section IV depicts the prototyping of Ovalyzer, an OVAL to Cfengine translator, whereas Section V shows an extensive set of experiments and the obtained results. Section VI presents conclusions and future work.

[1]Open Vulnerability Assessment Language

II. Towards Vulnerability Prevention in Autonomic Networks

Autonomic computing provides a promising paradigm for dealing with large scale network management [22], [16]. However, several challenges must be addressed first in order to enable this approach into daily computer systems and networks in an efficient manner. One of them, and also the main target in this article, is the ability of autonomic networks and systems to deal with security issues, particularly, to detect vulnerabilities and maintain safe configurations. In this section we present a review of related work done within several areas of interest including change management, risk assessment, and management of unknown and known vulnerabilities.

Usually, vulnerability management refers to the cyclical practice of identifying, classifying, remediating and mitigating vulnerabilities [21]. When autonomic networks and systems perform changes on their environments, unsafe states may appear, thus change and risk management techniques should be considered. A large variety of techniques have been proposed to evaluate the impact of changes in networks and systems [3], [20], [15]. These contributions provide strong foundations for their automation. Under an autonomic perspective, automated techniques for assessing change associated risks as proposed in [26] and [29], are extremely important because they provide a key support for the change management process, particularly for taking decisions about effective change implementations. Our work further focuses on security aspects, however, such previous research work on change management highlight key challenges that must be taken into account when self-configuration activities are performed.

But even when techniques for assessing changes impact are available, knowing which actions may imply vulnerable states is also challenging. Indeed we can see such problem in the opposite way, that is to say, knowing how vulnerable states look like and the implications that may have a scheduled change, it would be possible to infer whether such change will negatively impact on the system security. This fact enforces the need of increasing systems vulnerability awareness. When considering an autonomic behavior, means for better understanding the surrounding environment are deeply required. As the autonomic nervous system, autonomic networks and systems must be able to perform diagnosis on the environment they are working on. Because systems high-level state is typically changing, these capabilities provide the basis for adaptation. In this sense, different methodologies and techniques have been proposed for increasing security knowledge in systems and networks [18], [11], [10].

Potential vulnerabilities can be present on a system without explicit awareness of it, thus techniques for learning and discovering new vulnerabilities are required. Taking advantage of fuzzing methods may provide a powerful approach to unknown vulnerability detection within autonomic environments [19], [28]. Digital forensics techniques can be exploited as well [9], [17]. Digital forensics provides a deep understanding of discovering mechanisms about the anatomy of an attack, thus its robust technical background on data collection and analysis establishes a solid framework for performing computer system investigations, providing support to the vulnerability management activity. Other approaches such as case-based reasoning (CBR) provide interesting and useful perspectives for detecting unknown vulnerabilities as described in [23]. Several works also consider the idea of modeling a network intrusion as a sequence of steps where each gained privilege by the attacker opens new intrusion capabilities [25], [27]. This concept provides robust foundations for attack graphs, a widely used approach for performing network vulnerability analysis as occur in [12] and [24].

Different approaches for specifying vulnerabilities have been proposed in the past, but they lack of standardized languages and platform independence, leading to compatibility and interoperability problems. In order to cope with these problems, the MITRE corporation [5] has introduced the OVAL language [7], an information security community effort to standardize how to assess and report upon the machine state of computer systems. OVAL is an XML-based language that allow to express specific machine states such as vulnerabilities, configuration settings, patch states. Real analysis is performed by OVAL interpreters such as Ovaldi [8]. Several related technologies have evolved around the OVAL language. NIST [6] is responsible for the development of emerging technologies including the SCAP[2] protocol [13] and the XCCDF[3] language [30]. The SCAP protocol is a suite of six specifications that includes OVAL and XCCDF, and it can be used for several purposes, including automating vulnerability checking, technical control compliance activities, and security measurement. XCCDF is a language for authoring security checklists/benchmarks and for reporting results of checklist evaluation. The use of SCAP, particularly OVAL and XCCDF, not only allows to specify vulnerabilities, but also to bring a system into compliance through the remediation of identified vulnerabilities or misconfigurations. While OVAL provides means for describing specific machine states, XCCDF allows to describe certain actions that should be taken when these states are present on the system under analysis.

Currently, OVAL repositories offer a wide range of vulnerability descriptions. Such existing knowledge can highly increase the vulnerability awareness of autonomic networks and systems. This work aims at defining a solution for enabling autonomic environments to automatically capitalize such external knowledge source and take into account this security related information when self-management operations are performed.

III. OVAL-aware self-configuration

Within the autonomic computing field, the self-configuration property refers to the ability of networks and systems for automatically configuring themselves in order to obey high-level policies, typically linked to business-level objectives.

[2]Security Content Automation Protocol

[3]eXtensible Configuration Checklist Description Format

When autonomic networks and systems perform changes in order to be compliant with the specified policies, collateral effects can be introduced in an involuntary manner. Such unexpected effects can vary from internal malfunction to the exposure of vulnerable states, thus vulnerability management mechanisms are deeply required to ensure safe configurations and to reduce the probability of potential attacks and failures of the involved self-managed entities. In this section we present our approach for supporting vulnerability awareness in autonomic networks and systems. The objective is to integrate vulnerability descriptions provided by OVAL repositories into the autonomic management plane, particularly in the context of the Cfengine autonomic maintenance tool.

A. *Overall architecture*

Our work proposes the integration of vulnerability descriptions by providing an infrastructure where OVAL vulnerabilities descriptions can be translated into policy rules interpretable by Cfengine. Due to the automation provided by Cfengine for managing large-scale environments, the OVAL process can be integrated into Cfengine devices when maintenance operations are performed. The overall objective is to provide autonomic maintenance mechanisms for several platforms using Cfengine as illustrated in Figure 1, and taking into account the existing and future security related knowledge specified in the OVAL language.

The proposed architecture illustrated in Figure 1 involves an OVAL repository where the descriptions of known vulnerabilities are stored. Such descriptions are intended to be translated and introduced within a distributed Cfengine configuration. Generated policies are deployed by the Cfengine server into its several Cfengine agents (points in the cloud) which are in charge of managing the devices present in the target network, in order to detect and prevent vulnerable configurations when self-management activities are performed. When a vulnerability is found on a specific monitored device, Cfengine agents are capable of generating specific alerts and shall be able to perform correction operations.

B. *OVAL vulnerability descriptions*

Nowadays, the OVAL language is mostly used by vendors and leading security organizations in order to publish security related information that warn about current threats and system vulnerabilities. OVAL repositories offer a wide range of security advisories that can be used for avoiding vulnerable states as well as augmenting networks and systems security considering best practices recommendations.

The usual or intuitive way to think about a vulnerability is to consider it as a combination of conditions that if observed on a target system, the security problem described by such vulnerability is present on that system. Each condition in turn can be understood as the state that should be observed on a specific object. When the object under analysis exhibits the specified state, the condition is said to be true on that system. Under this context, the manner in which OVAL represents a vulnerability can be directly mapped to the usual way a

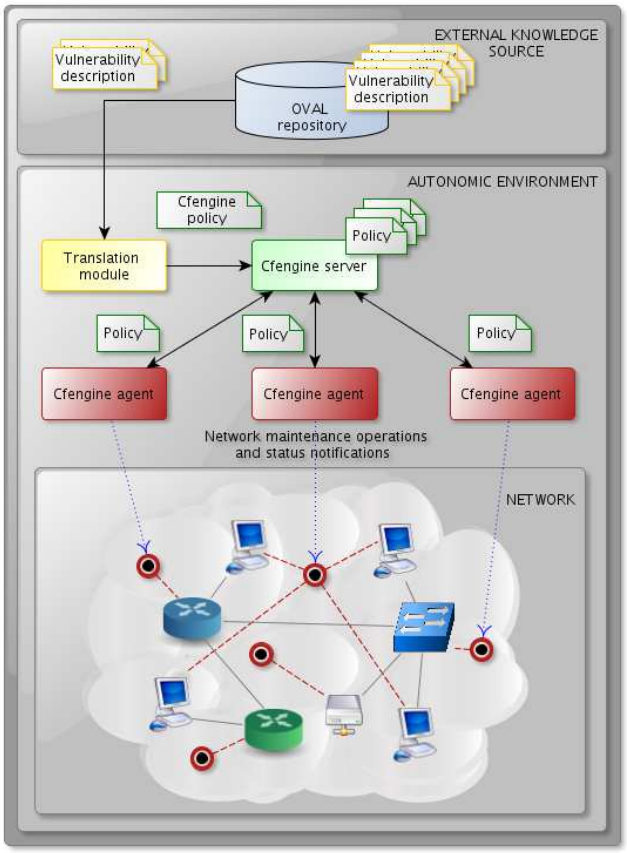

Fig. 1: High-level architecture

vulnerability is understood, as shown in Figure 2. Within the OVAL language, a specific vulnerability is described using an *OVAL definition*. An *OVAL definition* specifies a criteria that logically combines a set of *OVAL tests*. Each *OVAL test* in turn represents the process by which a specific condition or property is assessed on the target system. Each *OVAL test* examines an *OVAL object* looking for a specific state, thus an *OVAL test* will be *true* if the referred *OVAL object* matches the specified *OVAL state*. The overall result for the criteria specified in the *OVAL definition* will be built using the results of each referenced *OVAL test*.

As an example, we can consider an hypothetical vulnerability for the Cisco IOS[4] platform where two conditions must hold simultaneously: the version of the platform must be *11.3* and the service *ip finger* must be enabled (thus N would be 2 in Figure 2). Such vulnerability can be expressed within an *OVAL document* by defining an *OVAL definition* that arranges two *OVAL tests* as a logical conjunction where one test is in charge of assessing the system version and the other one must check the service status. The *OVAL objects* used in these tests will be an object that represents the version of the system and other object that represents the running configuration, respectively. Finally, the *OVAL states*, one for the version and one for the service, will express the states expected to be observed on each

[4]Cisco Internetwork Operating System

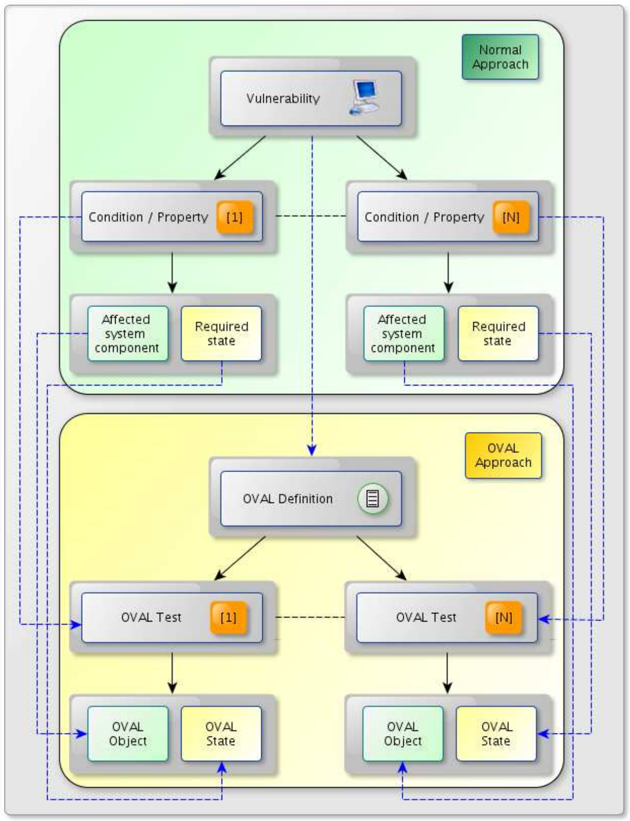

Fig. 2: Vulnerability conception mapping

object for the tests to be true and hence, defining the truth or falsehood of the *OVAL definition*.

C. *Translation formalization*

The translation module identified in Figure 1 has as a main goal the generation of Cfengine rules that accurately represent the OVAL advisories present in the OVAL repository. However, we consider the OVAL language should be seen from a logical perspective, as a first-order language. In our model, we understand the OVAL language as a means for predicating on the underlying system. From a logical point of view, its discourse universe is composed by each testable system component for each supported platform. Each OVAL object defines a family of *items* to be tested on the target system. For example, an OVAL *process object* with name *”httpd”* can define a set of several processes with that name, where each one of them is an identified OVAL item and will be tested independently. The overall result will be computed according to the parameters specified in the OVAL test. Because each collected OVAL item is what is actually tested within the OVAL process, the discourse universe of the OVAL language refers to such OVAL items and not to the OVAL objects that represent them.

Under this perspective, we consider a *predicate* as the very essential construction within the OVAL language. The most simple case can be seen as the evaluation of an OVAL item gathered from the system against a specified OVAL state. Mathematically, checking such item is the same as verifying whether the specified item belongs to a defined mathematical relationship. We believe that such formalization has potential within autonomic environments and that might be successfully exploited by reasoning engines such as done in [24] over standard networks. The example presented in Figure 3 depicts how the OVAL language can be used for expressing a predicate over the *httpd.conf* configuration file, assessing that its owner is the user *root*.

As mentioned before, the main core activity within the OVAL language is about predicating over the underlying system; this is, identify the system items (individuals of our discourse universe) and perform the assessment checking if they match specific states (check if retrieved individuals belong to specific mathematical relationships). The properties of the system under analysis can be seen as predicates where atomic formulas – OVAL tests – can be compounded to build more complex expressions – OVAL definitions –. Within the OVAL language, definitions typically search for a combination of specific characteristics that can reveal security holes on the underlying system. Figure 4 presents a summarized mapping between OVAL main constructors, their corresponding components within a first-order logic and the respective Cfengine building blocks.

Within our approach, *Cfengine classes* are particularly important as they are the main constructs for expressing results of predicates over the system. For instance, when a collected item is compared against a defined OVAL state, compliance truth or falsehood will be represented by a Cfengine class. If this item has to be compared against several OVAL states, several Cfengine classes will be defined. The overall result for this assessment will also be a Cfengine class based on each

```
<definitions>
   <definition id=”oval:org.mitre.oval:def:PHI” ...>
      <criteria>
         <criterion comment=”single formula ALPHA”
              test_ref=”oval:org.mitre.oval:tst:ALPHA”/>
      </criteria>
   </definition>
</definitions>

<tests>
   <file_test id=”oval:org.mitre.oval:tst:ALPHA”>
      <object object_ref=”oval:org.mitre.oval:obj:MyOBJ”/>
      <state state_ref=”oval:org.mitre.oval:ste:MySTE”/>
   </file_test>
</tests>

<objects>
   <file_object id=”oval:org.mitre.oval:obj:MyOBJ” ...>
      <path operation=”equals”>/etc/httpd/conf/</path>
      <filename operation=”equals”>httpd.conf</filename>
   </file_object>
</objects>

<states>
   <file_state id=”oval:org.mitre.oval:ste:MySTE” ...>
      <user_id operation=”equals”>root</user_id>
   </file_state>
</states>
```

Fig. 3: Basic predicate within OVAL

one of the previous classes. On the other hand, a test result will be also represented by a Cfengine class, hence, an OVAL definition result will be based on the Cfengine classes defined for each one of the referred tests. The steps followed by the translation process are described in Algorithm 1.

The algorithm takes as input an OVAL document that will be represented by the main configuration file within the Cfengine policy. Each OVAL definition in turn will have its own policy file that will be imported from the main Cfengine configuration file. Each OVAL test is translated as a Cfengine method that is invoked from the file that represents the OVAL definition. OVAL objects are represented by Cfengine prepared modules while OVAL states are specified using Cfengine control variables. Results for OVAL tests and OVAL definitions are specified using Cfengine classes that in turn are combined using the same logical structure described in the OVAL definitions. Due to space limitations it is not possible to detail further Cfengine grammar but more information can be found at [14].

Mapping		
First-order logic	**OVAL**	**Cfengine**
Arrangement of compound logical formulas	OVAL document	Cfengine main configuration file
Compound logical formulas	OVAL definitions	Cfengine input files
Atomic predicates	OVAL tests	Cfengine methods
Family of individuals in the discourse universe	OVAL objects	Cfengine prepared modules
Mathematical relationships	OVAL states	Cfengine control variables

Fig. 4: First-order logic, OVAL and Cfengine mapping

Since the OVAL language allows to express specific system states, OVAL definitions can be used in several ways; particularly for defining states that should not happen (e.g. configuration vulnerabilities) or states that should happen (e.g. recommendations and good practices). Under this perspective, OVAL definitions that model configuration vulnerabilities should generate an alert on the translated Cfengine policy when they are true. On the other hand, OVAL definitions that model recommendations and good practices should generate an alert when they are false. For the moment, only alerts are considered when configuration vulnerabilities are observed. In order to achieve fully autonomy, remediation and mitigation measures have to be taken into account. XCCDF [30] is a promising standard that can be used for expressing specific system states that if hold, defined tasks can be performed in an automatic manner.

```
Data: an OVAL document
Result: Cfengine policy rules
1  $mainFile$ ← create <Cfengine main configuration file>;
2  foreach $def$ ∈ OVAL definitions do
3      $defFile$ ← create <Cfengine input file> for $def$;
4      add import sentence at $mainFile$;
5      foreach test referred by $def$ do
6          on $defFile$ do {
7              $obj$ ← OVAL object referred by $test$;
8              add prepared module call at "control section"
               for gathering $obj$;
9              foreach ste referred by test do
10                 add ste control variables at
                   "control section";
11             end
12             add test call method at "methods section"
               specifying objects and states;
13         }
14         $methodFile$ ← create <Cfengine method file>
           for $test$;
15         on $methodFile$ do {
16             add method name and parameters at
               "control section";
17             add obj variable at "control section";
18             foreach atomic predicate on the specified obj
               do
19                 add result as a Cfengine class at
                   "classes section";
20             end
21             combine classes for defining final method result
               class;
22         }
23     end
24     add logical test criteria at "alerts section" on
       $defFile$;
25 end
```

Algorithm 1: Translation algorithm

IV. Implementation prototype

In order to provide a computable infrastructure to the proposed approach, a translator is needed capable of interpreting OVAL documents and generating the required Cfengine directives code. In this section we present Ovalyzer, an extensible plugin-based OVAL to Cfengine translator.

Ovalyzer has been purely written in *Java 1.6* over *Fedora Core* (kernel version *2.6.30.10*). The tool has been built over the *Spring framework 3.0* and uses *JAXB* [4] for managing XML related issues. The translator provides serveral customization options for building, deployment and logging tasks. The Cfengine policy rules currently generated by the translator are compliant with *Cfengine 2.2.10* [1].

The prototyped translator is responsible for the translation of OVAL documents to Cfengine policy rules that represent them. The translator takes as input the content of OVAL documents and produces Cfengine code that is structured as Cfengine policy files that can be later consumed by a Cfengine running instance. Figure 5 describes Ovalyzer main components and the high-level interaction between them.

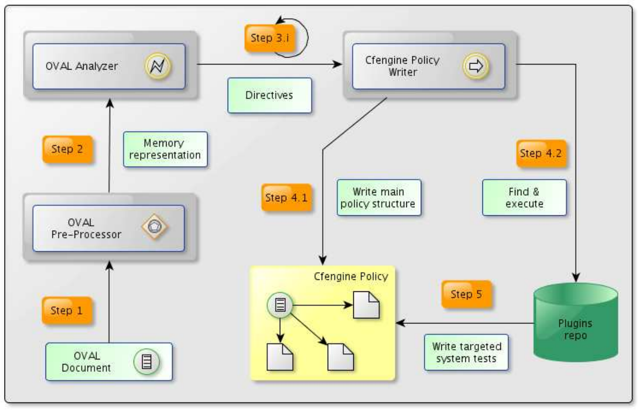

Fig. 5: Ovalyzer - High level operation

At step 1, an OVAL document is consumed as the input of the translator. An OVAL pre-processor is in charge of parsing the content of the specification, adjusting some configuration aspects and feeding the OVAL analyzer module at step 2 with a memory representation of the specified input. The OVAL analyzer module is the component that orchestrates the translation flow and provides the required directives for generating Cfengine code at step 3.i. Several calls are made by the OVAL analyzer module to the Cfengine policy writer depending on the content of the OVAL document. The Cfengine policy writer is in charge of generating the main Cfengine policy entries at step 4.1 and delegating at step 4.2, specific platform rules to plugins specifically designed for generating this type of Cfengine code. Plugins will produce the required Cfengine code that will be included at step 5 inside the generated Cfengine policy files.

The translator core is in charge of managing every high-level aspect of the OVAL documents it processes while available plugins provide the required functionality for generating the appropriate Cfengine code. The data model used by Ovalyzer is automatically generated using the JAXB technology. JAXB provides means not only for modeling XML documents within a Java application data model but also for automatically reading and writing them. Such feature provides to Ovalyzer the ability to evolve with new OVAL versions with almost no developing cost. While declarative extensibility of the translator is achieved by automatic code generation using the JAXB technology, functional extensibility is supported by a plugin-based architecture.

Plugins can be added on the plugin repository providing new translation capabilities. Each plugin knows how to translate a specific type of OVAL test to the appropriate Cfengine rules. This approach provides extensibility features, enabling a seamless functional evolution with the OVAL language. Moreover, plugin developers have access to the same data model built as a JAR library, simplifying eventual OVAL evolution impacts. When an OVAL document is processed by Ovalyzer, the required plugins are loaded at runtime from the plugins repository and the operations available in the plugins API are executed. Within the OVAL language, an OVAL definition can be seen as a logical formula compounded by OVAL tests. Because each type of test has it associated plugin, an OVAL definition can be translated only if the required plugins are present in the repository. Ovalyzer implements a plugin search mechanism based on name patterns. For example, if the name of the test belonging to the IOS platform is *line_test*, its associated plugin will be *CfengineIosLine.jar*; on the other hand, if the name of the test is *version55_test*, its associated plugin will be *CfengineIosVersion55.jar*. During the translation, Ovalyzer relies on the functionality of plugins for generating Cfengine code, thus an API has been specified in order to define the required methods for achieving a successful translation. Such methods have been specified based on how the Cfengine language structures its content. The current version of Ovalyzer provides an API of five methods that plugins shall implement.

V. A CASE STUDY

In this section we present a case study based on the IOS Operating System for Cisco devices.We consider an emulated environment where we show how the proposed framework can be used for augmenting the awareness of known vulnerabilities on Cisco routers. We also present the results obtained from the performed experiments.

Ovalyzer has been executed on an emulated environment in order to evaluate several factors such as functionality, performance and characteristics of the generated Cfengine code. Cisco devices have been emulated using *Dynamips / Dynagen* [2] running the operating system IOS version 12.4(4)T1.

A. *IOS coverage and execution time*

The official OVAL repository has 134 vulnerability definitions for the IOS platform, at the moment of writing this article. These definitions are based on three types of test, namely, *line_test* (*L*), *version55_test* (*V55*) and *version_test* (*V*). As we mentioned before, one plugin per each type of test is needed in order to provide the required translation capabilities. For this case study, three plugins have been written, namely, *CfengineIosLine.jar*, *CfengineIosVersion.jar* and *CfengineIosVersion55.jar*. Such plugins together provides a coverage of 100% of OVAL definitions for the IOS platform. Figure 6 depicts how the addition and combination of the required plugins increase the translation capabilities.

It can be also observed that each plugin does not provide a large coverage by itself. For instance, *line_test* only covers 1.49% of the available IOS definitions. This is because typically vulnerability definitions use more that one test for specifying the required conditions to be met on the target system. When combined, plugins shall cover a wider range of OVAL definitions. Different platforms may require a larger family of components to analyze, thus requiring more type of tests and hence, more plugins. In the case of the IOS platform, only three plugins were required for translating the 100% of available definitions in the OVAL repository.

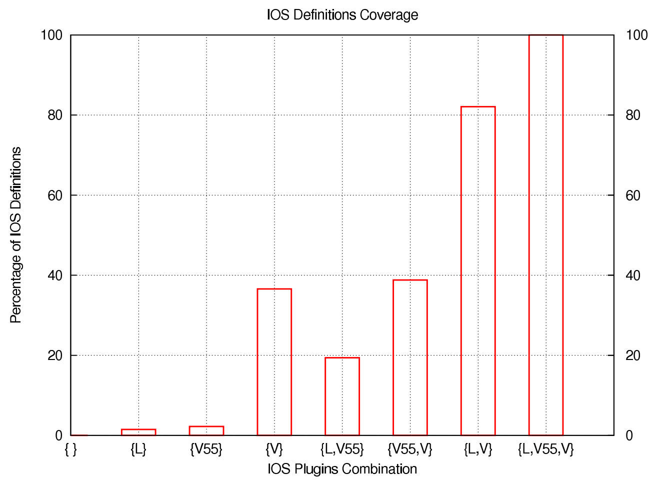

Fig. 6: IOS plugins coverage

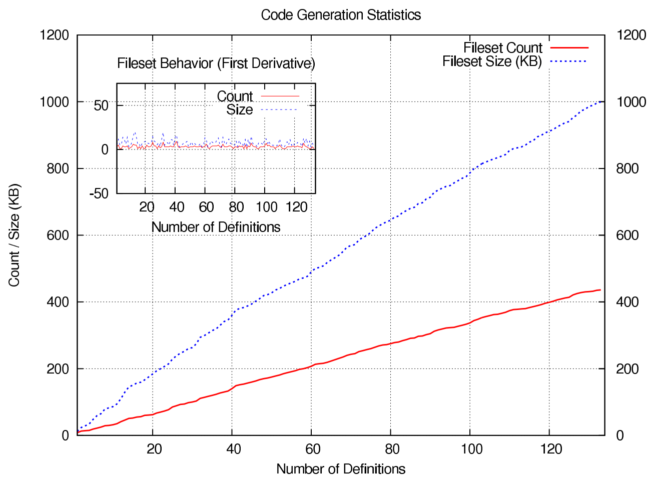

Fig. 8: IOS generation statistics

Since such translation shall be made in an automatic manner, several tests for evaluating Ovalyzer's performance have been done. We have particularly focused in the time required for generating Cfengine policy files over different sets of IOS vulnerability definitions. Figure 7 shows the observed timing values while varying the amount of translated OVAL definitions.

The experiment consists in executing Ovalyzer with a set of only one definition and measure the generation time, then with a set of two definitions and measure the generation time, and so on, until 134 definitions. Intuitively, one might expect a curve that monotonically grows with the number of definitions to translate, however, the obtained results are quite far from what expected. Within some executions for translating more than 100 definitions, the processing time is near from those executions translating less than 20 definitions. On the other hand, executions with a high translation time can be observed on a regular basis during the experiment. Because such experiments are run within an emulated and non-dedicated environment, we hold the hypothesis that this behavior is due to scheduling strategies of the operating system, not only with memory processes but also with I/O resources. We believe that such behavior is interesting for two reasons. First, involved equipment within autonomic networks may present similar scheduling issues; second, it gives a realistic overview of the expected behavior so autonomic strategies can take such conduct into account. The graph also identifies the average and median time of the executions performed, which respectively are of 9.5 and 6.1 seconds. Even when occasionally high time values occur and hence more experiments must be done for explaining why, the extremes seem to be bounded in the general case.

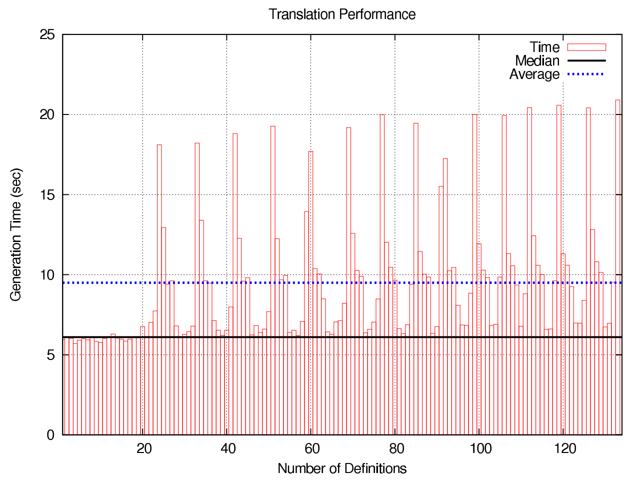

Fig. 7: IOS translation performance

B. Size of generated Cfengine policies for Cisco IOS

As happens with the generation time, the number and size of generated files constitutes an important dimension for analysis as well. We have experimented with the generated policies in the same way we did before, computing results for one definition, then two definitions and so on, until 134 definitions. Figure 8 illustrates the amount and total size of the generated files according to the number of definitions translated. For instance with 100 definitions, the translator generates a fileset of 333 files with a total size of 775 KB.

Both, the number of files and the size of the generated fileset, describe a linear growth when the number of IOS definitions is increased. This is in part due to the nature of the IOS definitions themselves, because in the average, each one of them uses a similar amount of tests and resources. With other platforms this behavior may not be observed because if we consider two definitions, one using several tests and objects and the other one, only one or two; the former will require several policy files – according to the way the translation is done – while the last one will be represented by a smaller set of files. Considering the case study of the Cisco IOS platform, the generation behavior (depicted by the first derivative of the curves), is stable as illustrated in the inner graph in Figure 8. Based on the experiments and results presented here, we aim to define for future work a mathematical and well founded mechanism for determining the size of a Cfengine policy fileset for any given set of OVAL definitions.

VI. Conclusions and future work

This work is mainly focused on integrating vulnerability descriptions in the management plane of autonomic networks and systems. Taking advantage of external knowledge sources such as OVAL repositories enables the ability of highly increasing vulnerability awareness in such self-managed environments. Cfengine has been taken as the autonomic part of this approach while the OVAL language is the resource that provides support for vulnerability descriptions. A formalization of the translation between OVAL descriptions and Cfengine policies has also been done by considering the OVAL language as a first-order language. As a case study we have chosen the IOS platform for Cisco devices, generating Cfengine policy rules capable of analyzing and detecting vulnerabilities over such platform, thus increasing vulnerability awareness in an autonomic manner. In addition, several experiments have been performed whose results successfully indicate the feasibility of the proposed approach in terms of functionality and integration into the Cfengine autonomic maintenance tool.

Vulnerability management integration into autonomic environments poses hard challenges. Supporting vulnerability awareness constitutes the first step towards secure self-managed infrastructures capable of detecting and remediating potential security breaches. We argue that networks and systems can achieve real autonomy if they are able to manage the required activities for understanding the surrounding environment, ensuring safe configurations and taking corrective actions when vulnerable states are found. For future work we plan to perform a deeper analysis of the actual impact of introducing our solution in real autonomic networks and systems. We also aim at extending the proposed approach to the execution of remediation actions. XCCDF is a promising language that can be used for expressing actions to take when a vulnerable condition is observed, being aware that such modifications may also have collateral effects thus risk assessment techniques must be taken into account. We are also interested in the formalization of the underlying used languages as it provide mathematical support for reasoning about security issues that can be integrated into autonomic networks and systems as well.

Acknowledgements

This work was partially supported by the EU FP7 Univerself Project and the FI-WARE PPP.

References

[1] Cfengine. http://www.cfengine.org/. Last visited on April 4, 2011.
[2] Dynamips/Dynagen Cisco Router Emulator. http://www.dynagen.org/. Last visited on April 4, 2011.
[3] ITSM - IT Service Management. http://www.itsm.info/ITSM.htm.
[4] Java Architecture for XML Binding. http://java.sun.com/developer/technicalArticles/WebServices/jaxb/.
[5] MITRE Corporation. http://www.mitre.org/. Last visited on April 4, 2011.
[6] NIST, National Institute of Standards and Technology. http://www.nist.gov/. Last visited on April 4, 2011.
[7] OVAL Language. http://oval.mitre.org/. Last visited on April 4, 2011.
[8] Ovaldi, the OVAL Interpreter Reference Implementation. http://oval.mitre.org/language/interpreter.html.
[9] H. Achi, A. Hellany, and M. Nagrial. Network Security Approach For Digital Forensics Analysis. *Proceedings of the International Conference on Computer Engineering and Systems (CCES'08)*, pages 263–267, November 2008.
[10] M. S. Ahmed, E. Al-Shaer, and L. Khan. *A Novel Quantitative Approach For Measuring Network Security*. IEEE, April 2008.
[11] M. S. Ahmed, E. Al-Shaer, M. M. Taibah, M. Abedin, and L. Khan. Towards Autonomic Risk-Aware Security Configuration. *Proceeding of the IEEE Network Operations and Management Symposium (NOMS'08)*, pages 722–725, April 2008.
[12] P. Ammann, D. Wijesekera, and S. Kaushik. Scalable, Graph-Based Network Vulnerability Analysis. *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS'02)*, page 217, 2002.
[13] J. Banghart and C. Johnson. The Technical Specification for the Security Content Automation Protocol (SCAP). *Nist Special Publication*, 2009.
[14] M. Burgess and Æ. Frisch. *A System Engineer's Guide to Host Configuration and Maintenance Using Cfengine*, volume 16 of *Short Topics in System Administration*. USENIX Association, 2007.
[15] M. Chiarini and A. Couch. Dynamic dependencies and performance improvement. In *Proceedings of the 22nd conference on Large installation system administration conference*, pages 9–21. USENIX Association, 2008.
[16] Autonomic Computing. An Architectural Blueprint For Autonomic Computing. *IBM White Paper*, 2006.
[17] V. Corey, C. Peterman, S. Shearin, M. S. Greenberg, and J. V. Bokkelen. Network Forensics Analysis, 2002.
[18] R. Costa Cardoso and M.M. Freire. Towards Autonomic Minimization of Security Vulnerabilities Exploitation in Hybrid Network Environments. *Proceedings of the Joint International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services (ICAS-ISNS'05)*, pages 80–80, 2005.
[19] J. Demott. The Evolving Art of Fuzzing 2 . Software Testing. pages 1–25, 2006.
[20] Y. Diao, A. Keller, S. Parekh, and V. V. Marinov. Predicting Labor Cost through IT Management Complexity Metrics. *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management (IM'07)*, (1):274–283, May 2007.
[21] P. Foreman. *Vulnerability Management*. Taylor & Francis Group, 2010.
[22] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, January 2003.
[23] M. J. Khan, M. M. Awais, and S. Shamail. Enabling Self-Configuration in Autonomic Systems Using Case-Based Reasoning with Improved Efficiency. *Proceedings of the 4th International Conference on Autonomic and Autonomous Systems (ICAS'08)*, pages 112–117, March 2008.
[24] X. Ou, S. Govindavajhala, and A. W. Appel. MulVAL: A Logic-Based Network Security Analyzer. *on USENIX Security*, 2005.
[25] N. K. Pandey, S. K. Gupta, S. Leekha, and J. Zhou. ACML: Capability Based Attack Modeling Language. *Proceedings of The Fourth International Conference on Information Assurance and Security*, pages 147–154, September 2008.
[26] J. Sauve, R. Santos, R. Reboucas, A. Moura, and C. Bartolini. Change Priority Determination in IT Service Management Based on Risk Exposure. *IEEE Transactions on Network and Service Management*, 5(3):178–187, September 2008.
[27] S. J. Templeton and K. Levitt. A requires/provides model for computer attacks. *Proceedings of the Workshop on New Security Paradigms (NSPW'00)*, pages 31–38, 2000.
[28] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. *Proceedings of the IEEE Symposium on Security and Privacy*, pages 497–512, May 2010.
[29] J. A. Wickboldt, L. A. Bianchin, and R. C. Lunardi. Improving IT Change Management Processes with Automated Risk Assessment. *Proceedings of the IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'09)*, pages 71–84, 2009.
[30] N. Ziring and D. Waltermire. Specification for the extensible configuration checklist description format (XCCDF). *NIST (National Institute of Standards and Technology)*.