Département de formation doctorale en informatique
UFR STMIA

École doctorale IAEM
Université de Lorraine, France

Vulnerability Management for Safe Configurations in Autonomic Networks and Systems

DISSERTATION

publicly presented on June 12, 2014

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in Information and Computer Science

by

Martín BARRÈRE CAMBRÚN

Dissertation committee:

President:	Le président
Rapporteurs :	Dr. Michelle SIBILLA. Professor at Université Toulouse III - Paul Sabatier, France. Dr. Raouf BOUTABA. Professor at University of Waterloo, Canada.
Examiners :	Dr. Emil LUPU. Reader at Imperial College London, U.K. Dr. Nacer BOUDJLIDA. Professor at Université de Lorraine, France. Dr. Rémi BADONNEL. Associate Professor at Université de Lorraine, France. Dr. Olivier FESTOR. Professor at Université de Lorraine, France.

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thloria.

Remerciements

Many people encouraged and supported this thesis. First, I would like to thank my supervisors Rémi Badonnel and Olivier Festor. They gave me the opportunity to join the Madynes Research Group, encouraged my research interests, and supported me all along the way. I am also indebted to all the members of this great team for their constant help and feedback.

During these years, I have been surrounded by bright and beautiful people at INRIA. I am deeply grateful to Laura for her strong support and patience. I would also like to thank my friends and colleagues Victor, Lautaro, Juan Pablo, Hernán, Iñaki, César, François, Gaëtan, Emmanuel, Renaud and Thao, for their friendship and helpful discussions.

This thesis would not have been possible if it were not for the constant encouragement and love of my family and friends in Uruguay. I would like to thank mom and dad, my sister Gabriela, Jesús, my loving nieces, my grandmother, Matilde, Viterbo, Gabriel, Fernando, Nicolás, Braulio, Laura, and Cristina, for supporting me in everything I do and always being there.

Finally, I would also like to thank all the people at the Computing Department of the Engineering School of the University of the Republic in Uruguay, for encouraging me to pursue this thesis. In particular, I am greatly thankful to Gustavo Betarte, Marcelo Rodríguez, Alejandro Blanco, and the whole Computer Security Team for their continuous support.

A mi familia
(Dedicated to my family)

Table of Contents

Table of Contents v

List of Figures xi

Introduction 1

Chapter 1 General introduction 3

1.1 The context 3

1.2 The problem 4

1.3 Organization of the document 5

1.3.1 Part I: State of the art 5

1.3.2 Part II: Contributions 5

1.3.3 Part III: Implementation 8

Part I Autonomic environments and vulnerability management 9

Chapter 2 Network management and autonomic computing 11

2.1 Introduction 11

2.2 Large-scale network management 12

2.2.1 Computer networks and the Internet 12

2.2.2 Technological evolution 13

2.2.3 End-users behavior 15

2.3 Autonomic computing overview 16

2.3.1 Key concepts of autonomics 16

2.3.2 Behavioral and architectural models 18

2.4 Security issues in autonomics 20

2.4.1 Vulnerability management in autonomic environments 21

2.5 Synthesis . . . 22

Chapter 3 Vulnerability management **23**

3.1 Introduction . . . 23

3.2 The vulnerability management process . . . 24

3.2.1 A brief history of vulnerability management . . . 24

3.2.2 On the organization of vulnerability management activities . . . 25

3.3 Discovering Vulnerabilities . . . 26

3.3.1 Exploiting testing methods . . . 26

3.3.2 Using network forensics . . . 27

3.3.3 Taking advantage of experience . . . 28

3.4 Describing Vulnerabilities . . . 29

3.5 Detecting vulnerabilities . . . 33

3.5.1 Analyzing device vulnerabilities . . . 33

3.5.2 Analyzing network vulnerabilities . . . 35

3.5.3 Correlating vulnerabilities with threats and attack graphs . . . 36

3.6 Remediating vulnerabilities . . . 37

3.6.1 Change management . . . 38

3.6.2 Risk and impact assessment . . . 39

3.7 Research challenges . . . 40

3.8 Synthesis . . . 41

Part II An autonomic platform for managing configuration vulnerabilities **43**

Chapter 4 Autonomic vulnerability awareness **45**

4.1 Introduction . . . 45

4.2 Integration of OVAL vulnerability descriptions . . . 46

4.3 OVAL-aware self-configuration . . . 48

4.3.1 Overall architecture . . . 48

4.3.2 OVAL to Cfengine translation formalization . . . 49

4.4 Experimental results . . . 52

4.4.1 IOS coverage and execution time . . . 52

4.4.2 Size of generated Cfengine policies for Cisco IOS . . . 54

4.5 Synthesis . . . 55

Chapter 5 Extension to distributed vulnerabilities **57**
5.1 Introduction 57
5.2 Specification of distributed vulnerabilities 58
5.2.1 Motivation, definition, and mathematical modeling 58
5.2.2 DOVAL, a distributed vulnerability description language 60
5.3 Assessing distributed vulnerabilities 62
5.3.1 Extended architecture overview 63
5.3.2 Assessment strategies 64
5.4 Performance evaluation 66
5.5 Synthesis 69

Chapter 6 Support for past hidden vulnerable states **71**
6.1 Introduction 71
6.2 Modeling past unknown security exposures 72
6.2.1 Understanding past unknown security exposures 73
6.2.2 Specifying past unknown security exposures 74
6.3 Detecting past hidden vulnerable states 75
6.3.1 Extended architecture overview 75
6.3.2 Assessment strategy 76
6.4 Experimental results 77
6.5 Synthesis 79

Chapter 7 Mobile security assessment **81**
7.1 Introduction 81
7.2 Background and motivations 82
7.3 Vulnerability self-assessment 83
7.3.1 Self-assessment process model 84
7.3.2 Assessing Android vulnerabilities 85
7.3.3 Experimental results 88
7.4 Probabilistic vulnerability assessment 91
7.4.1 Probabilistic assessment model 92
7.4.2 Ovaldroid, a probabilistic vulnerability assessment extension 95
7.4.3 Performance evaluation 99
7.5 Synthesis 101

Chapter 8 Remediation of configuration vulnerabilities **103**
8.1 Introduction 103
8.2 Background and motivations 104

8.3 Remediating device-based vulnerabilities 105
8.3.1 Vulnerability remediation modeling 105
8.3.2 The X2CCDF specification language 109
8.3.3 Extended framework for remediating device-based vulnerabilities . . . 111
8.3.4 Performance evaluation 113
8.4 Towards the remediation of distributed vulnerabilities 115
8.4.1 Modeling vulnerability treatments 116
8.4.2 DXCCDF, a distributed vulnerability remediation language 117
8.4.3 A strategy for collaboratively treating distributed vulnerabilities . . . 119
8.4.4 Performance evaluation 121
8.5 Synthesis 123

Part III Implementation 125

Chapter 9 Development of autonomic vulnerability assessment solutions 127
9.1 Introduction 127
9.2 Autonomic vulnerability assessment with Ovalyzer 128
9.2.1 Implementation prototype 128
9.2.2 OVAL to Cfengine generation example with Ovalyzer 131
9.3 Extension to past hidden vulnerable states 135
9.4 Mobile security assessment with Ovaldroid 137
9.4.1 Implementation prototype 137
9.4.2 A probabilistic extension 141
9.5 Synthesis 141

Conclusion 143

Chapter 10 General conclusion 145
10.1 Contributions summary 145
10.1.1 Autonomic vulnerability management 146
10.1.2 Implementation prototypes 147
10.2 Perspectives 148
10.2.1 Proactive autonomic defense by anticipating future vulnerable states . 148
10.2.2 Unified autonomic management platform 148
10.2.3 Autonomic security for current and emerging technologies 148
10.3 List of publications 149

Bibliography **151**

Annexes **161**

Annexe A CAS, a Configuration Assessment Service for UMF **163**
A.1 Introduction and problem statement 163
A.2 Background 164
A.3 Configuration modeling for UMF 164
A.4 Configuration assessment service architecture 168
A.5 UMF, conclusions and perspectives 168

Table des figures

1.1 Organization of the document 5
1.2 Organization of contributions 6

2.1 Large-scale networking evolution 13
2.2 Stages towards autonomic computing [82] 17
2.3 Autonomic management lifecycle [81] 18
2.4 Autonomic computing architecture [81] 19
2.5 Positioning of vulnerability management with respect to self-management activities 21
2.6 Mapping of the vulnerability management activity into the autonomic lifecycle . 22

3.1 Vulnerability assessment - D^3 classification 25
3.2 Automated vulnerability assessment classification 26
3.3 Forensic investigation process [54] 28
3.4 OVAL-based vulnerability assessment [117] 34
3.5 Scientific maturity of vulnerability management activities with respect to autonomic networks 40

4.1 Vulnerability conception mapping 46
4.2 OVAL example over Cisco IOS 47
4.3 High-level architecture 48
4.4 Basic predicate within OVAL 49
4.5 First-order logic, OVAL and Cfengine mapping 50
4.6 IOS plugins coverage 53
4.7 IOS translation performance 53
4.8 IOS generation statistics 54

5.1 Distributed vulnerability scenario 58
5.2 Distributed vulnerability matching process 60
5.3 DOVAL logical description 61
5.4 Overall architecture 63
5.5 Aggregation algorithm execution for role discovery 64
5.6 Statistics with uniform distribution on role assignment 68
5.7 Statistics with an increased device participation 68

6.1 Vulnerability lifecycle events 73
6.2 High-level imaging and exposure detection process 76
6.3 Vulnerability definitions assessment time 78
6.4 Tests assessment time 78

6.5 Repository scalability statistics . . . 79

7.1 OVAL-based vulnerability assessment architecture for the Android platform . . . 86
7.2 Scalability statistics in a simulated environment . . . 90
7.3 Scalability statistics in a real device . . . 90
7.4 Memory load in both emulated and real device . . . 91
7.5 Regular vs. probabilistic approach . . . 92
7.6 Test execution distribution . . . 95
7.7 Ovaldroid global architecture . . . 96
7.8 Ovaldroid client-server interactions . . . 98
7.9 Coverage convergence . . . 99
7.10 Collected objects . . . 100
7.11 Vulnerability evaluation rate . . . 100

8.1 Distributed vulnerability scenario with remediation tasks . . . 105
8.2 Change sequence search example . . . 108
8.3 VMANS high-level architecture . . . 111
8.4 VMANS control loop . . . 112
8.5 Vulnerability conversion statistics . . . 114
8.6 SAT solving analysis time for change detection . . . 114
8.7 NETCONF-Cisco statistics . . . 115
8.8 Mapping the model into the DXCCDF language . . . 117
8.9 Collaborative treatment - High level operation . . . 119
8.10 Vulnerability treatment scenario . . . 120
8.11 Time statistics for vulnerability assessment and treatment activities . . . 122

9.1 Ovalyzer's high-level operation . . . 128
9.2 JAXB process [89] . . . 129
9.3 OVAL vulnerability description for Cisco IOS . . . 131
9.4 Ovalyzer execution . . . 132
9.5 Cfengine code (main) . . . 133
9.6 Cfengine code (main) . . . 133
9.7 Cfengine code (method) . . . 134
9.8 Cfengine execution . . . 135
9.9 SVN-based assessment . . . 136
9.10 Self-assessment service high-level operation . . . 138
9.11 Ovaldroid agent . . . 138
9.12 Ovaldroid provider . . . 139
9.13 Ovaldroid reporter . . . 140
9.14 Ovaldroid reporter details . . . 141

A.1 UMF configuration error description with OVAL . . . 165
A.2 UMF distributed configuration error description with DOVAL . . . 166
A.3 DOVAL scenario for best practices . . . 167
A.4 Configuration assessment service architecture . . . 168

Résumé / Abstract

Le déploiement d'équipements informatiques à large échelle, sur les multiples infrastructures interconnectées de l'Internet, a eu un impact considérable sur la complexité de la tâche de gestion. L'informatique autonome permet de faire face à cet enjeu en spécifiant des objectifs de haut niveau et en déléguant autant que possible les activités de gestion aux réseaux et systèmes eux-mêmes. Cependant, lorsque des changements sont opérés par les administrateurs ou directement par les équipements autonomes, des configurations vulnérables peuvent être involontairement introduites, même si celles-ci sont correctes d'un point de vue opérationnel. Ces vulnérabilités offrent un point d'entrée pour des attaques de sécurité. Les environnements autonomes doivent être capables de se protéger pour éviter leur compromission et la perte de leur autonomie. À cet égard, les mécanismes de gestion des vulnérabilités sont essentiels pour assurer une configuration sûre de ces environnements.

Cette thèse porte sur la conception et le développement de nouvelles méthodes et techniques pour la gestion des vulnérabilités dans les réseaux et systèmes autonomes, afin de leur permettre de détecter, d'évaluer et de corriger leurs propres expositions aux failles de sécurité. Nous présenterons tout d'abord un état de l'art sur l'informatique autonome et la gestion de vulnérabilités, en mettant en relief les défis importants qui doivent être relevés dans ce cadre. Nous décrirons ensuite notre approche d'intégration du processus de gestion des vulnérabilités dans ces environnements, et en détaillerons les différentes facettes, notamment : extension de l'approche dans le cas de vulnérabilités distribuées, prise en compte du facteur temps en considérant une historisation des paramètres de configuration, et application en environnements contraints en utilisant des techniques probabilistes. Nous présenterons également les prototypes et les résultats expérimentaux qui ont permis d'évaluer ces différentes contributions.

Mots clés: sécurité, gestion de réseaux, informatique autonome, gestion de vulnérabilités.

Over the last years, the massive deployment of computing devices over disparate interconnected infrastructures has dramatically increased the complexity of network management. Autonomic computing has emerged as a novel paradigm to cope with this challenging reality. By specifying high-level objectives, autonomic computing aims at delegating management activities to the networks themselves. However, when changes are performed by administrators and self-governed entities, vulnerable configurations may be unknowingly introduced. Vulnerabilities constitute the main entry point for security attacks. Hence, self-governed entities unable to protect themselves will eventually get compromised and consequently, they will lose their own autonomic nature. In that context, vulnerability management mechanisms are vital to ensure safe configurations, and with them, the survivability of any autonomic environment.

This thesis targets the design and development of novel autonomous mechanisms for dealing with vulnerabilities, in order to increase the security of autonomic networks and systems. We first present a comprehensive state of the art in autonomic computing and vulnerability management, and point out important challenges that should be faced in order to fully integrate the vulnerability management process into the autonomic management plane. Afterwards, we present our contributions which include autonomic assessment strategies for device-based vulnerabilities and extensions in several dimensions, namely, distributed vulnerabilities (spatial), past hidden vulnerable states (temporal), and mobile security assessment (technological). In addition, we present vulnerability remediation approaches able to autonomously bring networks and systems into secure states. The scientific approaches presented in this thesis have been largely validated by an extensive set of experiments which are also discussed in this manuscript.

Keywords: security, network management, autonomic computing, vulnerability management.

Introduction

Chapter 1

General introduction

Contents

1.1 The context **3**
1.2 The problem **4**
1.3 Organization of the document **5**
  1.3.1 Part I: State of the art 5
  1.3.2 Part II: Contributions 5
  1.3.3 Part III: Implementation 8

1.1 The context

Over the last years, the massive deployment of computer devices over interconnected heterogeneous infrastructures has dramatically changed the perspective of network management. In particular, the Internet has played a fundamental role providing a platform for thousands of mixed technologies which currently constitute our globalized digital world. Nowadays, almost any network including the Internet itself, is expanding as a response to many factors. As end-users become more connected with computing technologies, or maybe technology becomes more friendly with end-users, a growing demand for useful digital advances goes along with the process. These new requirements appear in different forms, which directly impact in the mechanisms and resources used to meet them. In the other way around, technology can be also observed as a proactive stream, which shapes end-users' behavior to some extent. This symbiosis between technology and end-users makes a vehicle for their evolution.

This evolution however, is not without dangers. Constructive approaches tend to test the boundaries of current technologies. Since networks became the key platform for exchanging information and providing all kind of services on top of them, their management rapidly shifted into higher levels of complexity. These scenarios in turn are dynamic, which challenges their limits even more. A simple phone call over the Internet, understood as a service, may involve dozens of underlying running software, with different configurations, over disparate hardware, speaking distinct protocols, and geographically distributed all around the world. This might be only one service, however, we should imagine hundreds of different services, probably interacting between them and with the end-user, which in addition may have distinct requirements at different moments. It is clear that to cope with this complex landscape, smart and scalable management approaches are required to align the desired network behavior.

In that context, the autonomic computing approach has been conceived as a response to this problem. What if specific administration tasks to adapt our networks to each concrete service are not needed anymore ? What if we can express what we need from networks, and just tell the underlying infrastructure to manage it for us ? From a high-level viewpoint, these sentences may provide the intuition and spirit of autonomic computing. Indeed, autonomic computing aims at freeing administrators from the burden of heavy and error-prone management tasks. The main idea is to specify what networks have to do by means of high-level objectives, and delegate the responsibility of accomplishing these specific goals to the networks themselves. Under this perspective, self-governed networks and systems can positively tackle the management of the overwhelming technological development we are witnessing today. However, in order to make this approach work, security is essential. The ability of being autonomous implies self-protection. If this requirement is not met, autonomous entities might get compromised, not only affecting their own behavior but the surrounding environment as well. For this reason, **the ability of autonomic networks and systems to manage vulnerabilities and handle their own exposure is a critical factor for their survivability**. This matter constitutes the heart of our work. **This thesis aims at providing novel autonomous mechanisms for dealing with vulnerabilities, in order to increase the security of self-governed networks and systems**.

1.2 The problem

In computer security, vulnerabilities are flaws or weaknesses in the design, implementation, or configuration of a system that may allow an attacker to exploit them in order to bypass the security policies of such system. Vulnerabilities constitute the key entry point for breaking into computer systems and gaining unauthorized access to assets within these systems. Therefore, the ability to manage vulnerabilities is crucial for any computer system. Autonomic networks and systems are not an exception, although their autonomous nature challenges the vulnerability management process at higher levels. As a matter of fact, related tasks usually performed by human administrators over regular systems, must now be performed by self-governed entities on their own. The vulnerability management process basically involves the detection and remediation of vulnerabilities. Nevertheless, conceiving autonomic networks and systems featuring this process poses hard challenges. **How can we provide autonomic environments with mechanisms for increasing their vulnerability awareness ? What methods should be employed for identifying security weaknesses in an autonomous manner ? How should they proceed to mitigate and eradicate detected vulnerabilities while maintaining the system operative and safe ?** These and other questions constitute the issues that this thesis aims at dealing with.

Autonomic computing has opened new horizons for addressing problems where traditional methods seem to fail. Particularly, mechanisms able to properly scale with evolving dynamic networks and capable of reasonably tackling their increasing management complexity are simply, essential. Autonomic computing perfectly fits with these requirements. However, if autonomic infrastructures do not develop mechanisms and techniques to protect themselves from security threats, their real power and utility will eventually come apart. The focus of this thesis is to contribute in this direction, by providing a state of the art in autonomic computing and vulnerability management, and filling out missing scientific issues required to harden the foundations and security of autonomic computing. In the next section, we present the structure of this manuscript, detailing each main part of the document and their purpose.

1.3 Organization of the document

This manuscript is organized in three main parts (Parts I, II and III). Figure 1.1 illustrates the structure of the document. First, we present the state of the art in autonomic computing and vulnerability management, and the connection between both worlds (Chapters 2 and 3). The second part presents the contributions of this thesis, which are classified in two main categories according to the vulnerability management process, namely, vulnerability assessment (Chapters 4, 5, 6, 7) and vulnerability remediation (Chapter 8). The third part of this document presents three implementation prototypes that we have developed as proof of concepts derived from our research work (Chapter 9). The manuscript ends with general conclusions about our investigation and describes promising perspectives of this research work (Chapter 10). The three main parts of the document are detailed in the next subsections.

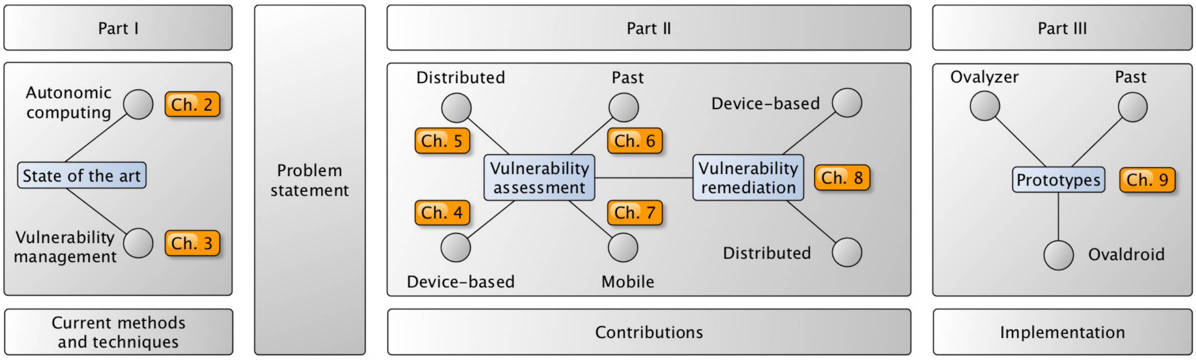

Part I
Autonomic computing
Ch. 2
State of the art
Vulnerability management
Ch. 3
Current methods and techniques
Problem statement
Part II
Distributed
Past
Device-based
Ch. 5
Ch. 6
Vulnerability assessment
Vulnerability remediation
Ch. 8
Ch. 4
Ch. 7
Distributed
Device-based
Mobile
Contributions
Part III
Ovalyzer
Past
Prototypes
Ch. 9
Ovaldroid
Implementation

FIGURE 1.1 – Organization of the document

1.3.1 Part I: State of the art

The first part of this document (Part I) describes the state of the art related to autonomic computing and vulnerability management. This part aims at positioning the autonomic computing paradigm in the context of large scale network management, and presenting current methods and techniques used nowadays for dealing with security vulnerabilities. Particularly, Chapter 2 briefly presents the evolution of networks during the last decades, and highlights the role of autonomic computing on this evolution. In that context, we identify security issues in autonomics that require special attention, specifically those related to vulnerability management. Chapter 3 presents our research work about current mechanisms and scientific approaches for managing vulnerabilities, and discusses how they can contribute to enhance the security of autonomic environments. Indeed, we identify benefits and limitations of these approaches, and put forward research challenges and open issues that must be addressed in order to achieve real autonomy on self-governed networks and systems.

1.3.2 Part II: Contributions

The second part of this manuscript (Part II) presents the contributions of this thesis. According to the vulnerability management process, we classify our contributions in two main categories, vulnerability assessment and vulnerability remediation. Figure 1.2 depicts our research work organized into different chapters where dashed lines illustrate the main reading flow across them. The first contribution, presented in Chapter 4, consists of a device-based approach for autonomic vulnerability assessment. From here, three dimensions represented with solid lines extend

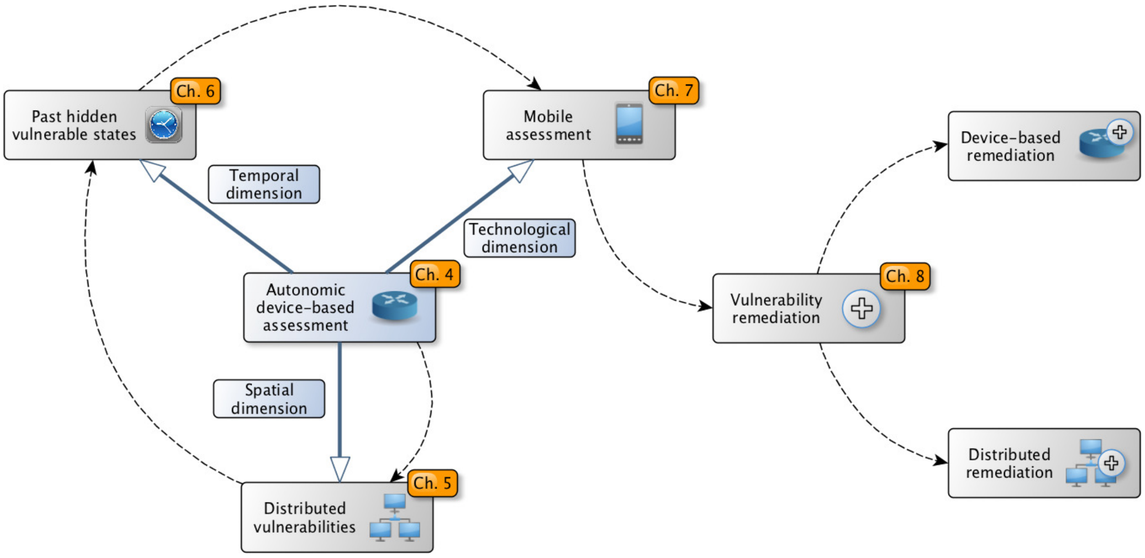

Past hidden vulnerable states
Ch. 6
Mobile assessment
Ch. 7
Device-based remediation
Temporal dimension
Technological dimension
Autonomic device-based assessment
Ch. 4
Vulnerability remediation
Ch. 8
Spatial dimension
Distributed remediation
Distributed vulnerabilities
Ch. 5

FIGURE 1.2 – Organization of contributions

the vulnerability assessment activity to novel scenarios considering spatial, temporal, and technological perspectives. Chapter 5 extends the concept of device-based localized vulnerabilities to composed vulnerabilities distributed across the network. We denominate this spatial extension, distributed vulnerabilities. Chapter 6 involves the second dimension which considers time. Indeed, this approach allows to increase the present security of computer devices by analyzing hidden vulnerable states in the past. Chapter 7 captures the technological dimension, where we have investigated novel approaches for assessing vulnerabilities in constrained environments such as mobile networks. All these chapters, from 4 to 7, fall into the vulnerability assessment category. Chapter 8 closes the vulnerability management process by considering vulnerability remediation activities. Contributions related to remediation activities are divided in two parts, namely, remediation of device-based vulnerabilities, and distributed ones, both located in Chapter 8. In the next subsections, we provide an overview of each chapter describing the contributions involved on them.

- **Autonomous vulnerability awareness**

Changes that are operated by autonomic networks and systems may generate vulnerabilities and increase their exposure to security attacks. 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, our first contribution presented in Chapter 4, introduces an autonomous approach for assessing device-based vulnerabilities. To this end, we have integrated vulnerability descriptions into the management plane of autonomic systems. We have particularly chosen the Cisco IOS platform as a proof of concept [86]. By automatically translating these security advisories into policy rules that are interpretable by an autonomic configuration system, autonomic agents distributed across the network become able to assess their own exposure. We have used the OVAL language [117] as a means for specifying vulnerability descriptions, and Cfengine [38] as the autonomic component of our solution. This approach provides an autonomous mechanism for increasing the vulnerability awareness of self-governed environments.

- **Distributed vulnerabilities**

Vulnerability assessment is traditionally performed over individual network devices, independently of each other. Sometimes however, two or more devices combined together may produce a vulnerable network state that host-based approaches are not able to detect. We refer to these security weaknesses as distributed vulnerabilities, which constitute our extension within the spatial dimension. Distributed vulnerabilities must be assessed with a consolidated view of the network in order to detect vulnerable states that may simultaneously involve two or more network devices. Chapter 5 presents our approach for describing and assessing distributed vulnerabilities in autonomic environments. We emphasize a mathematical construction for formally specifying distributed vulnerabilities as well as a machine-readable language for describing them. We also present an autonomous framework for assessing distributed vulnerabilities that exploits the knowledge provided by such descriptions. Therefore, our strategy permits to increase the vulnerability awareness of both individual devices and the network as a whole.

- **Past hidden vulnerable states**

Vulnerability assessment activities usually analyze new security advisories only over current running systems. However, a system compromised in the past by a vulnerability unknown at that moment may still constitute a potential security threat in the present. Indeed, a backdoor installed by an attacker for instance, may remain in the system even though the original vulnerability has been eradicated. Accordingly, past unknown system exposures are required to be taken into account. Chapter 6 presents our approach for increasing the overall security of computing systems by identifying past hidden vulnerable states, which constitutes our extension in the temporal dimension. In that context, we propose a modeling for detecting unknown past system exposures as well as an OVAL-based distributed framework for autonomously gathering network devices information and automatically analyzing their past security exposure.

- **Mobile security assessment**

The development of mobile technologies and services has contributed to the large-scale deployment of smartphones and tablets. These environments are exposed to a wide range of security attacks and may contain critical information about users such as contact directories and phone calls. Assessing configuration vulnerabilities is a key challenge for maintaining their security, but this activity should be performed in a lightweight manner in order to minimize the impact on their scarce resources. Chapter 7 presents two complimentary approaches for assessing configuration vulnerabilities in mobile devices, which constitute our extension within the technological dimension. The first approach considers a self-assessment strategy which allows mobile devices to assess their own exposure. In order to reduce the workload on the mobile side even more, we also propose a probabilistic cost-efficient strategy integrated into a client-server architecture. Both approaches target the Android platform [11] as a proof of concept, though these approaches could be adapted to other mobile platforms as well.

- **Remediation of configuration vulnerabilities**

Vulnerability assessment constitutes a key activity within the vulnerability management process. However, once a vulnerability has been detected, remediation activities to eradicate such security weakness are essential. Indeed, the management of known vulnerabilities plays a crucial role for ensuring safe configurations and preventing security attacks. However, this activity should not generate new vulnerable states. Chapter 8 presents two remediation approaches targeted on device-based and distributed vulnerabilities respectively. Our first approach formalizes the remediation decision process of device-based vulnerabilities as a SAT problem. In that context, we present an autonomous framework that is able to assess OVAL vulnerability descriptions and perform corrective actions by using XCCDF-based descriptions [167] of future machine states and the NETCONF protocol [63]. The second approach targets distributed vulnerabilities. There, we propose an autonomous strategy where network elements collaborate to remediate the vulnerabilities they are involved in.

1.3.3 Part III: Implementation

In order to evaluate the feasibility and scalability of the proposed approaches, we have developed different implementation prototypes that serve as the computable infrastructure for our experiments. Chapter 9 describes three implementation prototypes targeting autonomous device-based vulnerability awareness, past hidden vulnerable states, and mobile security assessment. Our first prototype, called Ovalyzer, is an OVAL to Cfengine translation system, which permits the integration of OVAL vulnerability descriptions into the autonomic management plane. Ovalyzer generates Cfengine policy rules that represent these security advisories. Then, generated Cfengine policies are consumed by autonomic agents deployed in the network, thus becoming able to assess their own security exposure. Our second implementation prototype aims at dealing with past unknown security exposures. Reusing the idea behind Ovalyzer, this prototype is able to autonomously generate XML-based snapshots of the state of the systems under surveillance, by following Cfengine policy rules. These images are then efficiently stored in an SVN-based repository. When new vulnerability descriptions become available, an exposure analyzer automatically assess stored images in order to identify past unknown security exposures. Our third prototype, called Ovaldroid, targets vulnerability assessment activities on the Android platform. Indeed, we have implemented both approaches presented in Chapter 8. First, we present a lightweight self-assessment service able to monitor an external provider for new vulnerability definitions and assess its own security exposure. Afterwards, we present our probabilistic extension where the assessment activities are controlled and performed by an external server, thus reducing even more the workload on the client side.

Part I

Autonomic environments and vulnerability management

Chapter 2

Network management and autonomic computing

Contents

2.1 Introduction **11**
2.2 Large-scale network management **12**
2.2.1 Computer networks and the Internet 12
2.2.2 Technological evolution 13
2.2.3 End-users behavior 15
2.3 Autonomic computing overview **16**
2.3.1 Key concepts of autonomics 16
2.3.2 Behavioral and architectural models 18
2.4 Security issues in autonomics **20**
2.4.1 Vulnerability management in autonomic environments 21
2.5 Synthesis **22**

2.1 Introduction

The growing development of networks and the multiplication of the services offered over them have dramatically increased the complexity of network management. The paradigm of autonomic computing has been introduced to address this complexity through the design of networks and services which are responsible for their own management. While high-level objectives are provided by network administrators, management operations are delegated to the networks themselves. This alleviates the administrative burden required for maintaining large-scale expanding networks as well as dozens of heterogeneous services. In this chapter, we first present a holistic perspective of evolving aspects that boost large-scale network management. We also discuss why the paradigm of autonomic computing may deal with these management issues. Following, we illustrate the main architectural characteristics of autonomics and put forward essential security concerns that need to be addressed. Finally, we explain why vulnerability management constitutes a critical activity to ensure real autonomy, and provide an overview of the approach taken in this thesis to achieve this goal.

2.2 Large-scale network management

In the broad sense of the concept, technology has been used by men for thousands of years. Defined as the creation, manipulation, and knowledge of tools and methods for solving problems, achieving goals, and controlling the environment that surround us [149], technology constitutes a cornerstone in the evolution of human kind. The inherent nature of technology projects a dependency relationship with human beings. While the desire of exploration, discovery, automation and innovation has made technology evolve to encompass human needs and goals, the impact of technology on people's culture and life has also become a form of feedback that aligns its direction. The era of information technology, where we are living now, is not an exception to this cyclical effect. Within this revolution, computer technologies have played a fundamental role, not only for end users, but also for orthogonal scientific and industry domains that at the end will eventually impact on human beings in some way or another. From small resource-less wired computers in the early 1980's to ubiquitous communication devices and smart apartments in the 2010s, the evolution of computing technologies has turned into a mesh of systems and devices where everything gets interconnected. In that context, the management of these large-scale, convoluted and heterogeneous networks becomes extremely challenging. The aim of this section is to show why autonomic computing is important to address the management of current and future networks. To that end, we analyze two aspects that directly impact on the evolution of computer networks: technological advances and the behavior of end-users. The objective is to provide an overview of current network trends and what we should expect, and motivate the development of autonomic solutions to success in the design and management of future networks.

2.2.1 Computer networks and the Internet

Computer networking has its origins in the early 1960s. Aiming at interconnecting and sharing computing resources, several research projects using a wide variety of protocols were developed at that time. The ARPANET project, led by the Advanced Research Projects Agency (ARPA) within the U.S. Department of Defense, was one of them [15, 83]. The first deployment of the network ARPANET was performed in the late 1960s. Involving four nodes at four different American universities (University of California, Los Angeles ; Stanford Research Institute ; University of California, Santa Barbara ; University of Utah), ARPANET became the first operational packet-switching network in the world. But it was not until the 1970s that this and other governmental and academic networks became interconnected using a common program called the Internet Transmission Control Program [127]. The specification of this program was the first document to coin the term Internet as a shorthand for *internetworking*. The fundamentals of this program gave rise to a new networking model released in the early 1980s called the Internet Protocol suite, currently maintained by the Internet Engineering Task Force (IETF) [84]. This set of communication protocols, commonly known as TCP/IP because of its two leading protocols, the Transmission Control Protocol (TCP) [129] and the Internet Protocol (IP) [128], would lay the architectural foundations of what we know today as the Internet.

The Internet is nowadays a global and decentralized system of interconnected computer networks. The Internet Protocol suite provides standards that allow computers within these networks to communicate with each other. This capability has provided a world-wide infrastructure over which thousands of systems and services have been built, reaching millions of users every day. The World Wide Web (WWW), whose standards are maintained by the World Wide Web Consortium (W3C), constitutes a representative example [156]. Current computer networks as well as the Internet itself also feature an interesting characteristic, it is an expanding technology.

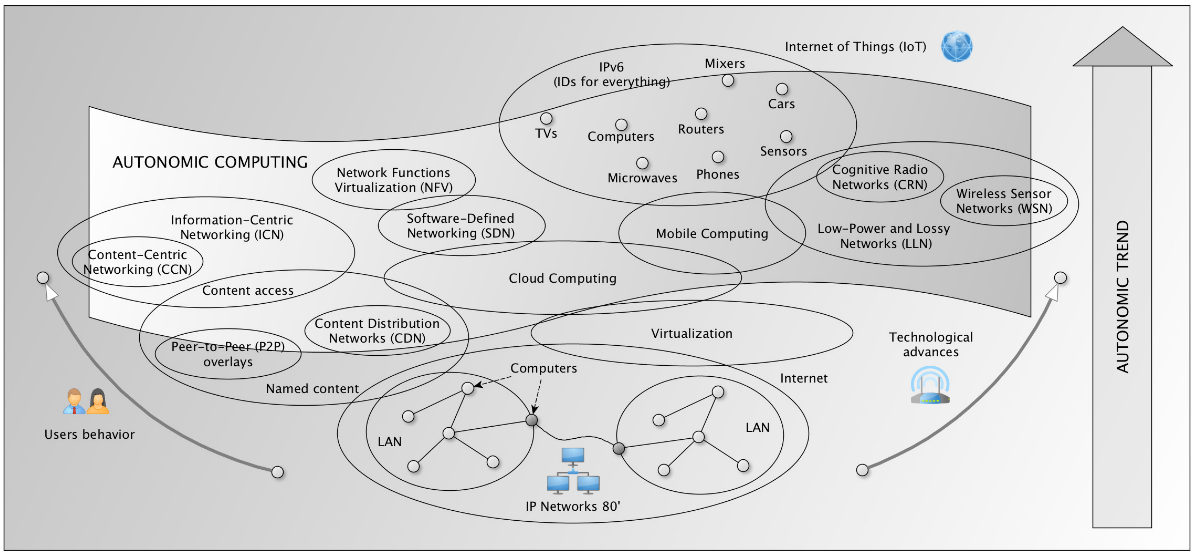

Internet of Things (IoT)
IPv6
(IDs for everything)
Mixers
Cars
TVs
Computers
Routers
Sensors
Microwaves
Phones
AUTONOMIC COMPUTING
Network Functions
Virtualization (NFV)
Cognitive Radio
Networks (CRN)
Wireless Sensor
Networks (WSN)
Information-Centric
Networking (ICN)
Software-Defined
Networking (SDN)
Mobile Computing
Low-Power and Lossy
Networks (LLN)
Content-Centric
Networking (CCN)
Cloud Computing
Content access
Content Distribution
Networks (CDN)
Virtualization
Technological
advances
Peer-to-Peer (P2P)
overlays
Computers
Internet
Named content
Users behavior
LAN
LAN
IP Networks 80'
AUTONOMIC TREND

Figure 2.1 – Large-scale networking evolution

Dozens of scientific domains contribute to some extent in the development of current and future networks. Some of them are focused on architectural and management issues, others deal with challenging problems that occur on top of networked infrastructures. As an example, enabling the interconnection of different devices may be classified as a networking problem. On the other hand, extracting accurate results for a specific search over thousands of gigabytes of information may be more connected to Big Data, Semantics or Data Mining techniques [77]. However, the interconnection of different techniques coming from disparate domains to solve computing problems is currently more and more common.

In the context of large-scale network management, it is important to have a holistic perspective of how current technologies are affecting and shaping the structure of today networks and therefore, the Internet. Even though this is not an exhaustive enumeration of current scientific research domains, we consider two prominent, yet very abstract, lines that can describe the forces that may mold future networks. These are: technological advances that provide new hardware and software systems with more and different capabilities, and end-users behavior, which guides to some extent, the final purpose of new technology and hence, how it should be constructed to meet people's needs and expectations. This vision is illustrated in Figure 2.1, where several interconnected research fields have arisen after the introduction of IP networks in the 1980s. The management of expanding and dynamic networks, where on top, diverse approaches and techniques are built to provide new services and applications, is becoming more and more complex everyday. Transversely, we have identified the autonomic computing paradigm as a perspective that goes beyond the resolution of network management problems as explained later in this chapter. Figure 2.1 aims at positioning network management trends under the influence of two evolving paths, the advances in computer technologies and the users behavior over services built on top of these technologies. Both aspects are discussed in the following subsections.

2.2.2 Technological evolution

Over the last decade, novel methods in engineering and electronics, non-expensive construction materials, and enhanced methodologies for large-scale production have arisen, triggering a massive flood of disparate powerful hardware with different abilities. This aspect has made room for new different technologies to emerge. With cheaper and more powerful hardware, the idea of virtualizing hardware platforms and general computing resources became popular [131]. The use of virtualization techniques has augmented the amount of network resources, real or virtualized, increasing the complexity of network management. With the incorporation of high-speed communication lines, cloud computing has gone one step further [31]. Cloud computing offers services whose processing is transparently performed combining resources distributed all around the globe. The decoupling of services from underlying resources has opened new horizons for emerging technologies such as Software-Defined Networks (SDN) [140, 64]. SDN allows administrators to manage computer networks independently from the real devices that actually implement low level network functionalities [22]. Complementary, Network Functions Virtualization (NFS) has come up with a novel perspective that leverage technology virtualization to consolidate disparate network devices into a single standard high-performance network platform [65]. These efforts aim at simplifying network management, to which autonomic computing can highly contribute.

The rapid evolution of the hardware industry has also promoted the increasing use of mobile devices such as smartphones and tablets. Indeed, it is expected that the number of mobile-connected devices will exceed the number of people on Earth by the end of 2013 [42]. With thousands of applications and services, mobile end-users expect seamless service provisioning from the cloud and clean communications on the move. This issue becomes a challenging problem for network operators when the reality tends to fast expanding mobile networks. Even though mobile devices spread fast, they still lack of powerful computation resources. In light of this, a new research domain denominated mobile cloud computing is emerging, which aims at making mobile devices resource-full in terms of computational power, memory, storage, energy, and context-awareness. This issue makes their management even more complex.

But mobile networks are not actually the only technological field with power-less devices. Low-Power and Lossy Networks (LLN) is a classification of networking environments with resource constrained devices [93], which includes Wireless Sensor Networks (WSN) [14]. WSNs are dynamic networks constituted by powerless sensors and short wireless signal range, e.g. temperature and home automation, which collaborate each other to route messages to their destination. However, energy or range limitations might break all communication links passing through them, posing hard management problems including routing, security and interoperability issues. In addition, WSNs use the ISM (industrial, scientific and medical) radio bands to transmit packets across the network [90]. But ISM is also used by other technologies, and therefore, the throughput of communication links can be degraded due to sensors usually have to wait for available channels to send their packets. To overcome this and many other issues, a novel research area called Cognitive Radio Networks (CRN) is arising [90]. CNRs feature autonomic strategies, making a smart use of the radio-frequency spectrum for opportunistically transmitting information.

All these technologies coexist within an environment that requires to be properly managed. Indeed, the Internet constitutes a target infrastructure where all these technologies can be integrated. With the advent of IPv6, the unique identification of any network device in the world becomes possible [125]. This has given room to a new way of understanding the Internet named the Internet of Things (IoT) [18, 23]. IoT describes the concept of a global network where heterogeneous and ubiquitous devices, from standard computers to cars and appliance on-board wireless sensors, become interconnected through high-speed communication channels. The revolutionary vision of an Internet integrated by any type of object providing smart services to others over a dense mesh of communication links, is for several scientists, the Internet of the future.

In brief, computer technologies evolve fast and constitute very complex internetworked environments. To make their management sustainable, systems involved in these convoluted networks must expose higher levels of autonomy, as reported in [16, 143]. To that end, autonomic computing provides strong foundations to tackle the management complexity of current and future networks.

2.2.3 End-users behavior

With the overwhelming avalanche of new devices and technologies, the trend to the Internet of Things seems indeed natural. However, there exist other influential factors that cannot be ignored, for instance, people's behavior with respect to the use of these technologies. As illustrated in Figure 2.1, people also determinate to some extent, in which way technology can be used, or whether its purpose makes sense. Social networks constitute an outstanding example of technologies that have greatly influenced people's behavior and vice versa [95], like Facebook [67] and LinkedIn [98]. During the last decade, it has been observed that social interactions mostly involve exchange and sharing of information and media content. The mechanisms that provide access to this content have dramatically evolved since then. Media aggregators such as YouTube [165], and photo sharing sites such as Picasa [121] and Flickr [70], are some examples.

Even though these web-based technologies are used on a daily basis, they are not the only means for accessing content. Dedicated infrastructures designed to share digital material have also arisen such as Peer-2-Peer (P2P) overlays (e.g. BitTorrent [28], eMule [60]), and Content Distribution Networks (CDN) (e.g. Akamai [7], Limelight [97]). The overwhelming consumption of media content over the Internet is currently so strong that several academics and industry entities have proposed radical approaches to deal with the burden of information exchange across the global network. Information-Centric Networking (ICN) is a new paradigm that provides a clean architectural approach to address these new relevant requirements [164, 3]. The main argument is that the Internet, as originally conceived, has been designed to exchange data packets between identified machines. However, end users do not really care about who provides the information (node endpoints), they just want to access the exact content they are looking for, no matter where it is stored. This vision drives to radical changes on the architectural design of the Internet itself, orienting its organization to named content, and using in-network caching for storing popular content so close users can retrieve the same information faster. Content-Centric Networks (CCN) is a well-known approach inside the ICN paradigm [139, 36]. The main goal is to provide a network infrastructure service that is better suited to today's use, particularly content distribution and mobility, and more resilient to disruptions and failures [3].

These novel information-based network models appear as a response to the inability of current networks to face end-users requirements. Therefore, a potential lecture of this phenomenon is that the community is trying to recreate or adapt more than thirty years of IP-based technologies to support current needs, which is indeed natural, but also extremely hard. This issue shows why flexible technologies are important. In that context, the adaptive nature of autonomic approaches may better fit current and future end-users behavior, providing more flexibility and scalability.

In summary, we have illustrated in this and the previous section, two different evolving paths that can have a strong impact on the future Internet. The objective is not to make an exhaustive analysis of each current edge technology, but to provide an insight of what is happening within current networks and the Internet, as well as disparate technologies running on top of them. In order to support their natural evolution, new perspectives to manage these evolving networks in a smart and controlled manner, constitutes a crucial problem. These perspectives must provide novel ways to deal with network management complexity. With this issue in mind, we have observed that no matter what technology or trend is considered, a common feature remains suitable to these challenges, autonomy. In order to tackle the overwhelming demand of end users as well as the fast large-scale deployment of heterogeneous network devices, autonomous mechanisms for managing these networks are essential to achieve scalability and reliability. In the next section, we present an overview of autonomic computing and detail architectural aspects as well as internal design issues.

2.3 Autonomic computing overview

The autonomic computing paradigm was born as a response to cope with the evident arising complexity of managing computing systems [91, 57]. In 2001, IBM [80] communicated a manifesto where it was explained that the management methods used at that moment, including software and hardware installation, configuration, integration, and maintenance, were not suitable to scale with the future landscape that was coming. In that context, a radical new way to deal with computer and network management was proposed, autonomic computing. The vision of autonomic computing is strongly inspired on the autonomous nervous system. Everyday, we, as human beings, perform several tasks conscientiously that to some extent, are the result of following some high-level objective within our lives. We eat when we are hungry, we sleep when we are tired, we take decisions expecting to succeed in our goals, and so many complex tasks as we can imagine. However, there are some other tasks, which are as much important as eating or sleeping, that are unconsciously and automatically done by our own bodies. Indeed, every time we breath, we do not do it conscientiously, we do not think about it. However, it is still a vital function that actually follows a high-level human law, being alive. In the same manner, the autonomous nervous system also governs our heart rate and body temperature, thus freeing our conscious brain from the burden of dealing with these and many other low-level, yet vital, functions [91]. In that context, autonomic computing aims at providing an infrastructure where networks can be managed by establishing high level-objectives. The underlying networking components will then perform in accordance to these rules and the changing environment, without explicit human intervention. This perspective aims at providing strong foundations for developing scalable and flexible infrastructures able to support a demanding and changing technological reality [133], [79].

2.3.1 Key concepts of autonomics

The first step towards the construction of autonomic solutions is in fact to understand what autonomic actually is. In other words, we should be able to distinguish between autonomic solutions and those that are just automatic. In 2009, NASA published a book where it is explained how autonomic systems are applied to NASA intelligent spacecraft operations and exploration systems [153]. There, the differences between automation, autonomy and autonomicity are very well discussed. Both automation and autonomy refer to the ability of executing a complete process without human intervention. The main difference is that while an automated solution sticks to the step-by-step process it was built for, an autonomous solution may involve a decision process under certain circumstances that affects the way tasks are executed, in order to accomplish its goals or final purpose. Such circumstances are usually related to the environment perceived by the autonomous agent. In other words, automated solutions replace step-by-step processes done by humans. Autonomic solutions on the other hand, aim at emulating human behavior during the process, by reasoning and taking decisions for the successful operation of the system. According to the dictionary definitions used in [144], autonomous and autonomic solutions means almost the same, except for a very slight property, spontaneity. While autonomy means self-governance and self-direction, independent and not controlled by external forces; autonomic means self-management, that occurs automatically and involuntarily, just as the autonomic nervous system does. For the sake of clarity and simplicity, we will use in this thesis both terms, autonomous and autonomic, without distinction except where explicitly noted, considering the following definition.

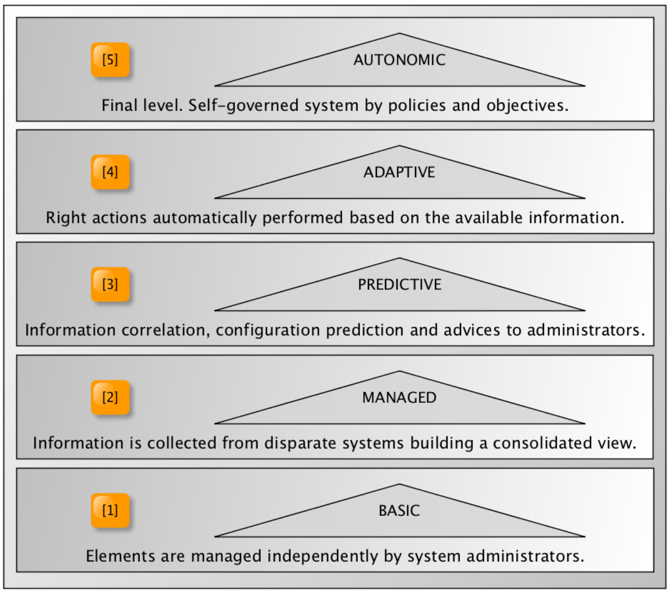

[5]
AUTONOMIC
Final level. Self-governed system by policies and objectives.
[4]
ADAPTIVE
Right actions automatically performed based on the available information.
[3]
PREDICTIVE
Information correlation, configuration prediction and advices to administrators.
[2]
MANAGED
Information is collected from disparate systems building a consolidated view.
[1]
BASIC
Elements are managed independently by system administrators.

FIGURE 2.2 – Stages towards autonomic computing [82]

Definition 1 (Autonomic system)**.** *An autonomic system is a self-governed entity, able to manage itself without any type of external control, and to perform required tasks without human intervention in order to accomplish the goals it was created for. While the purpose of the autonomic system is defined by high-level objectives, the achievement of this purpose is delegated to the system itself. The internal activities performed by the autonomic system may include environment perception, analysis, reasoning, decision making, planning and execution of actions that must ensure the successful operation of the system.*

As an effort to classify existing management solutions and their positioning with respect to the autonomic vision, IBM established what they called the evolutionary path to autonomic computing [82]. This path is represented by five levels as illustrated in Figure 2.2. The first level depicts the basic approach where network elements are independently managed by system administrators. Following, the managed level integrates the information collected from disparate network systems into one consolidated view. At the predictive level, new technologies are incorporated for correlating information, predicting optimal configurations and providing advices to system administrators. The ability to automatically take actions based on the available information constitutes the adaptive level. Finally, the autonomic level is achieved when the system is governed by business policies and objectives. These objectives define the purpose of the autonomic system that will be inherently pursuit by its logical implementation and supported by its internal know-how as well as its ability to sense the environment. The fact of being governed by policies or high-level objectives is what makes the key difference between autonomic and automated solutions.

Autonomic solutions are usually composed of smaller components called autonomic entities. Autonomic entities are designed to serve a specific purpose such as monitoring a network device or providing routing services. These entities can then be combined to construct more complex autonomic solutions. In order to organize their self-governing nature, autonomic systems involve a set of functional areas called self-* properties, defined as follows [105]:

- **self-configuration**, providing mechanisms and techniques for automatically configuring components and services,
- **self-optimization**, covering methods for monitoring and adapting parameters in order to

achieve an optimal operation according to the laws that govern the system,

- **self-healing**, for automatically detecting, diagnosing and repairing localized software and hardware problems, and
- **self-protection**, supporting activities for identifying and defending the system against potential threats.

These areas permit to classify the mechanisms used for regulating the behavior of autonomic entities. This aspect is discussed in the following subsection.

2.3.2 Behavioral and architectural models

From a high-level perspective, autonomic entities work under closed control loops that govern their behavior. In control theory, a closed control loop refers to a feedback mechanism that controls the dynamic behavior of a system based on the sensed environment as well as its own state (feedback) [17]. As shown in Figure 2.3, the resource manager interface provides means for monitoring and controlling the managed element (hardware, software, others). While its sensor enables the autonomic manager to obtain data from the resource, the effector allows it to perform operations on the resource. The autonomic manager is composed of a cycle of four activities that determines the behavior of the specific autonomic entity. This cycle or control loop includes monitoring its current state, analyzing the available information, planning future actions, and executing generated plans compliant to specified high-level goals [81]. In addition, the autonomic manager also exposes a management interface, in the same way the managed element does. This interface allows other autonomic managers to use its services and provides composition capabilities on distributed infrastructures.

It is important to highlight that a wide range of software and systems could embody, to some extent, autonomic solutions. To achieve this, the involved elements should be adapted to interact with the environment by means of sensors and effectors, as illustrated in Figure 2.3. In addition, they should work guided by rules and policies intended to achieve a specific purpose. Under a behavioral perspective, the autonomic manager will continuously monitor the managed elements and will perform an analysis of the perceived state. This information is then used to plan and execute changes required to align the state of the managed elements with the specified high-level objectives. This process describes the internal and finest working level of an autonomic

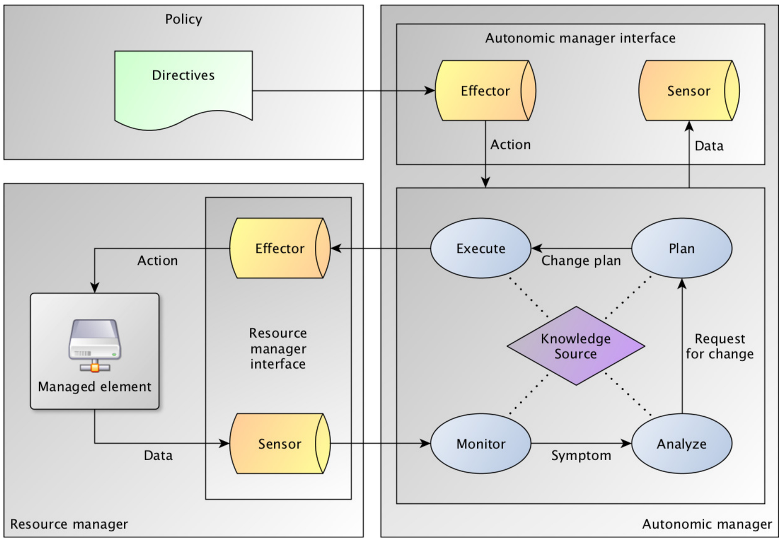


Figure 2.3 – Autonomic management lifecycle [81]

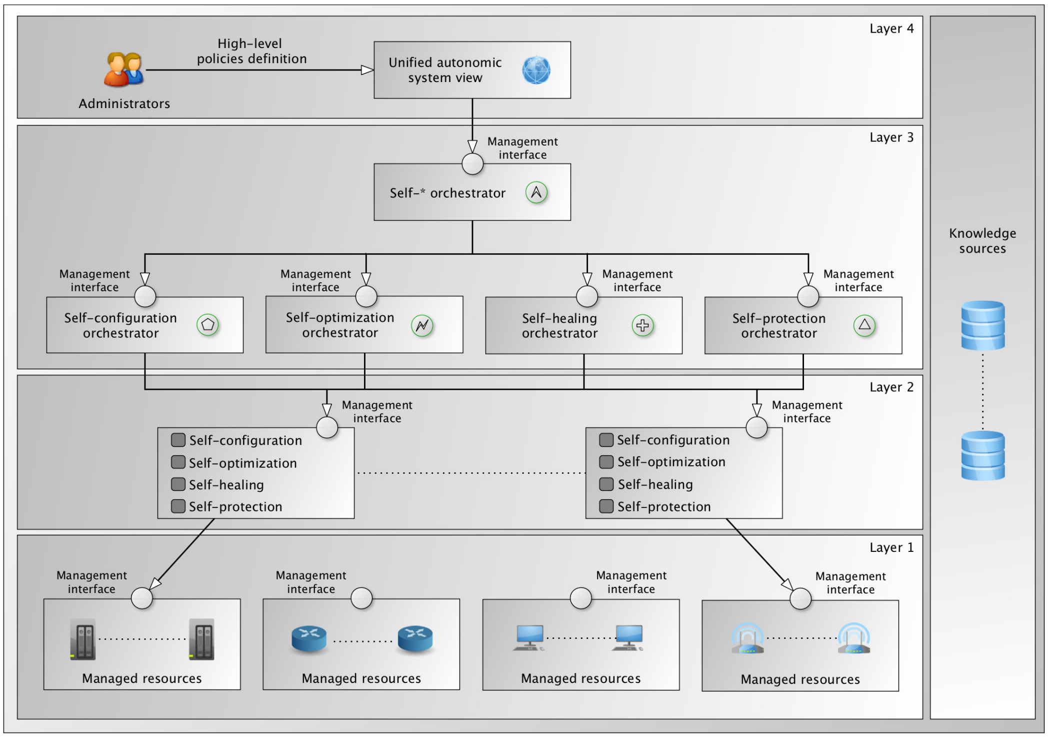

Layer 4
High-level policies definition
Administrators
Unified autonomic system view
Layer 3
Management interface
Self-* orchestrator
Management interface
Self-configuration orchestrator
Management interface
Self-optimization orchestrator
Management interface
Self-healing orchestrator
Management interface
Self-protection orchestrator
Layer 2
Management interface
Self-configuration
Self-optimization
Self-healing
Self-protection
Management interface
Self-configuration
Self-optimization
Self-healing
Self-protection
Layer 1
Management interface
Managed resources
Management interface
Managed resources
Management interface
Managed resources
Management interface
Managed resources
Knowledge sources

FIGURE 2.4 – Autonomic computing architecture [81]

system. However, the management interfaces exposed by these self-governed components, permit to combine them and to obtain easy solutions for complex problems. Indeed, the ability to integrate autonomic solutions into broader autonomic systems provides support for accompanying evolving technological landscapes [105].

The basic architecture of the autonomic computing approach proposes a layered organization as illustrated in Figure 2.4. Starting from the bottom, layer 1 contains managed resources over the IT infrastructure. These resources can be hardware devices such as servers, routers and access points, services and software components. Their management interfaces provide means for accessing and controlling the managed resources. Layer 2 describes autonomic managers for, but not limited to, the four aforementioned categories of self-management denominated self-* properties, i.e., self-configuration, self-optimization, self-healing and self-protection. Each autonomic manager within layer 2 is in charge of controlling a specific group of resources. To do so, these managers involve control loops for each self-* property, which control the state and behavior of the underlying devices. In order to line up the general directives of the autonomic system, it is required an autonomic component able to organize the overall activity of each specific self-* property across the whole system. Layer 3 contains autonomic managers that orchestrate other autonomic managers incorporating control loops that have the broadest view of the overall IT infrastructure. For instance, the self-configuration orchestrator will control the autonomic managers underneath related to self-configuration. This provides a consistent outlook of the system for each self-* property. The self-* orchestrator component is in charge of controlling every specific orchestrator, which is essential to achieve consistency with respect to the laws that govern the autonomic system. The top layer illustrates a manual manager that provides a common system management interface for the IT professional using an integrated solution console. The various manual and autonomic manager layers can obtain and share knowledge via knowledge sources.

Self-* properties are intended to autonomously solve high-level requirements, however, their implementation is complex and poses hard challenges. Along with administration tasks done by humans, changes performed by autonomic entities may inadvertently generate vulnerable states when following high-level objectives. Even though these changes can operationally improve the environment, insecure configurations may be produced increasing the exposure to security threats. Therefore, enabling autonomic networks and systems to manage vulnerabilities and maintain safe configurations constitutes a major challenge. This topic is discussed in the next section.

2.4 Security issues in autonomics

Autonomic systems must act on their own, taking any necessary decision to obey the rules that govern their behavior and to achieve their goals. As such, these systems must deal with security aspects, ensuring a proper functionality and guaranteeing their results. As explained before, autonomic systems are governed by high-level policies. Therefore, a major problem in autonomics relies on the successful operation of the system while also respecting all the rules that control the system. In that context, rules or policies can sometimes enter into a conflict. As an example, we can state the question: what happens when a service X has been identified as vulnerable to some kind of attack, but other rule says that service X must be always active ? If we do not have a countermeasure for this vulnerability, should we deactivate service X and violate the other rule to ensure security ? Should we leave service X activated so as to respect the other rule ? These are transversal requirements that pose consistency issues. At some point, solving these issues autonomously might not even be possible, and they should be addressed by human administrators. We could ask: are operational aspects more important than security ones ? The answer to this question might be found, perhaps, in the objectives and purpose of the specific autonomic system. Nevertheless, this is not a design problem of autonomics itself. This is a problem that human beings face everyday. Autonomic entities must be adaptive, with respect to the environment and to their own performance. When new knowledge contradicts information they already have, autonomic systems must be able to deal with this inconsistency problem and ensure appropriate functionality. They must learn and make decisions to accomplish their objectives. This problem may actually happen at any level of an autonomic architecture. Therefore, it is important to keep this problem in mind when designing an autonomic component.

During the last years, several efforts have been made to provide standard autonomic frameworks over which autonomic solutions can be built. The European UniverSelf project is an example to which we have contributed [154]. The UniverSelf project provides an infrastructure called Unified Management Framework (UMF), composed of three main blocks, governance, coordination and knowledge, which controls the overall behavior of the system. In addition, it offers well structured means for developing autonomic components to be executed over this infrastructure. These components are called Network Empowerment Mechanisms (NEMs). In the context of this thesis, we have contributed with a NEM for empowering vulnerability management features in UMF. Consistency issues regarding high-level policies are managed by the UMF framework. This thesis is oriented to deal with vulnerabilities in an autonomous manner. We do not deal with consistency problems at other operational levels of self-governed environments. However, we aim at providing consistent solutions so as to decrease, or at least to not increase, the burden of dealing with broader consistency problems. Our objective is to provide autonomous and consistent mechanisms for assessing and remediating vulnerabilities, in order to ensure safe configurations within autonomic environments as explained in the following subsection.

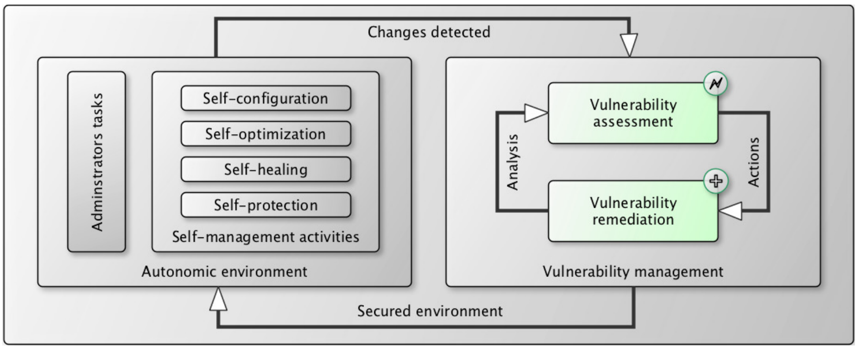


FIGURE 2.5 – Positioning of vulnerability management with respect to self-management activities

2.4.1 Vulnerability management in autonomic environments

In order to explain our approach, it is important to first understand what a vulnerability actually is. All along this work, the concept of vulnerability in computing security will be described by considering the following definition.

Definition 2 (Vulnerability)**.** *A vulnerability can be understood as a flaw or weakness in system security procedures, design, implementation, or internal controls that could be exercised (accidentally triggered or intentionally exploited) and result in a security breach or a violation of the system's security policy [108], [85].*

Under this perspective, vulnerability management is a cross-cutting concern strongly related but not limited to self-configuration and self-protection activities of autonomic networks. This process is depicted in Figure 2.5 where a control loop enables the assessment and remediation of potential vulnerable states generated by both administrators tasks and self-management activities, thus securing the environment. The main idea is that actions and changes performed in the system are constantly monitored and analyzed looking for vulnerabilities. When vulnerable states are detected, corrective actions are performed until the environment is secured. The vulnerability management control loop remains active during the whole lifetime period of the autonomic environment under surveillance. In that context, the establishment of a secure process for dealing with vulnerabilities requires the specification of a policy defining the desired system state, and a well-known secure initial state to identify vulnerabilities and policy compliance [162]. The main activities performed during the lifecycle of the vulnerability management process can be mapped to the same activity line present in autonomic components. Figure 2.6 describes the general lifecycle of an autonomic component where the main activities done for dealing with vulnerabilities have been mapped to the task loop performed during the autonomic manager lifecycle [81]. As it can be observed, vulnerability identification activities take place in the monitoring phase where tasks for assessing and analyzing vulnerable states are performed (I) taking advantage of the available security knowledge. When a security problem is found, it is classified (II) and changes for correcting the situation must be performed. Therefore, vulnerability counter-measures are planned based on several factors such as importance, risks and impact. Finally, a change plan is generated and remediation tasks are executed (III) in order to maintain safe configurations and to be compliant with the current policy. Figure 2.6 illustrates the overall approach taken in this thesis for integrating vulnerability management activities into the autonomic plane. Actual existing methods and techniques for dealing with vulnerabilities within autonomic and non-autonomic systems are widely discussed in Chapter 3.

Autonomic computing provides strong theoretical and practical foundations to face the large-scale network deployment that we are observing today. However, it is essential to incorporate

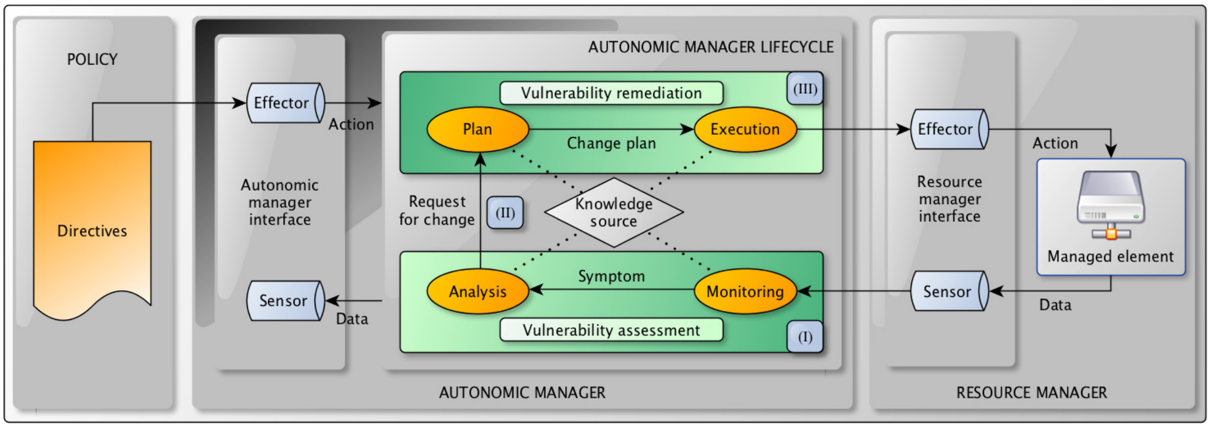

POLICY
Directives
Effector
Action
Autonomic manager interface
Sensor
Data
AUTONOMIC MANAGER LIFECYCLE
Vulnerability remediation
(III)
Plan
Change plan
Execution
Request for change
(II)
Knowledge source
Analysis
Symptom
Monitoring
Vulnerability assessment
(I)
AUTONOMIC MANAGER
Effector
Action
Resource manager interface
Managed element
Sensor
Data
RESOURCE MANAGER

Figure 2.6 – Mapping of the vulnerability management activity into the autonomic lifecycle

security mechanisms into autonomic environments in order to get practical and working solutions. As happens in the real world, autonomic elements coexist within dynamic environments, interacting with other autonomic and non-autonomic elements. Nevertheless, there are also continuous threats that may compromise autonomic elements safety. 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. Autonomic systems must be able to manage their own state and perform required activities to achieve secure configurations. Autonomic elements unable to support this capability will age with time, becoming more vulnerable, insecure and useless. Real automation can be possible only if autonomic networks and systems fully integrate vulnerability management mechanisms for ensuring safe configurations. In that context, this thesis aims at contributing to the security of autonomic computing, with a particular focus on the management of configuration vulnerabilities.

2.5 Synthesis

In this chapter, we have presented a broad outlook of issues that are shaping the structure of today's networks and the Internet. The overwhelming advent of new technologies featuring more power, ubiquity and usability in disparate contexts, requires novel techniques and methodologies for managing the underlying networks that support them. The technological landscape changes fast and users collaborate to mold its future as well. Therefore, it is important to leverage clean and adaptive approaches to face this evolving reality. Autonomic computing provides robust foundations that may encompass this evolution and can help to address several current network management challenges. However, autonomic systems must ensure safe configurations if we want to trust autonomic solutions. In that context, the aim of this thesis is to provide novel autonomous mechanisms for dealing with vulnerabilities, in order to increase the security of self-governed networks and systems. In the next chapter, we present an in-depth investigation of current existing methods and techniques for dealing with vulnerabilities, discussing their benefits and limitations with respect to their application in autonomic systems and networks.

Chapter 3

Vulnerability management

Contents

3.1 Introduction **23**
3.2 The vulnerability management process **24**
  3.2.1 A brief history of vulnerability management 24
  3.2.2 On the organization of vulnerability management activities 25
3.3 Discovering Vulnerabilities **26**
  3.3.1 Exploiting testing methods 26
  3.3.2 Using network forensics 27
  3.3.3 Taking advantage of experience 28
3.4 Describing Vulnerabilities **29**
3.5 Detecting vulnerabilities **33**
  3.5.1 Analyzing device vulnerabilities 33
  3.5.2 Analyzing network vulnerabilities 35
  3.5.3 Correlating vulnerabilities with threats and attack graphs 36
3.6 Remediating vulnerabilities **37**
  3.6.1 Change management 38
  3.6.2 Risk and impact assessment 39
3.7 Research challenges **40**
3.8 Synthesis **41**

3.1 Introduction

Managing large-scale networks is a complex task and by nature, humans make errors when configuring them. In addition, changes performed by autonomic entities may increase their own security exposure. Because of this, vulnerable configurations are likely within such environments and they may potentially lead to a wide spectrum of negative and unwanted issues such as instability, unavailability, confidentiality problems, and many more. In that context, managing vulnerabilities becomes a crucial and challenging activity. Autonomic computing must integrate vulnerability management mechanisms so as to ensure safe configurations. In this chapter, we present a detailed outlook of the vulnerability management process as well as direct and orthogonal research efforts that may potentially contribute to the integration of this process into autonomic environments.

3.2 The vulnerability management process

In the same way bank robbers prepare their strikes identifying weak points to take advantage of, attackers seek for weaknesses that can exploit to gain access to computer systems. These weaknesses range from end-users (e.g. we could apply social techniques on someone to obtain critical information such as passwords), to computer vulnerabilities and network security policies. This thesis however, is focused on the management of computer vulnerabilities. To formally define this concept, let us recall the concept of vulnerability given in Definition 2. A vulnerability can be understood as a flaw or weakness in system security procedures, design, implementation, or internal controls that could be exercised (accidentally triggered or intentionally exploited) and result in a security breach or a violation of the system's security policy [108], [85]. With this concept in mind, we consider the following definition for vulnerability management.

Definition 3 (Vulnerability management)**.** *In computer security, vulnerability management refers to a continuous process which involves the (I) identification, (II) classification, and (III) remediation and mitigation of vulnerabilities [71].*

The activities involved in the vulnerability management process previously mentioned are in fact very general. They could be broken down into more granular tasks, however, they describe well the main challenging areas within the vulnerability management process. In this thesis, we are mainly focused on the identification and remediation activities. In this section, we first provide a brief historical review that puts in context the importance of vulnerability management. Afterwards, we present a classification of research areas that contribute to vulnerability assessment, which accompanied by remediation activities, completes the vulnerability management process.

3.2.1 A brief history of vulnerability management

The process of managing vulnerabilities has been exercised since long time ago in different fields. Military for instance, considers a vulnerability as the inability to withstand an adverse circumstance produced by a hostile environment. Therefore, security procedures are defined to state how to proceed in these situations, thus constituting part of a vulnerability management program [20]. In information technology, vulnerabilities have existed from the beginning. As an example, in 1903 the Marconi wireless telegraph was reported to contain a flaw that allowed an attacker to intercept any message sent by the device thus leading to unauthorized information disclosure [114]. In 1962, the Multics CTSS operating system running on IBM 7094 was reported to have a flaw allowing an unauthorized user to disclose the password of every user on the system. Reports of identified vulnerabilities have continued coming up since the 60's until our days. With the incursion of computing systems into human activities, the diversification of programs and services have set up more and more vulnerabilities compromising the security of such systems. These undesired effects made it clear the need of developing security programs able to deal with such security issues. In 1972, a computer security technology planning study was created by the U.S. Air Force Systems Command (AFSC). The objective of this program was to specify directives for securing the use and development of computing systems [108]. Since those days, managing vulnerabilities became an essential activity for any organization involving the use of computers or telecommunication equipments. Nowadays, several technologies that will be later detailed, are widely used for supporting this process such as the Common Vulnerabilities and Exposures system (CVE) [46] for enumerating known vulnerabilities or the Security Content Automation Protocol (SCAP) [21] for automating vulnerability management activities.

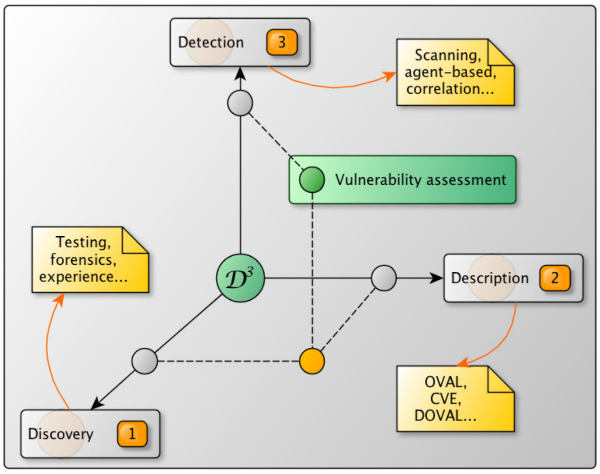


FIGURE 3.1 – Vulnerability assessment - D^3 classification

3.2.2 On the organization of vulnerability management activities

During our research work, we have investigated different methods and techniques contributing to vulnerability management in autonomic environments. However, we have realised that the identification of vulnerabilities itself constitutes a complex activity that requires to be organized so as to be properly executed. Such a setup should clearly consider the final target of this activity, machines. Therefore, machine-readable mechanisms able to automate tasks involved in the autonomic process are essential. In light of this, we decompose vulnerability assessment activities by considering what we call a D^3 (*D cube*) classification as illustrated in Figure 3.1, D^3 standing for *Discovery, Description* and *Detection*.

The D^3 classification provides a basis, divided into three axes, for organizing the foundations of vulnerability assessment which constitutes the first step for the vulnerability management process to be embedded into autonomic environments. Autonomic entities must be provided with knowledge about current vulnerabilities, either with mechanisms for discovering threats by themselves or with machine-readable specifications about security alerts. Regardless of the mechanism chosen, vulnerability discovery techniques (axis 1) must be analyzed in order to unveil unknown vulnerabilities, and to explore and understand the constantly evolving threatening environment. Taking advantage of these mechanisms, new knowledge becomes available for increasing the vulnerability awareness of self-governed environments. Such consciousness must be formally specified in order to be understood by computing devices, thus standard languages and protocols must be provided for describing and exchanging security advisories (axis 2). Such security knowledge increases the capability of autonomic networks and systems for detecting vulnerabilities in the surrounding environment (axis 3) and provides a strong support for taking decisions when performing self-management activities.

Vulnerability assessment activities increase the awareness of autonomic environments that along with remediation activities complete the vulnerability management process. In order to cover the automation of the vulnerability assessment process, we propose the classification depicted in Figure 3.2 where we divide the activity into three main areas following the D^3 approach. First, we present current approaches for discovering unknown vulnerabilities connecting their applicability over autonomic environments. Then, we detail description languages able to represent security advisories about known vulnerabilities. Afterwards, we describe techniques that take advantage of such knowledge for performing security analysis. A section covering remediation activities is provided at the end of this chapter, completing the vulnerability management loop.

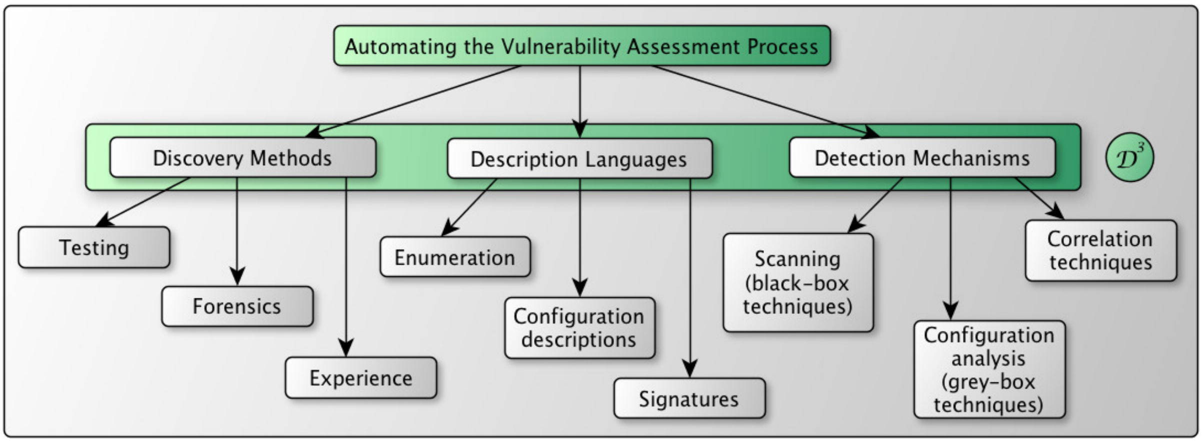

Automating the Vulnerability Assessment Process
Discovery Methods
Description Languages
Detection Mechanisms
$\mathcal{D}^3$
Testing
Forensics
Experience
Enumeration
Configuration descriptions
Signatures
Scanning (black-box techniques)
Configuration analysis (grey-box techniques)
Correlation techniques

FIGURE 3.2 – Automated vulnerability assessment classification

3.3 Discovering Vulnerabilities

Discovering vulnerabilities is a fundamental activity which indeed gives meaning to the vulnerability management process itself. With no known vulnerabilities, the process would not make sense in the first place. Therefore, the ability to unveil threats present on the environment becomes an essential requirement. Because the whole set of potential vulnerabilities on each system is typically unknown, techniques for learning and discovering vulnerabilities must be developed. Under this perspective, it is also important to consider how such security information actually becomes available for protecting autonomic networks and systems. Indeed, there is a bigger ecosystem that not only involves vulnerabilities and security defects, but also people and their motivations [29]. New vulnerability information usually follows complex paths before users get benefits from it [72]. This security ecosystem is frequently governed by economical laws where buyers and sellers of new security findings establish complex vulnerability markets. It is not the target to discuss here how vulnerability information is traded in both white and black markets, however it is important to keep in mind that computer security is not only about technologies but also about people's behavior and motivations. In this section, we focus on mechanisms and means for discovering unknown vulnerabilities. This topic has been barely addressed within the field of autonomic networks, therefore the objective is to explore different approaches that can be potentially integrated into self-governed systems. Usually, almost every solution designed for standard systems can be embedded to some extent into autonomic closed loops. The ease of such integration depends on the nature of these approaches. However, we can normally think or design autonomic elements with sensors capable of consuming the required input, that will feed the existing solution, and adapting the performed actions to be wrapped by autonomic effectors. While automating the operation of these solutions might be, in some cases, quite straightforward ; making them self-adaptive to the surrounding environment as well as to work under a policy-based perspective (autonomous), constitutes an open and highly challenging problem. In light of this, a subset of prominent perspectives has been selected to provide an overview of available strategies for unveiling security issues. Our research includes some of the most studied fields including testing methods, network forensics techniques and case-based reasoning.

3.3.1 Exploiting testing methods

Exploiting testing methods provides a powerful approach to unknown vulnerability detection. Software applications are commonly designed with a set of specific goals in mind in order to provide effective solutions to the stated requirements. While developers pursue efficient functional constructions, testers perform tasks for identifying correctness, completeness, quality and security of developed computer software. Several approaches under the tester point of view are used when software tests are designed [120]. White-box testing allows testers to have access to internal structures, algorithms and the code itself of the software being tested, e.g. static analysis, code review. Black-box testing on the other hand does not provide information about the internal implementation and the software is seen as an input-output black box, e.g. dynamic analysis, performance tests. Grey-box testing combines the previous approaches by considering knowledge about the software internals but executes the tests at a black-box level, e.g. internal database testing. Under a self-governing perspective, autonomic elements could be analyzed using these techniques in order to identify abnormal behavior. This first step would provide useful information to the underlying government mechanism about unsafe components. Even though traditional testing techniques unveil an important amount of software problems, it is unfortunately common for testers to focus on functionality correctness and to omit strong security tests. At the time of software construction and testing, normal input tests are frequently more numerous than anomalous input tests. Because of this, several unknown vulnerabilities remain hidden behind untested input data. Fuzzing techniques are intended to find these kind of software bugs.

The fuzzing approach complements traditional testing to discover combinations of code and data by exploiting the power of randomness, protocol knowledge, and attack heuristics. Instead of using normal input data, fuzzing methods generate unexpected or malformed inputs for feeding the target software. Software behavior is then assessed in order to identify potential vulnerability hotspots. A wide view of current fuzzing techniques is presented in [53] where different approaches are explored, highlighting fuzzing contributions to vulnerability detection. Since application's input space is in most cases impossible to enumerate, fuzzing techniques use two main approaches: data generation and data mutation (randomly modifying well-formed inputs). However, traditional fuzzing tools present some randomness related drawbacks when working with applications that perform various types of integrity checks (e.g. checksum). Checksum mechanisms reject an important part of the generated input set at initial execution stages, decreasing the fuzzer effectiveness and code coverage. The work presented in [159] identifies the stated problems and presents an approach to overcome early malformed input rejection due to checksum failures. The work reported in [50] introduces a fuzzing-based methodology called configuration fuzzing where the configuration of the running application is randomly modified at certain execution points in order to check for vulnerabilities that only arise in certain conditions. Considering the fact that autonomic systems are ruled by high-level policies, the same mechanism could be used for specifying properties and the expected behavior of a piece of software. This would allow testing solutions to be embedded into self-governing entities in order to analyze their operation by checking the current state against the defined policies. As a first step, this process could inform administrators about abnormal or unexpected behavior. It could also be taken one step further by automatically generating reports about the current system configuration for future use, looking for available solutions, configuring changes, and applying patches.

3.3.2 Using network forensics

Techniques based on network forensics can also be used for discovering unknown vulnerabilities. Network forensics is known as the process of archiving all network traffic and analyzing subsets as necessary [44]. This activity generally involves traffic reconstruction to assess network activity, providing useful information for further network-related events analysis. Network forensics belongs to a wider computing field called digital forensics. Digital forensics is defined as the

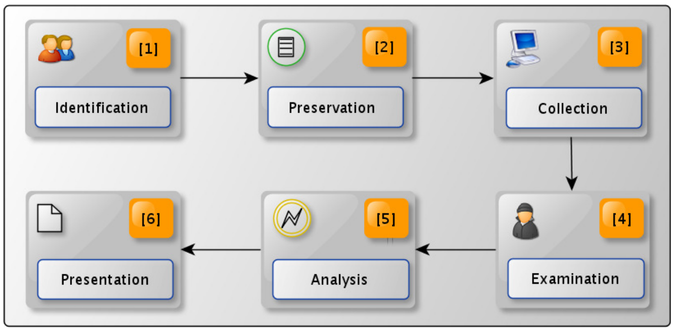


FIGURE 3.3 – Forensic investigation process [54]

use of scientifically derived and proven methods towards the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations [54]. Figure 3.3 shows the stages involved in a forensic investigation.

Digital forensics benefits go beyond criminal prosecution. Several contributions to computer security have been born within this field and they are widely used over different scenarios [44, 2]. Even though they are mostly targeted on traditional networked environments and not self-governing systems, these works provide a strong basis for being integrated into autonomic networks. Digital forensics provides a deep understanding of discovering mechanisms about the anatomy of an attack, how to gather pieces of evidence and put them together in order to determine how an attack took place on the system, when it was committed, who are the perpetrators and where they come from. Because of this, its robust technical background on data collection and analysis establishes a solid framework for performing computer system investigations, thus providing support to vulnerability management activities. The work presented in [2] provides an overview of digital forensics methodologies, computer and network vulnerabilities and security measures, and forensics tracking mechanisms to detect and deter intruders.

Forensic tools are extremely important on forensic scenarios. Effectiveness, efficiency, reproducibility, evidence consistency and integrity, traceability, security, are some of several factors that have to be considered when designing a forensic tool. Depending on the type of environment where the tool will be used, the previous features are required for a successful activity execution. Fundamental concepts related to network forensics and important features that a network forensic analysis tool (NFAT) should implement are presented in [44], namely, NFAT place and purpose, data capture, traffic analysis, and NFAT interaction and discovery services it should provide. Such concepts also support decentralized approaches where various forensic tools can collaborate in order to analyze the whole network. Evidence collection is a highly active study field as it constitutes an essential stage within digital forensic investigations. Results of this stage feed the analysis stage. The work proposed in [160] presents a network evidence graph-based approach which facilitates evidence presentation and automated reasoning. Such approaches may highly contribute to the integration and positioning of forensic actuators into autonomic environments aiming at identifying, analyzing and providing reports about suspicious or abnormal behavior, and therefore highly contributing to the first dimension of the D^3 approach.

3.3.3 Taking advantage of experience

Past experience in dealing with vulnerabilities strengthens the ability to face new security problems. Under this perspective, performing case-based reasoning also provides interesting and useful outlooks for detecting unknown vulnerabilities. Case-based reasoning (CBR) is a problem solving methodology which exploits past experience. Past experience is maintained in the form of problem-solution pairs, also called cases. On the arrival of new problems, solutions of similar past problems are used after appropriate adaptation. The work presented on [92] applies the CBR approach for enabling self-configuration capabilities in autonomic systems. This approach can be applied on unknown situations, where some kind of nearness concept may be used in order to classify how similar the new problem is to the problems already known. Using different algorithms, solutions for known similar problems can be modified to achieve the solution of the new problem. Indeed, an approach for dealing with fault management issues using CBR has been proposed in [151]. The authors outline a distributed case-based reasoning system over a self-organizing platform capable of assisting operators in finding solutions for faults. Such approaches can provide strong support for developing autonomic solutions based on previous experience. Such experience can be thought as part of the know-how that autonomic systems use to operate themselves. Moreover, considering self-configuration as a response for repairing vulnerable configurations, case-based reasoning strategies can provide expertise and feed a database of known vulnerabilities, which is the heart of the next sections.

System improvements usually provide new or better technological capabilities, however, they also carry new space for security concerns as well. Discovering unknown vulnerability constitutes an important security feature of self-governing systems. A wide spectrum of methods and techniques may be used to achieve this point. This section has covered some of the most important and promising areas for discovering software flaws and configuration misuse, from fuzzing methods (proactive) to forensics techniques (reactive). Even though there exist autonomic approaches for unveiling vulnerable configurations, our research work indicates that most of the prominent contributions are not oriented to self-governed environments. Taking advantage of such approaches remains as a challenging activity. Autonomic environments should incorporate these capabilities in order to become adaptive with the changing environment being able to unveil potential unknown security threats. In addition, we consider that no matter what technique is used for discovering vulnerabilities, describing vulnerabilities in a standardized and machine-readable manner is essential for integrating such approaches into the autonomic management plane. This topic constitutes the central point of the next section.

3.4 Describing Vulnerabilities

By the time a vulnerability is discovered, a time span will occur before system administrators are noticed about its existence. Another time will pass before a corrective solution exists and yet another will pass until all systems are patched. Attacker's activity usually takes place during this period of time, that can last from a few hours to several months or years. Because of this, it is important to develop a robust background as well as mechanisms and techniques in order to establish consistent and uniform means for describing vulnerabilities, analyzing and detecting them, and exchanging related information. The Common Vulnerabilities and Exposures or CVE system [46] has been introduced by the MITRE Corporation [104] as an effort for standardizing the enumeration of known information security vulnerabilities. The CVE dictionary, widely used today, allows the community to be aware of current existing threats and exposures by providing unique identifiers to each known security alert as well as descriptions written in natural language. This is extremely useful for increasing the security awareness of autonomic systems. However, the CVE standard only provides means for informing about their existence but not for their assessment. Describing the anatomy of known vulnerabilities and the techniques developed to this end are fundamental as they provide essential means for dealing with vulnerability management. This knowledge can highly increase the know-how of self-governing systems providing strong support for developing and integrating autonomic security solutions.

During the last years, several approaches on vulnerability analysis have been taken. Vulnerability signatures have been widely used by intrusion prevention systems (IPS) and intrusion detection systems (IDS). They are intended to describe the characteristics of the input that lead the execution of a specific program to a vulnerable point and the state that such program must hold for the vulnerability to be exploited [32]. Vulnerability signatures are mostly used for analyzing traffic looking for specific patterns and detecting potential attacks. The work proposed in [32] contributes to the second dimension of the D^3 approach by automatically generating high coverage vulnerability-based signatures. However, there are no fully developed up-to-date standards available for their representation and the generation as well as their coverage still remains an open problem. In addition to this issue, IDSs also lack of fully mature standards for exchanging alerts. The Intrusion Detection Message Exchange Format (IDMEF) is a data model to represent information exported by intrusion detection systems proposed by the Internet Engineering Task Force (IETF), but its status is currently experimental and it will not change [126]. Much of the work done in vulnerability analysis has defined the assessment infrastructure using its own vulnerability specification language arising compatibility and interoperability problems. Languages such as VulnXML [135] and the Application Vulnerability Description Language (ADVL) [19] have been developed as an attempt to mitigate these problems and to promote the exchange of security information among applications and security entities. However, these languages are only focused on web applications covering a subset of the existing vulnerabilities in current computer systems.

In order to cope with the problems described previously, the Open Vulnerability and Assessment Language (OVAL), supported by MITRE Corporation, standardizes how to assess and report upon the machine state of computer systems [117]. OVAL is an XML-based language and therefore, it inherits all XML features like platform independence, interoperability, transportability and readability. The OVAL specification is supported by XML schemas which serve as both the framework and vocabulary for the language. These schemas specify what content is valid within an OVAL document and what is not. OVAL is organized in three main XML Schemas, namely, (i) the OVAL Definition Schema that expresses a specific machine state; (ii) the OVAL Characteristics Schema that stores configuration information gathered from a system; and (iii) the OVAL Results Schema that presents the output from a comparison of an OVAL Definition against an OVAL System Characteristics instance. Valid XML instances typically represent specific machine states such as vulnerable states, configuration settings and patch states. Usually, a vulnerability is considered as a logical combination of conditions that if observed on a target system, the security problem described by such vulnerability is present on the system. The OVAL language follows the same idea by considering a vulnerability description as 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 OVAL state. Components found in the system matching the OVAL object description are called OVAL items. These items are compared against the specified OVAL state in order to build the OVAL test result. The overall result for the criteria specified in the OVAL definition will be built using the results of each referenced OVAL test.

We now put forward an illustrative OVAL example of a vulnerability description for the Cisco Internetwork Operating System (IOS) [86]. This example aims at providing an overview of OVAL's main building blocks. It is based on a real vulnerability specification but it was simplified

```
 1. <?xml version="1.0" encoding="UTF-8"?>
 2. <oval_definitions xsi:schemaLocation="http://oval.mitre.org/XMLSchema/oval-definitions-5 oval-
      definitions-schema.xsd http://oval.mitre.org/XMLSchema/oval-definitions-5#ios ios-definitions-
      schema.xsd http://oval.mitre.org/XMLSchema/oval-common-5 oval-common-schema.xsd" xmlns="http://
      oval.mitre.org/XMLSchema/oval-definitions-5" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
      instance" xmlns:oval="http://oval.mitre.org/XMLSchema/oval-common-5" xmlns:oval-def="http://oval
      .mitre.org/XMLSchema/oval-definitions-5">
 3.   <generator>
 4.     <oval:product_name>The OVAL Repository</oval:product_name>
 5.     <oval:schema_version>5.7</oval:schema_version>
 6.     <oval:timestamp>2010-06-18T15:02:46.614-04:00</oval:timestamp>
 7.   </generator>
 8.   <definitions>
 9.     <definition id="oval:org.mitre.oval:def:6086" version="2" class="vulnerability">
10.       <metadata>
11.         <title>Cisco IOS SIP Denial of Service Vulnerability</title>
12.         <affected family="ios">
13.           <platform>Cisco IOS</platform>
14.         </affected>
15.         <reference source="CVE" ref_id="CVE-2008-3800" ref_url="http://cve.mitre.org/cgi-bin/
    cvename.cgi?name=CVE-2008-3800"/>
16.         <description> Vulnerable SIP implementation ... </description>
17.         <oval_repository>
18.           <dates> .... </dates>
19.           <status> .... </status>
20.         </oval_repository>
21.       </metadata>

22.       <criteria operator="AND">
23.         <criterion comment="IOS vulnerable versions" test_ref="oval:org.mitre.oval:tst:9025"/>
24.         <criterion comment="SIP Test using runnning config. result contains: 5060" test_ref="
    oval:org.mitre.oval:tst:24211"/>
25.       </criteria>
26.     </definition>
27.   </definitions>
```

Listing 3.1 – Cisco IOS vulnerability example (part 1)

to show how a basic OVAL definition looks like. An OVAL definition is typically written in one XML file but here we divide it in two parts just for didactic purposes. The first part illustrated in Listing 3.1 contains the OVAL definition that represents our vulnerability description. A definition is the key structure in OVAL. It is analogous to the logical sentence or proposition: if a computer's state matches the configuration parameters laid out in the criteria, then that computer exhibits the state described. Within this example, the vulnerability definition with id *oval:org.mitre.oval:def:6086* (lines 9-26) states that the referred vulnerability is present on the system if both following conditions hold: (i) the IOS version belongs to a set of affected IOS versions (line 23), and (ii) VoIP is configured (line 24). The second part of our example illustrated in Listing 3.2 defines the rest of required components referred on the first part, namely, OVAL tests (lines 28-37), OVAL objects (lines 38-43) and OVAL states (lines 44-52). An OVAL Test is used by one or more definitions to compare an object(s) against a defined state. An OVAL Object describes a unique set of items to look for on a system. This unique set of items can then be used by an OVAL Test and compared against an OVAL State. An OVAL State is a collection of one or more characteristics pertaining to a specific object type. The OVAL State is used by an OVAL Test to determine if a unique set of items identified on a system meet certain characteristics. The first condition is analyzed by the first test with id *oval:org.mitre.oval:tst:9025* (lines 29-32). This *version test* refers to one OVAL object (line 39) and one OVAL state (lines 45-47). It will be true if and only if the specified object match the specified state. The *pattern match* expression allows to specify a family of IOS versions using a regular expression (line 46). The second condition is analyzed by the second test with id *oval:org.mitre.oval:tst:24211* (lines 33-36). This *line test* refers to one OVAL object (lines 40-42) and one OVAL state (lines 48-51). It will be true if and only if the sub-command *show running-config* result contains the port number *5060* (line 50).

The OVAL language currently constitutes a de facto standard for describing vulnerabilities as well as good practices. Autonomic environments should take advantage of this capability in order

```
  <tests>
    <version55_test id="oval:org.mitre.oval:tst:9025" version="1" comment="IOS vulnerable
   versions" check_existence="at_least_one_exists" check="at least one" xmlns="http://oval.mitre.org
   /XMLSchema/oval-definitions-5#ios">
      <object object_ref="oval:org.mitre.oval:obj:6804"/>
      <state state_ref="oval:org.mitre.oval:ste:4432"/>
    </version55_test>
    <line_test id="oval:org.mitre.oval:tst:24211" version="1" comment="SIP Test using ip socket.
   config contains: 5060 .may generate few false positive" check_existence="at_least_one_exists"
   check="at least one" xmlns="http://oval.mitre.org/XMLSchema/oval-definitions-5#ios">
      <object object_ref="oval:org.mitre.oval:obj:6385"/>
      <state state_ref="oval:org.mitre.oval:ste:6946"/>
    </line_test>
  </tests>

  <objects>
    <version55_object id="oval:org.mitre.oval:obj:6804" version="1" xmlns="http://oval.mitre.org/
   XMLSchema/oval-definitions-5#ios"/>
    <line_object id="oval:org.mitre.oval:obj:6385" version="1" xmlns="http://oval.mitre.org/
   XMLSchema/oval-definitions-5#ios">
      <show_subcommand>show running-config</show_subcommand>
    </line_object>
  </objects>

  <states>
    <version55_state id="oval:org.mitre.oval:ste:4432" version="1" xmlns="http://oval.mitre.org/
   XMLSchema/oval-definitions-5#ios">
      <version_string operation="pattern match">12\.3\(\d+\w*\)XF(\d.*|$)</version_string>
    </version55_state>
    <line_state id="oval:org.mitre.oval:ste:6946" version="1" xmlns="http://oval.mitre.org/
   XMLSchema/oval-definitions-5#ios">
      <show_subcommand>show running-config</show_subcommand>
      <config_line operation="pattern match">\s+5060($|\s+)</config_line>
    </line_state>
  </states>

</oval_definitions>
```

Listing 3.2 – Cisco IOS vulnerability example (part 2)

to augment their vulnerability awareness. Within our contributions, we have heavily exploited the benefits of the OVAL language as detailed later in Part II. During the last years, several related languages have evolved around the OVAL language. The National Institute of Standards and Technology (NIST) [108] has supported the development of the Security Content Automation Protocol (SCAP) [21]. The SCAP protocol is a suite of specifications that standardize the format and nomenclature by which security software communicate information about publicly known software flaws and security configurations. These advisories are annotated with common identifiers and embedded in XML. SCAP also utilizes software flaw and security configuration standard reference data, also known as SCAP content. This reference data is provided by the National Vulnerability Database (NVD) [111], which is managed by NIST and supported by the Department of Homeland Security (DHS) [55]. Other public vulnerability databases exist as well, such as the Open Source Vulnerability Database (OSVDB) [114]. However, the vulnerability descriptions provided by them are usually understandable by humans and not by machines, thus difficulting an automated consumption of this security knowledge.

SCAP can be used for several purposes, including automating vulnerability checking, technical control compliance activities, and security measurement. The integration of SCAP into self-governing environments constitutes a major challenge, however its automation-targeted nature can highly benefit future autonomics development. The SCAP protocol includes the OVAL language and complements it with enumeration languages such as the Common Platform Enumeration (CPE), a nomenclature and dictionary of hardware, operating systems, and applications [45] ; the Common Configuration Enumeration (CCE), a nomenclature and dictionary of security software configurations [35] ; and CVE for enumerating security-related software flaws. SCAP also considers the eXtensible Configuration Checklist Description Format (XCCDF) for authoring security benchmarks and reporting checklist evaluation results [167], and the Common

Vulnerability Scoring System (CVSS) for measuring and scoring the relative severity of software flaw vulnerabilities [47]. The specifications involved in the SCAP protocol not only allow us to specify vulnerabilities (with OVAL), but also to bring a system into compliance through the remediation of identified vulnerabilities or misconfigurations (with XCCDF). These features perfectly fit requirements for expressing which actions autonomic systems should perform when vulnerable states are detected.

These specifications highly contribute to security automation and to the vulnerability management activity. Other works have been done as well, such as the one proposed in [158] where an ontology-based approach for dealing with vulnerability management activities called OVM is presented. However, its connection with autonomic technologies is not addressed, nor the scalability or actual impact on real networks. Moreover, OVM only considers vulnerability management activities from a high-level perspective, focusing on the process rather than fine-grained concepts that allow vulnerabilities to be described. Languages such as OVAL are crucial for representing security knowledge that in turn involves technical details. The OVAL language has been further detailed as a means for performing the assessment activity. Standardization efforts are essential for exchanging this knowledge and it requires a strong support of the community. Autonomic networks and systems should be able to manage these security advisories and capitalize the knowledge provided by vulnerability descriptions repositories in order to increase their vulnerability awareness. Moreover, autonomic elements should be able to provide appropriate sensing and actuation mechanisms, as depicted in Figure 2.3, in order to be autonomously assessed and eventually corrected. In this section we have investigated different mechanisms for describing vulnerabilities and exchanging related information which provide a strong solid for achieving this goal. Next section is dedicated to explore existing methods and techniques used for detecting computer system vulnerabilities.

3.5 Detecting vulnerabilities

Once a vulnerability is known and described, mechanisms used for detecting it constitute a central concern on autonomic networks and systems. Self-governed environments should be able to incorporate and take advantage of security advisories provided by different sources when vulnerability assessment activities are performed. In this section we will discuss different methods and systems for assessing both device and network vulnerabilities. These methods contribute to the third dimension of the D^3 approach. We will also present several approaches for correlating security information and analyzing potential attacks and security policies violations.

3.5.1 Analyzing device vulnerabilities

The assessment of local vulnerabilities on a device requires the investigation of specific states and conditions that may allow an attacker to compromise the system. While black-box techniques, such as network scanning discussed in subsection 3.5.2, can provide useful security information without requiring specific tools in the device under analysis, grey-box techniques can highly enhance the obtained information by accessing the device itself and inspecting its internal state and particular configurations. Assessing vulnerabilities using the OVAL language can be understood as a grey-box approach since it not only allows to specify vulnerability descriptions but also standardizes the three main steps of the assessment process, namely, representing configuration information of systems for testing ; analyzing the system for the presence of the specified machine state (vulnerability, configuration, patch state, etc.) ; and reporting the results of the assessment. Figure 3.4 describes the main steps of the OVAL process [117]. At step 1, security advisories are

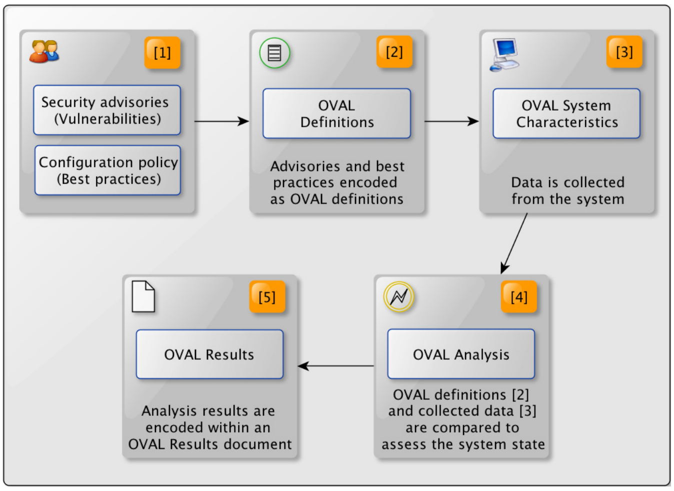

[1]
Security advisories (Vulnerabilities)
Configuration policy (Best practices)
[2]
OVAL Definitions
Advisories and best practices encoded as OVAL definitions
[3]
OVAL System Characteristics
Data is collected from the system
[4]
OVAL Analysis
OVAL definitions [2] and collected data [3] are compared to assess the system state
[5]
OVAL Results
Analysis results are encoded within an OVAL Results document

FIGURE 3.4 – OVAL-based vulnerability assessment [117]

published and encoded as OVAL definitions at step 2. These definitions are then interpreted at step 3 to gather all the required information in order to perform the analysis at step 4. Once the OVAL analysis is done, a report is generated at step 5 identifying if the specific machine states described at step 2 are present or not on the target system. The integration of such process into the management plane of self-governing environments provides a strong basis for autonomously assessing the exposure of autonomic elements. This is one of the cornerstone of this thesis. It is important to notice that OVAL is a specification language and it allows to describe content ; real analysis is performed by OVAL interpreters. However, interpreter's activity is guided by the underlying OVAL language structure, thus we can think of OVAL as a language for specifying, analyzing and reporting vulnerabilities. Moreover, because OVAL allows to describe specific machine states, semantics can be used in several ways, i.e., states that can not hold (vulnerabilities), states that should hold (best practices).

Several OVAL-based systems have been developed since the OVAL language was released. The work proposed in [94] presents the design and implementation of a vulnerability assessment tool based on the OVAL language to detect weak points in Linux System. The proposed approach has more readability, reliability, scalability and simplicity than traditional tools. Although this work was published in 2004, it clearly highlights OVAL's potentiality. Others up-to-date OVAL-based tools exist as well. Ovaldi [118] is a free OVAL interpreter maintained by MITRE intended to provide a reference implementation for evaluating OVAL definitions. Current releases of the interpreter cover a wide, but not complete, part of OVAL's specification. This incomplete coverage arises difficulties to extrapolate its usage within other fields such as forensic scenarios. Although Ovaldi is a robust tool, its main development language (C) is platform-dependent, thus increasing maintenance efforts for each OVAL supported platform. Moreover, its internal design and continuous official releases make it quite difficult to use as a base start point for customized or extended OVAL-based tools. The work proposed in [26] presents XOvaldi, a live forensic, multi-platform and extensible OVAL-based system for digital evidence collection. XOvaldi has been purely written in Java [88] and its plugin-based architecture, as well as its automatic model adaptation, provide easy means for naturally evolving with dynamic forensics scenarios. As explained later in Part III, we have heavily used XOvaldi, and also extended it, for conducting various experiments in this thesis. The effort invested in the development of OVAL-based assessment systems provides a strong background for automating the detection of known vulnerabilities. These systems can be then combined and integrated into autonomic environments in order to enhance their ability for detecting security threats.

Autonomic networks must be capable of adapting according to specific policies or security issues. Because of this, analyzing network vulnerabilities positions strong challenges that autonomic entities should be able to solve. Network vulnerability analysis, also known as vulnerability scanning, involves activities to determine vulnerabilities and security holes exploitable within the target network. In order to perform this analysis, data collection and examination has to be done over members of the network and correlation techniques must be applied to analyze the target network as a whole.

3.5.2 Analyzing network vulnerabilities

The ability of identifying host-based and distributed vulnerabilities constitutes the first step for the vulnerability management process to be completely embedded into the management plane of autonomic networks and systems. Network scanning constitutes one of the most used techniques for discovering devices in a network. This process allows to identify active hosts either for security assessment or for performing different kinds of attacks. The enumeration of a network provides useful information such as users, groups and running services on each network member. Port scanners are usually used within this process for analyzing each device in order to detect which ports are open and which services are listening on them. Probes against these ports and the behavior presented by the target device may allow port scanners to infer useful information about the software running on each port such as the type of application and its version.

The kind of response emitted by the device under analysis indicates whether the port is in use, and if so, it can be further explored for detecting weaknesses. Fingerprinting for instance, is a technique used for interpreting the responses of an operating system by sending to it different combinations of data and analyzing its responses against a fingerprint database [101]. Fingerprints are usually generated by the application of a hash function over a specific piece of data where the obtained hash value uniquely identifies the input data. Behavior patterns for well and bad-formed messages are correlated with the observed responses in order to obtain a match of known systems and applications, and related vulnerabilities. Currently, several network scanners exist for assessing vulnerabilities on a target network such as Nessus [106], OpenVAS [113] or SAINT [132]. Some of them use the functionalities provided by powerful port scanners such as Nmap [109]. However, these tools do not provide standard means for describing and exchanging the vulnerabilities they are able to assess. Languages such as OVAL are highly required. In addition, none of them have currently shown trends or means for being embedded into self-governed environments.

Regardless of the mechanisms used for individually assessing devices, grey-box techniques such as agent-based vulnerability assessment or black box techniques such as network scanning, it is essential to develop approaches capable of analyzing the network and its relations as a whole. Steps taken by an intruder usually respond to some favorable conditions present on the system. By modeling these capabilities and actions to take, inference can be performed. Reasoning engines are widely used in this field to achieve automated approaches. As an example, the work presented in [116] and enhanced in [115, 130] introduces a logic-based network security analyzer called MulVAL. MulVAL is a framework and reasoning system that conducts multi-host, multistage vulnerability analysis on a network. MulVAL uses the OVAL language to analyze each host on the network. The reasoning engine consists of a collection of Datalog rules [52] that captures the operating system behavior and the interaction of various components in the network. After gathering all required information from the environment, the analysis performed by MulVAL has two main parts. First, all possible multi-steps accesses and inferred privileges on each user are computed. Then, results are compared against the stated global policy. If the analysis results show a user with some kind of privilege that is not present on the global policy, a security breach has been found. Due to autonomic networks are by nature governed by policies, such approach and the methodology used are particularly appropriated to be embedded into policy-driven environments such as autonomic networks and systems.

3.5.3 Correlating vulnerabilities with threats and attack graphs

The mechanisms used for detecting vulnerabilities in autonomic networks provide an extremely useful overview of the potential security problems that might be exploited on a target network. However, this information can be yet enhanced by correlating security threats found in the assessment phase, as shown in Figure 2.3, and analyzing how the activity of an attacker could take advantage of them. Attack modeling languages such as ACML [119] allows to express the capability gained by an attacker at each step of the intrusion process. This approach allows to link network events and detect multi-steps attacks. This can be very useful in the context of autonomic environments as it could support the analysis of scenarios where an autonomic element is compromised, and the impact over those elements connected to it and the relationships between them, e.g. service provisioning requirements. Previous work done in [150] considers the idea of a requires-provide model where each gained privilege by an attacker opens new intrusion capabilities. This concept is extremely important when analyzing attack sequences and provides robust foundations for attack graphs approaches.

A deep review on attack graphs is presented in [99] where several contributions on this topic are analyzed. The authors make clear the achievements and limitations of attack graphs by discussing fundamental construction concepts as well as their use in network security approaches. In this thesis, we do not deal with attack graphs and multi-step attacks. However, insightful research work on network security assessment and remediation techniques using attack graphs has been previously reported in [142], [10]. Bayesian attack graphs have been also used for assisting administrators on mitigation plans [122]. Our approach is indeed complementary. While a vulnerability represents a potential security problem that could be exploited by an attacker in order to compromise the system, an attack graph describes the actual activity and steps performed by an attacker in order to achieve a desired goal. In other words, a vulnerability is focused on the system by identifying insecure states, and an attack graph is focused on the behavior of the attacker that takes advantage of these security weaknesses.

Recently, the Common Attack Pattern Enumeration and Classification (CAPEC) language [33] has been proposed by MITRE for describing attack patterns. CAPEC involves a collection of common methods for exploiting software systems, including network attack patterns. The CAPEC schema also enables the use of the Cyber Observable eXpression (CybOX) language [48] as a means for describing cyber observables that exist for various steps and portions of the attack pattern. Such cyber observables refer to events or stateful measures that can be observed in the operational domain. CybOX also uses the Malware Attribute Enumeration and Characterization language (MAEC) [102] for characterizing the behavior of malware, and the Common Event Expression (CEE) [37] for unifying the representation and classification of events found in the lifecycle of systems and networks. Currently, MITRE is also considering automated mechanisms for converting MAEC and CybOX content into OVAL checks in order to detect malware artifacts and other host-based cyber observables. These initiatives are still in an early stage though they are quite promising as their contributions might harden the security of autonomic environments.

While some authors focus on attacks anatomy using attack graphs, other authors also propose metrics for quantifying attack potentiality which depends on several factors. For instance, the work presented in [137] proposes a framework for measuring the vulnerability of individual hosts based on current and historical operational data for vulnerabilities and attacks. Metrics are particularly important within autonomic environments as they can be used for autonomously parametrizing the behavior of the entire system. Such measurements can be successfully integrated into the closed loop that depicts the lifecycle of self-governed elements in order to feed and control their behavior. The work proposed in [1] presents a method for calculating a policy security metric which can be used to evaluate how good a policy is, as well as compare policies and assess policy changes. Such approach provides support not only for assessing the dynamics of individual policy-based self-governed systems, but also for evaluating the overall behavior of autonomic environments in which vulnerabilities play a critical role.

Assessing vulnerabilities constitutes a crucial activity that enables autonomic networks and systems to identify threats that potentially may compromise their security. This ability is in turn complemented by approaches capable of correlating exploitable network weaknesses in order to identify potential successful attacks. The integration of such mechanisms within the management plane of autonomic environments provides a powerful basis for assessing their own exposure. In this section we have presented methods for detecting device and network vulnerabilities, and we have discussed different approaches for correlating security information and inferring potential attacks. However, in order to ensure safe configurations in autonomic environments, remediation activities to eradicate these weaknesses are essential. This is the topic of the next section.

3.6 Remediating vulnerabilities

When a system is found to be vulnerable, remediation actions must be executed in order to bring the system into a secure state. However, these actions must also encompass general system's operational policies. Indeed, every management activity, whatever its impact on the system is, should care about respecting the overall governing rules. In autonomics, the ability to orchestrate different management aspects under a common and single vision constitutes a hard and challenging problem. We have realised that even with optimal solutions for autonomously managing different aspects of self-governing environments, a common layer where all these components might interact, is essential for achieving real and functional autonomic systems. Indeed, this is one of the main goals proposed by the UniverSelf project [154]. In that context, the use of standard languages and mechanisms that leverage interoperability becomes an important requirement of autonomic approaches and solutions. As explained before, we do not deal with rule consistency problems at other operational levels of self-governed environments ; we are focused on autonomous solutions for vulnerability management. Nevertheless, the contributions presented in this thesis make a vast use of standard languages and technologies. This vision not only fosters the use of existing solutions but also makes available new scientific contributions that might feed and support future ones.

Considering a common, consistent and coherent governing framework, able to speak the same language among different constituting autonomic components, is essential to achieve real autonomic computing. This kind of frameworks provide support to line up general policies under a holistic perspective. Therefore, when changes are performed, global mechanisms in charge of controlling affected autonomic elements, can be safely applied. Commonly, vulnerabilities can take the form of software flaws or misconfiguration errors, and they can be usually corrected by means of different methods such as applying software patches, adjusting configuration settings or removing the affected software [103]. However, when corrective actions are performed, changes are introduced in the environment, thus change management mechanisms must be taken into account. Risk assessment methods are also important, as they provide a strong basis for analyzing the impact of remediation activities within the vulnerability treatment process. It is crucial to ensure safe changes not only from an operational viewpoint but from a security perspective too. Within this section, we point out related work about change management on networks and systems considering vulnerability treatments, and we also cover different approaches contributing to the risk assessment activity.

3.6.1 Change management

Change management already constitutes a challenging activity when performed by human administrators, the automation of such process is even more complex. Decisions are usually based on factors that depend on the nature of the system, laws that rule the behavior and purpose of the system. While some entities will prioritize functionality over security, others might follow the other way around. But most importantly, no matter what the chosen action is, performed changes should be effective as to the objective they were designed for, and consistent with the rules that govern the system. The latter is not always easy to achieve thus mechanisms for solving conflicts and techniques for reducing the impact of these changes must be taken into account as well. While several works have been focused on vulnerability management such as [158], just a few works address this topic into autonomic environments, mainly focused on the vulnerability assessment activity [40]. Orthogonal works have been proposed in the area of change management. They contribute to ensure the correctness of configuration operations and their positive impact over services, but they do not consider security aspects with respect to vulnerable configurations. Therefore, vulnerability management activities and change management techniques become interconnected. Network changes must be evaluated in order to ensure safe modifications and at the same time, vulnerable states must be remediated by performing controlled changes in the environment.

A large variety of techniques have been proposed to deal with changes in networks and systems. Information Technology Service Management (ITSM) is a fundamental work field for institutions and corporations ; intended to expose mechanisms for dealing with changes within an organization, trying to minimize the impact and, at the same time, maximize the utility provided by them [87]. The work proposed in [56] allows to determine if a given process transformation is likely to improve business performance based on process associated complexity metrics. Some approaches consider future changes already at system design such as design rationale, which involves an explicit documentation of the reasons behind the decisions taken, as used in [148]. The work reported in [41] presents a very interesting approach for predicting the effects of changes based on dynamic modeling dependency techniques. The work presented in [8, 9] targets the ability to validate configuration changes and their application on runtime, which increases the correctness and safety of reconfiguration activities within self-managed environments. This kind of works are very important because they provide a key support for the change management process, particularly for taking decisions about effective change implementations. Even though their analysis are usually focused on the operational impact rather than security concerns, they highlight key challenges that must be taken into account when vulnerability management activities are performed.

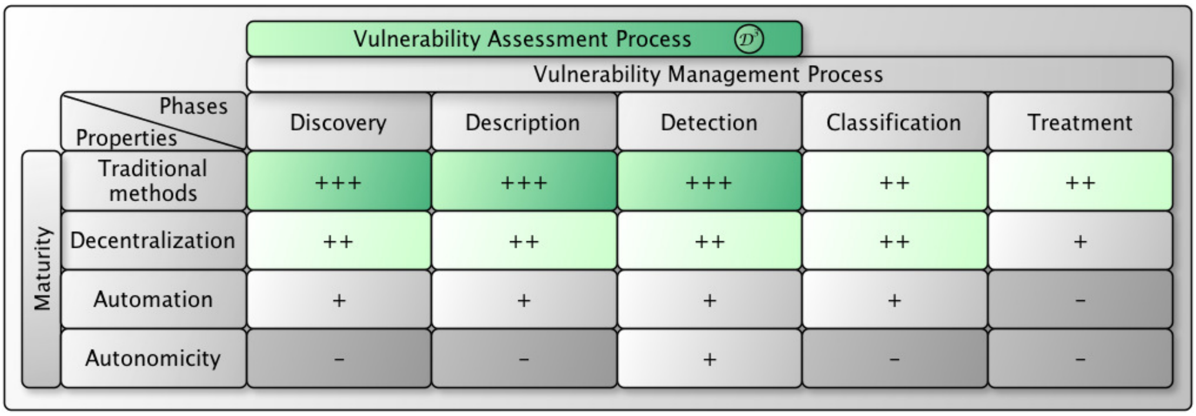
As an effort to automate the management of computer systems, different protocols have been also proposed in the past. As already mentioned, the SCAP protocol involves several specifications to automate security management mechanisms. Particularly, by using the XCCDF language, a system can be brought into compliance through the remediation of identified vulnerabilities or misconfigurations. In addition, the CVSS language can be used for rating IT vulnerabilities, thus contributing to the classification and impact analysis of security weaknesses. In order to deal with network management operations and changes, IETF has developed NETCONF [63], a network configuration protocol that provides mechanisms to install, manipulate and delete the configuration of network devices. The NETCONF protocol specification is a standard, though its deployment, as well as complete vendors implementations, seem to be still in an early stage. Nevertheless, very interesting works have already been presented showing evaluations of its maturity as well as diverse technical aspects [152], [157]. The integration of change management techniques into the vulnerability management plane may positively contribute to the overall security of current and future computer systems. In that context, risk and impact assessment techniques are also required for ensuring coherent automated security processes.

3.6.2 Risk and impact assessment

Usually, the security risk level of a system is based on three main combined factors, namely, the potentiality of a threat in conjunction with the exposure of that system to such threat, and the impact that a successful attack related to this threat may have in that system [49]. The exposure of a system in turn is directly related to the vulnerabilities present in such system. Therefore, managing vulnerabilities that might be exercised by a given threat constitutes a critical activity for quantifying the system exposure and hence, the risk level of autonomic networks and systems. Assessing change associated risks provides a key support for change management, as they are used take decisions about effective change implementations. Sometimes however, these decisions are not one hundred percent clear, therefore, measurable mechanisms are required to perform appropriate cost-benefit analysis. The work presented in [136] describes a method for evaluating the risk exposure associated with a change, which can be used to organize and take business-level decisions about required changes. Other works use past experience to analyze the impact of new changes [161]. Interconnected operational risks are considered in [138], which are used to schedule service changes with the lowest expected impact on the business.

As the autonomic nervous system, autonomic systems and networks must be able to perform diagnosis on the environment they are working on. Nevertheless, in practice, it is almost impossible to be aware of each security and exploitable hole for each system. Vulnerability detection provides large amounts of information that allow systems to be aware of threats, but autonomic systems need to see the big picture, not only as a snapshot but also considering past experience, in order to identify risk factors and perform progressive adaptation to achieve secure states. The work presented in [5] and [4] proposes a security metric-based framework that identifies and quantifies objectively security risk factors, including existing vulnerabilities, historical trend of vulnerabilities of remotely accessible services, prediction of potential vulnerabilities for any general network service and their estimated severity and finally propagation of an attack within the network. From an autonomic point of view, automated techniques to assess change associated risks like those presented here, are extremely important in order to achieve full autonomy.

Vulnerability management is an essential activity to ensure safe configurations within autonomic environments. When these systems are found to be vulnerable, remediation activities must be performed in order to erradicate these security weaknesses. In that context, changes are introduced in the environment and therefore, they must me properly managed. In this section, we have presented different change management techniques, as well as various mechanisms for assessing and evaluating the impact and the effectiveness of these changes. During our research, we have also observed that various problems still need to be addressed in order to integrate vulnerability management activities into autonomic environments. In the next section, we resume our findings and highlight different research axes that require to be further investigated.


Vulnerability Assessment Process
Vulnerability Management Process
Maturity

Phases / Properties	Discovery	Description	Detection	Classification	Treatment
Traditional methods	+++	+++	+++	++	++
Decentralization	++	++	++	++	+
Automation	+	+	+	+	-
Autonomicity	-	-	+	-	-

FIGURE 3.5 – Scientific maturity of vulnerability management activities with respect to autonomic networks

3.7 Research challenges

During the realization of our investigation, we have detected several challenges that must be addressed in order to be able to really integrate the autonomic computing approach into daily computer systems and networks. In that context, Figure 3.5 summarizes the scientific maturity of the vulnerability management process with respect to autonomic environments highlighting properties and issues that should be further investigated. As depicted in the first column, traditional methods for discovering unknown vulnerabilities count with a strong foundation, though decentralized as well as automated approaches require further investigation. Autonomic methods for addressing this capability have been barely or even at all discussed. Autonomic computing should incorporate these capabilities in order to unveil potential existing threats. As to describing known vulnerabilities, shown in the second column, several scientific contributions have been done, mostly from a device perspective. However, automatic generation as well as autonomic mechanisms for describing security problems are still in an early stage thus requiring research efforts in order to harden the foundations and maturity of such activity. Autonomic environments should capitalize such security knowledge in order to analyze themselves and assess their own exposure. A variety of methods have been proposed for detecting vulnerabilities in non-autonomic environments as shown in the third column of Figure 3.5. However, decentralized mechanisms exist in a minor degree, and automated and autonomic approaches have been weakly discussed. Once security problems are detected, they need to be classified according to their impact and risk, and remediated through the application of appropriate treatments. Vulnerability classification and treatment mechanisms, fourth and fifth columns of Figure 3.5 respectively, have only been partially addressed in the past for non-autonomic environments and mostly from a centralized perspective. Automated and autonomic approaches for dealing with these activities remain an open problem.

In light of this, we have observed several lacks mostly located on the automation and autonomicity properties of Figure 3.5 that should be further investigated. We highlight here three transversal research axes that are important to leverage the maturity and robustness of these properties.

- **Integration of vulnerability models into the management plane of networks and services**. This integration requires automated means for exchanging vulnerability descriptions in a standardized manner as well as detecting them. Vulnerability detection can be performed by dynamically translating vulnerability descriptions into configuration policy rules interpretable by autonomic configuration systems. In addition, such perspective can be enriched with automated vulnerability discovery mechanisms. This feature can enable the alignment of network components to desired and secure states. However, mechanisms for dealing with rules conflicts and policy consistency must be in place as well.

- **Investigation of collaborative methods and techniques for performing vulnerability management activities in a decentralized manner, with multiple vulnerability description datasources**. Autonomic elements need automated mechanisms for healing security holes. Control mechanisms and algorithms for classifying vulnerabilities and executing vulnerability treatments (apriori or aposteriori configuration operations) in an optimal manner must be analyzed. The SCAP protocol and particularly, the XCCDF language, in combination with the NETCONF protocol, can be extremely useful to achieve this point.
- **Formalized approaches for supporting the two previous themes are highly required**. Robust data collection mechanisms, mature system assessment techniques and efficient environment discovery methods constitute cornerstones within the integration of the vulnerability management process into self-governed environments. Autonomic networks and systems can take advantage of disparate computing fields such as digital forensics for threat discovery, machine learning for adaptation, or statistical models for prediction. Methods for managing and planning changes as well as techniques for assessing their impact are essential within the vulnerability management process. Reasoning systems capable of capitalizing security knowledge can provide new horizons for dealing with dynamic environments, not only from an operational viewpoint but from a security perspective too.

3.8 Synthesis

The management of computer systems and networks is becoming more and more complex over time. Conventional approaches do not scale well with respect to this evolving landscape, thus leading to new management problems. The autonomic paradigm aims at releasing administrators from low-level details by considering self-management approaches that work on a high-level, goal-oriented basis. This scenario allows to specify how things work while functional details are solved by the underlying autonomic system. Autonomics provides a scalable new perspective for dealing with the management of growing heterogeneous networks. However, both administrators and self-governed entities may introduce vulnerable states when perform their operations. Therefore, vulnerability management constitutes an essential activity in order to ensure safe configurations within autonomic environments. In this chapter, we have performed a deep review of current vulnerability management mechanisms. We have classified the vulnerability assessment process in three dimensions: discovery, description and detection (D^3 approach) ; which serves as an organizational framework for discussing different existing assessment methods and techniques. We have also discussed mechanisms that contribute to the remediation of vulnerabilities, thus closing the vulnerability management cycle. We state that **real autonomic solutions can only be achieved if mechanisms for dealing with vulnerabilities are fully integrated into the management plane of autonomic networks and systems**. However, there still exist several challenging problems that must be addressed in order to achieve this goal. The aim of this thesis is to contribute in that direction.

Considering these research challenges, we aim at dealing with the integration of vulnerability and remediation descriptions into the management plane of autonomic environments. These descriptions, understood as policies, enable autonomic entities to manage their own security exposure. In the following chapters, we present our research work which includes autonomic assessment strategies for device-based vulnerabilities and extensions in several dimensions, namely, distributed vulnerabilities (spatial), past hidden vulnerable states (temporal), and mobile security assessment (technological). In addition, our general approach also considers remediation activities able to bring networks and systems into secure states. In that context, we have conducted research in remediation approaches for device-based and distributed vulnerabilities. In particular, collaborative approaches, which constitute another axis identified as a research challenge, are also considered within distributed scenarios. In addition, our approaches are supported by mathematical models over which other scientific contributions can be built upon. Finally, the contributions presented in this thesis have been evaluated through an extensive set of experiments which are widely discussed all along this document.

Part II

An autonomic platform for managing configuration vulnerabilities

Chapter 4

Autonomic vulnerability awareness

Contents

4.1 Introduction 45
4.2 Integration of OVAL vulnerability descriptions 46
4.3 OVAL-aware self-configuration 48
4.3.1 Overall architecture 48
4.3.2 OVAL to Cfengine translation formalization 49
4.4 Experimental results 52
4.4.1 IOS coverage and execution time 52
4.4.2 Size of generated Cfengine policies for Cisco IOS 54
4.5 Synthesis 55

4.1 Introduction

Autonomic networks and systems are responsible for their own management. However, changes that are performed by administrators and self-governed entities may generate vulnerabilities and increase the exposure to security attacks. Therefore, vulnerability management is a crucial activity for ensuring safe configurations and reducing the exposure of such autonomic systems. While strong standardization efforts have been done for describing vulnerabilities, in particular with the OVAL language, there is no full 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 this chapter, we present an autonomic approach for supporting vulnerability awareness in self-governing networks and systems using the OVAL language and the Cfengine tool [38]. Cfengine is an autonomic maintenance system that provides support for automating the management of large-scale environments based on high-level policies. Even though we are focused on the Cfengine tool, currently used in millions of managed devices, our general approach could be applied to other policy-based configuration tools such as Puppet [124] or Chef [39]. Our strategy consists in integrating OVAL vulnerability descriptions into the management plane, in order to enable autonomic systems to detect and prevent configuration vulnerabilities. For that purpose, the OVAL vulnerability descriptions are dynamically translated into policy rules directly interpretable by Cfengine. Therefore, Cfengine agents become able to autonomously assess their own security exposure. The remainder of this chapter is organized as follows. Section 4.2 presents a review of OVAL vulnerability descriptions and discusses its integration into autonomic environments. Section 4.3 describes the underlying architecture for increasing vulnerability awareness within autonomic networks and systems, and depicts the formalism for supporting the translation of vulnerability descriptions into Cfengine policy rules. Section 4.4 shows an extensive set of experiments performed over the Cisco IOS platform and the obtained results. Section 4.5 concludes this chapter presenting conclusions and further work.

4.2 Integration of OVAL vulnerability descriptions

Nowadays, the OVAL language is mostly used by vendors and leading security organizations in order to publish security related information that warns 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. Previously in Definition 2, we have provided the conceptual meaning of a vulnerability in computing systems. From a technical perspective, a vulnerability can be also considered 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. In that context, the manner in which OVAL represents a vulnerability can be directly mapped to the usual way a vulnerability is understood, as shown in Figure 4.1.

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, let us consider an hypothetical situation, illustrated in Figure 4.2, where a vulnerability for the Cisco IOS has just been disclosed. For this vulnerability to be present, two conditions must hold simultaneously: (I) the version of the platform must be *12.4* and (II) the

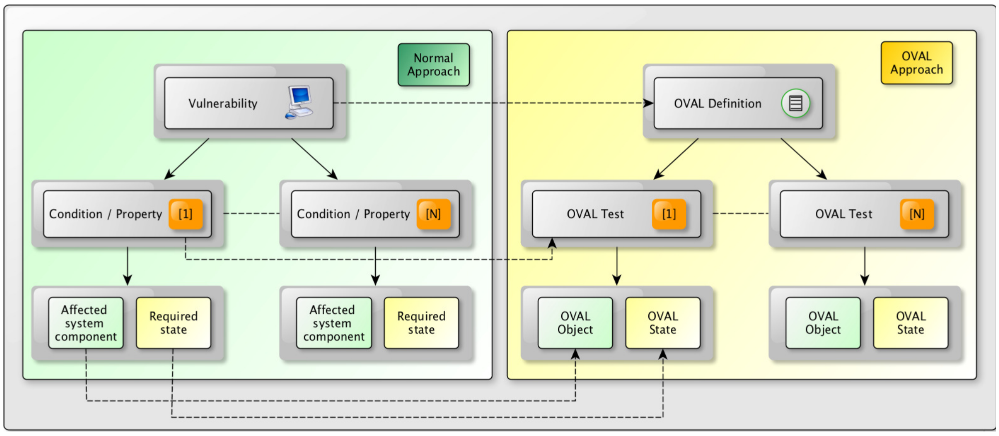


FIGURE 4.1 – Vulnerability conception mapping

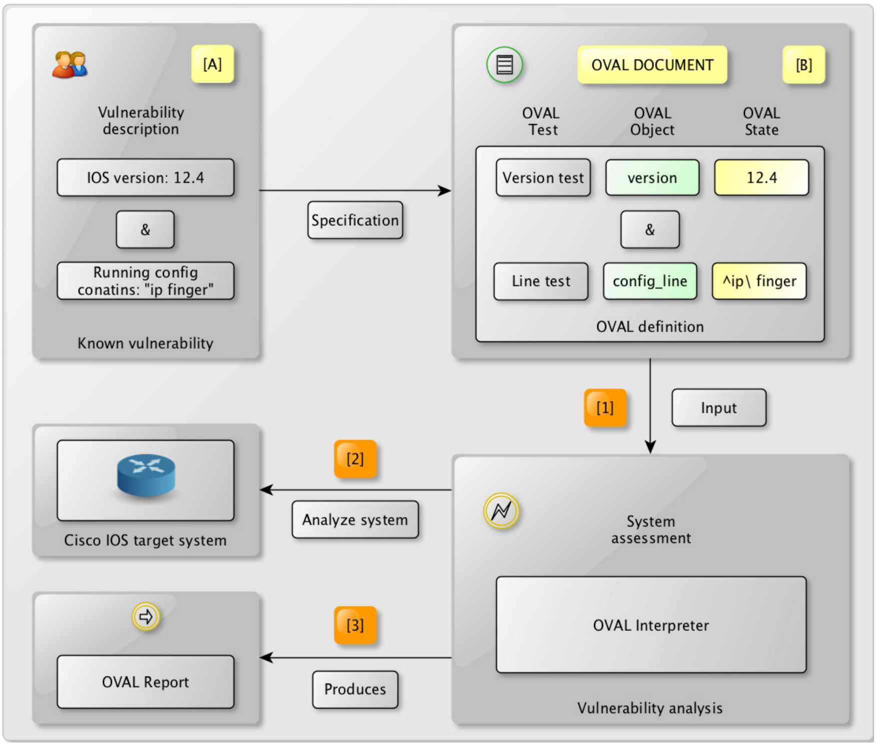

[A]
Vulnerability description
IOS version: 12.4
&
Running config conatins: "ip finger"
Known vulnerability
Specification
OVAL DOCUMENT
[B]
OVAL Test
OVAL Object
OVAL State
Version test
version
12.4
&
Line test
config_line
^ip\ finger
OVAL definition
[1]
Input
[2]
Analyze system
Cisco IOS target system
System assessment
OVAL Interpreter
Vulnerability analysis
[3]
Produces
OVAL Report

FIGURE 4.2 – OVAL example over Cisco IOS

service *ip finger* must be enabled (thus N would be 2 in Figure 4.1). Such vulnerability can be expressed within an *OVAL document* by defining an *OVAL definition* that arranges two *OVAL tests* as a logical conjunction. 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 object for the tests to be true and hence, defining the truth or falsehood of the *OVAL definition*. In this particular example, it is expected to observe the value 12.4 as the version of the system, and the running configuration file must have a line starting with the directive *ip finger*. If these two properties are observed, then the vulnerability is present on the target system.

Once an OVAL document has been specified, the regular approach to perform its assessment over a target system can be resumed in three main steps. As shown in Figure 4.2, step 1 consists in interpreting the document that specifies the objects and tests to be evaluated. At step 2, the target system is analyzed looking for present vulnerabilities. As a remainder, the OVAL analysis, as previously described in Section 3.5.1, involves two parts, namely, the collection of required OVAL objects to be analyzed, and the comparison of collected OVAL items against the specified OVAL states. Finally, a report is produced at step 3 indicating the results of the assessment process. Our approach aims at making from this approach an autonomous process. To do so, we automate the exploitation of OVAL descriptions warning about current threats and system vulnerabilities, and translate them into Cfengine policy rules. In this manner, current and future security advisories can be integrated and assessed by autonomous Cfengine agents deployed across the network. The global architecture of the proposed approach is detailed in the next section.

4.3 OVAL-aware self-configuration

In autonomics, 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. 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.

4.3.1 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 4.3, and taking into account the existing and future security related knowledge specified in the OVAL language.

The proposed architecture illustrated in Figure 4.3 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. To do so, a translation module is placed between the OVAL repository and the Cfengine server. This module, explained in detail in the next section, consumes available OVAL vulnerability descriptions from the repository and produces Cfengine policy rules that allow Cfengine agents to be aware of these security weaknesses. The Cfengine architecture is based on a client-server model. The server keeps these

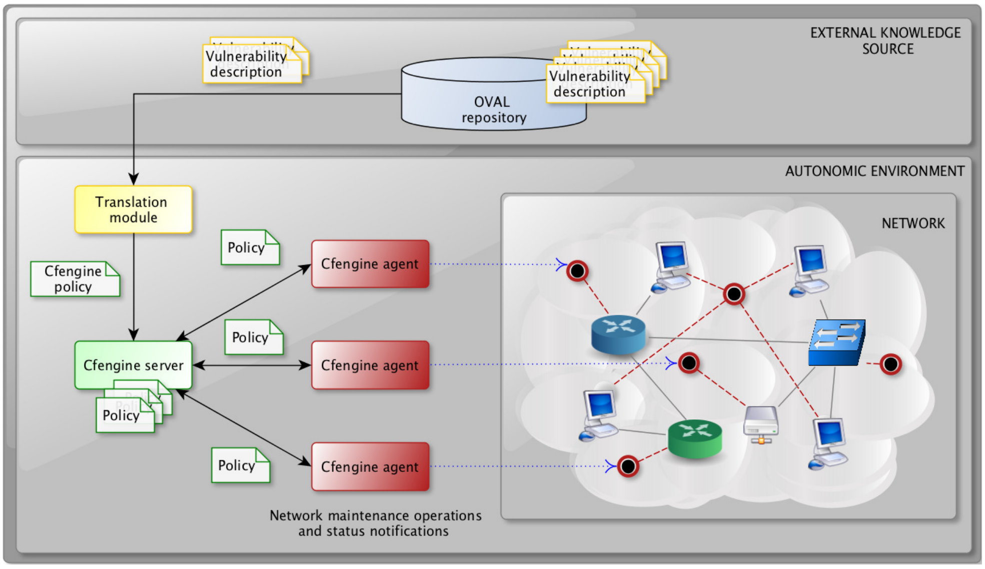

EXTERNAL KNOWLEDGE SOURCE
Vulnerability description
OVAL repository
Vulnerability description
AUTONOMIC ENVIRONMENT
Translation module
Cfengine policy
Policy
Cfengine agent
Policy
Cfengine server
Cfengine agent
Policy
Policy
Cfengine agent
NETWORK
Network maintenance operations and status notifications

FIGURE 4.3 – High-level architecture

generated policies on its own and autonomous agents will pull these new policies from the server when convenient. In this manner, generated policies are deployed by the Cfengine server into its several Cfengine agents (points in the cloud). These autonomous agents 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. In the next section, we detail and formalize the translation process performed to convert OVAL-based advisories into Cfengine policy rules.

4.3.2 OVAL to Cfengine translation formalization

The translation module, identified in Figure 4.3, has as a main goal the generation of Cfengine rules that accurately represent the OVAL advisories present in the OVAL repository. However, we consider that 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 of 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.

```
<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>
```

Figure 4.4 – Basic predicate within OVAL

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 [116] over standard networks. The example presented in Figure 4.4 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*. There, the configuration file is represented by an OVAL object of type *file_ object*. In it, two attributes specify the target object, the path and its filename. Finally, the predicate is completed with the specification of the OVAL state, expressing that the user id of this file must be equal to *root*.

As mentioned before, the main core activity within the OVAL language is about predicating over the underlying system, i.e. identify the system items (individuals of our discourse universe) and 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 4.5 presents a summarized mapping between OVAL main constructors, their corresponding components within a first-order logic and the respective Cfengine building blocks.

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

FIGURE 4.5 – First-order logic, OVAL and Cfengine mapping

The following list describes the main building blocks of the OVAL language and the corresponding Cfengine constructs used to represent them.

- **OVAL documents**. Within an OVAL document, several definitions can be found within the XML tag <*definitions*>. This set of definitions can be arranged on a high level Cfengine configuration file including each one of the referred definitions. The inclusion can be made using the *import* directive within the Cfengine language. Hence, for each OVAL definition present within the OVAL document, an import sentence will be present on the main Cfengine configuration file.

- **OVAL definitions**. Each OVAL definition can be defined in a separate Cfengine configuration file that will be imported by the main Cfengine configuration file. Imported files are typically located on the Cfengine *input* folder, hence one file per OVAL definition will be located there. Within each one of these files, the involved tests are referred, and their results are computed to represent the same logical structure specified on the original OVAL definition.
  Because OVAL tests involve objects from the system that will be assessed against specific states, each Cfengine file corresponding to one OVAL definition file will involve mechanisms to collect the required objects. Collection can be made using Cfengine *prepared modules* that are launched before any policy control sequence is executed. Once the required information is available, the specified tests can be executed. Following the Cfengine philosophy, each test is mapped to a Cfengine method that is called when needed from any Cfengine input.

- **OVAL tests**. Cfengine methods are usually located in the *modules* folder, hence, for each OVAL test, a file will exist on this place that represents it. The parameters of this Cfengine method represent the data needed to effectively compute the test result ; this is, the item information to be tested and depending on the taken approach, the expected states.
- **OVAL objects**. As previously explained, system objects to be evaluated can be gathered using Cfengine *prepared modules*. Calls to these modules are placed on the corresponding Cfengine input file. Collected information will be later used by the involved tests.
- **OVAL states**. States are used by tests in order to verify whether gathered objects match specific properties. States definitions can be specified on the Cfengine method file using Cfengine control variables, or they can be specified on the Cfengine input file and provided as parameters to the corresponding method.

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 one of the previous classes. On the other hand, a test result will be also represented by

```
Data: an OVAL document
Result: Cfengine policy rules
$mainFile \leftarrow$ create <Cfengine main configuration file>;
foreach $def \in$ OVAL definitions do
    $defFile \leftarrow$ create <Cfengine input file> for $def$;
    add import sentence at $mainFile$;
    foreach test referred by $def$ do
        on $defFile$ do {
            $obj \leftarrow$ OVAL object referred by $test$;
            add prepared module call at ”control section” for gathering $obj$;
            foreach ste referred by test do
                add ste control variables at ”control section”;
            end
            add test call method at ”methods section” specifying objects and states;
        }
        $methodFile \leftarrow$ create <Cfengine method file> for $test$;
        on $methodFile$ do {
            add method name and parameters at ”control section”;
            add obj variable at ”control section”;
            foreach atomic predicate on the specified obj do
                add result as a Cfengine class at ”classes section”;
            end
            combine classes for defining final method result class;
        }
    end
    add logical test criteria at ”alerts section” on $defFile$;
end
```

Algorithm 4.1: Translation algorithm

a Cfengine class, hence, an OVAL definition result will be based on the Cfengine classes defined for each one of the referred tests. Considering the mapping introduced before, we present in Algorithm 4.1 the proposed approach for translating OVAL documents into Cfengine policies. This algorithm has been fully developed and integrated into an implementation prototype called Ovalyzer which is described in Chapter 9.

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. Further details on Cfengine grammar as well as technical Cfengine related information can be found at [30].

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. The autonomicity provided by Cfengine combined with the automated translation of security advisories accommodates an autonomic support for increasing the vulnerability awareness of these systems. In this approach however, only alerts are considered, and they should be taken into account by a human administrator. The integration of remediation actions has not be done yet though it could be performed as discussed in Chapter 8. In the next section we present a set of experiments performed over the Cisco IOS platform and the obtained results.

4.4 Experimental results

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 have developed a Java-based implementation prototype called Ovalyzer in charge of translating OVAL security advisories into Cfengine policy rules. Ovalyzer's technical specification is described in Chapter 9, which includes an example of an OVAL vulnerability description and its corresponding Cfengine generated code. 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* [58] running the operating system IOS version 12.4(4)T1. The *Expect* [66] tool provided by NIST [108] has been also used for automating the communication between Cfengine and Cisco devices. We present in this section the results obtained from the performed experiments.

4.4.1 IOS coverage and execution time

The official OVAL repository has 134 vulnerability definitions for the IOS platform, by the moment of writing this document. These definitions are based on three types of OVAL tests, namely, *line_test* (*L*), *version55_test* (*V55*) and *version_test* (*V*). Ovalyzer, as explained later in this manuscript, has been designed using a plugin-based architecture. In that context, one

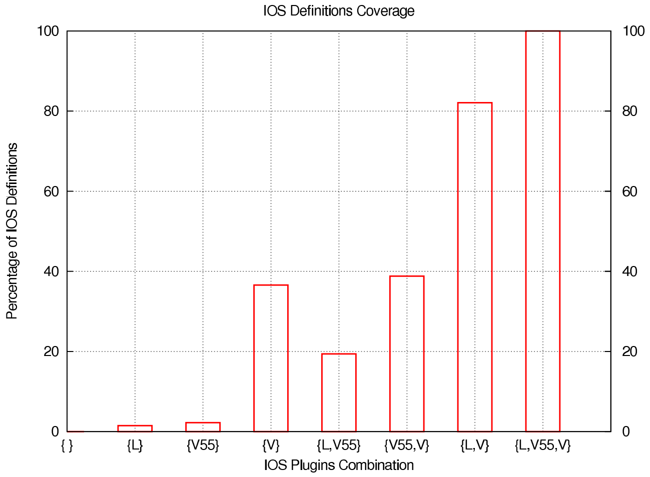


FIGURE 4.6 – IOS plugins coverage

plugin per type of OVAL 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 provide a coverage of 100% of OVAL definitions for the IOS platform. Figure 4.6 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 types 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.

Since such translation shall be made in an automatic manner, several tests for evaluating Ovalyzer's performance have been done. We have particularly focused on the time required for generating Cfengine policy files over different sets of IOS vulnerability definitions. Figure 4.7 shows the observed timing values while varying the amount of translated OVAL definitions.

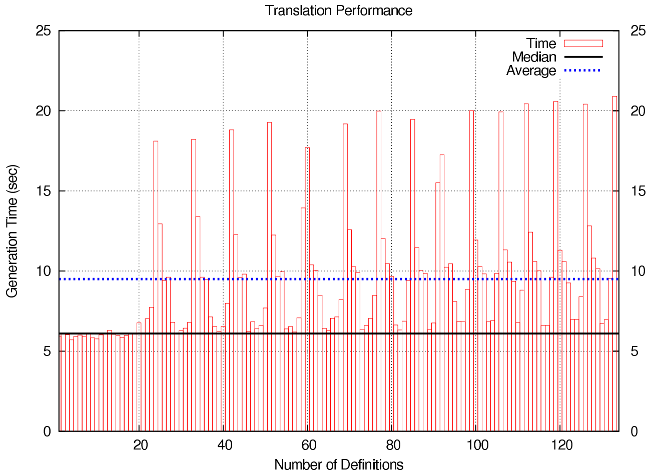


FIGURE 4.7 – IOS translation performance

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 close to 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.

4.4.2 Size of generated Cfengine policies for Cisco IOS

As happens with the generation time, the number and size of generated files constitute 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 4.8 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 on 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

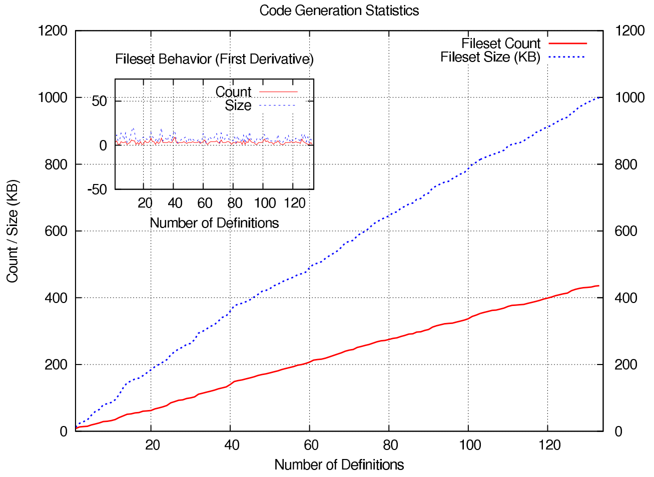


Figure 4.8 – IOS generation statistics

platform, the generation behavior (depicted by the first derivative of the curves), is stable as illustrated in the inner graph in Figure 4.8. Based on the experiments and results presented here, an interesting future line of work would be to define a mathematical and well founded mechanism for determining the size of a Cfengine policy fileset for any given set of OVAL definitions.

4.5 Synthesis

In this chapter, we have presented our approach for 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. The implementation prototype used within the experiments is fully described in Chapter 9.

Supporting vulnerability awareness constitutes the first step towards secure self-managed infrastructures capable of detecting and remediating potential security breaches. Indeed, real autonomy can only be possible if networks and systems are able to manage the required activities for understanding the surrounding environment, ensuring safe configurations and performing corrective actions when vulnerable states are found. The latter constitutes a hot and challenging aspect which is deeply discussed in Chapter 8. However, assessing vulnerabilities over individual network elements may not provide a global view of how vulnerable a network can be. Sometimes, two or more devices may seem to be secure, but when combined, a security weakness might arise. We call to this concept a distributed vulnerability, which is the heart of the next chapter.

Chapter 5

Extension to distributed vulnerabilities

Contents

5.1 Introduction **57**
5.2 Specification of distributed vulnerabilities **58**
  5.2.1 Motivation, definition, and mathematical modeling 58
  5.2.2 DOVAL, a distributed vulnerability description language 60
5.3 Assessing distributed vulnerabilities **62**
  5.3.1 Extended architecture overview 63
  5.3.2 Assessment strategies 64
5.4 Performance evaluation **66**
5.5 Synthesis **69**

5.1 Introduction

As systems and technologies evolve, new space for vulnerabilities comes into scene. Autonomic networks must integrate support mechanisms for preventing vulnerabilities. Nowadays, networks are analyzed in order to detect vulnerabilities that may allow a malicious user to perform an attack. However, traditional mechanisms perform a global analysis by investigating each network element individually. Indeed, our approach for increasing the vulnerability awareness of self-governed environments previously presented in Chapter 4, considers independent analysis on each network member. Even though such approaches can detect sets of vulnerabilities that may allow an attacker to perform a multi-step attack, they do not provide the capability of detecting vulnerabilities that simultaneously involve two or more devices under specific conditions. The underlying problem relies in that each network device can individually present a secure state, but when combined across the network, a global vulnerable state may be produced. In order to cope with this problem, the following issues must be attended. Formal mechanisms for describing distributed vulnerabilities are required. Moreover, using standard means for achieving such objective can promote the exchange of security knowledge among practitioners and organizations. Such descriptions in turn must be integrated into the management plane of autonomic networks and systems. Mechanisms for interpreting and assessing theses security advisories must be provided. In addition, optimized algorithms and strategies for collaboratively assessing the network should be developed.

In this chapter, we present a framework for describing and assessing distributed vulnerabilities in autonomic networks and systems. This approach complements the host-based perspective presented in Chapter 4. We put forward mechanisms for specifying distributed vulnerable states that are taken into account by the proposed framework thus increasing the vulnerability awareness of such self-governed environments. We also perform an analytical and technical evaluation of the proposed approach in order to analyze and show the feasibility of our solution. The remainder of this chapter is organized as follows. Section 5.2 presents the proposed approach for specifying distributed vulnerabilities in autonomic networks and systems. In this section, we formalize the definition of distributed vulnerabilities and put forward a machine-readable language called DOVAL for representing them. The architecture of the proposed framework as well as algorithms and strategies for the assessment of distributed vulnerabilities are described in Section 5.3. Section 5.4 provides an evaluation of our solution through a comprehensive set of experiments and the obtained results. Section 5.5 concludes this chapter presenting conclusions and further work.

5.2 Specification of distributed vulnerabilities

During our research on vulnerability assessment, a recurrent question used to come up about the actual definition of a distributed vulnerability. Our findings indicate that its meaning is currently being misused, and a revision on the conception of distributed vulnerabilities is required. In this section, we motivate and mathematically formalize the concept of distributed vulnerabilities. In addition, we present DOVAL, a language for describing such vulnerabilities in a machine-readable manner.

5.2.1 Motivation, definition, and mathematical modeling

The concept of a distributed vulnerability may usually be understood as a set of individual vulnerabilities distributed in the network that potentially might allow a multi-step attack. A multi-step attack actually describes a sequence of steps performed by an attacker in order to achieve a desired goal. Within this sequence, the attacker may exploit known vulnerabilities at each step, in the same or a different network device, in order to scale and move forward to the final objective. We think that even though this vision of individually assessing vulnerabilities provides a useful perspective to the topic in question, it does not offer a complete outlook of the problem. Let us motivate this issue by considering the following example. The scenario described in [166] and depicted in Figure 5.1 involves two related hosts, a SIP (Session Initiation Protocol) server and a DNS (Domain Name System) server. Each one has specific properties, however, they constitute together a potential exploitable network vulnerability.

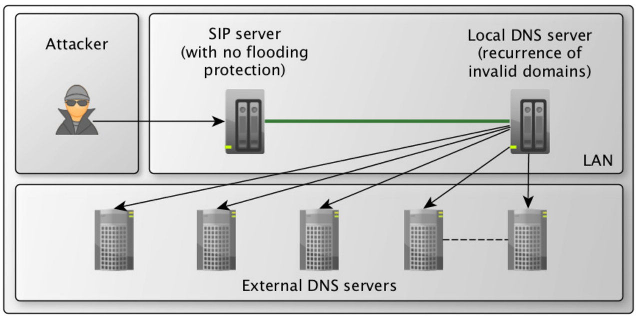


FIGURE 5.1 – Distributed vulnerability scenario

In this example, a denial of service (DoS) attack over the SIP server can be performed by flooding it with unresolvable domain names that must be solved by a local DNS server. The local DNS server in turn, is configured for requesting the resolution of unknown domains to external servers, increasing the number of waiting requests and therefore the response time for each SIP request. Under these configuration states, flooding a SIP server with such type of messages will prevent it to respond to legitimate requests. It is important to highlight that both servers and the relationship between them are required conditions for the distributed vulnerability to be present. If the DNS server is not present or if it is not compliant with the required specific conditions, the SIP server would immediately respond to a SIP client that its SIP request has failed. Even in such a situation, thousands of SIP requests may collapse the SIP server anyway, though it is a slightly different scenario that could be specified using standard OVAL definitions. On the other hand, if there is no SIP server, it is quite clear that the distributed vulnerability has no place in this environment. Considering the insight provided by the previous example, we characterize the concept of a distributed vulnerability by proposing the following definition.

Definition 4 (Distributed vulnerability)**.** *A distributed vulnerability is a security weakness that arises when specific conditions over two or more network devices occur simultaneously, providing a potential exploitable entry point for security attacks in the network under analysis.*

As a remark, it is important to distinguish the main difference between considering a set of individual vulnerabilities over different network devices and the proposed definition. The main difference is that in a distributed vulnerability, the required conditions to be observed over one network device may not constitute a complete vulnerability description.

In order to formalize this conceptualization, we now present some required definitions to mathematically specify a distributed vulnerability.

- ◇ $H = \{h_1, h_2, \dots\}$ denotes the set of devices or systems in the network (e.g. hosts, routers).
- ◇ $P = \{p_1, p_2, \dots\}$ denotes the set of device properties in the form of unary predicates $p_i(h), h \in H$. Such predicates are used for both specifying required properties to be observed for a vulnerability to be present as well as properties the device already possesses.
- ◇ $S = \{s_1, s_2, \dots\}$ denotes the set of device states where a state s_i describes a set of properties required to be observed over a network device (called *role*) as well as for describing existing specific network devices states. The set S is inductively defined as follows:
  - i. if $p_i \in P$, then $p_i \in S \qquad (i \in N)$
  - ii. if $\alpha, \beta \in S$, then $(\alpha \diamond \beta) \in S \qquad \diamond \in \{\wedge, \vee\}$
  - iii. if $\alpha \in S$, then $(\neg\alpha) \in S$.
- ◇ $R = \{r_1, r_2, \dots\}$ denotes the set of relationships between network devices such as reachability and service provisioning. The relationships are modeled in the form of n-ary predicates $r_i(h_i, ..., h_j)$ and they are used for representing existing relationships between network devices as well as those relationships required to be observed for a distributed vulnerability to be present.

Based on the previous definitions, a distributed vulnerability DV is defined as the compliant projection of the pattern (P^H, P^R) over the network (H, R), illustrated in Figure 5.2, where the constructs P^H and P^R are defined as follows:

- ◇ $P^H = \{s_1, ..., s_k\}$ denotes the set of machine states or roles ($s_j \in S$) required to be observed on specific network devices.
- ◇ $P^R = \{r_1, ..., r_v\}$ denotes the set of relationships ($r_i \in R$) between those devices matching the required roles.

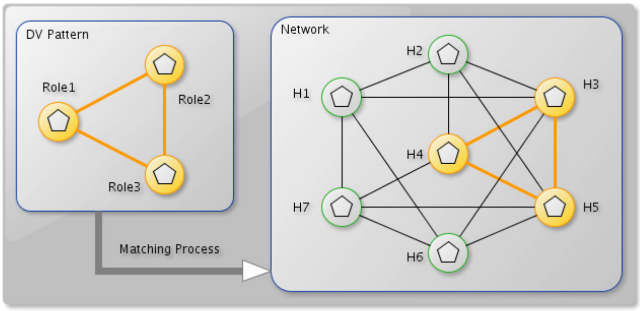


FIGURE 5.2 – Distributed vulnerability matching process

Under a logical perspective, a compliant projection of the pattern (P^H, P^R) over the network (H, R) makes the following sentence to be true:

$$\exists(h_1, ..., h_n)\ (s_1(h_1) \wedge ... \wedge s_n(h_n) \wedge r_1(h_i, ..., h_j) \wedge ... \wedge r_v(h_k, ..., h_l)) \tag{5.1}$$

We specify the previous sentence in short by considering the predicate $DV(H, R)$ that expresses the evaluation of a distributed vulnerability DV based on the pattern (P^H, P^R) over a generic network (H, R). It is important to notice that the model allows to specify a device-based vulnerability by just defining $P^H = \{s_1\}$ and $P^R = \{\}$, or a sequence of vulnerabilities spread across the network by considering $P^H = \{s_1, ..., s_k\}$ and $P^R = \{\}$.

In order to incorporate the ability to detect distributed vulnerabilities into autonomic environments, a means for expressing these patterns is required. In that context, we have developed DOVAL, an XML-based language built on top of OVAL for describing distributed vulnerabilities. In the next section, we describe the main aspects of the DOVAL language as well as its applicability over the motivational example previously presented.

5.2.2 DOVAL, a distributed vulnerability description language

We have designed the DOVAL language (Distributed OVAL) on top of OVAL as a means for describing distributed vulnerabilities in a machine-readable manner. The OVAL perspective can be seen as a host-based approach, capable of describing specific host states independently. DOVAL leverages the OVAL language by providing mechanisms for describing vulnerabilities that involve two or more network devices at the same time. While the universe of discourse in the OVAL language is composed of digital components (e.g. processes, files), DOVAL extends it by considering network devices as well. In addition, we extend the semantics of the language by allowing to express relationships between objects in order to describe conditions involving several devices simultaneously. In this manner, we can for instance specify that a network is vulnerable, if a given traffic between specific processes and devices is allowed.

Within the DOVAL language, the required conditions over each involved device are described using standard OVAL definitions. An OVAL definition is intended to describe a specific machine state using a logical combination of tests that must be performed over a host. If such logical combination is observed, then the specified state is present on that host (e.g. vulnerability, specific configuration). As explained in the previous chapter, this combination can be understood from a logical perspective as a first order formula where each test corresponds to an atomic unary predicate over that system. DOVAL extends this concept by enabling the expression of predicates that involve more than one device, thus allowing the specification of required relationships over

The DOVAL language		
DOVAL constructs	**Description**	**First-order logic**
DOVAL document	Set of distributed vulnerabilities	Arrangement of compound logical formulas
DOVAL definition	Distributed vulnerability	Compound logical formula
DOVAL test	Assessment of a condition between several devices	Container of an atomic n-ary predicate
DOVAL object	Devices with specific conditions using OVAL definitions	Family of individuals in the discourse universe
DOVAL state	Network condition between devices characterized by DOVAL objects	Mathematical relationship specification

Figure 5.3 – DOVAL logical description

the network. Figure 5.3 depicts the main DOVAL constructs and provides a description of the intended purpose of each main building block.

A DOVAL document is intended to meet the specification of several components required to describe a set of distributed vulnerabilities. Each distributed vulnerability is specified by a DOVAL definition, which provides the capability of expressing a logical formula that involves several DOVAL tests. Each DOVAL test in turn constitutes the container of an n-ary predicate over a set of network devices $(h_i, ..., h_j)$ and it is in charge of putting together the required devices and the states expected to be observed between them.

We consider a required network device as a device that meets certain conditions (s_j) and that is required to be present for the distributed vulnerability DV to be true in the network under analysis (H, R). Required network devices (P^H) are described by means of DOVAL objects. In order to express DOVAL objects, we take advantage of the very final objective of the OVAL language, this is to say, to express specific machine states. Thus, a DOVAL object is actually a set of references to OVAL definitions, where each one describes a required specific machine state (p_i) on that device. Finally, the expected relationships over the network (P^R) are expressed using DOVAL states. A DOVAL state (r_i) specifies properties between devices and the roles each device has within such relationship, and can be seen as an actual predicate itself.

There exist several situations where neither individual host assessments nor inference chains over individual exploitable vulnerabilities can expose potential security threats in a wide network. Mechanisms for globally specifying and evaluating network distributed vulnerabilities are essential. In order to illustrate the utilization of the DOVAL language for describing this kind of situations, let us retake the example presented in Figure 5.1. The specification of such scenario using the DOVAL language is presented in Listing 5.1.

A DOVAL definition with id *"doval:fr.inria.doval:def:1"* identifies which DOVAL tests must be performed in order to detect the distributed vulnerability that such definition is intended to describe. The DOVAL test with id *"doval:fr.inria.doval:tst:4141"* identifies the required devices as DOVAL objects and the relationships between them by referencing DOVAL states. The required devices, namely a SIP server with no flooding protection (s_1) and a local DNS server with external unknown domain resolution (s_2), are specified using two DOVAL objects, *"doval:fr.inria.doval:dev:222"* and *"doval:fr.inria.doval:dev:256"* respectively. Each DOVAL object enforces the required properties over the device it describes by considering a set of OVAL definitions, one for each needed condition. Each DOVAL state specifies the characteristics of the relationship expected to be observed between both devices.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<doval_document>
  <doval_definitions>
    <doval_definition id="doval:fr.inria.doval:def:1" class="distributed_vulnerability">
      <metadata>
        <title>SIP DoS attack using DNS flooding</title>
        <description>...</description>
        <doval_repository>...</doval_repository>
      </metadata>
      <criteria>
        <criterion comment="..." test_ref="doval:fr.inria.doval:tst:4141"/>
      </criteria>
    </doval_definition>
  </doval_definitions>

  <tests>
    <doval_test id="doval:fr.inria.doval:tst:4141" comment="DOVAL test combining two specific tests,
         reachability and configuration." check_existence="at_least_one_exists" check="at least one">
      <object device_ref="doval:fr.inria.doval:dev:222"/>
      <object device_ref="doval:fr.inria.doval:dev:256"/>
      <state state_ref="doval:fr.inria.doval:ste:4444"/>
      <state state_ref="doval:fr.inria.doval:ste:7777"/>
    </doval_test>
  </tests>

  <objects> <!-- devices -->
    <device_object id="doval:fr.inria.doval:dev:222"/>
      <prop ovaldef_ref="oval:org.mitre.oval:def:1000"/> <!--SIP server running-->
      <prop ovaldef_ref="oval:org.mitre.oval:def:1001"/> <!--Port 5060 open-->
      <prop ovaldef_ref="oval:org.mitre.oval:def:1002"/> <!--Security module not installed-->
    </device_object>

    <device_object id="doval:fr.inria.doval:dev:256"/>
      <prop ovaldef_ref="oval:org.mitre.oval:def:2000"/> <!--DNS server running-->
      <prop ovaldef_ref="oval:org.mitre.oval:def:2001"/> <!--Port 53 open-->
      <prop ovaldef_ref="oval:org.mitre.oval:def:2002"/> <!--Unknown domains solved by external DNS
           servers-->
    </device_object>
  </objects>

  <states> <!-- relationships -->
      <link_state id="doval:fr.inria.doval:ste:4444"/> <!--Traffic capability-->
        <protocol operation="equals"> udp </protocol>
        <src_port device_ref="doval:fr.inria.doval:dev:222"
         operation="pattern match"> .* </src_port>
        <dst_port device_ref="doval:fr.inria.doval:dev:256"
         operation="equals"> 53 </dst_port>
      </link_state>

      <service_state id="doval:fr.inria.doval:ste:7777"/> <!--Service config-->
        <name operation="equals"> dns </name>
        <consumer device_ref="doval:fr.inria.doval:dev:222"/>
        <provider device_ref="doval:fr.inria.doval:dev:256"/>
      </service_state>
  </states>
</doval_definitions>
```

Listing 5.1 – DOVAL document

Within the current example, two relationships are required: (1) DNS traffic is allowed between the SIP server and the DNS server (r_1), specified by a *DOVAL link_state* with id *"doval:fr.inria.doval:ste:4444"*, and (2) the SIP server has configured the DNS server as its DNS service provider (r_2), specified by a *DOVAL service_ state* with id *"doval:fr.inria.doval:ste:7777"*. In order to assess such a specification, a deployable distributed infrastructure capable of enforcing its evaluation and notification is required. This aspect is presented in the next section.

5.3 Assessing distributed vulnerabilities

The main objective of DOVAL is targeted on describing conditions involving several network devices that if observed, the underlying network presents a vulnerable state that can be exploited by an attacker. In order to detect such scenarios, DOVAL descriptions must be interpreted and evaluated on the target network. We propose a framework based on Cfengine, a widely deployed configuration and administration system, capable of enforcing security policies for discovering distributed vulnerabilities across the network.

5.3.1 Extended architecture overview

Due to the size and dynamics of current networks, the assessment and detection of vulnerable distributed states is not a trivial task. We partition the problem into several steps, namely, (1) generation of a minimized loop-free topology of the underlying network, (2) collection of hosts and network information, and (3) assessment of DOVAL specifications over the gathered data. Figure 5.4 illustrates the steps and the architecture of the proposed approach.

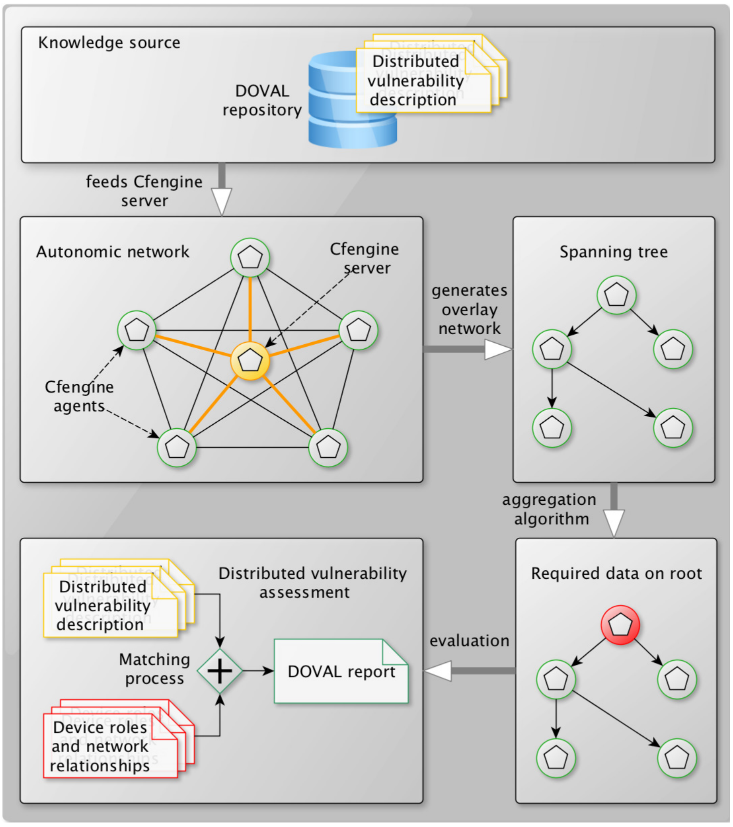

Knowledge source
DOVAL repository
Distributed vulnerability description
feeds Cfengine server
Autonomic network
Cfengine server
Cfengine agents
generates overlay network
Spanning tree
aggregation algorithm
Distributed vulnerability assessment
Distributed vulnerability description
Matching process
DOVAL report
Device roles and network relationships
evaluation
Required data on root

FIGURE 5.4 – Overall architecture

Within this architecture, distributed vulnerabilities are specified using the DOVAL language and stored in a database. A Cfengine server is fed with such knowledge and translated as Cfengine policy rules, in the same way we have done for host-based vulnerabilities as presented in Chapter 4. Our approach considers a deployment of Cfengine agents across the network, where each agent is in charge of controlling one network device. In order to evaluate the existence of a distributed vulnerability, a spanning tree is built on top of the target network to minimize paths and avoid network loops. The DOVAL specification is then transmitted across the tree and the required information is gathered by performing an aggregation algorithm over the nodes. Each Cfengine agent assesses the device it controls in order to discover which roles such device can play within the distributed vulnerability specification. This information is returned back until all the information is stored at the root node of the spanning tree. Finally, the specification of the distributed vulnerability is projected over the information gathered from the network, and a DOVAL report is generated informing about distributed vulnerable states across the network.

5.3.2 Assessment strategies

Several strategies for assessing the required properties on each network device can be used. Considering the number of potential combinations, an optimized algorithm for evaluating the network is required. Within our approach, we consider the aggregation algorithm proposed in [30] for building a tree-based overlay network with the information of each device in the network under analysis. We now proceed to explain our strategy in a constructive manner considering two situations. The first situation presents a simplified scenario where full connectivity between each pair of nodes is in place and no further relationships between nodes are required. The second situation puts forward a more realistic scenario that extends the first one by considering network constraints such as reachability restrictions or service provisioning requirements.

When starting a DOVAL definition assessment, the set of required devices for the distributed vulnerability to be present can be seen as an empty tuple $t = (\underline{\ \ }_1, \underline{\ \ }_2, ..., \underline{\ \ }_k)$. Each blank field represents the placeholder for a required role characterized by $P^H = \{s_1, ..., s_k\}$ according to the definitions given in Section 5.2.1. During the assessment across the spanning tree, tuple t will be collaboratively fulfilled as each Cfengine agent will indicate which of these fields the device it controls can play. At the end, several combinations can occur thus the final computation will be performed over a set $T = \{t_1, ..., t_w\}$. Let DV be a distributed vulnerability where $t = \{\overset{s1}{\underline{\ \ }}, \overset{s2}{\underline{\ \ }}, \overset{s3}{\underline{\ \ }}\}$ specifies the set of roles required to be observed in the network. Figure 5.5 depicts the steps taken during the algorithm execution for discovering the roles each network device is able to play. At the initial step, a spanning tree covering every node in the network is considered. The

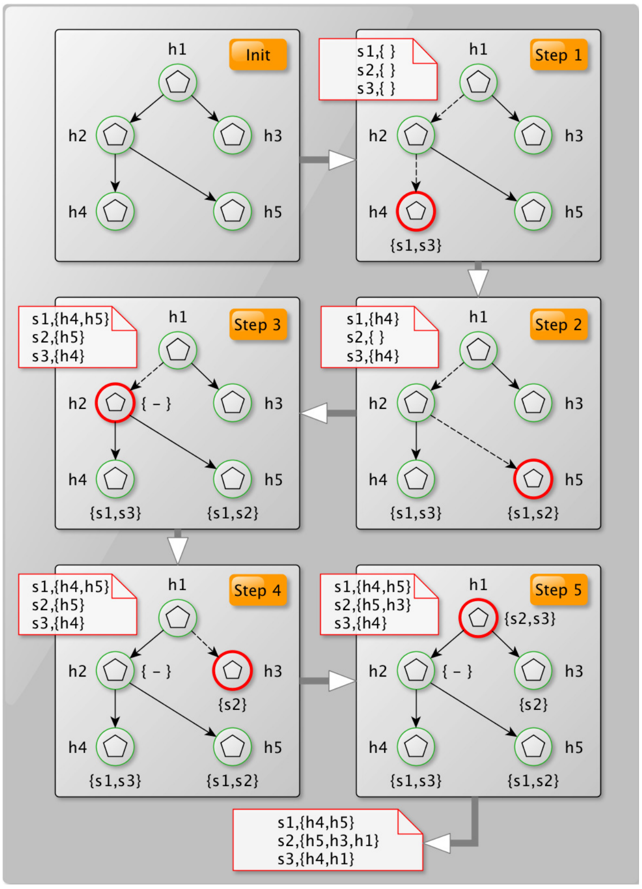


FIGURE 5.5 – Aggregation algorithm execution for role discovery

tree is explored using a post-order traversal. At Step 1, the node $h4$ is assessed reporting that it matches roles $s1$ and $s3$. Then, the node $h2$ fulfills its temporal role list and continues with the node $h5$ as shown in Step 2. At Step 3, $h2$ is assessed in order to detect which roles it can play and the role list for the sub-tree with root $h2$ is returned to the caller $h1$. The node $h3$ is assessed at Step 4 identifying its ability to perform the role $s2$. At Step 5, the temporal role list describes the roles that the nodes $h2$, $h3$, $h4$ and $h5$ are able to play. The assessment of the node $h1$ will complete the role list indicating that role $s1$ can be performed by the nodes $h4$ and $h5$, whereas the role $s2$ can be performed by $h1$, $h3$ and $h5$, and $s3$ by $h1$ and $h4$.

Usually, networks present complex topologies imposing reachability restrictions. We consider such constraints in the second situation, where distributed vulnerabilities require the existence of relationships among the involved network devices such as reachability or service provisioning. Under a logical perspective, a distributed vulnerability involving n roles may require at most the evaluation of n-ary predicates $r_i(h_1, ..., h_n)$ where each device h_i covers one specific role s_i though it could cover more than one at the same time. Such scenarios require not only to discover the roles that each network device is able to play but also to assess the relationships between them. In order to deal with these more realistic situations, we extend the previous algorithm by performing a neighborhood discovery at each node and assessing the relationships each node supports with its neighbors. The idea behind this approach is to *reduce* the participation of nodes that can endorse an expected role but they do not satisfy the required relationships with other nodes. Considering the example given in Section 5.2.1, the fact of finding a SIP server h_1 and a DNS server h_2 under the required conditions does not mean that the distributed vulnerability is necessarily present. For instance, if DNS traffic is not allowed between them, the predicate $r_1(h_1, h_2)$ does not hold; then such pair of network devices does not constitute a candidate combination for the distributed vulnerability. In order to analyze the surrounding environment at each node, we take advantage of Cfengine's functionalities for performing a neighborhood discovery [30]. The process described by Algorithm 5.1 depicts the steps performed by each Cfengine agent while assessing the roles the device can play as well as the required relationships it supports with its neighbors.

```
Input: DOVAL document
Output: List of tuples specifying devices and relationships between them
foreach role $s \in$ DOVAL objects do
    if currentNode n is compliant with role s then
        $N \leftarrow SelectPartitionNeighbors$;
        foreach predicate $r \in$ DOVAL states do
            if r requires role s then
                $w \leftarrow getPredicateArity(r)$;
                foreach sequence $\{h_2, ...h_w\} \subseteq N$ do
                    $result \leftarrow$ eval state $r(n, h_2, ..., h_w)$;
                    if result is true then
                        add tuple to output list $[r, (n, s), (h_2, s_2), ..., (h_w, s_w)]$;
                    end
                end
            end
        end
    end
end
```

Algorithm 5.1: Node roles and relationships assessment

Within our approach, a DOVAL object describes the required properties (s) that must be observed over a network device (h) that is part of a distributed vulnerability. Each device compliant with such description is able to perform the object's intended role. The proposed algorithm is executed on each agent and analyzes every role (line 1) the device it controls can perform. For each supported role (line 2), its neighbors are stored in the set N (line 3) and the required relationships are assessed against them (line 4). Such relationships are described by means of DOVAL states and they can be seen as predicates involving w network devices. Only those relationships involving the current role s are assessed (line 5). Depending on the arity of the predicate (line 6), subsets of $w-1$ network devices are built (line 7) and the relationship among them is assessed (line 8). If such relationship holds, a tuple is added to the output list (line 10) indicating that the node n is able to perform the role s, and that under that role, the relationship r (described by a DOVAL state) holds when considering devices $(h_2, ..., h_w)$ performing potential roles $(s_2, ..., s_w)$ respectively. The actual verification of devices $(h_2, ..., h_w)$ performing roles $(s_2, ..., s_w)$ is done at the end, when each node in the spanning tree has been evaluated and the root node has all the required information for the assessment of the distributed vulnerability in the network. In order to evaluate the performance of the proposed approach, we have performed different experiments which are presented in the next section.

5.4 Performance evaluation

In this section we present an analytical evaluation and a technical discussion of the proposed approach. We put forward the specification of two metrics in order to analyze the performance during the evaluation of a generic distributed vulnerability, namely, (1) the number of messages sent across the network, and (2) the total time required for the assessment. We also show the scalability of the proposed approach by modeling different scenarios based on the instantiation of various parameters within the specified metrics.

Let N be the network under analysis with a set of devices $H = \{h_1, ..., h_n\}$ and relationships $R = \{(h_i, ..., h_j)*\}$. Let DV be a distributed vulnerability that requires a set of k roles defined by $P^H = \{s_1, ..., s_k\}$ and v relationships defined by the set $P^R = \{r_1, ..., r_v\}$. We consider the following assumptions during the assessment of DV over N:

- ◇ a binary spanning tree is built on top of N,
- ◇ each device h has in average q neighbors,
- ◇ each predicate r_i has in average arity b (we define the variable $a = b-1$ in order to simplify the equations),
- ◇ the probability for a device to play a role s_j (event A) is given by $P(A) = \alpha$,
- ◇ the probability for a role s_j to occur on a predicate r_i (event B) is given by $P(B) = \beta$,
- ◇ the evaluation of any role s_j over any device h takes in average γ units of time,
- ◇ the evaluation of any predicate r_i takes in average δ units of time.

Number of messages. In order to analyze the traffic generated across the network, we consider the number of messages sent during the evaluation of a generic distributed vulnerability DV. We define the metric M that estimates the amount of messages transmitted as follows:

$$M = n * (1 + k * P(A) * P(B|A) * v * M_{pred})$$

$$\text{where} \quad M_{pred} = P_R(q+1, a) * a = (q+1)^a * a$$

When traversing the spanning tree (n nodes), each node receives one message in order to start its own evaluation. At each node, k roles must be evaluated. Because not every device will play every role, we model this uncertainty as an event A that occur with probability[1] $P(A) = \alpha$. Given the event A for a role s_j, we model the probability of such role to be involved on each predicate r_i by considering the conditional probability $P(B|A) = \beta$ and the multiplier v. The final factor M_{pred} represents an upper bound of the number of messages sent when evaluating one single predicate among the node under analysis and its neighbors. $P_R(q+1, a)$ denotes the number of a-permutations with repetition of a set of q neighbors plus the node itself. For each possible permutation that may fulfill the arguments (roles) required by the predicate under analysis, a messages must be sent. M_{pred} represents an upper bound because we consider that multiple roles can be covered by one single device, which means that for those combinations assigning for instance the same device to each role, only one instead of a messages will be sent.

Assessment time. We analyze here two distributed approaches that have a direct impact on the total assessment time for a distributed vulnerability DV. The first one consists on sequentially assessing the network, analyzing one node at a time. In this case, we define the total assessment time as:

$$T_S = n * k * (\gamma + P(A) * P(B|A) * v * T_{pred})$$

$$\text{where} \quad T_{pred} = \delta * P_R(q+1, a) = \delta * (q+1)^a$$

For each node in the network, k roles are evaluated where each one takes γ units of time. The probability for the node to perform a role is given by $P(A)$ whereas $P(B|A)$ defines the probability for that role to be involved on each one of the v predicates. T_{pred} involves the same permutations among neighbors as M_{pred} does, but for each possible sequence T_{pred} considers the parameter δ that models the average time for evaluating a predicate r_i.

An alternative to the sequential modality consists on using a parallel computing approach. Such approach enables the evaluation of every node simultaneously thus reducing the sequential assessment time. Under this perspective, the total assessment time becomes:

$$T_P = max_{node}(k * (\gamma + P(A) * P(B|A) * v * T_{pred}))$$

$$\text{and the worst case is:} \quad T_{Pw} = k * (\gamma + v * T_{pred})$$

T_{Pw} describes the extreme case where a node is able to perform every required role that in turn is involved in each predicate. Such case maximizes the amount of time among the times required by the nodes in the spanning tree.

The proposed metrics have several parameters that affect the final result such as the number of nodes (n) or the probability for an arbitrary node to play a given role ($P(A)$). Typical scenarios usually involve predicates conceived as peer to peer properties (binary predicates, $a = 1$) as well as a restricted number of roles (small k). Each required predicate in turn usually involves all roles, thus simplifying conditional probabilities ($P(B|A) = 1$). Under this perspective, we have performed several experiments using the example illustrated at Section 5.2.2 as a case study in order to analyze the behavior of our approach. In this scenario we have $k = 2$, $v = 2$, $a = 1$ and $q = n - 1$ depicting a full mesh network. We have also simplified the units of time considering $\gamma = \delta = 1$. Networks nature can vary depending on the context and the purpose they have been built for. Therefore, we use the parameters n and $P(A)$ for analyzing such situations. The experiment depicted in Figure 5.6 considers a uniform distribution for role assignment ($P(A) = \frac{1}{n}$), meaning that only one of n nodes may play a given role s_j. These curves

---

[1] This is a simplified measure of the likelihood for the event A to occur since depending on the network nature, the probability of choosing, for instance, a SIP server within a standard company network will be presumably lower than picking up a standard workstation running any version of Windows 7.

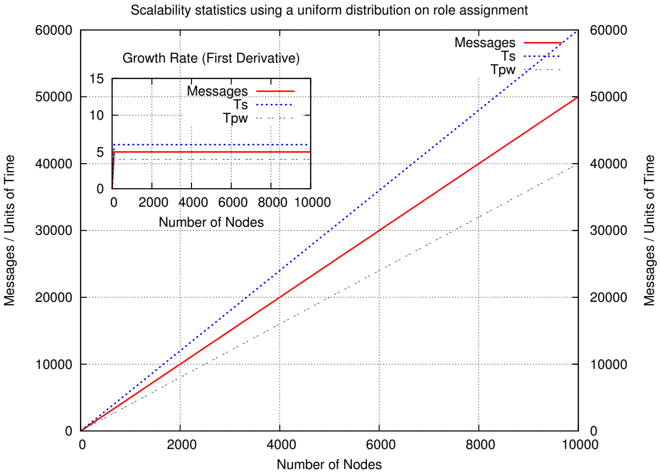


FIGURE 5.6 – Statistics with uniform distribution on role assignment

depict how much grows the number of messages (solid line) as well as the assessment time under a sequential computing approach (dotted line) and a parallel computing approach (dashed line) when the number of nodes in the network becomes bigger. We can observe a proportional growth on every metric (M, T_S, T_{Pw}) under a constant rate. This is easily verifiable by looking at the inner graph of Figure 5.6 that shows the first derivatives of each curve.

Considering that only one device can perform a required role may apply just on special cases so we have analyzed the behavior of our approach when increasing the number of potential devices able to perform the required roles. This is achieved by modifying the probability for a device to play a role ($P(A)$) as shown in Figure 5.7. We observe that even though the assessment time using a sequential approach is not linear, its growth rate illustrated in the inner graph of Figure 5.7 remains linear. We get a maximum assessment time when $P(A) = 1$ (dotted line) because every node is able to play every role. When $P(A) = \frac{1}{2}$ (solid line), a half of the network can perform each role and the assessment time is lower though the curve demonstrates similar behavior as with $P(A) = 1$. When a parallel approach is used (dashed line), a constant behavior is observed in the worst case ($P(A) = 1$) making it clear that such approach is better than the sequential one.

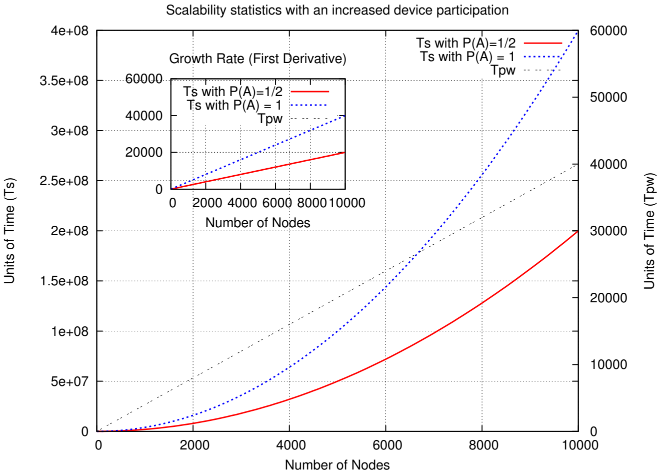


FIGURE 5.7 – Statistics with an increased device participation

From a more technical point of view, the proposed approach requires mechanisms for interpreting and assessing descriptions of distributed vulnerabilities. As indicated in the previous chapter, we have developed Ovalyzer, a tool capable of translating OVAL advisories to Cfengine policy rules, detailed in Chapter 9. Within our approach, the Cfengine system is the component to be embedded within target autonomic networks and systems. It is in charge of enforcing security policies including the assessment of distributed vulnerabilities. To achieve this, policy rules directly interpretable by Cfengine are needed. In light of this, an extension to Ovalyzer (not developed yet) would be able to translate DOVAL specifications into Cfengine policy rules that represent them. Such an extension, that we could call Dovalyzer, could use the functionalities provided by Ovalyzer for translating OVAL definitions and complement the generated code in order to cover distributed assessment tasks specified by DOVAL definitions. The translator must take as input the content of DOVAL documents and produce Cfengine code, structured as Cfengine policy files, that can be later consumed by a Cfengine running instance.

5.5 Synthesis

In this chapter, we have proposed an extension to distributed vulnerabilities which enables autonomic networks and systems to assess such security advisories. This perspective complements the approach presented in Chapter 4 for host-based vulnerabilities by considering a holistic overview of the network. We have mathematically defined the concept of a distributed vulnerability and we have developed DOVAL, an OVAL-based language for expressing these formal constructions. A case study has been presented showing DOVAL's main constructs. As in OVAL, DOVAL descriptions can constitute useful security repositories that in turn can be exploited by self-managed environments in order to ensure safe configurations. We have proposed a framework based on the Cfengine system for assessing distributed vulnerabilities in autonomic networks as well as optimized algorithms and collaborative strategies for performing such evaluations. We have analyzed the proposed algorithms by mathematically defining computation costs that show the feasibility of the model through a comprehensive set of experiments. We also have presented a technical discussion about implementation perspectives, where Cfengine policies could be fed by a translator capable of producing Cfengine policy rules that represent DOVAL security advisories.

The applicability of the DOVAL language goes beyond the expression of distributed vulnerabilities. Indeed, it can be used for describing general distributed scenarios involving different entities with specific configurations. Semantics can be used for both identifying wrong configurations as well as best practices. In the context of the Univerself project and the UMF framework, we have proposed a configuration assessment services called CAS that involves both the OVAL and DOVAL languages. A more detailed explanation of the CAS service can be found in Annex A. As stated before, vulnerability assessment constitutes the first step within the vulnerability management process. Therefore, remediation activities must be considered as well, for host-based and distributed vulnerabilities. This topic is discussed in detail in Chapter 8. When considering vulnerability assessment mechanisms, the extension performed from a host-based perspective, presented in Chapter 4, to a distributed one, could be considered as a spatial dimensional extension. However, this is not the only dimension that may contribute to provide better mechanisms to autonomic security. Indeed, the ability to understand the state of autonomic elements in the past and potential hidden vulnerabilities, might provide robust support for enhancing security mechanisms in the present. This perspective is presented in the next chapter.

Chapter 6

Support for past hidden vulnerable states

Contents

6.1 Introduction **71**

6.2 Modeling past unknown security exposures **72**

6.2.1 Understanding past unknown security exposures 73

6.2.2 Specifying past unknown security exposures 74

6.3 Detecting past hidden vulnerable states **75**

6.3.1 Extended architecture overview 75

6.3.2 Assessment strategy 76

6.4 Experimental results **77**

6.5 Synthesis **79**

6.1 Introduction

In the previous chapter we have discussed about distributed vulnerabilities, which extends the concept of regular host-based vulnerabilities towards complete networks as a whole. This spatial extension constitutes an important improvement in the way autonomic networks can be analyzed and protected. However, time is also an important dimension that must be considered when managing vulnerabilities. By the time a piece of software is being constructed, several errors may be unintentionally introduced providing room for security vulnerabilities. These vulnerabilities can survive within active systems for a long period of time without being detected. During this period, attackers may perform well-planned and clean attacks (e.g., stealing information) without being noticed by security entities (e.g., system administrators, intrusion detection systems, self-protection modules). Indeed, unaware entities do not even think about such a potential breach due to the very nature of being under-informed, constituting blind and easy targets for attackers. As a matter of fact, such attacks might never be detected. Changes in the system or even its normal activity can alter or erase the remaining evidence. This issue makes it clear why it is so important to increase the awareness of our systems as soon as security information becomes available. In that context, our approach aims at taking advantage of current security information for analyzing system security in the past. If unknown security exposures are detected, response actions can be performed in the present for bringing system states to secure levels.

The ability to identify past unknown system exposures due to hidden vulnerabilities allows forensic activities to be performed in order to detect malicious activity [54], [2]. For instance, a bank that has detected a potential intrusion compromising data about credit cards would be able to take actions before consequences become out of control. It would be easier for the bank to block compromised credit cards and make new ones than waiting for notifications of anomalous activity from its clients. Other scenarios apply as well in general computing systems. Usually, intruders leave entry points (backdoors) to come back to compromised hosts. Later, if a past exposure that may allow an attacker to install backdoors is detected, forensic analysis could be performed in order to reveal such security issue. The consequences of this investigation not only allow to know if the system has been actually compromised but also to correct a security breach in the present that could be used for future attacks. The acknowledge of current vulnerabilities is a critical factor for reducing the exposure of computing systems. Under this perspective, there is a race for getting security information early. Both security entities and attackers can benefit from their speed, being for self-defence or for breaking security barriers. Open and mature standards such as the OVAL language are cornerstones at this point as they provide a strong support for openly exchanging security information within the community.

Historical vulnerability information as well as security metrics and trends are highly useful as proposed in [137, 1]. However, these contributions do not take advantage of new security information that could have been useful in the past for detecting security exposures. As explained in the next section, the exploit for a vulnerability can be released long time before the vulnerability is publicly known. Hence, affected systems can be exposed during this period without actually knowing it. To the best of our knowledge, no previous contributions have taken advantage of current security advisories for assessing past hidden vulnerable states. This would enable current systems to increase their own exposure awareness and to take actions in consequence if unknown past exposures are detected.

In this chapter we propose a novel approach for increasing the overall security of computing systems by identifying past hidden vulnerable states. This information can be used for detecting potential unknown attacks in the past, identifying compromised assets and bringing systems up to secure states. Taking advantage of the OVAL language for representing system states and analyzing vulnerabilities, our strategy consists in autonomously generating images of network devices that represent their current state, building a history of their evolution, and capitalizing new security advisories for automatically assessing past system states in order to detect potential security breaches. The remainder of this chapter is organized as follows. Section 6.2 presents our approach for mathematically modeling and detecting unknown past security exposures. Section 6.3 details the proposed framework describing its architecture and the strategies for performing assessment activities. Section 6.4 provides an evaluation of our solution through a comprehensive set of experiments. Section 6.5 concludes this chapter and discusses further work.

6.2 Modeling past unknown security exposures

Since the construction of a software program, errors are unintentionally introduced producing security vulnerabilities. At a certain time, system administrators, security modules or self-protection components, system security entities from now, may be unaware of these issues permitting attackers to take advantage of them and to breach the security measures without being noticed. However, the awareness of such potential attacks later in time provides the ability to inspect possible security breaches and to take actions to ensure the security of the system. In this section we present a mathematical model that defines and supports the process for detecting past unknown security exposures.

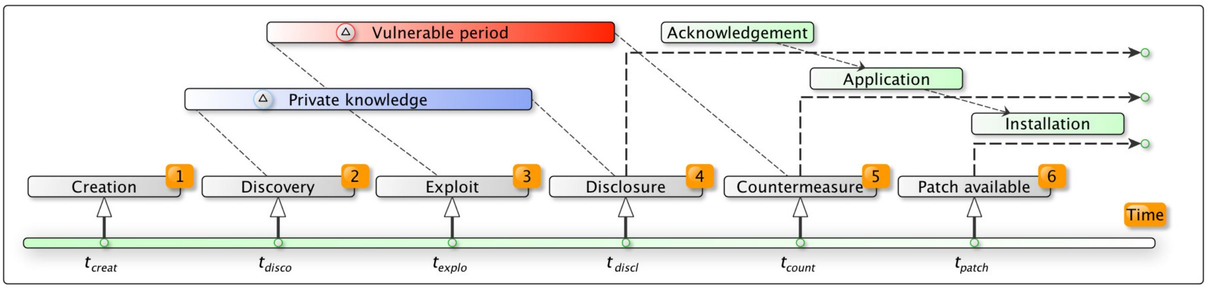


Figure 6.1 – Vulnerability lifecycle events

6.2.1 Understanding past unknown security exposures

Security exposures can inadvertently occur during long periods of time. Unaware of this fact, systems become victims of unnoticed security incidents that may compromise their information and functionalities in the long term. Once a vulnerability has been introduced in a software program, a sequence of events constitutes what is called the vulnerability lifecycle [72] described in Figure 6.1. Event 1 indicates the vulnerability creation time denoted by t_{creat}. Event 2 records the time where the vulnerability is discovered, specified by t_{disco}. Event 3 denoted by t_{explo} indicates the first time an exploit becomes available. Its disclosure time specified by t_{discl} occurs in event 4 where the vulnerability information becomes freely available to the public. Since the vulnerability discovery time until this point, the information about it is considered as private knowledge denoted by $\triangle_{private} = t_{discl} - t_{disco}$. Beyond this point, system security entities may acknowledge its existence. Event 5 indicates the time where a vulnerability countermeasure becomes available, denoted by t_{count}. Vulnerable states may be partially mitigated by performing certain actions that do not correct the problem but avoid it to be exploited. Since an exploit exists until this point, systems are vulnerable to security attacks. This period is denoted by $\triangle_{vulnerable} = t_{count} - t_{explo}$. Event 6 specified by t_{patch} indicates the time where a patch becomes available to the public. System security entities can install this patch in order to eradicate the vulnerability.

It is important to notice that such a sequence of events describes the general lifecycle of a vulnerability but it can actually differ from one system to another. For instance, system security entities may acknowledge the existence of a vulnerability later in time after its disclosure. The same happens with the application of countermeasures and patch installations. In some cases, such actions may never occur. Because of this, we have modified the original event sequence proposed in [72]. Within our approach we consider the existence of potential countermeasures at time t_{count}. We understand that its application as well as the installation of a patch are inherent to the environment where the vulnerability lives in and not as a lifecycle component. In addition, these events usually occur in the order they are listed. However, they also depend on the context and may vary among different vulnerabilities. For instance, the exploit might be published after the vulnerability has been disclosed, or a countermeasure may not exist until the patch is available thus t_{count} will coincide with t_{patch}.

Based on the previous definitions, we specify a past unknown security exposure by considering the following definition.

Definition 5 (Past unknown security exposure)**.** *A past unknown security exposure is an exploitable vulnerable state that exposes a system to security threats during a certain period of time ($\triangle_{vulnerable}$) in which neither the system nor its security entities were aware of such security weakness.*

In order to unveil such security exposures, an infrastructure capable of managing snapshots of the system across time would be able to analyze past system states by taking advantage of current security information. In this manner, exposure time gaps of the system can be detected in order to perform further analysis such as forensic activities over valuable assets. In the next section we present our model for supporting the proposed infrastructure.

6.2.2 Specifying past unknown security exposures

In order to define a mathematical specification of unknown past security exposures, we first introduce a set of core definitions that constitute the main building blocks of the model. We present here three definition groups: (1) domains, (2) predicates and (3) functions, that are used for defining how a system is evaluated in order to detect past exposures. The universe of discourse is constituted by the following domains:

- $P = \{p_1, p_2, \dots\}$ denotes the set of device properties in the form of unary predicates $p_i(h)$ where h is the device under analysis. Such predicates are used for specifying both required properties to be observed for a vulnerability to be present as well as properties the device already possesses.
- $S = \{s_1, s_2, \dots\}$ denotes the set of device states where a state s_i is used for describing in a compact manner the set of properties required to be observed over the device as well as for describing existing specific device states. The set S is inductively defined as follows:
    i. if $p_i \in P$, then $p_i \in S \qquad (i \in N)$
    ii. if $\alpha, \beta \in S$, then $(\alpha \diamond \beta) \in S \qquad \diamond \in \{\wedge, \vee\}$
    iii. if $\alpha \in S$, then $(\neg\alpha) \in S$.
- $R = \{r_1, r_2, \dots\}$ denotes the sequence of system revisions (snapshots) through time, where a revision r_i precedes a revision r_j only if $i < j$. R is called the revision repository.
- $V = \{v_1, v_2, \dots\}$ denotes the set of known vulnerability definitions and it is also called the knowledge source.

The predicates applied over individuals of our discourse universe are defined as follows:

- All the defined domains act as membership predicates, e.g., $R(r)$ is *true* if and only if r is a system revision.
- $isVulnerable : S \times V \to Boolean$ denotes a predicate that takes a system state $s \in S$ and a vulnerability definition $v \in V$ as input and returns *true* if and only if the vulnerability v is present in the system state s.
- $isNew : V \to Boolean$ denotes a predicate that takes a vulnerability definition $v \in V$ as input and returns *true* if and only if the vulnerability v is new within the current knowledge source.

The functions used in the approach are the following:

- $revision : N \to R$ denotes a function that takes a revision number $n \in N$ as input and returns the associated system revision $r \in R$.
- $number : R \to N$ denotes a function that takes a system revision $r \in R$ as input and returns the associated number $n \in N$.
- $state : R \to S$ denotes a function that takes a system revision $r \in R$ as input and returns its associated state $s \in S$.
- $time_R : R \to N$ denotes a function that takes a system revision $r \in R$ as input and returns the time elapsed since the revision was created.
- $time_V : V \to N$ denotes a function that takes a vulnerability definition $v \in V$ as input and returns t_{explo} if known, otherwise t_{discl} is returned.

Based on the previous core definitions, we define $E(R, V)$ shown in Equation 6.1 as a predicate that based on a revision history R and a vulnerability knowledge source V, indicates if the system under analysis has been unknowingly exposed in the past.

$$\begin{aligned} E(R,V) = \exists(r)\ \exists(v) \quad (R(r) \wedge V(v) \wedge isNew(v) \\ \wedge time_V(v) \leq time_R(r) \\ \wedge isVulnerable(state(r), v)) \end{aligned} \tag{6.1}$$

Equation 6.1 mathematically states the main concept of our approach. If a new vulnerability is available, and exists at least one system revision made after the exploit was created or the vulnerability was disclosed, and such revision is found to be vulnerable, then the system has been unknowingly exposed in the past, even if the vulnerability is not observable in the present configuration. This warning can be used for performing a deeper analysis within the vulnerable period in order to detect malicious activity or compromised data. In the next section we present a framework capable of capitalizing security advisories and analyzing historical revisions in order to detect and warn about unknown past exposures.

6.3 Detecting past hidden vulnerable states

Detecting past unknown security exposures relies on the ability to see beyond the current status of a given system. In order to achieve this goal, we propose a distributed autonomous framework capable of organizing historical information about computing systems and analyzing them when new security information is available. In this section we present the overall architecture and explain our strategy for detecting past security exposures by taking advantage of new advisories over past system states.

6.3.1 Extended architecture overview

In order to build a framework capable of identifying security exposures in the past, we consider two independent cyclical processes. One process for imaging systems in an autonomous manner and the second one for actually detecting past security exposures. Figure 6.2 illustrates the proposed architecture identifying the main components as well as the communication processes between them. The sequence denoted by Steps I, II and III constitutes the image generation process. At Step I, the exposure analyzer provides directives for data collection that will be used for building system images. These directives are specified by means of OVAL documents that are automatically translated to Cfengine policy rules. The ability to express OVAL objects without actually expecting any particular state allows us to use OVAL documents as the inventory of required objects to be collected. At Step II the generated Cfengine policy rules are transmitted to the autonomic agents distributed in the network. These agents are in charge of controlling network devices and they will perform data collection activities in order to build their system images. Finally, these images are automatically stored in the revision repository at Step III. The image generation process constitutes an autonomic activity and it is performed independently from the past exposure detection process. The latter is composed of two steps. First at Step 1, the exposure analyzer monitors the knowledge source on a regular basis checking for new vulnerability definitions. When new definitions become available, it analyzes system images stored in the revision repository at Step 2 in order to detect past unknown security exposures.

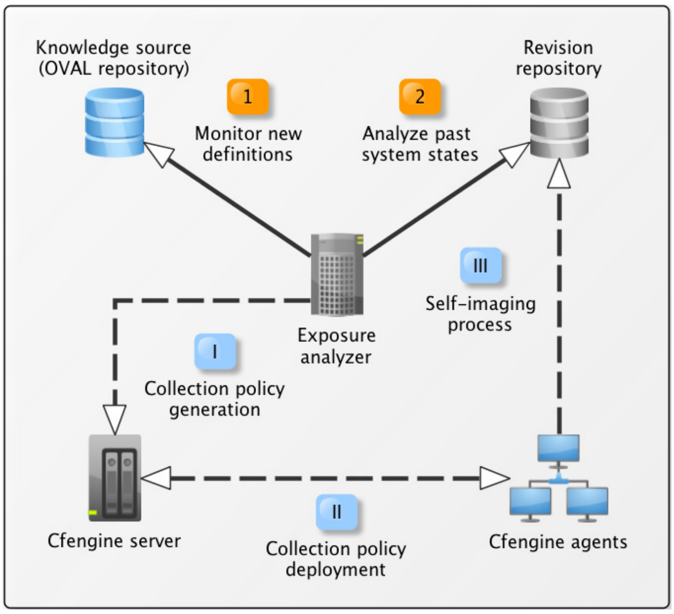


FIGURE 6.2 – High-level imaging and exposure detection process

This framework in fact can be easily coupled to autonomic frameworks that perform assessment activities such as the one presented in Chapter 4. In this manner, a combined solution of past and present vulnerability assessment could highly increase the security of autonomic environments. Moreover, forensic activities could be partially automated by collecting forensic evidence using machine-readable procedures that may warn administrators about past exposures and current threats [26]. In addition, the proposed architecture allows to outsource assessment activities. By analyzing system images, target devices may provide the required data while the exposure analyzer may perform a security evaluation of it. As we mentioned before there is a period of time, noted $\triangle_{private}$, where the information about a vulnerability is not publicly known. The capability of outsourcing security assessment activities may allow organizations to perform analysis with their own information without actually violating their disclosure restrictions. At the same time, clients can be warned about security exposures and be advised about actions to take without knowing internal mechanisms for detecting such threats. In the next section we illustrate the proposed strategy for performing assessment activities in the past and detecting unknown security exposures.

6.3.2 Assessment strategy

Given a new vulnerability definition, the objective of our strategy is to identify affected system states across time after its exploit was publicly available. This process provides a period of time where the system was potentially exposed to the security threat represented by the specified vulnerability. The steps followed by the proposed strategy are depicted in Algorithm 6.1. First, the exposure time is set, depending on the available information, between the exploit or vulnerability disclosure time and the current time (lines 1-2). Then, a sequence of available system revisions during this period is gathered, ordered by time starting with the newest revision first (line 3). For each revision within the sequence (line 4), the system state is analyzed cheking if the specified vulnerability is present or not (line 5). If the system state is found to be vulnerable, the algorithm takes the longest period of potential exposure. If the vulnerable system state is not the newest one (lines 6-7), the exposure end time is set to the next revision time found

```
   Input: Vulnerability v
   Output: ExposureTime e
1  $e.startTime \leftarrow time_V(v)$;
2  $e.endTime \leftarrow now()$;
3  $revs \leftarrow getRevisionsFromTo(e.endTime, e.startTime)$;
4  foreach $Revision\ r \in revs$ do
5      if $isVulnerable(state(r), v)$ then
6          $nextRev \leftarrow revision(number(r) + 1)$;
7          if $nextRev \in revs$ then
8              $e.endTime \leftarrow time_R(nextRev)$;
9          end
10         $return\ e$;
11     end
12 end
13 $e.endTime \leftarrow time_V(v) - 1$;
14 $return\ e$;
```

Algorithm 6.1: Exposure assessment algorithm

not vulnerable (line 8). If the vulnerable system state is the newest one, the exposure end time corresponds to the current time. Afterwards, the exposure time is returned (line 10). If none of the revisions is found to be vulnerable, the time period is set to a negative value and the exposure time is returned (lines 13-14). This strategy has been integrated within our implementation prototype which is described in detail in Chapter 9. Considering the XML-based nature of the OVAL language, we have taken advantage of the SVN (Apache Subversion) versioning system to efficiently store past system states [146]. In the next section, we present a case study where a comprehensive set of experiments has been made for determining the feasibility and limits of our solution.

6.4 Experimental results

Past system security exposures can provide unnoticed pathways for performing attacks on current system states. In this section we present a case study based on the IOS operating system for Cisco devices. We illustrate the application of the proposed approach for increasing the present security by analyzing past hidden vulnerable states in an emulated environment. We use the GNS3 emulator [75] over a regular laptop (2 Ghz Intel Core i7 with 8GB RAM) and present the results obtained through an extensive set of experiments.

The complexity involved within each vulnerability description usually depends on its very own nature, meaning that some vulnerability definitions may require a small set of tests to be evaluated while others may need a higher amount of systems checks. In order to analyze the performance of the proposed implementation prototype, we have taken the whole set of IOS vulnerability descriptions available within the official OVAL repository [117] and we have cyclically tested them as shown in Figure 6.3. The testing strategy consists on increasing the number of OVAL definitions by one each time and measuring the accumulated assessment time over one system image. We observe a low time cost at the beginning of the analysis due to definitions involving a small amount of tests. The inclusion of definitions with more tests clearly increases the assessment time though its behavior depicted in the inner graph describes a general stable execution, taking about 5 seconds of assessment time for 138 IOS vulnerability descriptions over one system image.

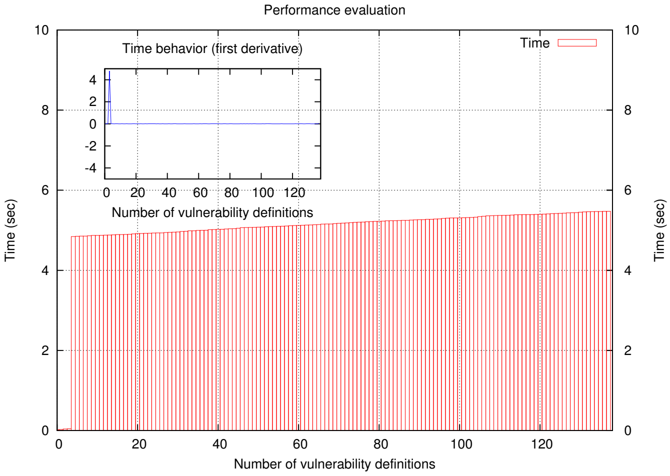


FIGURE 6.3 – Vulnerability definitions assessment time

In order to avoid the nature-based size discrepancy among different vulnerability descriptions, we have increased the granularity of our experiments by independently analyzing the involved OVAL tests. Assessing the whole set of OVAL definitions for the IOS platform requires the evaluation of approximately 2400 OVAL tests. Figure 6.4 illustrates the accumulated time required for assessing each system property involved in the IOS vulnerability descriptions. Within the performed experiments, it takes as expected approximately 5 seconds for evaluating the whole set of involved OVAL tests. We also observe a linear time growth rate when the number of OVAL tests is increased as depicted in the inner graph, meaning that the proposed approach scales properly regarding the size and nature of the IOS vulnerability descriptions.

As we mentioned before, a storage mechanism able to scale with the size of system images is imperatively required. In order to measure the efficiency of our SVN-based implementation, fully detailed in Chapter 9, we have performed experiments to analyze both the size of the repository and the assessment time required for evaluating the history of past system states when new revisions are generated. Figure 6.5 shows the behavior of our implementation prototype when the number of system images is increased, instrumented as a range from 1 to 100 revisions. We have cyclically analyzed the required assessment time (red solid line) when a new vulnerability defini-

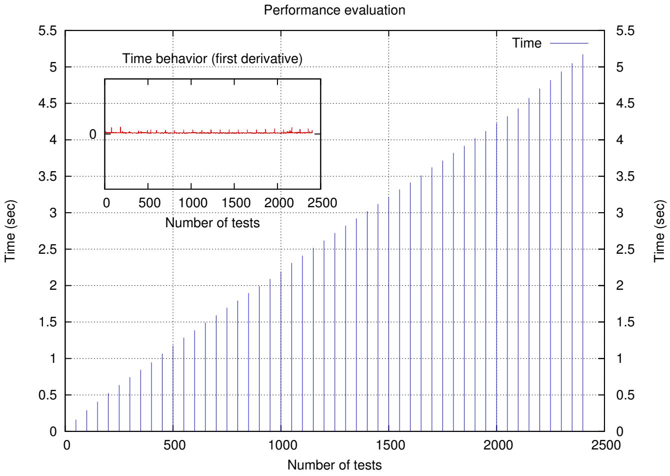


FIGURE 6.4 – Tests assessment time

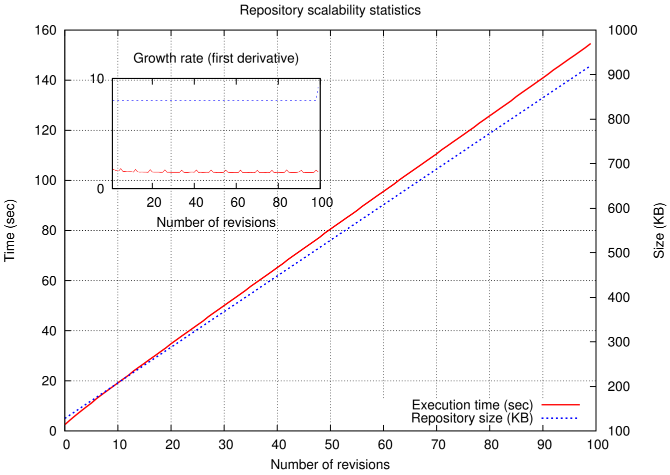


FIGURE 6.5 – Repository scalability statistics

tion becomes available over the proposed image range. As expected, it can be clearly identified a linear time growth along the number of revisions augments. In addition, the repository size (blue dashed line) also presents a stable growth rate in terms of storage requirements as shown in the inner graph. The frequency with which a network device changes its configuration can vary among platforms and usage. Nonetheless, our experiments performed over the IOS platform show that the proposed implementation prototype is capable of preserving a history of about 1 year with system images performed every 4 days in less than 1 MB of storage space. In addition, the assessment time of the whole year history can be performed in less than 3 minutes. The results obtained from these experiments confirm the feasibility and scalability of the proposed approach.

6.5 Synthesis

Vulnerability assessment tasks constitute a critical activity that is usually performed only over running systems. However, even though a known vulnerability may not be present on a current system, it could have been unknowingly active in the past providing an entry point for attacks that may still constitute a potential security threat in the present. In this chapter we have proposed an approach for increasing the overall security of computing systems by identifying past hidden vulnerable states. We have proposed a mathematical model for describing and detecting past unknown security exposures. Taking advantage of the OVAL language and the Cfengine-based approach presented in Chapter 4, we have proposed a framework able to autonomously build and monitor the evolution of network devices, and also to outsource the assessment of their exposure in an automatic manner. We have also developed an implementation prototype, which is described in Chapter 9, that efficiently performs assessment activities over an SVN repository of IOS system images. Performed experiments confirm the feasibility and scalability of our solution.

The integration of vulnerability management mechanisms into autonomic environments poses hard challenges. The approach presented in this chapter considers a centralized solution for assessing the security exposure of network devices. However, a mechanism for providing downloadable exposure analyzers would allow autonomous agents to perform actions on their own in order to move up to secure states. In that context, automated forensic investigations over past system states could provide essential information for performing appropriate corrective activities in the present. During the realization of this work, we have learnt that vulnerability assessment can be understood as an autonomic function. Indeed, the ability to outsource assessment activities provides strong support to achieve autonomic features. With this concept in mind and considering the diversity of devices involved in current networks and the Internet, a challenging domain come up, mobile devices. Mobile devices are ubiquitous computers with scarce resources that provide with applications and services to millions of users world-wide. Mobile computing exhibits constrained scenarios where autonomic computing gets challenged. Being focused on vulnerability management and autonomic environments, we decide to investigate to what extent autonomic computing may increase the security of mobile devices. Our investigation is presented in the next chapter.

Chapter 7

Mobile security assessment

Contents

7.1 Introduction **81**
7.2 Background and motivations **82**
7.3 Vulnerability self-assessment **83**
  7.3.1 Self-assessment process model 84
  7.3.2 Assessing Android vulnerabilities 85
  7.3.3 Experimental results 88
7.4 Probabilistic vulnerability assessment **91**
  7.4.1 Probabilistic assessment model 92
  7.4.2 Ovaldroid, a probabilistic vulnerability assessment extension 95
  7.4.3 Performance evaluation 99
7.5 Synthesis **101**

7.1 Introduction

The overwhelming technological advances in the broad sense of mobile computing have made end users to experience real computers in their pockets. Android [2] [11], a Linux-based operating system for mobile devices, is nowadays the election of millions of users as the platform for governing their mobile devices. Only in the second quarter of 2013, worldwide sales of smartphones to end users reached 225 million units where Android-based devices leaded the market share owning the 78.9% followed by iOS [3] with 14.2% [74]. However, despite of the many security improvements that have been done since Android's creation, the underlying operating system as well as services and applications have also evolved providing room for new vulnerabilities. Moreover, the open and barely protected mobile environment facilitates attackers to take advantage of such vulnerabilities. Sensitive data handled by mobile users becomes easily exposed. Under this perspective, lightweight and effective mechanisms for managing vulnerabilities must be provided in order to ensure safe configurations and to increase the overall security of the system.

Mobile devices are widely used with different purposes such as telephony, Internet browsing, handling of personal information, messaging and gaming. In addition, background and transparent services are also executed for controlling the overall behavior of each device. All these activities have a consumption of resources that should be taken to a minimum in order to maximize the performance and responsiveness of these mobile devices. Sometimes users may prefer to deactivate security processes such as antivirus software instead of having a short battery lifetime. This is a blocking point that we are trying to tackle. Indeed, the large-scale deployment of mobile devices combined with present security issues and their limited resources poses hard challenges that must be addressed. Such scenario makes it clear the need for non-invasive, lightweight and effective security solutions able to efficiently increase vulnerability detection capabilities in mobile environments.

---

[2] Android is developed by Open Handset Alliance, led by Google [112]

[3] Apple iOS [13]

In this chapter, we present our approach for increasing the security of the Android platform, though it could be applied over other mobile platforms as well, using the OVAL language as a means for describing Android vulnerabilities. We put forward two complementary perspectives for increasing Android security awareness. First, we present a lightweight autonomous framework for performing self-assessment activities on mobile devices. Afterwards, we enhance the proposed approach by considering a probabilistic vulnerability assessment model that efficiently reduces the resource consumption on the mobile side. The remainder of this chapter is organized as follows. Section 7.2 describes key concepts of Android security, identifying existing work and their limits, which motivates in turn the approaches presented in this chapter. Section 7.3 presents our approach for performing vulnerability self-assessment activities in the Android platform, as well as several experiments and the obtained results. Section 7.4 describes a probabilistic assessment framework that efficiently reduces computation costs as illustrated in the presented results. Section 7.5 concludes this chapter presenting conclusions and further work.

7.2 Background and motivations

Android is an open source operating system that integrates some security features by design. It uses the Dalvik virtual machine [51] for executing end user applications written in Java [88]. It is not the same standard Java virtual machine used in most popular platforms such as Linux, Mac OS X or Windows. It has its own API[4] that is almost the same as the standard one. The Dalvik virtual machine takes the Java application classes and translates them into one or more *.dex* (Dalvik Executable) files generating optimized and smaller code. The internal design of the Android platform provides important security features such as the sandbox execution approach [155]. Such approach executes Android applications within separate instances of the Dalvik virtual machine that in turn are represented by different Linux kernel processes. In order to manage the underlying system resources, Android uses an access control policy based on unique identifiers for each application to ensure that they can not interfere between each other.

Despite of the many security features provided by the Android platform [62, 141], end users still face a wide range of security threats such as denial of service and privacy bypass attacks. These threats are supported by existing vulnerabilities within the system itself, misuse of personal data performed by applications and malicious third party software [61, 68]. Several approaches have been proposed for analyzing Android applications and their risks [27, 96]. These contributions provide a strong support for increasing the security of the Android platform. Nevertheless, vulnerability assessment mechanisms have been barely or not at all discussed. Currently, dozens of security applications exist for the Android platform developed by different providers [100, 110, 163]. However, they generally use private knowledge sources as well as their own assessment techniques, and they do not provide standardized and open means for describing and exchanging vulnerability descriptions within the community.

---

[4] Application Programming Interface

Once a vulnerability is discovered in almost any typical software product, its patch cycle normally describes a time gap until the vulnerability is disclosed, another time span until the patch is available and yet another time span until the end user applies the patch [72]. It is usually during this period that attackers activity takes place. Within the Android environment, this issue gets worse. Android is distributed as open source and device manufacturers and telecommunications carriers customize it in order to provide specific services as well as added value to their customers. When a patch is released by Google, an extra time gap will occur until the manufacturer adapts it to work with its own hardware and another time span will pass until the patch is released by the carrier [155]. In addition to this problem, several application markets allow to fast distribute third party applications with only some security checks expecting that the community identifies and reports malicious software. With thousands of applications in the market, Android users are very likely to encounter malware[5] on their devices [100].

Such scenario imperatively requires solutions for rapidly identifying new vulnerabilities and minimizing their impact. Even though no patch might be available for a new vulnerability at a given time, countermeasures can be taken in order to mitigate the problem until the disclosure of an official patch. In that context, vulnerability assessment mechanisms are highly required in order to increase the vulnerability awareness of the system. In addition, mobile devices usually have limited resources thus optimized lightweight tools should be developed to ensure efficiency without losing functionality. Moreover, there are no current solutions built over solid foundations as well as open and mature standards that foster its adoption and speed up general vulnerability information exchange. Currently, OVAL repositories offer a wide range of vulnerability descriptions though Android is not yet an official supported platform. In this work, we have instrumented our approach with an experimental OVAL extension for Android within the OVAL Sandbox project [134]. Such extension enables practitioners and experts within the field to specify known vulnerabilities for Android in a machine-readable manner and at the same time, it promotes the exchange and enrichment of Android security information within the community. Our work aims at defining a solution for increasing the security of Android devices by capitalizing Android vulnerability descriptions specified with the OVAL language. Indeed, our investigation involves two perspectives. The first one considers security advisories which are automatically integrated into an autonomous distributed architecture, where lightweight self-assessment activities are performed in order to ensure safe mobile configurations. Our second approach goes one step further by considering a probabilistic model able to reduce computation costs and resource allocation. Both approaches are presented in detail in the following sections.

7.3 Vulnerability self-assessment

The process by which vulnerabilities are assessed is critical for efficiently analyzing a target system and minimizing computation costs at the same time. In order to increase the security awareness of mobile devices, our first approach considers a self-assessment perspective where mobile devices are in charge of assessing their own exposure. In this section we present a mathematical model that defines and efficiently supports the vulnerability assessment process, an autonomous framework for performing mobile vulnerability self-assessment, and several experiments that show the feasibility of the proposed approach.

---

[5] Malicious software including viruses, worms and spyware among others

7.3.1 Self-assessment process model

Usually, a vulnerability can be understood as a logical combination of properties that if observed in a target system, the security problem associated with such vulnerability is present on that system. Properties can vary depending on the nature of the vulnerability being described, some examples are: a specific process is running (e.g., httpd), a specific port is open (e.g., 80), the system has a specific version (e.g., 2.6.10.rc). Frequently, one property is required by several vulnerability descriptions and naturally one vulnerability description may require several properties. Under this perspective, the set of vulnerability descriptions that constitutes a knowledge base can be compactly represented by using a boolean pattern matrix PM defined as follows:

$$PM = \begin{array}{c} \\ v_1 \\ v_2 \\ \vdots \\ v_m \end{array} \begin{array}{c} \begin{array}{cccc} p_1 & p_2 & \cdots & p_n \end{array} \\ \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \end{array} \quad a_{i,j} \in \{0,1\}$$

Each matrix row encodes the properties required to be observed for the vulnerability v_i to be present. Thus, each entry $a_{i,j}$ denotes if the vulnerability v_i requires the property p_j. Considering for instance a scenario with three vulnerabilities v_1, v_2 and v_3, a pattern matrix PM can be built as follows:

$$\left.\begin{array}{l} v_1 = (p_1, p_3, p_5) \\ v_2 = (p_2, p_4) \\ v_3 = (p_1, p_2, p_5) \end{array}\right\} \quad PM_{3,5} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

The pattern matrix can also provide useful information for performing statistics. The $vflatten$ operation aggregates the number of times that each property occurs within the whole set of known vulnerabilities. The resulting vector provides an indicator that helps to identify most common properties involved in vulnerabilities. Such indicator provides valuable information that can be used for closer monitoring and controlling critical components changes.

$$vflatten(PM) = (\sum_{i=1}^{m} a_{i1}, \sum_{i=1}^{m} a_{i2}, \ldots, \sum_{i=1}^{m} a_{in})$$

Other useful metric can be extracted from the pattern matrix when the aggregation operation is performed horizontally, as indicated by $hflatten$. A column vector is obtained from its application where each entry j denotes the amount of properties required by each vulnerability v_j. This metric can be utilized, among other uses, for identifying those vulnerabilities that are most likely affected by changes performed in the environment, thus assessment activities should be taken into account as well.

$$hflatten(PM) = (\sum_{j=1}^{n} a_{1j}, \sum_{j=1}^{n} a_{2j}, \ldots, \sum_{j=1}^{n} a_{mj})^T$$

The state of a system can be encoded in the same manner as done with vulnerabilities, indicating for those properties under control, which ones are present and which ones are not. Thus, a system state is a boolean vector s defined as follows:

$$s = (s_1, s_2, \ldots, s_n) \quad s_i \in \{0,1\}$$

Each entry s_i takes the value 1 if the property p_i is present in the system and 0 if it is not. Considering these constructs, the results of performing the vulnerability assessment process over a given system is defined by the following equation:

$$w = hflatten(PM) - [PM * s^T] \tag{7.1}$$

$$\Downarrow$$

$$w = \begin{pmatrix} \sum_{j=1}^{n} a_{1j} \\ \sum_{j=1}^{n} a_{2j} \\ \vdots \\ \sum_{j=1}^{n} a_{mj} \end{pmatrix} - \left[\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \times \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{pmatrix} \right]$$

The resulting assessment vector $w = (w_1, w_2, \cdots, w_m)$ denotes the status of each vulnerability v_i in the target system. The semantic of the vector w is given by the *Kronecker delta* function as follows:

$$\delta_i = \begin{cases} 0, & \text{if } i \neq 0 \\ 1, & \text{if } i = 0 \end{cases}$$

A null entry w_i indicates that the vulnerability v_i is present in the system while non null values denotes the absence of the corresponding vulnerability. This fact can be understood as a distance metric where a positive value indicates a positive distance between the vulnerability and the target system, and a null distance indicates that the vulnerability is actually in the system. Computing matrix operations in optimized manners constitutes a field that has been studied for years [145]. The integration of the proposed model into real computing systems can take advantage of such expertise providing a compact and efficient representation for performing vulnerability assessment activities.

7.3.2 Assessing Android vulnerabilities

The previous model establishes a well-founded process for assessing vulnerabilities in an efficient manner. By taking advantage of OVAL security advisories, such model can be used for efficiently increasing the security of mobile computing devices. Mobile devices have become a daily useful resource for connecting people, entertainment, working, managing personal data and much more. This fact attracted the attention of legitimate users of these pocket-computers but also from attackers. In only the first semester of 2011, malware for the Android platform has grown at 250% [100]. It is critical to develop open security frameworks that can speed up the knowledge exchange among community users and also being able to take advantage of such information in order to augment their own security. In this section we present our approach for efficiently increasing the security of Android-based devices by automatically evaluating OVAL-based vulnerability descriptions and reporting analysis results.

Architecture and main components

We have designed the proposed architecture illustrated in Figure 7.1 as a distributed infrastructure composed of three main building blocks: (1) a knowledge source that provides existing security advisories, (2) Android-based devices running a self-assessment service and (3) a reporting system for storing analysis results and performing further analysis. The overall process is defined as follows. Firstly at step 1, the Android device periodically monitors and queries for new vulnerability descriptions updates. This is achieved by using a web service provided by the

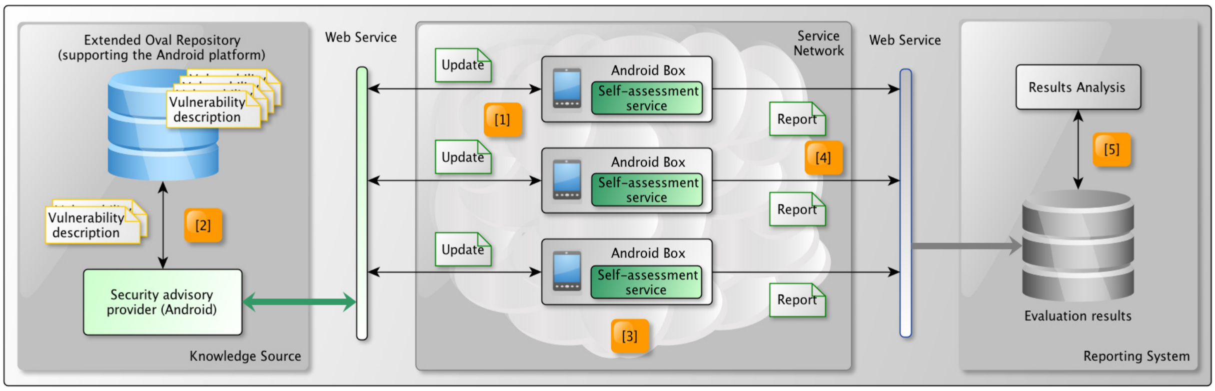

Extended Oval Repository
(supporting the Android platform)
Vulnerability description
Vulnerability description
[2]
Security advisory provider (Android)
Knowledge Source
Web Service
Update
Update
Update
[1]
Android Box
Self-assessment service
Android Box
Self-assessment service
Android Box
Self-assessment service
Service Network
Report
[4]
Report
Report
[3]
Web Service
Results Analysis
[5]
Evaluation results
Reporting System

FIGURE 7.1 – OVAL-based vulnerability assessment architecture for the Android platform

security advisory provider. At step 2, the provider examines its database and sends back new found entries. The updater tool running inside the Android device synchronizes then its security advisories. When new information is available or configuration changes occur within the system, a self-assessment service is launched in order to analyze the device at step 3. At step 4, the report containing the collected data and the results of the analyzed vulnerabilities is sent to a reporting system by means of a web service request. At step 5, the obtained results are stored in the external database. This information could be used later for different purposes such as forensic activities or statistical analysis.

Within the proposed approach, vulnerabilities are described by using OVAL definitions. As explained before, an OVAL definition is intended to describe a specific machine state using a logical combination of OVAL tests that must be performed over a host. If such logical combination is observed, then the specified state is present on that host (e.g. vulnerability, specific configuration). Under a logical perspective, this combination can be understood as a first order formula where each OVAL test corresponds to an atomic unary predicate over that system as presented in Chapter 4. The model presented in Section 7.3.1 denotes these predicates as the set of properties $P = \{p_1, p_2, \ldots, p_n\}$. P represents all the predicates (OVAL tests) involved in the vulnerability descriptions (OVAL definitions) available within our knowledge source. In this manner, a boolean matrix PM representing each involved OVAL test for each OVAL definition can be easily built in order to perform assessment activities. The self-assessment component depicted in Figure 7.1 constitutes a critical building block because it is in charge of orchestrating the entire lifecycle of the framework in an automatic manner. Hence, optimized algorithms for performing self-assessment activities are highly required. In order to achieve this objective, we have designed and implemented a strategy that uses the model presented in Section 7.3.1 for minimizing the system components required to be assessed.

Optimized assessment strategy

Due to the limited resources provided usually by mobile devices, it is important to optimize the use of such elements without losing functionality and performance. The proposed assessment strategy takes this issue into account and minimizes computation costs by using a boolean pattern matrix PM that represents known vulnerabilities and a system state vector s that holds the current system properties. The overall assessment is then efficiently performed using both the pattern matrix and the system vector defined in Section 7.3.1. Within our approach, two types of events can trigger self-assessment activities: (i) when changes occur in the system and (ii) when new vulnerability definitions are available. Algorithm 7.1 depicts the overall strategy for treating such events and minimizing the number of OVAL tests to be re-evaluated. In order to explain the proposed algorithm, we put forward an illustrative example that considers both situations and uses the matrix $PM_{3,5}$ illustrated in Section 7.3.1. Let consider the property p_2 = *{Package X has version Y}* and the system state $s = (1, 0, 0, 0, 1)$ meaning that only the properties p_1 and p_5 are present in the system. Within the OVAL language, p_2 is described using an OVAL test that involves an OVAL *package_ object* with its attribute $name = X$ and an OVAL *package_ state* with its attribute $version = Y$.

```
   Input: Event event, PatternMatrix matrix, SystemState state
   Output: VulnerabilityList list
 1 if event is of type SystemChange then
 2     $objs \leftarrow getAffectedObjectsByEvent(e)$;
 3     foreach Property $p \in state$ do
 4         $o \leftarrow getObjectFromProperty(p)$;
 5         if $o \in objs$ then
 6             $result \leftarrow evaluateProperty(p)$;
 7             $updateSystemState(state, p, result)$;
 8         end
 9     end
10 end
11 if event is of type DefinitionUpdate then
12     $defs \leftarrow getDefinitionsFromEvent(e)$;
13     $props \leftarrow getPropertiesFromDefinitions(defs)$;
14     foreach Property $p \in props$ do
15         if $p \notin state$ then
16             $addEmptyPropertyColumn(matrix, p)$;
17             $addEmptyPropertyColumn(state, p)$;
18             $result \leftarrow evaluateProperty(p)$;
19             $updateSystemState(state, p, result)$;
20         end
21     end
22     foreach Definition $d \in defs$ do
23         $addAndLoadDefinitionRow(matrix, d)$;
24     end
25 end
26 $w \leftarrow hSumMatrix(matrix) - (matrix * state)$;
27 $index \leftarrow 0$;
28 foreach Entry $v \in w$ do
29     if $v = 0$ then
30         $vulnDef \leftarrow getVulnDef(index)$;
31         $addToOutputList(list, vulnDef)$;
32     end
33     $index \leftarrow index + 1$;
34 end
```

Algorithm 7.1: Efficient event-based vulnerability assessment algorithm

Let suppose now that an event of type *package_ updated* has occurred in the system affecting the package X (line 1). Usually, a complete evaluation of each OVAL definition involving the OVAL test that describes the property p_2 should be carried out. However, only the truth value of the involved OVAL test for p_2 is required for recomputing the results of all the descriptions affected. In order to achieve this, the objects affected by the event are retrieved (line 2) and compared with the objects related to the system properties (lines 3-4). If the object of one property is seen to be affected (line 5), the property represented by an OVAL test is re-evaluated and reflected in the system state (lines 6-7). Within our example, such optimization point will only assess and change the second entry of the system state s. Due to both events are disjoint (system changes at line 1 and definition update at line 11), we now explain the end of the algorithm for the first case and then we discuss the behavior for the second case. Let suppose that the new value for the package version is Y thus the new system state becomes $s = (1, 1, 0, 0, 1)$. Once the assessment of the OVAL test for p_2 has been done, the overall assessment result is achieved by performing two operations between boolean matrices (line 26), within our example, as given by Equation 7.2.

$$w = \begin{pmatrix} 3 \\ 2 \\ 3 \end{pmatrix} - \left[\begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \right] = \begin{pmatrix} 1 \\ 1 \\ \boxed{0} \end{pmatrix} \tag{7.2}$$

For each entry in the result vector w (line 28), we use the *Kronecker delta* function (line 29) in order to detect if the vulnerability represented by that entry is present in the target system. If it is the case, the vulnerability definition is added in the output detected vulnerability list (lines 30-31). Within our example, it can be observed that the change performed in the system has exposed itself to new security risks due to the presence of the vulnerability v_3.

The second situation involves the arrival of new vulnerability descriptions (line 11). In this case, both the pattern matrix PM and the system state s have to be extended so as to cover the new properties involved in the OVAL definitions. In order to achieve this, the new definitions are retrieved from the event (line 12), and the properties involved within such definitions are analyzed (lines 13-14). For each uncovered property (line 15), an extension process must be applied. The extension process for the pattern matrix PM will include new columns with null entries for the new properties within existing vulnerability definitions (line 16). The system state s is extended (line 17) and updated as well with the result of the property assessment (lines 18-19). It is important to notice that the arrival of new vulnerability definitions does not imply changes on the system and that the assessment results for known properties are already loaded in the system state, thus there is no need to re-evaluate them again. Finally, for each new vulnerability definition (line 22), a new row is added in the pattern matrix PM indicating the required properties for that vulnerability to be present (line 23). The final assessment procedure is then performed in the same manner as explained in the first situation (lines 26-34). The proposed strategy constitutes a critical part of our framework and it has been integrated into an implementation prototype, which is fully described in Chapter 9. In that context, we have performed several experiments to measure the performance of the proposed framework. These experiments are illustrated in the next section.

7.3.3 Experimental results

Devices with limited resources imperatively require well-designed and optimized software that take care of such elements. In this section we present an analytical evaluation of the proposed mathematical model as well as a technical evaluation that involves a comprehensive set of experiments showing the feasibility and scalability of our solution.

Analytical evaluation

In the proposed approach, the vulnerability assessment process is governed by Equation 7.1. Given n as the number of system properties being monitored and m the number of available vulnerability definitions, the complexity of computing the result vector w is $n \times m$. Considering the worst case ($n = m$), the complexity is $O(n^2)$. Being $hflatten(PM)$ a known value, the number of operations performed during the process are n boolean multiplications plus $n - 1$ integer sums for each vulnerability definition. Then, the total number of boolean multiplications is $m \times n$ and the total number of integer sums is $m \times (n - 1)$. Hence, $m \times (n + (n - 1)) \approx n^2$ arithmetic operations are performed for assessing the entire knowledge repository in the worst case.

Considering a knowledge repository with 1000 vulnerability definitions involving 1000 different system properties, the size of the pattern matrix PM is 10^6. This means that the assessment process defined by the model will perform 10^6 arithmetic operations for assessing the entire knowledge base. Considering MFLOPS[6] as the performance measure, though boolean and entire operations are cheaper than floating point operations, the assessment requires 1 MFLOP. Within our experimental devices *Samsung Galaxy Gio* running Android 2.3.3, we have measured an average of 8.936 MFLOPS. With this information, we can infer that a dedicated application of our strategy over a 10^6 size matrix takes less than 1 second in almost any standard Android-based device.

Moreover, latest models may achieve more than 100 MFLOPS meaning that a knowledge source of 10000 vulnerability definitions involving 10000 different properties could be mathematically assessed in less than 1 second. Currently, the OVAL repository [117] offers 8747 UNIX vulnerability definitions including all versions and families after years of contributions made by the community. Such scenario provides real facts making the proposed approach highly suitable for efficiently performing vulnerability assessment activities.

Technical experimentation

We have performed several experiments in order to analyze the behavior of our implementation prototype. The proposed methodology cyclically tests the framework without other applications running in foreground. The OVAL definitions set is increased by 5 each time until a set of 100 definitions is evaluated. The used OVAL definitions are similar in size containing on average two OVAL tests. For instance, the vulnerability with the *CVE-2011-3874* id permits a locally installed application to gain root privileges by causing a buffer overflow within *libsysutils*. This vulnerability only affects specific Android versions (first OVAL test) and requires the existence of the library *libsysutils* (second OVAL test). Figure 7.2 illustrates the behavior of our implementation prototype over the emulated Android device. Our implementation prototype is heavily based on XOvaldi4Android. We have developed XOvaldi4Android as an extension to the XOvaldi OVAL interpreter [26] in order to also support the Android platform. Details of XOvaldi4Android are presented in Chapter 9. Within our experiments, we analyze three performance dimensions: (1) the CPU utilization when XOvaldi4Android is executed (red solid line with crossings), (2) the XOvaldi4Android execution time (green dashed line with triangular points) and (3) the total framework execution time (blue dashed line with rounded points). During the XOvaldi4Android execution, we have observed a stable and linear behavior in terms of CPU utilization, consuming 80% on average. Its execution time is also stable as shown by the first derivative within the inner graph. While assessing 50 definitions takes about 72 seconds, 100

---

[6] Million Floating Point Operations Per Second

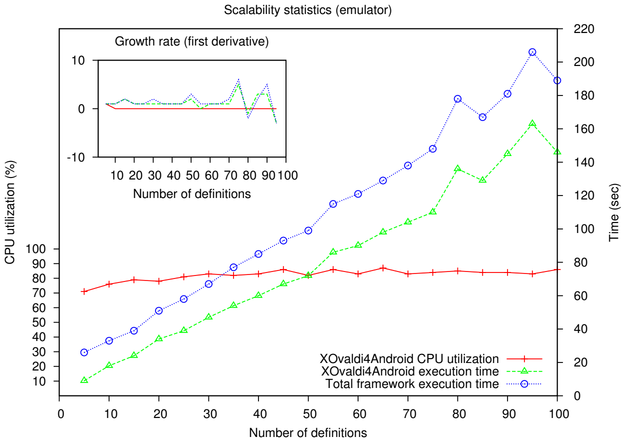


FIGURE 7.2 – Scalability statistics in a simulated environment

definitions takes almost twice the time. The overall execution time across the framework, including database updates and reporting results, shows the same behavior though slightly increased in time due to the sequential execution of its components. It is important to notice that these experiments consider extreme cases. As a matter of fact, only new definitions or a small set of definitions affected by system changes will be evaluated in most situations.

In order to analyze the framework behavior using a real device, we have performed the same experiments using a standard smartphone *Samsung Galaxy Gio S5660* (CPU 800 MHz, 278 MB of RAM, Android 2.3.3). Figure 7.3 illustrates the obtained results. We can observe the same behavior on each curve as with the emulated device, describing a linear growth for each analysis dimension as shown in the inner graph. Nevertheless, we have also detected an improvement in terms of speed and resource usage. The average value for the CPU utilization is now about 65%. In addition, the execution time of XOvaldi4Android is almost half the emulator execution time, taking 38 seconds for analyzing 50 vulnerabilities and 75 for 100 vulnerabilities. This is probably due to a slower emulated CPU. The overall execution time is also reduced due to the faster execution of the vulnerability assessment process. However, its growth rate, though linear, is faster because the internetwork connections are real in this case.

As a final but not less important dimension to analyze, we have experimented with the memory load. Within this analysis, we have considered the allocated memory required by XO-

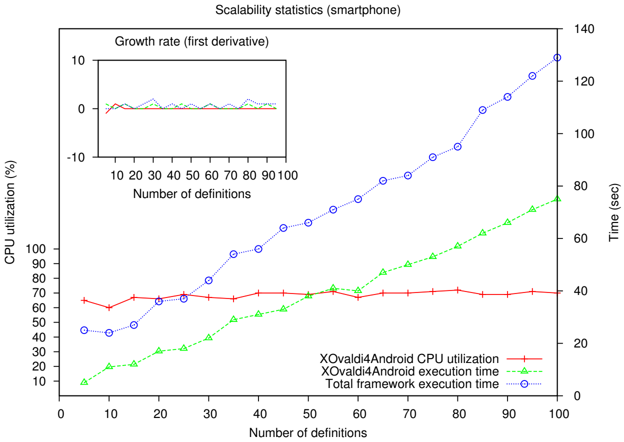


FIGURE 7.3 – Scalability statistics in a real device

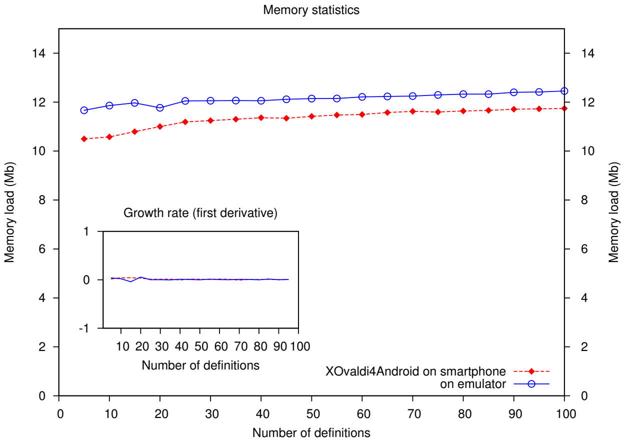


FIGURE 7.4 – Memory load in both emulated and real device

valdi4Android when it is executed. The system classifies the allocated memory in two categories, native and Dalvik, taking on average 40% for native memory and 60% for Dalvik memory. Figure 7.4 illustrates the total memory load considering both, the emulator and the smartphone. We have observed an almost constant utilization of the RAM memory. Within the emulator (blue solid line with rounded points), XOvaldi4Android requires 12 MB on average (4.8 MB of native memory, 7.2 MB of Dalvik memory). Within the smartphone (red dashed line with rhomboid points), XOvaldi4Android requires a little less memory, 11 MB on average (4.4 MB of native memory, 6.6 MB of Dalvik memory). The results obtained from the performed experiments show good performance in terms of resource consumption and scalability. However, the ability to outsource vulnerability assessment activities, as done in Chapter 6 for detecting past unknown security exposures, can reduce the workload even further. Performing a smart management of assessment activities in terms of time and type of vulnerabilities, may dramatically reduce the resource allocation required on the mobile side. This approach is discussed in the next section.

7.4 Probabilistic vulnerability assessment

Delegating vulnerability assessment activities to mobile devices provides higher levels of autonomy. However, when these activities are performed, there is still a resource consuming process in the mobile side that must control the overall behavior of the analysis. We have realized that the externalization of this control may still provide autonomicity and decrease the mobile workload. In this section, we present an extended approach that centralizes main logistic vulnerability assessment aspects as a service. Mobile clients only need to provide the server with required data to analyze known vulnerabilities described with the OVAL language. By configuring the analysis frequency as well as the percentage of vulnerabilities to evaluate at each security assessment, the proposed framework permits to bound client resource allocation and also to outsource the assessment process. Our strategy consists in distributing evaluation activities across time thus alleviating the workload on mobile devices, and simultaneously ensuring a complete and accurate coverage of the vulnerability dataset. This technique results in a faster assessment process, typically done in the cloud, and considerably reduces the resource allocation on the client side.

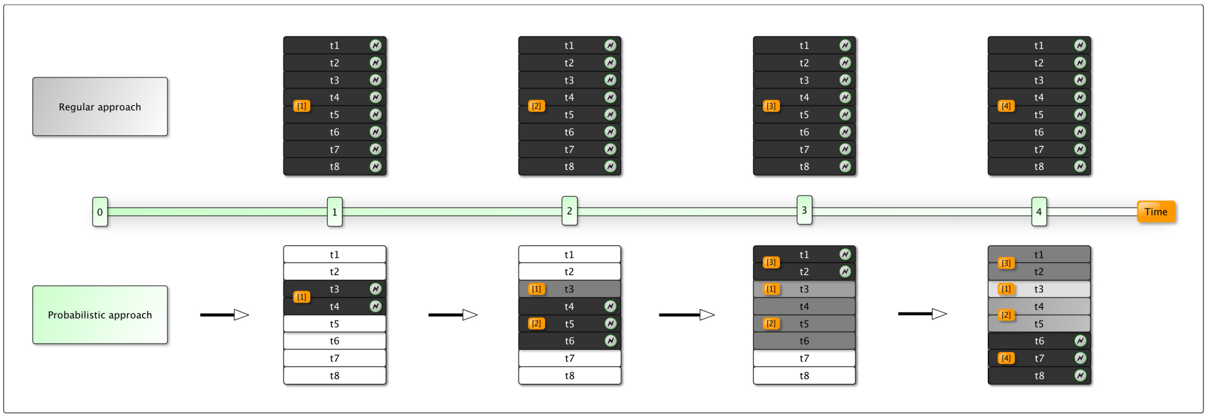


FIGURE 7.5 – Regular vs. probabilistic approach

7.4.1 Probabilistic assessment model

When developing mobile solutions, limited resources present on mobile devices must be carefully managed in order to increase the performance and responsiveness of such devices. In that context, our aims at reducing the resource consumption at the target device, e.g. battery, CPU, and at the same time increasing the vulnerability assessment accuracy.

Regular vs. probabilistic approach

Each time a security analysis is made, vulnerabilities descriptions are analyzed in order to detect security weaknesses on a target device. As a remainder, the OVAL language represents vulnerabilities by means of OVAL definitions. Each OVAL definition logically combines OVAL tests that represent atomic checks or evaluations over the target device. Each OVAL test in turn can be referenced by different OVAL definitions and contains an OVAL object that describes the component to be analyzed, and an OVAL state that describes the properties expected to be observed on the specified component. The test result will be *true* if the component actually exhibits the specified state, and $false$ otherwise. Let $T = \{t_1, t_2, \ldots, t_n\}$ be the set of available OVAL tests. Then, the set of known vulnerability descriptions $V = \{v_1, v_2, \ldots, v_m\}$ constituting our knowledge source can be built by respecting the following rules:

i. if $t_i \in T$, then $t_i \in V \qquad (i \in N)$
ii. if $\alpha, \beta \in V$, then $(\alpha \diamond \beta) \in V \qquad \diamond \in \{\wedge, \vee\}$
iii. if $\alpha \in V$, then $(\neg\alpha) \in V$.

Traditional assessment mechanisms usually evaluate these vulnerabilities in a one-step fashion by analyzing the whole set of vulnerability descriptions at once. Such methodology is highly time and resource-consuming. Our approach aims at dealing with this problem by probabilistically distributing vulnerability assessment activities across time and restricting resources affected by this task. Figure 7.5 exemplifies both regular and probabilistic approaches where a set of vulnerabilities involving eight single tests is evaluated during four periods of time. The regular approach analyzes the whole body of vulnerabilities at each period thus evaluating all tests each time. This is accurate but constitutes an extremely heavy task. The probabilistic approach on the other hand selects only a subset of tests to execute in order to cover a subset of vulnerabilities each time. Tests are probabilistically selected according to their utility on the resolution of vulnerability evaluations as well as the elapsed time since their last analysis. The test selection process constitutes the heart of this section and it is detailed in the following subsections. By following this methodology, the probabilistic approach highly reduces the activity load and resource allocation at each security analysis while rapidly converging to a complete assessment of the vulnerability set.

The probabilistic approach is also depicted in Figure 7.5 where only tests t_3 and t_4 are evaluated and tagged at period 1. At period 2, tests t_5 and t_6 are evaluated and tagged but also t_4, probably due to a high utility value thus being re-evaluated once again. Test t_3 has not been selected at this period thus becoming one period older in terms of its evaluation, illustrated with a less intense grey color. At period 3, tests t_1 and t_2 are evaluated while test t_3 becomes two periods older, and t_4, t_5 and t_6 only one. At period 4, tests t_6, t_7 and t_8 are selected for evaluation thus completing the whole vulnerability assessment. Notice the re-evaluation of t_6, probably due to a high utility value again. The selection process continues like this across time thus t_3, the oldest evaluated test so far, will have a higher probability of being selected but it will still compete with other high utility tests during future selection processes. The idea is that high utility tests are more frequently evaluated but low utility tests are also evaluated as they become older. Therefore, test starvation is avoided ensuring the convergence towards the analysis of the complete set of known vulnerabilities.

The proposed model considers different parameters that allow the user to adapt it according to specific needs, namely, (1) a threshold λ that indicates the percentage of vulnerabilities that must be evaluated at each security analysis, and (2) a time interval δ that specifies the amount of elapsed time between each security analysis. The overall idea is that during each security analysis made with frequency δ, an iterative evaluation process is performed, statistically guided by the utility that each test has over the current vulnerability database as well as the elapsed time since their last evaluation. Tests are probabilistically selected until the desired threshold λ is achieved. In order to minimize the load impact over mobile devices, the process by which tests are selected is critical because of two reasons, firstly it must consider the most useful tests at each security analysis and secondly, it must ensure that all tests will be eventually executed. These concepts are presented in the next subsections.

Test utility analysis

Within the proposed model, the utility of a test aims at expressing a metric that combines the ability of this test to speed up the overall evaluation and its security impact on the target system. Such concept relies on: (1) how much the body of vulnerability descriptions can be reduced towards a complete coverage when its value is determined, and (2) the security impact of the vulnerabilities in which the test is involved in. The concept of reduction refers to the idea of how much closer we are to determine the truth value of the vulnerabilities under analysis when a test value is known. For instance, let v be a vulnerability description with the form $v = t_1 \wedge (t_2 \vee t_3)$. If the value of t_1 is known and it is *false*, then there is no need to evaluate t_2 and t_3 as the final value for v will be *false* no matter what values take t_2 and t_3. In this case, the utility of t_1 is higher than the utility of t_2 and t_3 because its evaluation could potentially eliminate the need to evaluate the remaining tests in the formula. The other way around however is not true; if t_2 is *false* then t_3 must be evaluated, if it is *true* then t_1 must be evaluated. No matter what value takes t_2, a second test must always be evaluated. The same phenomenon occurs with t_3. Therefore, t_1 fits better in this situation and it will have a higher utility value than t_2 and t_3. During the reduction process, if the evaluation result of t_1 is $false$ then v will be reduced to $v = false$ thus completing the evaluation. If the result evaluation of t_1 is $true$ instead, then v will be reduced to $v = true \wedge (t_2 \vee t_3) = t_2 \vee t_3$. The process will then continue over t_2 and t_3 until obtaining the truth value for v.

In order to facilitate the quantification of the utility of a test, vulnerabilities are represented as formulas in conjunctive normal form (CNF). A vulnerability expressed in CNF is a conjunction of clauses where each clause is a disjunction of tests as follows:

$$v_i^{CNF} = \bigwedge(\bigvee(t_j | \neg t_j)) \qquad t_j \in T, v_i \in V \tag{7.3}$$

Accordingly, if the value of a test t is known, its utility over a specific vulnerability database V is expressed by a fitness function U defined as follows:

$$U(t, val, V) = \frac{\sum_{i=1}^{|V|} \left(\frac{testRed(v_i, t, val) * I(v_i)}{totalTests(v_i)} \right)}{|V|} \tag{7.4}$$
$$t \in T, val \in Boolean, v_i \in V$$

The *testRed* function represents the number of tests whose truth values do not contribute to the final resolution of v_i when the value of t is val. The *totalTests* function returns the number of tests involved in v_i. The I function returns a numerical value representing the impact security factor or criticality of v_i, e.g., its CVSS score [47]. Because the function represented by Equation 7.4 is used for selecting the next test to be executed, the evaluation values for those tests under selection are not known yet. Therefore, we define a weight function W for determining the average utility of a test t over a vulnerability database V as follows:

$$W(t, V) = \frac{U(t, true, V) + U(t, false, V)}{2} \qquad t \in T \tag{7.5}$$

In order to select the next test to be evaluated, tests are sorted by descending utility values producing an ordered list $T_W = \{t_1, t_2, \ldots, t_n\}$. This list provides statistical-based ranking information for unevaluated tests that combined with a temporal factor supports the probabilist test selection process.

Probabilistic test selection process

As there is a threshold λ that limits the execution of the whole set of tests, not every test will be executed during a single security analysis. If only best tests were selected at each analysis and the device state remains the same, there would be tests that would never be evaluated. This effect is called test starvation meaning that some tests might never come up with the opportunity to be evaluated because of their low utility values. Therefore, some vulnerabilities might never be covered either. In order to avoid test starvation, we consider two factors that shape the overall behavior of our strategy across time. The first factor is a weighted probability ρ for each test that is directly proportional to its utility value. This means that even when a test has the highest utility value, another test with a lower utility value could be selected in its place for execution. Such approach is less elitist though still fair as it provides the opportunity for lower tests to substitute higher tests with probabilities according to their ranking. In order to specify the probability for a test to be chosen according to its positioning in the weighted list T_W, we define the ρ function as follows:

$$\rho(t, V, T_W) = \frac{W(t, V)}{\sum_{i=1}^{|T_W|} W(t_i, V)} \qquad t, t_i \in T_W \tag{7.6}$$

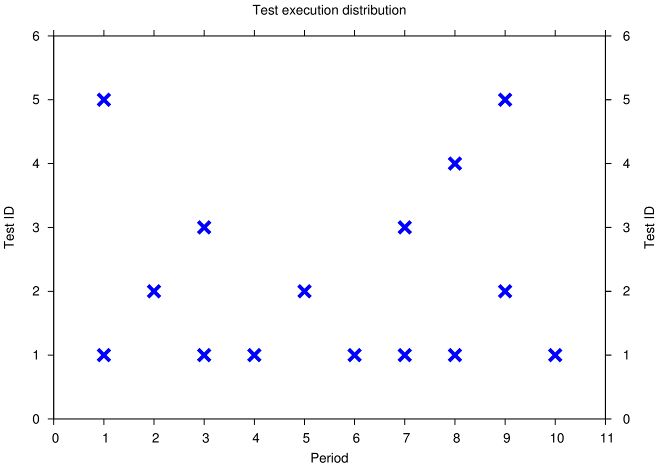


FIGURE 7.6 – Test execution distribution

The second factor to avoid test starvation is the elapsed time τ between each security analysis. The older the last evaluation of a test is, the higher is the chance for this test to be selected. This increase however must consider their ranking status indicated by the first factor ρ in order to respect the statistical analysis done for each test. In order to combine both factors in the selection process, we define the selectivity value for a test t in a given time x by the following equation:

$$S(t,x,V,T_W) = \rho(t,V,T_W) * \tau(t,x) \quad t \in T_W, x \in [0..\infty) \tag{7.7}$$

The main idea in Equation 7.7 is to prioritize high impact tests given their weighted probabilities but simultaneously promoting lower tests that turn more important as long as their last evaluations become older. The delta time τ for a test t is considered as the time elapsed between its last evaluation and a specific time x. τ is defined as follows:

$$\tau(t,x) = x - lastEvalTime(t) \quad t \in T_W, x \in [0..\infty) \tag{7.8}$$

The behavior of the selection process is illustrated in Figure 7.6 where five tests constitute the body of known vulnerabilities V and they are assessed over ten periods of time ($\delta = 1$). Tests have been ordered according to their utility values over V, being the first test the most useful test. It can be observed how the test with the highest utility has been selected seven times, much more than the other tests with lower utility values. However, lower utility tests also have been selected though in a lower rate. It can be also noticed that the fourth test is stronger than the fifth test in terms of utility, but in this specific experiment however, the latter shows a higher selection frequency (periods 1 and 9) than the former (period 8). This is an interesting effect due to the probabilistic nature of the process though in the general case, as illustrated later in Section 7.4.3, the test execution frequency tends to a coherent distribution according to test utility values. In the next section we present Ovaldroid, a probabilistic vulnerability assessment framework that integrates the proposed model in order to increase the overall security of Android devices.

7.4.2 Ovaldroid, a probabilistic vulnerability assessment extension

Ovaldroid is a probabilistic-based framework designed for assessing configuration vulnerabilities over Android devices. We explain here its architecture as well as the underlying strategy that has been cautiously designed for outsourcing as much as possible the involved assessment activities and dealing with issues such as resource usage and ubiquity.

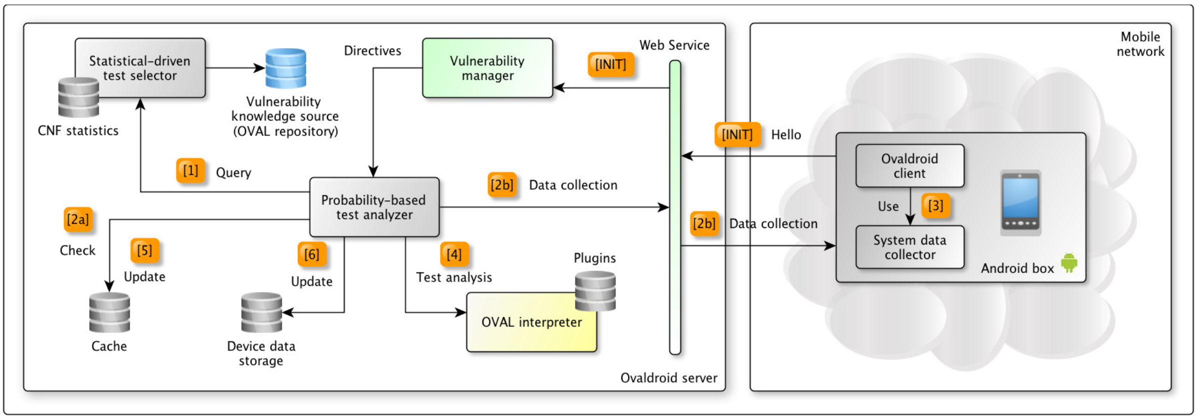

Statistical-driven test selector
CNF statistics
Vulnerability knowledge source (OVAL repository)
Directives
Vulnerability manager
[INIT]
Web Service
Mobile network
[INIT]
Hello
Ovaldroid client
Use
[3]
System data collector
Android box
[1]
Query
Probability-based test analyzer
[2b]
Data collection
[2b]
Data collection
[2a]
Check
[5]
Update
[6]
Update
[4]
Test analysis
Plugins
OVAL interpreter
Cache
Device data storage
Ovaldroid server

FIGURE 7.7 – Ovaldroid global architecture

Architecture overview

The architecture of Ovaldroid, described in Figure 7.7, has been designed as a centralized service-oriented infrastructure capable of analyzing vulnerabilities over Android-based devices. It is composed of two main building blocks, namely, a server that manages the whole assessment process and clients located on the mobile network that use the vulnerability assessment service. Mobile clients periodically communicate with the Ovaldroid server in order to inform about their assessment availability. This communication is started by the Ovaldroid client that sends an identified *Hello* message using the web-service provided by the server. Based on the historical evaluation registry, the server decides whether it is necessary to perform a new vulnerability assessment based on the pre-established assessment frequency (δ). If it does, the vulnerability manager subsystem located on the server side sends specific directives to the probabilistic-based test analyzer in charge of orchestrating the overall assessment activity. The probability-based test analyzer in turn, executes a sequence of OVAL tests until the specified percentage of vulnerabilities to be evaluated (λ) is reached.

In order to select which OVAL test must be evaluated at each iteration, the analyzer uses the services of the statistical-driven test selector (step 1). The latter builds, at the first call, a local CNF database representing the vulnerability descriptions available in the vulnerability knowledge source. Then, at each query sent by the analyzer, the statistical-driven test selector will produce an ordered list of tests suitable to be performed over the target device based on the impact that each unevaluated test has towards the desired vulnerability coverage. The analyzer then chooses the test to be executed from this list by considering its ranking combined with the elapsed time since its last evaluation as the probability to be selected. This means that high utility tests will be more likely to be selected because of their high ranking values. However, low utility tests still have the opportunity to be selected though in a minor rate.

Once a test has been selected for execution, the analyzer checks if a previous unexpired result for this test exists in the cache (step 2a). If it does, it is directly used thus saving computation resources on the client side. The cache also stores collected objects from previous tests due to sometimes the same object is used by different tests. Therefore, if no result for this test is found, the system looks for an unexpired version of the object previously collected from the device under analysis over which this test applies. If there is a hit, the object is used without interacting with the target device. Otherwise, the analyzer performs a data collection request on the target device (step 2b) in order to gather the required data and assess the corresponding OVAL test on the server side. Cache entries do not affect the test selection process itself because the oldness of these tests is already considered in the model. Therefore, the cache and its policy can be independently set to reduce the load even further on the target device.

Data collection is done on the client side by running a lightweight Android application (step 3). Once the required object is available, the services of an OVAL interpreter are used in order to evaluate the selected OVAL test (step 4). Depending on the nature of a vulnerability, different types of tests might be used when describing it, e.g., file tests, process tests, version tests. In that context, the OVAL interpreter uses plugins for each type of OVAL test where each plugin knows how to collect and analyze the information of the type of test it was created for. After the evaluation, the collected object and the test result are stored in the cache for future use (step 5). Finally, the test result is also placed in the results storage system on the server side (step 6). The process continues over steps 1 to 6 until the percentage of vulnerability coverage specified by the administrator is reached. Final assessment results are also saved in the results storage system.

Assessment strategy

The proposed methodology integrates a probabilistic component for selecting which tests must be evaluated at each security analysis. However, the spectrum of eligible tests is built following a statistical strategy. The steps followed by the combined assessment strategy are depicted in Algorithm 7.2. The general process consists in selecting and evaluating tests in the target device until the specified coverage threshold is reached (line 2). At each iteration, a test is selected as described in Section 7.4.1 by considering how much it contributes to achieve the specified coverage, the impact of the vulnerabilities this test participates in, and the elapsed time since its last evaluation (line 3). The algorithm looks for a previous unexpired evaluation result of this test in the cache (line 4). If a result is found, it is directly used (line 5). If it is not, the object referenced by this test is searched in the cache (line 8). If the object is found (line 9), it is directly used. If it is not, the data collection process is launched over the target device (line 11). After a cache hit or the collection process itself, the evaluation process is performed (line 13) and the cache is updated with the collected object and the result (line 14). Current results are then updated in the general assessment results (line 16). Considering these results and the remaining tests to be assessed, the vulnerability list is reduced as explained in Section 7.4.1 by replacing known test values within the CNF formulas that represent such vulnerabilities (line 17). Finally, the vulnerability coverage obtained until this point is updated (line 18). The algorithm ends when the percentage of assessed vulnerabilities satisfies the specified threshold.

The proposed strategy is performed each time the Ovaldroid server considers that a security analysis needs to be made over a specific device. However, the event that potentially triggers such analysis is initiated by the client side. Indeed, a periodic *Hello* message is sent by the Ovaldroid client to the server in order to indicate its assessment availability as shown in Figure 7.8. Communication messages are always sent by the client that analyses the response of the server. The responses of the server can be to start a new security analysis, to update the client policy and parameters, nothing to do at that moment (OK status) or an error such as busy error. If a new analysis is required based on the established frequency δ, the server will respond with the appropriate message and also the first OVAL object description to collect. The client will collect the items corresponding to the specified OVAL object and will send a new message to the server with the collected OVAL items. This mechanism is based on the piggybacking technique in order to reduce the amount of network messages transmitted during the process. The server will then respond with a new OVAL object request or a flag indicating the end of the assessment process. From the client point of view, it enters in a loop while the server keeps responding *ContinueAnalysis(OVAL object)* until it receives the assessment results. The collection of objects

```
Input: CNFVulnList vulnList, Threshold threshold
Output: AssessmentResults results
$coverage \leftarrow 0$;
while $coverage < threshold$ do
    $test \leftarrow computeBestUtilityTest(vulnList)$;
    if $test$ in cache then
        $testResult \leftarrow getResultFromCache(test)$;
    else
        $object \leftarrow getObjectDescription(test)$;
        if $object$ in cache then
            $objectData \leftarrow getFromCache(object)$;
        else
            $objectData \leftarrow collectFromDevice(object)$;
        end
        $testResult \leftarrow evaluate(test, objectData)$;
        $updateCache(test, objectData, testResult)$;
    end
    $updateAssessmentResults(results, testResult)$;
    $reduceCNFVulnList(vulnList, test, results)$;
    $updateCoverage(coverage, vulnList, test, results)$;
end
```

Algorithm 7.2: Probabilistic assessment algorithm

is quite simple and only uses two HTTP methods invoked from the client side. However, powerful network management protocols such as NETCONF [63] already exist and they could be envisioned in the future as soon as their linkage with OVAL and the Android platform become more mature. As shown later in Chapter 8, we have successfully used the NETCONF protocol for performing vulnerability remediation activities over the Cisco IOS platform. In the next section, we present the results obtained from an extensive set of experiments performed with our probabilistic framework.

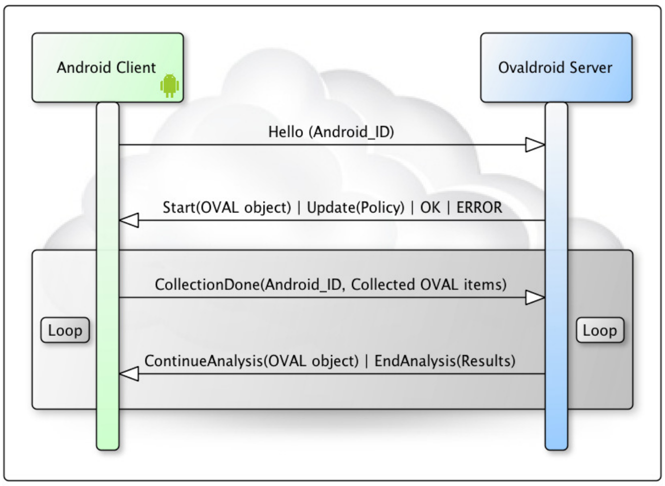

Android Client
Ovaldroid Server
Hello (Android_ID)
Start(OVAL object) | Update(Policy) | OK | ERROR
CollectionDone(Android_ID, Collected OVAL items)
Loop
Loop
ContinueAnalysis(OVAL object) | EndAnalysis(Results)

Figure 7.8 – Ovaldroid client-server interactions

7.4.3 Performance evaluation

In order to provide a computable infrastructure to the proposed approach, we have developed an implementation prototype that integrates the building blocks presented in the Ovaldroid framework. The implementation prototype is described in Chapter 9. We have also performed a deep behavioral analysis of the proposed framework through a comprehensive set of experiments. In this section we detail the experiments and the obtained results.

In order to evaluate the behavior and performance of our framework, we have performed our experiments using a regular laptop (Intel Core i7 2.20 Ghz, 8 GB of RAM, Linux kernel v.3.7.9) running the Ovaldroid server and a Samsung I9300 Galaxy S III smartphone (Quad-core 1.4 GHz, 4 GB of RAM, Android v.4.1.0) running the Ovaldroid client. The vulnerability database used within the experiments has been built taking real vulnerability descriptions for Android. In order to evaluate scalability aspects we have replicated their structure to construct more vulnerability descriptions involving two tests on average. Under a semantic perspective they represent the same vulnerability but under a technical perspective, these vulnerabilities and the involved tests, objects and states are different as they have different identifiers. Based on this methodology, we have constructed a database involving 500 vulnerability descriptions. Regarding Ovaldroid's parameters, we have experimented with several values for the vulnerability coverage λ while considering $\delta = 1$. As to the cache replacement policy, we have established an average of 3 periods before stored objects and results become expired.

We now present three different experiments that provide an insight of Ovaldroid's performance and show the feasibility of our solution. The first experiment shows how the proposed approach converges to a complete coverage of the vulnerability database across time. Indeed, one of the characteristics of Ovaldroid is the capability of distributing the load among different evaluation periods. By providing higher priority to those tests with higher utility as explained in Section 7.4.1 but simultaneously avoiding test starvation, the progression behaves as illustrated in Figure 7.9. We can observe that only covering the 33% of vulnerabilities at each period (solid blue line with crossings), the whole vulnerability database can be 100% covered at the end of the sixth period. If the vulnerability database keeps the same, following periods will re-evaluate vulnerabilities according to their impact and importance, but always providing vulnerabilities with lower utility to be evaluated as well. If new vulnerabilities become available, they will have higher priority as they have never been evaluated. This re-evaluation process smooths the load impact on the target

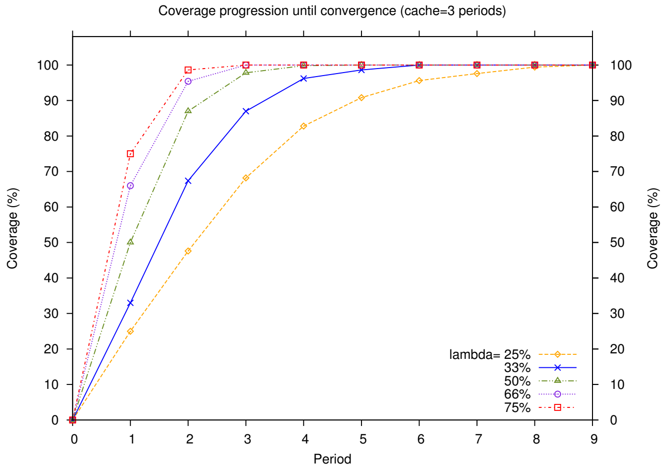


FIGURE 7.9 – Coverage convergence

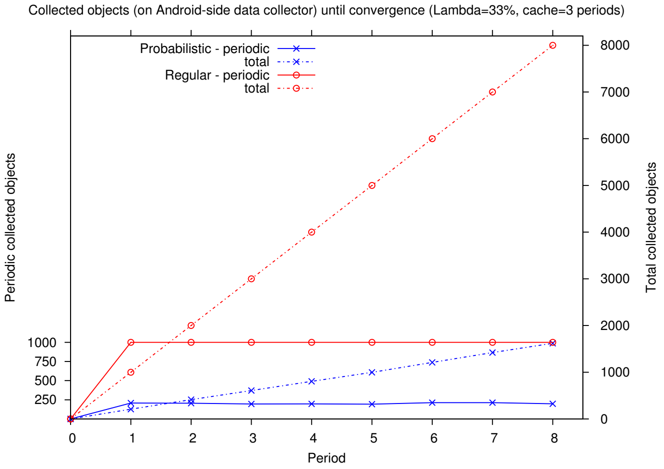


FIGURE 7.10 – Collected objects

device, produces frequent and more accurate results, and also fits its potential changing nature. By augmenting the vulnerability coverage λ, our experiments have shown a faster convergence as expected.

In order to analyze the load activity variation on the Android side, we have performed a second experiment where we analyze the object collection behavior by measuring the standard approach evaluating all vulnerabilities at once, and Ovaldroid's approach distributing the assessment activity across time. Figure 7.10 shows the observed behavior where two types of results are illustrated, namely, the number of collected objects per period (solid lines) and the total amount of collected objects (dashed lines). We can observe that while the standard approach collects 1000 objects per period (red solid line with circles), Ovaldroid's approach collects between 200 and 250 objects on average (blue solid line with crossings). This means that our approach only needs to collect approximately 25% of the objects required by the standard approach in this case, thus considerably reducing the load factor. Even though the proposed approach is slower than the standard one in terms of coverage speed, the load reduction achieved by Ovaldroid is really high and therefore, it positively contributes to the efficiency and responsiveness of the target device. The curves representing total accumulated objects show more clearly how the standard approach (dashed red line with circles) highly exceeds the interactions done by Ovaldroid with the mobile device (dashed blue line with crossings).

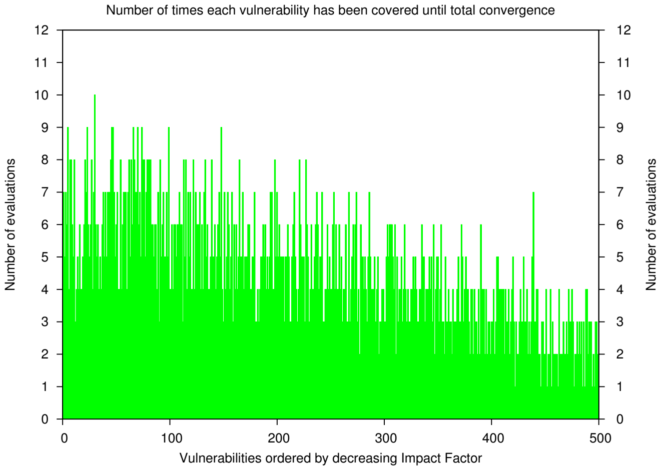


FIGURE 7.11 – Vulnerability evaluation rate

The experiments previously described consider a vulnerability dataset where each vulnerability has the same impact factor. In order to analyze the frequency with which each vulnerability is evaluated across time regarding their security impact, we have performed a third experiment depicted in Figure 7.11 involving 14 evaluation time periods. Vulnerabilities identifiers are ordered by decreasing impact factor. We can observe as expected that vulnerabilities with a higher impact factor have been evaluated more frequently than those vulnerabilities with a lower impact factor. However, vulnerabilities with a lower impact factor have been analyzed several times meaning that the model also solves the starvation problem.

7.5 Synthesis

Witnessing a strong trend to large evolving heterogeneous networks, traditional servers and personal computers are not the only targets for attackers anymore. Mobile computing has become a de facto technological standard in the modern society. We have already shown the power of autonomic computing on the management of core networks. In this chapter, we have also entered into the field of mobile computing where autonomic solutions have been barely or not at all discussed. First, we have proposed an approach that outfits mobile devices with the capability of assessing their own security exposure in an autonomous manner. A mathematical model, a functional framework, and several experiments have been presented to that end. We also have observed that outsourcing assessment activities may highly reduce the workload of mobile devices. In that context, we have presented a probabilistic approach for increasing the security of Android devices in a cost-efficient manner. A detailed mathematical model, smart strategies over an autonomous framework, and a comprehensive set of experiments support the proposed approach. As stated in Chapter 2, autonomic computing transcends the barrier of technological heterogeneity. All along this thesis, we have shown how autonomics contributes to achieve more affordable ways to manage the security of disparate increasing networks. Particularly, we have attacked core and mobile networks. However, vulnerability assessment constitutes one part of the vulnerability management process. Indeed, real autonomy can only be achieved if this process can be completed in a loop, permitting network devices to detect and remediate security exposures by themselves. This is the heart of the next chapter, where we discuss approaches for remediating host-based and distributed vulnerabilities from an autonomic perspective.

Chapter 8

Remediation of configuration vulnerabilities

Contents

8.1 Introduction 103
8.2 Background and motivations 104
8.3 Remediating device-based vulnerabilities 105
8.3.1 Vulnerability remediation modeling 105
8.3.2 The X2CCDF specification language 109
8.3.3 Extended framework for remediating device-based vulnerabilities . 111
8.3.4 Performance evaluation 113
8.4 Towards the remediation of distributed vulnerabilities 115
8.4.1 Modeling vulnerability treatments 116
8.4.2 DXCCDF, a distributed vulnerability remediation language 117
8.4.3 A strategy for collaboratively treating distributed vulnerabilities . . 119
8.4.4 Performance evaluation 121
8.5 Synthesis 123

8.1 Introduction

In the previous chapters we have presented several approaches for dealing with the assessment of configuration vulnerabilities in different dimensions, namely, device-based vulnerabilities, distributed vulnerabilities, past hidden vulnerabilities, and mobile security vulnerabilities. Autonomously assessing vulnerabilities helps to address the growing complexity of network management by providing awareness mechanisms to these convoluted and heterogeneous networked environments. However, remediation activities are still required in order to complete the vulnerability management control loop. In this chapter, we extend our previous approaches for not only identifying but also remediating vulnerabilities based on high-level policies. These policies describe vulnerable machine states as well as potential corrective activities in order to ensure safe configurations and prevent security attacks.

In order to achieve this goal, we present two complimentary approaches that aim at dealing with remediation activities of device-based and distributed vulnerabilities. First, we tackle the remediation of device-based vulnerabilities by proposing a SAT-based autonomous strategy. Using our previous logical model for OVAL vulnerability descriptions, the idea is to formulate the repository of known vulnerabilities as a logical boolean expression. Then, we use a SAT solver to identify which properties must be changed in the target system in order to eradicate all known vulnerabilities. Our second approach tackles the remediation of distributed vulnerabilities. In that context, we extend our model for specifying distributed vulnerabilities presented in Chapter 5 in order to cover distributed remediation tasks as well. We propose an infrastructure for describing distributed treatments as well as a collaborative strategy for performing corrective actions. For both approaches, we present several experiments that show the feasibility of the proposed solutions.

The remainder of this chapter is organized as follows. Section 8.2 describes important concepts related to vulnerability remediation as well as motivational issues to address this activity in the context of autonomic environments. Section 8.3 presents the proposed approach for dealing with device-based environment using a SAT-based strategy. Section 8.4 illustrates our approach for tackling the remediation of distributed vulnerabilities in an autonomous manner. Section 8.5 concludes this chapter and points out further work.

8.2 Background and motivations

The vulnerability remediation activity constitutes itself as a hard and challenging task. From a proactive perspective, it should be able to decide which potential states could be dangerous for the security of the system. In the same manner, but under a reactive perspective, effective vulnerable states should be rapidly eradicated to avoid potential attacks that could compromise the system. Finding those changes that can ensure the security of the system is also a complex activity. One single change may impact or activate other vulnerable states that were not present before the change. The same effect could occur over other system policies, in this work however, we only deal with security configuration vulnerabilities. In that context, looking for correct changes that together can provide a safe system configuration becomes an explosive combinatorial activity. This is indeed a decision problem classified as an NP-complete problem [43].

In order to cope with this problem, we propose to formalize the change decision problem as a satisfiability or SAT problem [123]. Given a boolean expression, the SAT problem consists of finding an assignment for variables such that the formula evaluates to *true*. By specifying our vulnerability knowledge source as a propositional logical formula, we fix those system properties that we cannot change and free those variables for which changes are available. We use a SAT solving engine to determine which changes have to be made to secure the system. In order to provide proactive and reactive solutions, we propose the concept of a future state. This describes how a system will look after applying a specific change. These descriptions can be used for analyzing the security impact of changes without actually changing the system. When this information is not available, we use the NETCONF protocol [63] and its notion of candidate state where changes can be applied, analyzed and rolled back if necessary.

NETCONF has been developed by IETF [84] in order to deal with network management operations and changes. NETCONF is a network configuration protocol that provides mechanisms to install, manipulate and delete the configuration of network devices. Its specification constitutes a standard, though its deployment, as well as complete vendors implementations, seem to be still in an early stage. However, very interesting works have already been presented showing evaluations of its maturity as well as diverse technical aspects [152], [157]. To the best of our knowledge, the integration of change management techniques into the vulnerability management plane constitutes a novel approach that may positively contribute to the overall security of current and future computer systems.

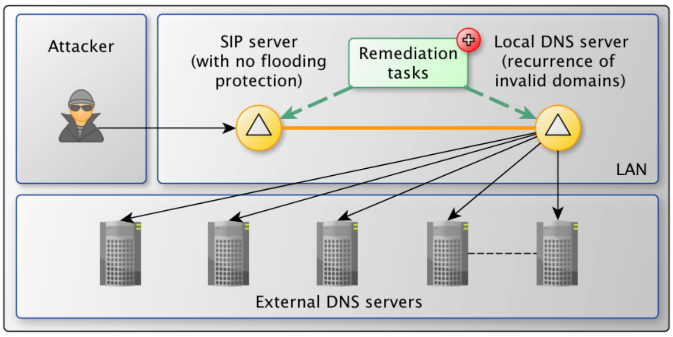


Figure 8.1 – Distributed vulnerability scenario with remediation tasks

On the other hand, remediating distributed vulnerabilities in an autonomous manner also remains a challenging problem. We have introduced in Chapter 5 a mechanism for specifying and assessing distributed vulnerabilities involving several devices simultaneously, and also covering device-based vulnerabilities as a particular case. Now, the idea is to provide a collaborative strategy for remediating both distributed and device-based vulnerabilities. The scenario presented in Figure 8.1 shows the same scenario illustrated in Chapter 5 but now considering remediation tasks as well. As a remainder, this scenario involves two devices, a SIP server with no flooding protection and a local DNS server with external unknown name resolution. Together, they constitute a distributed vulnerable state. In this situation, an attacker can perform a denial of service attack by flooding the SIP server with unresolvable domains that must be solved by a local DNS server. The local DNS server in turn is configured for solving unknown domains querying external servers, thus increasing the number of waiting requests as well as the response time for each SIP request. If at least one of the servers is not present or is not compliant with the required specific state, the distributed vulnerability has no place in the environment. In order to correct such security problem, different remediation tasks could be performed in the SIP server or in the DNS server. A key challenge is to define a strategy for determining how and by which devices the distributed vulnerability can be remediated. In the next section, we present our approach for dealing with device-based vulnerabilities. Subsequently, we describe our autonomous strategy for collaboratively remediating distributed vulnerabilities found across the network.

8.3 Remediating device-based vulnerabilities

Remediating device-based vulnerabilities constitutes a critical activity for ensuring secure network configurations. In this section, we put forward a mathematical model for performing device-based vulnerability assessment as well as remediation activities. We present a SAT-based autonomous framework able to capture these mathematical concepts and execute vulnerability management activities. The feasibility of the proposed approach is illustrated by several experiments presented at the end of this section.

8.3.1 Vulnerability remediation modeling

In order to provide sound foundations for performing vulnerability remediation tasks, we propose here a model that formalizes the main building blocks involved in the process. We also discuss the exponential nature of finding appropriate changes for securing a vulnerable system and we propose a SAT-based approach for dealing with this issue.

Specifying corrective changes

Previously in Section 5.2, we have mathematically described the main concepts involved in the OVAL language. As a remainder, we retake a few definitions in the following list.

- ◇ $H = \{h_1, h_2, \dots\}$ denotes the set of devices or systems in the network (e.g. hosts, routers).
- ◇ $P = \{p_1, p_2, \dots\}$ denotes the set of device properties in the form of unary predicates $p_i(h), h \in H$. Such predicates are used for both specifying required properties to be observed for a vulnerability to be present as well as properties the device already possesses.
- ◇ $S = \{s_1, s_2, \dots\}$ denotes the set of device states where a state s_i is used for describing in a compact manner a set of properties required to be observed over a network device as well as for describing existing specific network devices states.
- ◇ $state : H \rightarrow S \equiv$ function that takes a device $h \in H$ as input and returns its current state $s \in S$.

In the OVAL language, an OVAL definition representing a vulnerability v is actually a specific machine state $s \in S$ where the involved properties $p_i \in P$ are evaluated by OVAL tests. We therefore consider the set of known vulnerability descriptions constituting our knowledge source as $V = \{v_1, v_2, \dots, v_m\}$. As each vulnerability $v_i \in V$ can be specified as a logical formula, we can describe the whole vulnerability dataset as a disjunction of formulas as follows:

$$\phi = v_1 \vee v_2 \dots \vee v_m = \bigvee(v_i) \qquad v_i \in V \tag{8.1}$$

Considering this definition, we specify an evaluation function $\Phi : S \rightarrow Boolean$ that classifies a system $h \in H$ as vulnerable under V if and only if the assessment of ϕ over the state of h is *true*, i.e., $\Phi(state(h)) = true$. From a logical point of view, ϕ is a logical consequence of those formulas constituting the state of the device h, i.e., $state(h) \models \phi$.

In order to remediate configuration vulnerabilities, corrective changes must be performed on the target system. However, a change aimed at solving a specific vulnerability may introduce or activate other vulnerabilities. If this effect is not properly managed, this process would still expose the system to security threats. A system could be re-assessed after a change is made, and undo such modification if other vulnerabilities arise. Nevertheless, this approach does not take the big picture into account. Introduced vulnerabilities may also have potential fixes that would lead the system into a secure state. In that context, we consider the available information about known vulnerabilities and corrective tasks, as a whole. This potentially allows us to find a sequence of changes or fixes such that the final machine state is secure. Even though intermediate states in the sequence are vulnerable.

During the search of such sequence, changes could be applied on the target system and immediately assessed in a backtracking fashion. However, the ability to project the consequences of a vulnerability fix or change, i.e., how the affected part of the system will look after applying such change, allows us to analyze change sequences without actually changing the target system. This is one of the cornerstones of our approach. In order to formalize this concept, our remediation model involves the following definitions:

- ◇ $C = \{c_1, c_2, \dots\}$ denotes the set of changes or corrective actions applicable over network devices.
- ◇ $change : H \times C \rightarrow H \equiv$ function that takes a device $h \in H$ as input and returns the same device h after performing a change $c \in C$ that produces an observable change on its state. The following property holds in the considered model: $state(h) \neq state(change(h, c))$, $\forall h \in H, \forall c \in C$.
- ◇ $future : C \rightarrow S \equiv$ function that takes a change $c \in C$ as input and returns a state $s \in S$ that projects the affected characteristics of a system after applying the change c.

- $\Pi : S \times S \rightarrow S \equiv$ function that takes a projected system state $s_1 \in S$ and a machine system state $s_2 \in S$ as input and returns s_2 updated with the properties of s_1.

In order to analyze the impact of a change c_i over a system h, the $future$ and projection Π functions are combined with the Φ formula to check present vulnerabilities as follows:

$$\Phi(\Pi(future(c_i), state(h)) \qquad c_i \in C, h \in H \tag{8.2}$$

From a technical point of view, the $future$ function can be intuitively understood as observing the resulting state of a change over a rollback-capable system. However, this can be also achieved by considering a specification mechanism for describing those system properties that will be modified once the change is applied. Our approach considers both techniques, which are discussed in the following sections. In light of these definitions, we define a sequence of changes ω as follows:

$$\omega = c_1 \circ c_2 \circ \ldots \circ c_n \qquad c_i \in C \tag{8.3}$$

We say that ω constitutes a secure sequence of changes for a system $h \in H$ if and only if $\Phi(\omega(h)) = false$. Finding such a sequence for different system states and contexts constitutes an NP-complete problem as explained in the following section.

Addressing complexity of change sequence analysis

Each time a single change is made to fix a specific vulnerability, some system properties are naturally modified and therefore, the state for the next corrective change in the sequence is modified. In that context, the order and the combination of distinct changes for each vulnerability induce several different possible combinations. This issue falls into a family of problems called NP-complete where no solution in polynomial time is known. In this section, we first present an illustrative example that shows the exponential nature of finding a suitable change sequence ω and then we formalize our approach as a satisfiability (SAT) problem.

Let $h \in H$ be a target device where $s \in S$ constitutes its current state as follows: $s = state(h) = \{p_1, p_2, p_3, p_4\}$, meaning that properties p_1, p_2, p_3 and p_4 are present on the system. Let us also consider a vulnerability database V and a vulnerability fix database C_V as follows:

$$V \equiv \{v_1 = p_1 \wedge p_2 \wedge p_3,\;\; v_2 = (\neg p_1 \vee \neg p_2) \wedge p4,\;\; v_3 = \neg p_1 \wedge p_3 \wedge \neg p_4\} \quad p_i \in P, v_i \in V \tag{8.4}$$

$$C_V \equiv \{c_{1a} \mapsto \neg p_1,\; c_{1b} \mapsto \neg p_2,\; c_2 \mapsto \neg p_4\} \qquad p_i \in P, c_i \in C \tag{8.5}$$

This example is based on three real Cisco IOS vulnerabilities identified by *CVE-2008-3812*, *CVE-2008-3798* and *CVE-2008-3821* respectively [111]. Within our model, properties are mapped to the following propositions:

- $p_1 \equiv$ IOS firewall is enabled.
- $p_2 \equiv$ Deep Packet Inspection (DPI) is enabled.
- $p_3 \equiv$ HTTP server is enabled.
- $p_4 \equiv$ SSL/TLS is enabled.

As explained before, vulnerabilities are represented as specific machine states that are specified by logical formulas. For instance, vulnerability v_1 requires p_1, p_2 and p_3 to be active for the vulnerability to be present. Within the set of available changes C_V, c_{1a} and c_{1b} are alternatives changes for fixing vulnerability v_1 while c_2 constitutes the only remediation action for vulnerability v_2. No fix action is available for vulnerability v_3. In this scenario, it can be observed that

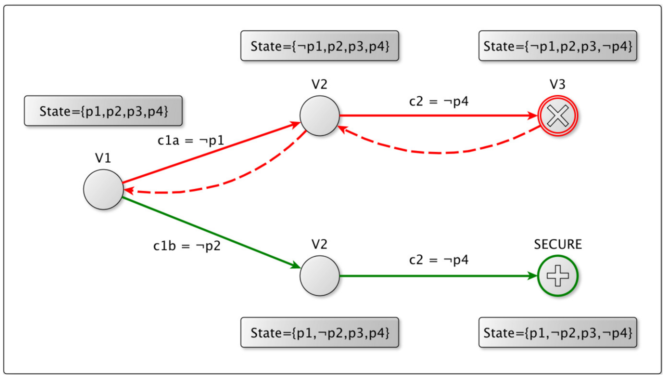


FIGURE 8.2 – Change sequence search example

the vulnerability v_1 is a semantic consequence of the properties present in the system, which are compactly represented as s. This means that v_1 is logically *true* under these hypothesis. This fact is represented by a node labeled v_1 in the graph illustrated in Figure 8.2. Beginning at this node, a search for a secure change sequence is launched. Two alternative changes are available for fixing the vulnerability v_1. Change c_{1a} deactivates the property p_1, changing the system state to $s = \{\neg p_1, p_2, p_3, p_4\}$. Under these conditions, v_2 becomes present in the system. However, a fix for v_2 exists so the change c_2 is applied. Such modification brings the state of the system to $s = \{\neg p_1, p_2, p_3, \neg p_4\}$ activating the vulnerability v_3. As no remediation action is available for v_3, this change sequence is considered as invalid. Backtracking to the beginning, fix c_{1b} is applied activating again the vulnerability v_2. Once again, change c_2 is applied but this time, the combination of remediation actions leaves the system as $s = \{p_1, \neg p_2, p_3, \neg p_4\}$ successfully eradicating all vulnerabilities. It can be noticed from this example that a naive approach for assessing all possible combinations cannot provide solutions in polynomial time.

This problem constitutes a decision problem that relies on changes being applied to ensure a secure system state. In our model, we have a boolean expression ϕ that indicates the vulnerable nature of a system when it is evaluated as *true*. Considering $\psi = \neg\phi$, we can say that a target system h is secure, or not vulnerable, when ψ is *true*. Our problem therefore consists of finding such a propositional assignment that makes the ψ formula *true*. In computational complexity theory, this is known as a satisfiability or SAT problem. Given the current state of a target system, ψ can be instantiated and evaluated, and changes can be understood as actions that can assign a specific value to the properties involved in the formula. Considering the proposed example, the ψ formula states that none of the known vulnerabilities in V can occur:

$$\psi = \neg(p_1 \wedge p_2 \wedge p_3) \wedge \neg((\neg p_1 \vee \neg p_2) \wedge p_4) \wedge \neg(\neg p_1 \wedge p_3 \wedge \neg p_4) \tag{8.6}$$

Because we usually only know a small set of actions to remediate vulnerabilities, not every property is likely to be changed. In that context, we need to find solutions for ψ respecting those property values that are not changeable, i.e., there are no available actions for modifying their states. This in turn reduces the search space. Within our example, property p_3 is not changeable, and therefore, it must take its current system value, *true*, giving the following expression:

$$\psi = p_1 \wedge \neg p_2 \wedge \neg p_4 \tag{8.7}$$

The solution in this case is trivial. It states that our target system can be classified as secure only if those properties not matching the current state are changed, i.e., p_2 and p_4. Therefore, changes c_{1b} and c_2 must be applied, deactivating or updating the DPI engine as well as the SSL/TLS service. In the worst case, these changes could consist in deactivating or uninstalling the services themselves, nevertheless, the idea is that changes might generally patch or update to newer versions that do not present the characteristics involved in the vulnerabilities. The proposed example constitutes a simple scenario aimed at showing the insight of our general approach. However, versions can also be modeled as properties, thus enriching the expressiveness of our vulnerability descriptions and the accuracy of our solutions.

8.3.2 The X2CCDF specification language

We have designed the X2CCDF language, built on top of the XCCDF language [167], in order to express the future state of target systems after applying vulnerability remediation actions. In this section we present the core building blocks of X2CCDF and explain its use in the context of change analysis.

Specifying corrective changes for remediating known vulnerabilities in such a way machines can interpret them is crucial to achieve higher levels of security automation. The XCCDF language provides great support for this point by allowing referenced vulnerability descriptions expressed with the OVAL language and linking them to rules that can be applied to correct the specified security weaknesses. Nevertheless, applying changes blindly, without actually analyzing the impact of such changes, does not ensure a secure corrective process. In that context, we introduce the idea of future or post-action states. Future states are intended to describe how the system will look like after applying a specific change. They do not describe the entire system but only the components affected by changes. In light of this and being designed for describing computer machine states, the OVAL language suitably fits for representing future machine states. Its interpretation however changes, i.e., in the general case, data is usually collected and compared against OVAL vulnerability descriptions. Within our approach, collected data is mixed with OVAL-based future states descriptions and compared against OVAL vulnerability descriptions.

The ability to express this concept in a machine-readable manner provides new capabilities for analyzing different ways of modifying and correcting computer systems without actually changing them. The main objective in describing future states within X2CCDF is to complement management protocols such as NETCONF [63] by allowing the projection of changes using the current system state combined with future states of known remediation actions. While XCCDF provides means for specifying remediation actions when specific states are detected, X2CCDF extends its capabilities by specifying also the consequences of such actions when performed by means of the OVAL language.

Listing 8.1 presents an illustrative example where X2CCDF is used for specifying the two alternative actions for vulnerability v_1 as described in Section 8.3.1. For the sake of clarity, we have omitted some XCCDF components that should be present in valid instances. Within this example, only one XCCDF group of management rules is defined (lines 3-5). The only referenced rule *v1-treatment* is declared below (lines 6-14). X2CCDF extends XCCDF by considering a new building block named *complex-Rule* under the *x2ccdf* namespace. This extension, permits the specification of a boolean expression involving alternative actions (lines 10-13) that can change different properties of a specific vulnerability (lines 7-9). Corrective changes from the model, c_{1a} (lines 15-20) and c_{1b} (lines 21-26), are described using standard XCCDF rules. However, the semantics of these rules express the future or post-action state of each change. While the particular actions to perform are specified inside the *fix* tag (lines 16 and 22 respectively), the *check* tag under the *x2ccdf* namespace serves as a semantic indicator for automated interpreters. It is a common practice to use scripts in XCCDF. It should be noticed that the example in Section 8.3.1 only deals with atomic changes and that is why the *OR* operator appears in Listing 8.1. However, the logical composition of changes constitutes an important issue to address

```
<cdf:Benchmark id="X2CCDF-test-1" xmlns:x2ccdf="..." xmlns:cdf="..."...>

 <cdf:title> X2CCDF example </cdf:title>
 <cdf:Group id="vulnerability-treatment-with-future-state" selected="1">
    <cdf:requires idref="v1-treatment"/>
 </cdf:Group>

 <x2ccdf:complex-Rule id="v1-treatment" selected="1" check="1">
   <x2ccdf:check system="http://oval.mitre.org/XMLSchema/oval">
     <x2ccdf:check-content-ref href="iosDefns.xml" name="oval:mitre:def:5302"/>
   </x2ccdf:check>
   <x2ccdf:criteria operator="OR">
     <x2ccdf:criterion idref="v1-fix-1a" check="1"/>
     <x2ccdf:criterion idref="v1-fix-1b" check="1"/>
   </x2ccdf:criteria>
 </x2ccdf:complex-Rule>

 <cdf:Rule id="v1-fix-1a" selected="1">
   <cdf:fix> ./disableFirewall.sh </cdf:fix>
   <x2ccdf:check system="http://oval.mitre.org/XMLSchema/oval">
     <x2ccdf:future-content-ref href="iosFuture.xml" name="x2ccdf:inria:def:1"/>
   </x2ccdf:check>
 </cdf:Rule>

 <cdf:Rule id="v1-fix-1b" selected="1">
   <cdf:fix> ./disableDPIEngine.sh </cdf:fix>
   <x2ccdf:check system="http://oval.mitre.org/XMLSchema/oval">
     <x2ccdf:future-content-ref href="iosFuture.xml" name="x2ccdf:inria:def:2"/>
   </x2ccdf:check>
 </cdf:Rule>

</cdf:Benchmark>
```

Listing 8.1 – X2CCDF example

that X2CCDF already supports using the *AND*, *OR* and *NOT* operators. This point has been already scheduled for future work.

In order to describe future or post-action states, we also use the OVAL language. However, its interpretation is different since we are comparing OVAL states against OVAL states, and not specific collected information (OVAL system characteristics) against OVAL states. The main idea is as follows. Each change is represented as a pair of OVAL object and OVAL state. The OVAL object represents the component over which the specific change is applied. The OVAL state represents the characteristics the object will present after applying this specific change. Finally, the impact of a change can be analyzed by looking for vulnerabilities which involve OVAL tests using the same OVAL object as the specified change. Affected OVAL tests are evaluated by comparing their OVAL states against the future OVAL-based state involved in the change. Other OVAL tests involved in the affected vulnerabilities are evaluated following the standard process, i.e., OVAL system characteristics against OVAL states.

While specifying the modifications that a change will have on a target system, the attributes inside an OVAL state are used to express the characteristics that the OVAL object will present. For instance, if a change is designed to change the version of a Cisco IOS system, the OVAL object will be the *version_ object* while the *version_ state* will contain the new version value, e.g., 12.4. Comparing instances of OVAL simple datatypes, such as integers and booleans, does not present difficulties. This can be done in the same way OVAL characteristics are compared against OVAL states. However, in future states, regular expressions can be utilized for specifying certain values inside information blocks that are a priori unknown. An example of this could be to look for a particular configuration line inside the running configuration file of a Cisco device. In that case, we need to potentially compare a regular expression against another regular expressions within OVAL states. According to the principles established in automata theory, the intersection of two regular languages e_1 and e_2 is a regular language e_3 [78]. Therefore, we can compute whether $e_1 \subseteq e_2$ by verifying if $e_1 = e_1 \cap e_2$. By operating over these regular expressions, we can say if a projected state might match the expressions present in the vulnerability descriptions.

Within our experiments, we have used the Greenery tool to support these operations on regular expressions [76].

8.3.3 Extended framework for remediating device-based vulnerabilities

In order to assess and remediate device-based vulnerabilities over computer systems, we propose an autonomous framework called VMANS which is presented in this section. We explain its architecture as well as the underlying strategy in charge of orchestrating the overall vulnerability management process.

Architecture overview

The proposed architecture, illustrated in Figure 8.3, comprises two independent processes, namely, one process for maintaining logical representations of OVAL vulnerabilities descriptions up-to-date, and a second process for performing vulnerability management activities. The first process is in charge of monitoring the OVAL vulnerability descriptions database (step I) and converting new vulnerability descriptions into equivalent boolean expressions when they become available (step II). Independently, a second process is in charge of dealing with vulnerabilities, which is orchestrated by the vulnerability manager component. At step 1, it communicates with the OVAL analyzer in order to launch the assessment process. The analyzer consumes OVAL vulnerability descriptions from the repository at step 2 and collects the required data from those devices under control at step 3. Once the assessment is performed, the analyzer sends the results back to the vulnerability manager. If the system is found to be vulnerable, the vulnerability manager analyzes the available remediation descriptions at step 4 and correlates them with the properties that can be changed in the target system. Considering the current system state and the available changeable properties, the SAT solver engine is used at step 5 to decide which changes must be applied in order to secure the system. At step 6, the SAT solver uses a logical

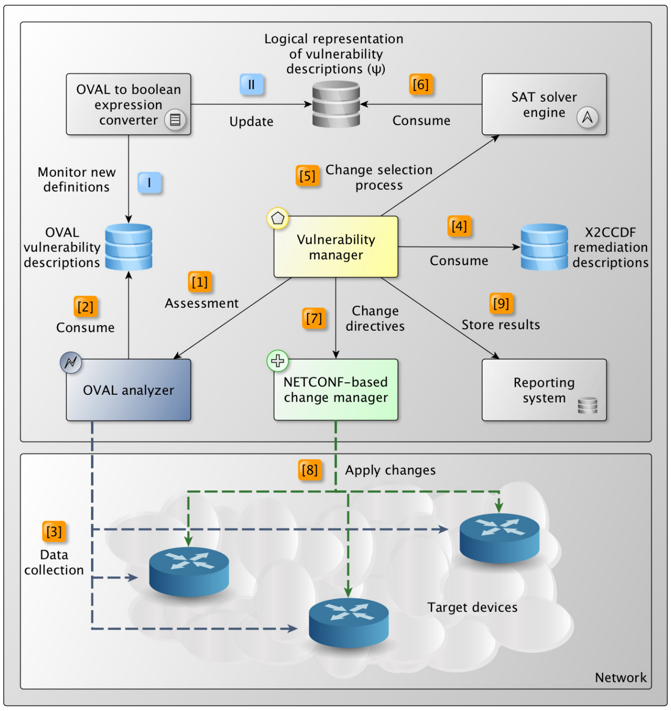

Logical representation of vulnerability descriptions (ψ)
OVAL to boolean expression converter
II
Update
[6]
Consume
SAT solver engine
Monitor new definitions
I
[5] Change selection process
OVAL vulnerability descriptions
Vulnerability manager
[4]
Consume
X2CCDF remediation descriptions
[1]
Assessment
[2]
Consume
[7]
Change directives
[9]
Store results
OVAL analyzer
NETCONF-based change manager
Reporting system
[8] Apply changes
[3] Data collection
Target devices
Network

Figure 8.3 – VMANS high-level architecture

representation ψ, such as the one illustrated in Eq. 8.7, specifying that none of the vulnerabilities can occur. A solution provided by the SAT solver indicates which properties must be changed in the system to present a secure state. The vulnerability manager interprets this information and sends specific directives to the NETCONF-based change manager subsystem at step 7 in order to effectuate these changes. At step 8, the NETCONF protocol is used to communicate and perform the specified changes on the target system. Finally, the obtained results are sent to the reporting system at step 9.

Vulnerability management strategy

In order to autonomously deal with vulnerabilities, the proposed strategy illustrated in Figure 8.4 is a closed control loop where three classes of events may potentially trigger vulnerability management activities. These activities can happen when new vulnerability or remediation descriptions become available and when the system presents changes that may compromise its security. In that context, an event monitoring component is in charge of observing these events and triggering the vulnerability management process when required. Once this process has been launched, vulnerabilities affected by the event that has triggered the process are computed (step 1). Depending on the case, these vulnerabilities can be recently added vulnerability descriptions, or vulnerabilities referenced by new remediation descriptions, or known vulnerabilities involving components that have been changed on the target system. Afterwards,

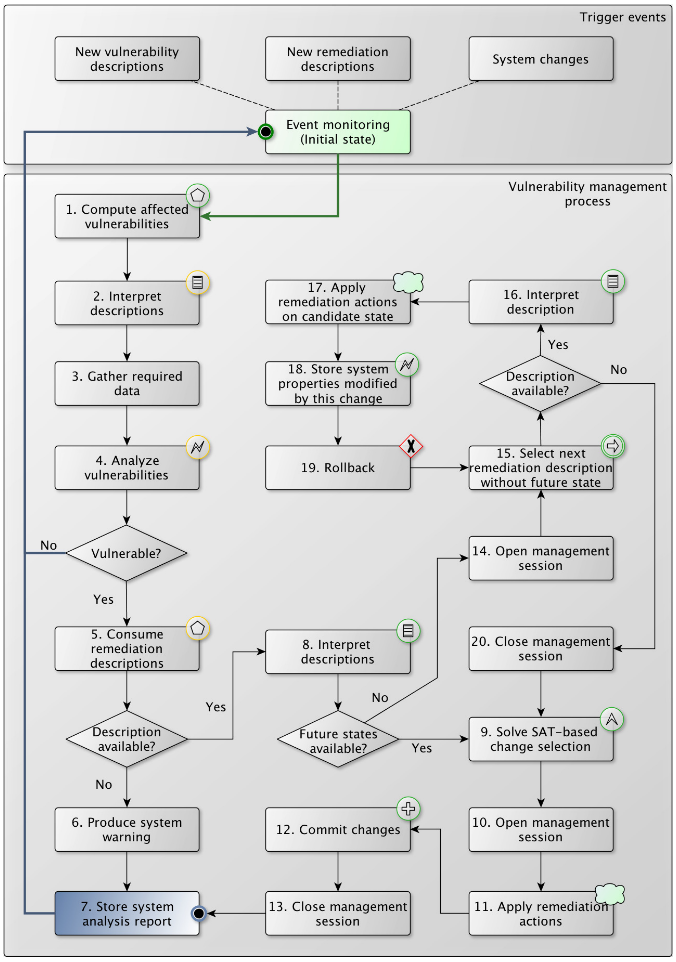

Trigger events
New vulnerability descriptions
New remediation descriptions
System changes
Event monitoring (Initial state)
Vulnerability management process
1. Compute affected vulnerabilities
2. Interpret descriptions
3. Gather required data
4. Analyze vulnerabilities
Vulnerable?
No
Yes
5. Consume remediation descriptions
Description available?
Yes
No
6. Produce system warning
7. Store system analysis report
8. Interpret descriptions
Future states available?
No
Yes
9. Solve SAT-based change selection
10. Open management session
11. Apply remediation actions
12. Commit changes
13. Close management session
14. Open management session
15. Select next remediation description without future state
Description available?
Yes
No
16. Interpret description
17. Apply remediation actions on candidate state
18. Store system properties modified by this change
19. Rollback
20. Close management session

FIGURE 8.4 – VMANS control loop

these vulnerability descriptions are evaluated on the system (steps 2-4). If the system is not found to be vulnerable, the process ends and returns to the initial monitoring state. If it is vulnerable, then available remediation descriptions are consumed (step 5). If no remediation is available for treating the security issues, then a system warning is produced (step 6) and an analysis report is created and stored (step 7) ending the process and returning to the initial state. Otherwise, remediation descriptions are analyzed (step 8). In the case every remediation description provides a specification of the future state after applying the involved changes, the process continues with the change selection process (step 9). Considering the current properties present in the target system and the properties that can potentially change by applying the available vulnerability treatments, a SAT solver is used to provide a logical assignment of every property related to any vulnerability to ensure a secure system state. Once the changes have been identified, they are applied within a NETCONF session (steps 10-13) and an analysis report is generated (step 7) going back to the initial state. When a remediation description does not specify a future state, its impact is empirically evaluated by applying the involved changes on a candidate state of the target system (steps 14-19). To do so, the candidate state feature included in the NETCONF specification is considered. For each remediation description under these circumstances, involved changes are applied (step 17), modified properties are collected and stored (step 18) and finally modifications are rolled back (step 19). When the loop is finished, the NETCONF session is closed (step 20) and the process continues normally with the change selection stage (step 9) as described before.

8.3.4 Performance evaluation

In order to evaluate the proposed approach, we have developed an implementation prototype able to perform the main activities shown in the VMANS framework. We have also performed a deep behavioral analysis using the Cisco IOS platform as a case study. In this section we illustrate the performed experiments and the obtained results. The model of Cisco routers used is c3725 with IOS version 12.4. This model implements a subset of NETCONF operations that permits the execution of basic configuration management activities. Cisco routers have been emulated using GNS3 [75] over a regular laptop (2 Ghz Intel Core i7 with 8GB RAM). The OVAL analyzer is an extension of XOvaldi [26] for Cisco IOS. OVAL vulnerability descriptions have been taken from the public OVAL repository [117]. We have used the SAT solver engine provided by the *Aima* project [6]. Operations between regular expressions for analyzing future states are performed with the Greenery tool [76]. We have used and slightly modified the Netconf4J project library [107] to communicate with Cisco routers via NETCONF.

In order to analyze the scalability of our framework, we have performed several experiments involving vulnerability representations as boolean expressions, SAT solving analysis time and behavioral aspects of the NETCONF protocol over Cisco. Our first experiment, illustrated in Figure 8.5, shows the behavior of VMANS while dealing with vulnerability logical representations. In the general case, SAT solvers consume boolean expressions in conjunctive normal form (CNF). If the input formula is not in CNF, SAT solvers transform it internally. In that context, we have measured the time required to load standard logical representations into memory (red solid line) as well as their transformation to CNF (blue dashed line). We have repeated this measurement while varying the amount of vulnerability descriptions. When all the OVAL descriptions for IOS are considered (around 140), their representations are loaded in 53 milliseconds while their transformation to CNF takes 7.5 seconds approximately. We have observed a stable behavior for both activities in the general case as shown by the first derivatives depicted in the inner graph of the figure.

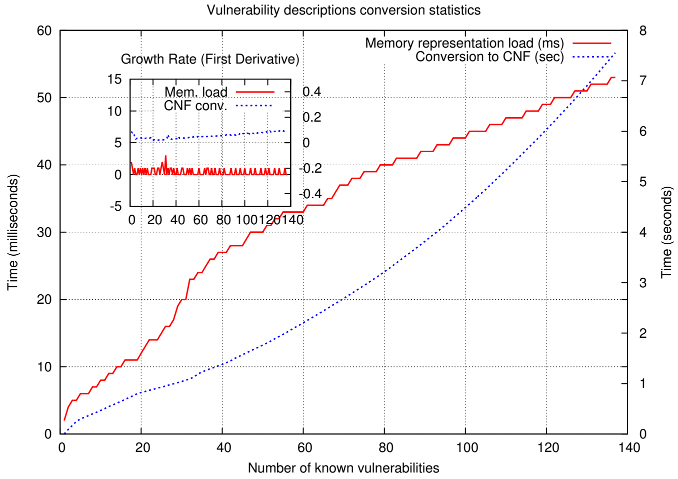


FIGURE 8.5 – Vulnerability conversion statistics

One of the critical points in the vulnerability remediation process is the change selection activity. We have analyzed the SAT solving time for different scenarios as shown in Figure 8.6. We have evaluated the same system with one, three, five and ten active vulnerabilities each time, while varying the amount of vulnerability descriptions in the database. In addition, a set of available changes has been provided to the framework to detect which corrective actions must be performed. In all cases, we have observed a linear behavior as illustrated in the inner graph of the figure, taking around 2 seconds in average to provide the answers for the whole dataset. Often, the SAT solving time depends on the nature of the equations being solved. The observed behavior is partially supported by the fact that the sets of properties (OVAL tests) involved in the IOS vulnerability descriptions are mostly disjoint. Thus, the SAT process is faster because each part of the formula does not impact on the other clauses. In addition, we have observed two interesting phenomena. The first one is that the SAT solving time (around 2 seconds), which includes a CNF transformation process, is faster than the CNF transformation time depicted in Figure 8.5 (around 7.5 seconds). After investigating this behavior, we have realized that only changeable properties participate actively in the SAT solving process. The remaining properties take their truth values from the system, so they are fixed and ignored, making the internal CNF

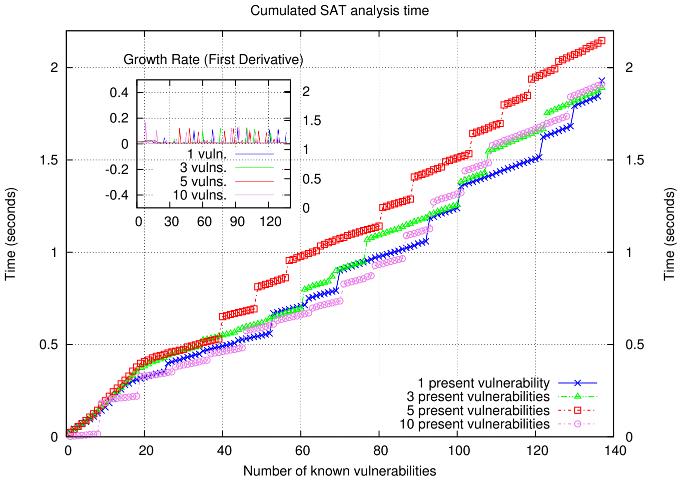


FIGURE 8.6 – SAT solving analysis time for change detection

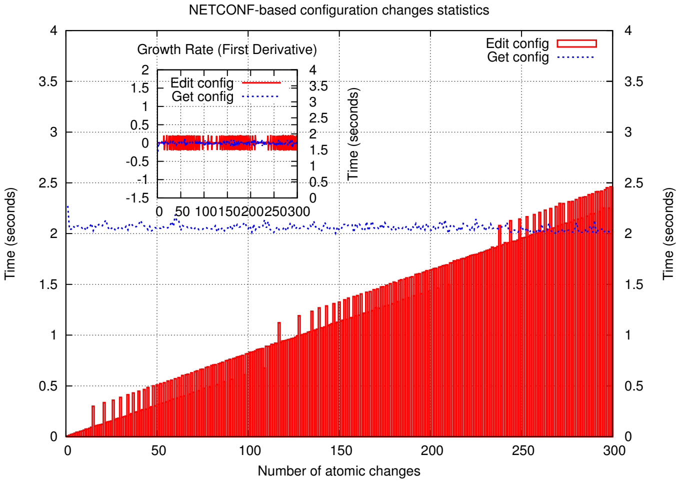


Figure 8.7 – NETCONF-Cisco statistics

transformation faster. The second phenomenon is that the curve depicting the system with ten present vulnerabilities (violet dashed lines with rounded points) has lower values that the one with five vulnerabilities (red dashed line with square points). Sometimes, more available changes facilitates the search of the SAT solver though this fact also leaves more free assignable variables increasing the search space. Even though this depends on the mechanisms used by the SAT solver, the general behavior for remediating IOS vulnerabilities has been observed to be linearly stable.

Finally, we also have measured the time required to query and perform atomic changes over Cisco IOS via NETCONF. A complete NETCONF session for getting the current configuration in our scenario, including network delays, takes around 2 seconds in average (blue dashed line). NETCONF also allows us to perform a set of various changes in one single session. We have varied the size of this change set from 1 to 300 as shown in Figure 8.7 (red bars). We have observed that the time grows linearly, as shown in the inner graph, and that it only requires about 2.5 seconds for performing 300 changes. Considering the overall behavior, these sets of experiments have shown the scalability of the proposed strategy in our context, in terms of representation conversion time, SAT solving time and NETCONF performance.

8.4 Towards the remediation of distributed vulnerabilities

In the previous section we have presented an approach for individually remediating vulnerabilities in network devices. However, there also exist situations in which security issues become visible when an upper view of the network is taken. This global perspective may show network weaknesses that otherwise could not be detected. In that context, the purpose of describing and detecting distributed vulnerabilities is to provide support for increasing network security awareness as a whole. In Chapter 5 we have presented the conceptualization of distributed vulnerabilities as well as a mathematical modeling and a framework for detecting them. In this section, we present an approach for supporting the remediation of distributed vulnerabilities. We put forward a mathematical model that formally supports the remediation activity, as well as a language for expressing remediation tasks. Finally, we propose an extension to the framework presented in Chapter 5 for remediating distributed vulnerabilities, and perform different experiments to analyze the feasibility of our solution.

8.4.1 Modeling vulnerability treatments

In this section we formalize configuration vulnerability treatments by extending and enhancing our previous mathematical presented in Chapter 5. A distributed vulnerability is defined as a set of conditions over two or more network devices that if observed simultaneously, a potential threat is present on that network. It is important to remark that the required conditions to be observed over a specific device do not necessarily constitutes a complete device-based vulnerability description.

Specifying distributed treatments

As explained in Chapter 5, a distributed vulnerability is mathematically defined as a compliant projection of the pattern (P^H, P^R) over the network (H, R) that makes *true* the predicate defined in Equation 5.1. As a reminder, the predicate is defined as follows:

$$DV(H,R) \equiv \exists(h_1,...,h_n)\ (s_1(h_1) \wedge ... \wedge s_n(h_n) \wedge r_1(h_i,...,h_j) \wedge ... \wedge r_v(h_k,...,h_l))$$

We recall that this predicate expresses the evaluation of a distributed vulnerability DV based on the pattern (P^H, P^R) over a generic network (H, R). Based on this modeling, we consider a distributed treatment DT as a body of tasks performed over a set of network devices that introduces configuration changes in order to eliminate the security weakness described by a specific distributed vulnerability DV. In order to formally define what a distributed treatment is, we extend the description model explained in Chapter 5 by defining the following domains:

- ◇ $A = \{a_1, a_2, \dots\}$ denotes the set of actions applicable over network devices.
- ◇ $T = \{t_1, t_2, \dots\}$ denotes the set of tasks applicable over network devices. A task t_i is a logical combination of actions and its logical value is computed based on the successful application of each action. The set T is inductively defined as follows:
  - i if $a_i \in A$, then $a_i \in T \qquad (i \in N)$
  - ii if $\alpha, \beta \in T$, then $(\alpha \diamond \beta) \in T \qquad \diamond \in \{\wedge, \vee\}$

In order to define the application of remediation tasks over the network, we specify the following set of *core functions*:

- ◇ $state_H : H \to S \equiv$ function that takes a device $h \in H$ as input and returns its current state $s \in S$.
- ◇ $state_R : R \to 2^S \equiv$ function that takes a network relationship $r \in R$ as input and returns a set with the current state $s_i \in S$ of each involved network device $h_i \in H$ in the relationship.
- ◇ $action : H \times A \to H \equiv$ function that takes a device $h \in H$ as input and returns the same device h after performing an action $a \in A$.
- ◇ $task_H : H \times T \to H \equiv$ function that takes a device $h \in H$ as input and returns the same device h after performing a task $t \in T$ that produces an observable change on its state. This means that at least one action $a_i \in A$ must introduce a change that cannot be rolled back by any other action in the task nor a combination of them. The following property holds in the considered model: $state_H(h) \neq state_H(task_H(h,t)), \quad \forall t \in T, \forall h \in H$.
- ◇ $task_R : R \times T \to R \equiv$ function that takes a network relationship $r \in R$ as input and returns the same network relationship r after performing a task $t \in T$ over its member

devices. Based on the definition of T, is can be noticed that the task t will produce an observable change on its state and that the following property also holds: $state_R(r) \neq state_R(task_R(r, t)), \quad \forall t \in T, \forall r \in R$.

- ◇ $T^H = \{t_1^H, ..., T_n^H\}$ denotes the body of available tasks for performing over network devices where each task t_i^H is semantically related to a specific state s_i. Usually, the following equation can hold $|T^H| < |P^H|$, meaning that treatment tasks are not always available for correcting certain device states.
- ◇ $T_R = \{t_1^R, ..., t_v^R\}$ denotes the body of available tasks for performing over network relationships where each task t_i^R is semantically related to a specific relationship r_i. Usually, the following equation can hold $|T^R| < |P^R|$, meaning that treatment tasks are not always available for correcting certain network relationships.

We therefore define a distributed treatment DT as the compliant application of (T^H, T^R) over the network (H, R) that eliminates every possible matching projection of the pattern (P^H, P^R) over (H, R). Under a logical perspective, $DT(H, R)$ is defined as the disjunction of task applications $\Pi(T^H, T^R)$ over each potential combination of devices and network relationships (H', R') performing the roles required by the distributed vulnerability DV as follows:

$$\begin{aligned} DT(H,R) = task_H(h_1, t_1^H) \vee \ldots \vee task_H(h_n, t_n^H) \vee task_R(r_1, t_1^R) \vee \ldots \vee task_R(r_v, t_v^R) \\ \forall H' = \{h_1, ..., h_n\} \subseteq H, R' = \{r_1, ..., r_v\} \subseteq R, \text{ such that } DV(H', R') \text{ holds.} \end{aligned} \tag{8.8}$$

Changes done for correcting different instances (H', R') of the distributed vulnerability must not shadow those remediation actions performed for any other observed vulnerable instances of DV. Therefore, $\neg DV(H, R)$ must hold after the DT application. In the next section, we present an XML-based language able to capture these mathematical constructions.

8.4.2 DXCCDF, a distributed vulnerability remediation language

We have conceived the DXCCDF language, built on top of XCCDF, as a means for expressing vulnerability treatments in a machine-readable manner. XCCDF rules allow to specify remediation information that can be used by automated systems to perform corrective actions when specific states are detected. These states can be specified by languages such as OVAL and DOVAL. DXCCDF extends XCCDF by considering a new building block named *complex-Rule* under the *dxccdf* namespace. This extension provides the ability to specify a boolean expression involving all the potential tasks that can be performed for remediating a specific machine state. Figure 8.8 depicts the mapping between the main components involved in the mathematical model and their representatives constructs within the DXCCDF language. An action a_i can be

The model and the DXCCDF language

Model block	**Insight**	**Applies to**	**Expressed with**
Action a_i	chmod 644 passwd	Property p_i	XCCDF rule
Task t_i	$a_1 \vee (a_2 \wedge a_3)$	State s_i	DXCCDF complex rule
Distributed treatment DT	$t_1 \vee t_2 \vee \ldots \vee t_k \vee \ldots$	Distributed vulnerability	DXCCDF complex rule
Distributed treatments	$\{DT_1, DT_2, \ldots\}$	$\{DV_1, DV_2, \ldots\}$	XCCDF group of complex rules

FIGURE 8.8 – Mapping the model into the DXCCDF language

understood as a simple operation, i.e. a shell command, that is performed for changing a system property p_i. This property can be checked and remediated using an XCCDF rule. A task t_i is a combination of actions in the form of a boolean expression intended to correct a specific state s_i. Tasks are represented by means of DXCCDF complex rules. A distributed treatment DT is also represented using DXCCDF complex rules and they are finally put together into XCCDF groups.

We now put forward an illustrative DXCCDF example illustrated in Listing 8.2 where a distributed treatment is specified in order to provide directives for remediating the distributed vulnerability previously depicted in Figure 8.1. For the sake of clarity, we have omitted some XCCDF components that should be present in valid instances. Within this document, the group for vulnerability treatments is selected for evaluation containing only one treatment. The construct DXCCDF complex rule represents the distributed treatment itself. A DXCCDF complex rule allows to refer a check system for assessing the distributed vulnerability under analysis by means of a DOVAL reference, and also to specify a logical criterion involving both XCCDF rules and DXCCDF complex rules. In the proposed scenario there exist two involved roles, the SIP server and the DNS server. Within the DXCCDF example, one remediation task has been defined for each role, namely, T^{SIP} and T^{DNS}. The task T^{SIP} is in turn a non-atomic task since two corrective tasks, namely, T^{SIP}_{yum} and T^{SIP}_{web}, can be alternatively performed. Thus, the distributed treatment expressed by the DXCCDF complex rule corresponds to the logical expression $(T^{SIP}_{yum} \vee T^{SIP}_{web} \vee T^{DNS})$ where each one of the referenced standard XCCDF rules are defined in the

```xml
<cdf:Benchmark id="sip-dos-test-1" xmlns:dxccdf="..." xmlns:cdf="..."...>

  <cdf:title> DXCCDF example </cdf:title>
  <cdf:Group id="vulnerability-treatments" selected="1">
     <cdf:requires idref="dv1-treatment"/>
  </cdf:Group>

  <dxccdf:complex-Rule id="dv1-treatment" selected="1" check="1">
    <dxccdf:check system="http://doval.inria.fr/XMLSchema/doval">
      <dxccdf:check-content-ref href="dvDefns.xml" name="doval:inria:def:1"/>
    </dxccdf:check>
    <dxccdf:criteria operator="OR">
      <dxccdf:criteria operator="OR">
        <dxccdf:criterion idref="flooding-protection-yum" check="0"/>
        <dxccdf:criterion idref="flooding-protection-custom" check="0"/>
      </dxccdf:criteria>
      <dxccdf:criterion idref="stop-bind-daemon" check="0"/>
    </dxccdf:criteria>
  </dxccdf:complex-Rule>

  <cdf:Rule id="flooding-protection-yum" selected="1">
    <cdf:fix> yum install asterisk-sip-dos-patch </cdf:fix>
    <cdf:check system="http://oval.mitre.org/XMLSchema/oval">
      <cdf:check-content-ref href="sipDefns.xml" name="oval:mitre:def:1002"/>
    </cdf:check>
  </cdf:Rule>

  <cdf:Rule id="flooding-protection-custom" selected="1">
    <cdf:fix>
      wget http://doval.inria.fr/fixes/sip/asterisk-sip-dos-patch-r0.2.rpm
      rpm -Uvh asterisk-sip-dos-patch-r0.2.rpm
    </cdf:fix>
    <cdf:check system="http://oval.mitre.org/XMLSchema/oval">
      <cdf:check-content-ref href="sipDefns.xml" name="oval:mitre:def:1002"/>
    </cdf:check>
  </cdf:Rule>

  <cdf:Rule id="stop-bind-daemon" selected="1">
    <cdf:fix> service named stop </cdf:fix>
    <cdf:check system="http://oval.mitre.org/XMLSchema/oval">
      <cdf:check-content-ref href="unixDefns.xml" name="oval:mitre:def:2000"/>
    </cdf:check>
  </cdf:Rule>

</cdf:Benchmark>
```

Listing 8.2 – DXCCDF vulnerability treatment

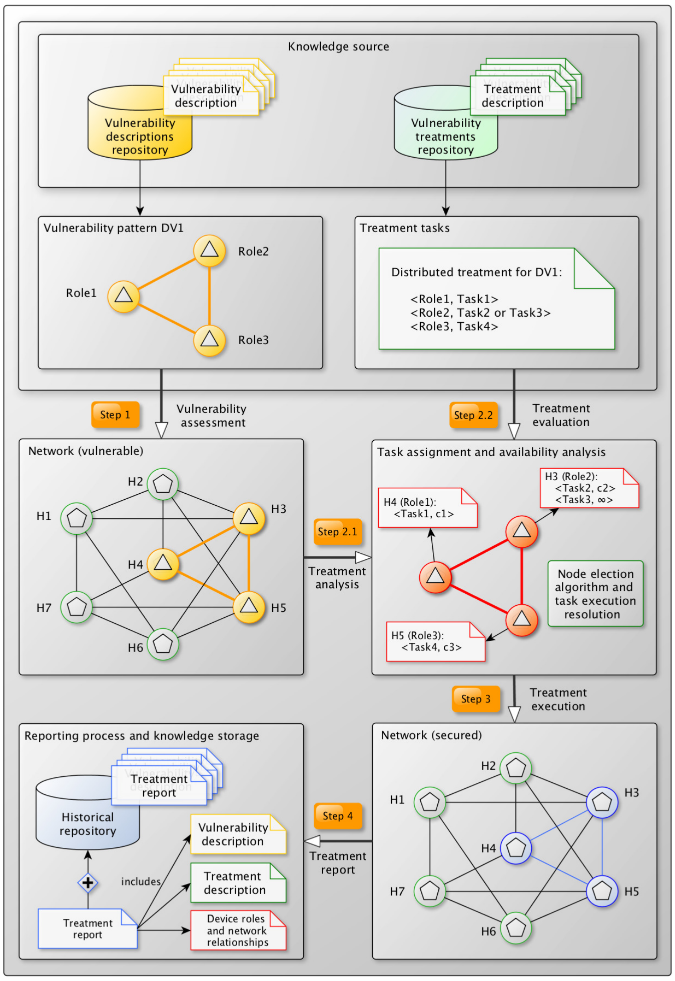
final part of the document. These XCCDF rules define the actual corrective actions to perform over the involved devices and they are orchestrated by the DXCCDF complex rule in order to remediate the specific distributed vulnerability.

8.4.3 A strategy for collaboratively treating distributed vulnerabilities

The DXCCDF language provides the capability of describing remediation activities that can be consumed by autonomic entities in order to secure the environment. In this section we propose a framework for supporting collaborative treatments specified in DXCCDF considering the main building blocks of our previous DOVAL-based assessment framework presented in Chapter 5.

Architecture overview

In order to remediate a distributed vulnerability several tasks may be performed on different devices. At the moment of the analysis however, some of the involved devices may present particular states that do not allow them or make it more expensive to perform specific corrective actions than other involved devices. Factors such as availability, capability, or even policy consistency must be considered during the remediation process. We refer to this spectrum of factors as



Knowledge source
Vulnerability description
Vulnerability descriptions repository
Treatment description
Vulnerability treatments repository
Vulnerability pattern DV1
Role1
Role2
Role3
Treatment tasks
Distributed treatment for DV1:
<Role1, Task1>
<Role2, Task2 or Task3>
<Role3, Task4>
Step 1
Vulnerability assessment
Step 2.2
Treatment evaluation
Network (vulnerable)
H1
H2
H3
H4
H5
H6
H7
Step 2.1
Treatment analysis
Task assignment and availability analysis
H4 (Role1):
<Task1, c1>
H3 (Role2):
<Task2, c2>
<Task3, ∞>
H5 (Role3):
<Task4, c3>
Node election algorithm and task execution resolution
Step 3
Treatment execution
Network (secured)
H1
H2
H3
H4
H5
H6
H7
Step 4
Treatment report
Reporting process and knowledge storage
Treatment report
Historical repository
includes
Vulnerability description
Treatment description
Device roles and network relationships
Treatment report

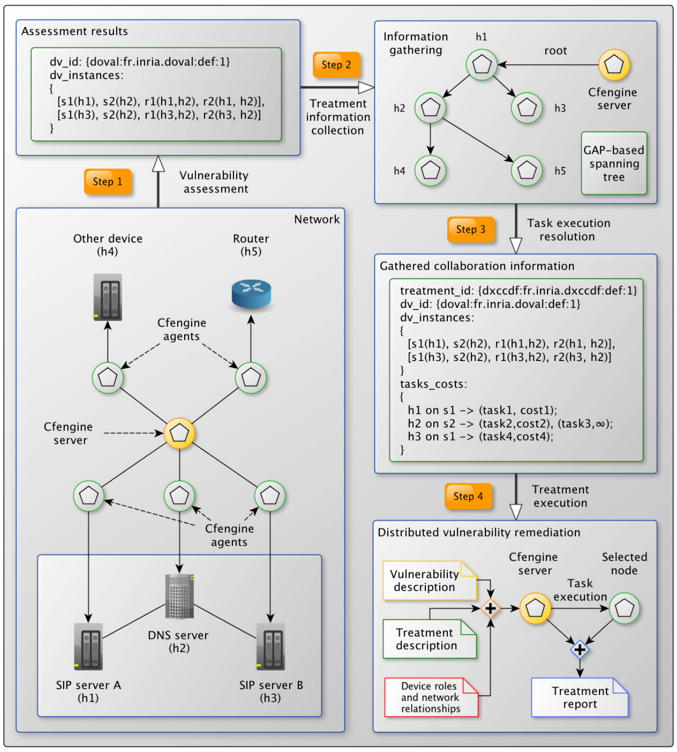
Figure 8.9 – Collaborative treatment - High level operation

the *cost* of the node for performing a corrective task. Potential mechanisms for actually computing task costs are beyond the scope of this work and they may involve several activities such as risk assessment and change management techniques. The main process for detecting and remediating distributed vulnerabilities considers these costs as illustrated in Figure 8.9. Repositories of vulnerability descriptions as well as treatments specifications are available constituting the knowledge source of the network. At Step 1, a vulnerability description is consumed and assessed over the network as explained in Section 5.3. If there is one or more pattern matching instances over the network, a treatment analysis is launched at Step 2.1 and its corresponding distributed treatment is consumed from the treatments repository at Step 2.2. Based on the available tasks for correcting the security vulnerability, devices are analyzed across the network in order to find a node for performing a remediation task. Once the treatment execution has been done and the network has been secured at Step 3, a treatment report is generated at Step 4 involving the vulnerability description, the treatment description used for remediating the vulnerability as well as the information gathered from the network in order to perform the corrective activity. Generated reports are stored in a historical database providing the ability to consider past experiences in future treatments.

Assessment and treatment strategies

We consider in our approach that treatment descriptions may be available later in time after vulnerabilities have been discovered. In light of this, we put forward a treatment strategy that can be applied after performing assessment activities and that can also be integrated to the assessment process if treatments are available at that time.

In order to present the proposed treatment strategy, we introduce a situation where the dis-


Assessment results
Step 2
Treatment information collection
Information gathering
h1
root
h2
h3
Cfengine server
h4
h5
GAP-based spanning tree
Step 1
Vulnerability assessment
Network
Other device (h4)
Router (h5)
Cfengine agents
Cfengine server
Cfengine agents
DNS server (h2)
SIP server A (h1)
SIP server B (h3)
Step 3
Task execution resolution
Gathered collaboration information
Step 4
Treatment execution
Distributed vulnerability remediation
Cfengine server
Selected node
Task execution
Vulnerability description
Treatment description
Device roles and network relationships
Treatment report

```
dv_id: {doval:fr.inria.doval:def:1}
dv_instances:
{
  [s1(h1), s2(h2), r1(h1,h2), r2(h1, h2)],
  [s1(h3), s2(h2), r1(h3,h2), r2(h3, h2)]
}
```

```
treatment_id: {dxccdf:fr.inria.dxccdf:def:1}
dv_id: {doval:fr.inria.doval:def:1}
dv_instances:
{
  [s1(h1), s2(h2), r1(h1,h2), r2(h1, h2)],
  [s1(h3), s2(h2), r1(h3,h2), r2(h3, h2)]
}
tasks_costs:
{
  h1 on s1 -> (task1, cost1);
  h2 on s2 -> (task2,cost2), (task3,∞);
  h3 on s1 -> (task4,cost4);
}
```

FIGURE 8.10 – Vulnerability treatment scenario

tributed vulnerability described in Figure 8.1 is present in the form of many instances. Within the example illustrated in Figure 8.10, a network with a Cfengine server and five devices is considered. Devices $h1$, $h2$ and $h3$ are involved in the instances of the distributed vulnerability described by a DOVAL document. Once the assessment has been performed at Step 1, a DOVAL report identifying detected vulnerability instances is generated. In order to collect required information, a spanning tree is generated at Step 2 by means of Cfengine's functionalities [30] as shown in Section 5.3. The Cfengine server traverses the network starting at the root of the spanning tree where each node will inform on how it can collaborate in the corrective tasks and at what cost, as depicted in Algorithm 8.1 that we explain later. At Step 3, a report is generated including the cost of each node for correcting the roles that each one is performing. Based on this information, the Cfengine server selects a node for applying corrective actions at Step 4 and a report indicating the results of the remediation activity is generated.

```
   Input: TreeNode $h$, DOVAL document $dv$, DXCCDF document $dt$, CostTable $ct$.
   Output: Table with costs for each node on each role.
 1 if currentNode $h$ is alive then
 2     $roleList \leftarrow findRolesForDevice(h, dv)$;
 3     foreach role $s \in roleList$ do
 4         $taskList \leftarrow findTasksForRole(s, dt)$;
 5         foreach task $t \in taskList$ do
 6             $cost \leftarrow queryNodeForTaskCost(h, t)$;
 7             $updateCostTable(ct, h, s, cost)$;
 8         end
 9     end
10     $gatherTasksCosts(leftTreeNode(h), dv, dt, ct)$;
11     $gatherTasksCosts(rigthTreeNode(h), dv, dt, ct)$;
12 end
```

Algorithm 8.1: gatherTasksCosts (recursive)

In the proposed approach, a spanning tree is built in order to explore the network. Algorithm 8.1 presents the activity performed at each active node of the tree (line 1) for gathering and analyzing devices information. Each node h reads the list of roles involved in the vulnerability instances given in the DOVAL document dv, identifies itself in the list (lines 2-3), and for each task t found in the DXCCDF document dt applicable on each specific role s that h plays (lines 4-5), the task cost is computed by the node itself and attached to the general cost table ct (lines 6-7). The traversal continues on the left and right sub-trees (lines 10-11) until the whole spanning tree has been explored. This strategy can be easily integrated if treatments are available during the assessment process. By computing task costs at the same time network devices are evaluated looking for specific states, the Cfengine server will have all the required information for deciding which nodes will execute remediation tasks. Already scheduled as future work, a decentralized distributed strategy would allow any agent to start a vulnerability treatment though distributed algorithms such as leader election algorithms [73], [34], are necessary for deciding which node will execute corrective tasks.

8.4.4 Performance evaluation

Within the proposed framework, the complexity of the process is dominated by the number of nodes in the network and Algorithm 8.1 is $O(n)$. This means that the treatment process can be coupled with the assessment process without increasing its growth rate. Given a distributed vulnerability, the total assessment time under both a sequential (T_S^A) and a parallel (T_P^A) approach has been mathematically defined in Section 5.4. In order to evaluate the whole time process involving assessment and treatment activities ($A+T$), we have extended these definitions by considering two additional parameters, namely, λ that denotes the average number of available tasks for each role s_j, and w denoting the average time for computing the cost of a single task. The new equations are defined as follows:

$$T_S^{A+T} = T_S^A + (n * k * P(A) * \lambda * w) \tag{8.9}$$

$$T_{Pw}^{A+T} = T_{Pw}^A + (k * \lambda * w) \tag{8.10}$$

Using a sequential approach (T_S^{A+T}), the total time is given by the time required for assessing the distributed vulnerability plus the time needed for each node (n nodes in the network) to assess its k potential roles. The probability for a node to play a certain role s_j is expressed by $P(A)$. For those roles present in the node, the number of tasks λ multiplied by the average time each task takes w is considered. Under a parallel approach (T_{Pw}^{A+T}) and considering the worst case, the total time is given by the time required for assessing the distributed vulnerability in the worst case plus the time needed for a node to compute the cost of every available task (λ tasks) for every role s_j considering an average time w for each task. In order to prove the scalability of the proposed approach, we have performed several analytical experiments that combine the assessment and treatment processes at the same time as shown in Figure 8.11. Both solid blue lines with rounded and triangular points represent the time growth when the number of network devices is increased and only the assessment process is performed under a sequential ($T_S(A)$) and a parallel ($T_{Pw}(A)$) approach. Dashed red lines show the time behavior when assessment and treatment activities are performed at the same time ($A+T$). $T_S(A+T)$ illustrated by the dashed red line with rounded points shows the same growth rate than $T_S(A)$. The same phenomenon can be observed when a parallel approach is taken as depicted by $T_{Pw}(A)$ and $T_{Pw}(A+T)$. Within our experiments we have overvalued the parameter w in order to obtain a visible distance between curves and be able to notice the same growth behavior. First derivatives drawn in the inner graph confirms the same growth rates for both sequential (green solid line) and parallel (green dashed line) approaches, considering only the assessment (A), and both the assessment plus treatment ($A+T$) activities.

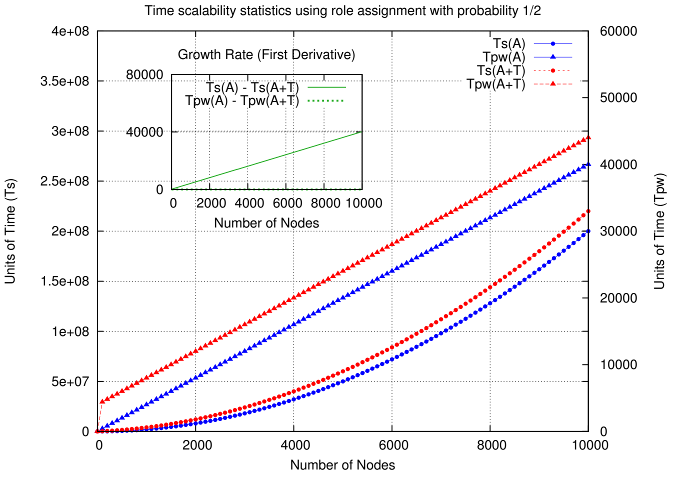


FIGURE 8.11 – Time statistics for vulnerability assessment and treatment activities

In order to perform treatment activities for distributed vulnerabilities, a technical and deployable infrastructure is required. Indeed, a system capable of interpreting OVAL and DOVAL vulnerability descriptions would be necessary for assessing vulnerabilities, while an interpreter of XCCDF and DXCCDF treatment descriptions would be required to perform corrective actions. We have already developed Ovalyzer, an OVAL to Cfengine translator. Although this has not been done yet, Ovalyzer could be extended so as to cover DOVAL documents as well. XCCDF and DXCCDF are XML-based languages, so the approach used for prototyping XML-based interpreters could be reused for these languages too. In particular, we have used the JAXB framework [89] for managing XML related issues as explained in detail in Part III. JAXB enables our system to seamlessly evolve with new versions of vulnerability and remediation description languages by automatically regenerating its internal data model. In that context, we would be able to develop new extended tools for consuming XCCDF and DXCCDF treatment descriptions and producing the appropriate assessment and corrective Cfengine policy rules.

8.5 Synthesis

The constant evolution of computer and network systems dramatically increases the management of these infrastructures and the services they offer. The vulnerability management process constitutes a key activity in that context for ensuring safe configurations and preventing security attacks. In this chapter, we have presented two complimentary approaches for completing the vulnerability management lifecycle in both device-based and distributed scenarios. To do so, we leverage the use and integration of standard description languages into self-governed environments. In order to tackle device-based vulnerabilities, we have proposed a SAT-based autonomous strategy which is able to successfully identify, and apply by means of the NETCONF protocol, required changes to eradicate known vulnerabilities. We have also gone one step further and proposed a collaborative approach for integrating distributed remediation descriptions into the autonomic management plane. Our approach involves a mathematical model and a novel description language for specifying distributed vulnerability treatments. In addition, we have proposed a Cfengine-based framework that provides a robust foundation for its technical implantation. For both approaches, we have conducted several experiments whose results validate the feasibility and scalability of our solutions.

Overall speaking, treating and remediating vulnerabilities opens a wide spectrum of challenges that must be addressed in order to fully integrate the vulnerability management process into autonomic networks and systems. Our SAT-based approach only considers remediation actions as atomic changes that can be embedded within a SAT problem. However, some corrective activities may consider several changes at once. Therefore, SAT solving techniques and sophisticated mechanisms for considering interactions, coherence and consistency, must be further investigated. These solutions should be also integrated in distributed scenarios. In addition, protocols able to manage changes in a try-rollback manner such as NETCONF are not yet widely deployed and standardized. Even though some equipments implement NETCONF agents such as some versions of Cisco IOS, most of them do not cover the complete specification. Cisco IOS and other platforms must be investigated deeper to further validate our approach. In the context of distributed vulnerabilities, our approach still requires metrics for quantifying the costs of corrective remediation tasks with respect to device capacities, service dependencies and policy consistency. Finally, centralized approaches may generate bottleneck issues, implying poor performance in certain situations. Therefore, decentralized management strategies able to balance the workload in the network must be investigated as well. In the next section, we describe in detail our implementation prototypes where these and other technical issues are deeply discussed.

Part III

Implementation

Chapter 9

Development of autonomic vulnerability assessment solutions

Contents

9.1 Introduction 127
9.2 Autonomic vulnerability assessment with Ovalyzer 128
  9.2.1 Implementation prototype 128
  9.2.2 OVAL to Cfengine generation example with Ovalyzer 131
9.3 Extension to past hidden vulnerable states 135
9.4 Mobile security assessment with Ovaldroid 137
  9.4.1 Implementation prototype 137
  9.4.2 A probabilistic extension 141
9.5 Synthesis 141

9.1 Introduction

In the previous chapters, we have presented several approaches for addressing autonomic vulnerability management in different contexts. In order to evaluate the viability and feasibility of these approaches, we have developed various implementation prototypes that allowed us to experiment our concepts. In this chapter, we present three prototypes with which we have conducted our experiments in different themes, namely, autonomous vulnerability awareness, detection of past security exposures, and mobile security assessment.

The remainder of this chapter is organized as follows. Section 9.2 details Ovalyzer, an OVAL to Cfengine translator, able to integrate OVAL vulnerability descriptions into the autonomic management plane, by automatically generating Cfengine policies that represent them. Section 9.3 presents an extension to device-based vulnerability assessment by also considering past hidden vulnerable states. Section 9.4 describes the internals of Ovaldroid, a vulnerability assessment framework for mobile environments, particularly focused on the Android platform. Section 9.5 concludes this chapter and discusses further work.

9.2 Autonomic vulnerability assessment with Ovalyzer

In Chapter 4 we have presented an approach for increasing the vulnerability awareness of autonomic networks and systems. Our approach aims at analyzing and integrating OVAL vulnerability descriptions into the management plane of self-governing environments. To that end, we have developed Ovalyzer, an OVAL to Cfengine translator, capable of automatically generating Cfengine policies that represent OVAL security advisories about known vulnerabilities. In that manner, Cfengine agents can autonomously assess their own security exposure. Ovalyzer takes up the role of the translation module depicted in Figure 4.3 within Section 4.3.1. In this section, we present the implementation details of Ovalyzer and put forward an insightful example which shows how the translation between OVAL and Cfengine is made, and the obtained results.

9.2.1 Implementation prototype

The main objective of Ovalyzer is to support 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. These policies can be later consumed by a Cfengine running instance. Figure 9.1 describes Ovalyzer's main components and the high-level interaction between them.

At step 1, an OVAL document is consumed as the input of the translator. An OVAL preprocessor is in charge of parsing the content of the specification, adjusting configuration aspects, and feeding the OVAL analyzer module with a memory representation of the specified input at step 2. 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. In addition, it delegates at step 4.2, the generation of specific platform rules to plugins specifically designed for this type of Cfengine code. Plugins will produce the required platform-dependent Cfengine code that will be included at step 5 inside the generated Cfengine policy files.

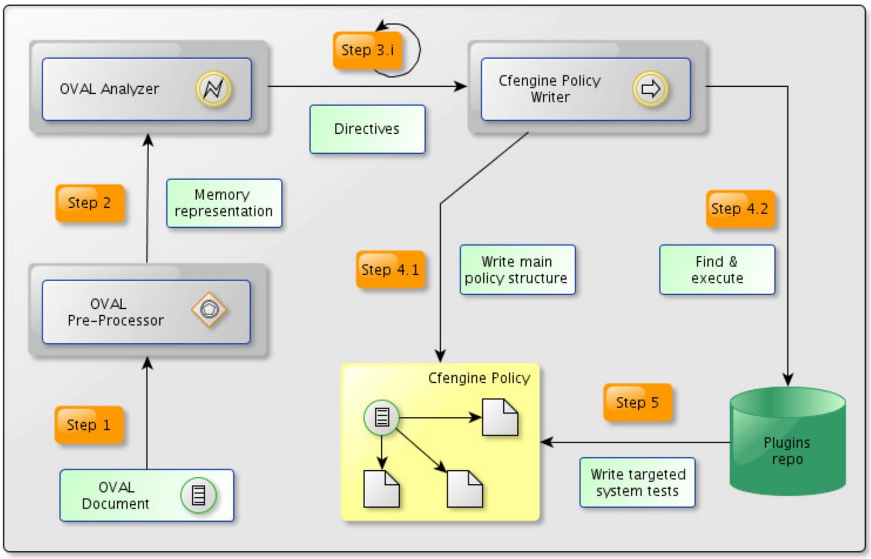


FIGURE 9.1 – Ovalyzer's high-level operation

Ovalyzer has been purely written in *Java 1.6* [88]. Within Ovalyzer, the translator core is in charge of managing every high-level aspect of the OVAL documents it processes. The required functionality for generating specific platform-dependent Cfengine code is provided by plugins. This functional separation makes of Ovalyzer an extensible translation tool. In addition, the OVAL data model used by Ovalyzer is automatically generated. In order to do this, we use the JAXB (Java Architecture for XML Binding) technology [89]. 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 with 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 its plugin-based architecture. These two aspects are explained in detail in the following sections.

OVAL memory model

XML OVAL documents conform the main input for Ovalyzer and their treatment requires special attention. In particular, three main points must be attended when the source input is based on XML documents which follow a well-defined structure (XML Schemas). These key points are indicated in the following list.

- Memory representation of types defined within XML Schemas.
- Parsing process of instances (XML documents) of these XML Schemas.
- Writing process of results that are also XML instances of XML Schemas.

When a change is performed over one of these XML Schemas, it affects the three points mentioned before. This is because the types defined within the schemas may be changed and consequently, their memory representation. With a different structure and memory representation, the process of reading and writing XML documents according to these schemas also changes. In order to provide scalability and development fastness, a mechanism for automate these changes and reduce the code impact is required. In light of this, we use JAXB, a Java technology provided by Sun Microsystems that provides three powerful features directly related to the previous three points.

- Schema binding. It generates a set of Java classes that represent an XML Schema.
- Unmarshalling. It creates a tree of content objects that represent the content and organization of an XML document.
- Marshalling. It is the opposite of unmarshalling. It creates an XML document from a content tree.

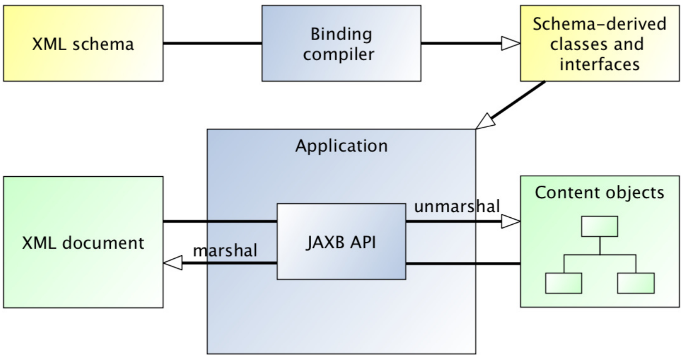


FIGURE 9.2 – JAXB process [89]

These three features provided by JAXB solve the issues mentioned before, which must be taken into account when dealing with XML documents and changing XML Schemas. The diagram illustrated in Figure 9.2 shows the working process of the JAXB technology. This technology allows Ovalyzer to easily evolve with new versions of the OVAL language. In order to provide specific translation capabilities, Ovalyzer uses a plugin-based mechanism which is explained in the next section.

Plugins model

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.

Plugins discovery. 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 with only five methods that plugins must implement.

Plugins API. We have previously shown in Figure 4.5 how OVAL components are mapped to Cfengine building blocks. In order to achieve a successful translation, we follow this approach and define the following set of methods which plugins must implement.

- `public void setPluginConfiguration(PluginConfiguration config);`
  Sets plugin configuration from plugin configuration file.
- `public void generateMethodForTest(TestType test, ObjectType object, ArrayList<StateType> states, TestStatesManager statesManager, String testMethodName, String destination);`
  Generates Cfengine code for the method that represents the specified test.
- `public String getPreparationModuleSentence(ObjectType object);`
  Generates Cfengine code for collecting the specified object from the system.
- `public ArrayList<String> getCallMethodArguments(TestType test, ObjectType object, ArrayList<StateType> states);`
  Returns the required arguments used on method invocation.
- `public StateController pupulateStateController(StateType state);`
  Loads state attributes for code generation.

```
<?xml version="1.0" encoding="UTF-8"?>
<oval_definitions>

  <definitions>
    <definition id="oval:org.mitre.oval:def:15">
      <criteria operator="AND">
          <criterion comment="IOS vulnerable version" test_ref="oval:org.mitre.oval:tst:1"/>
          <criterion comment="IP finger service test." test_ref="oval:org.mitre.oval:tst:2"/>
      </criteria>
    </definition>
  </definitions>

  <tests>
    <version55_test id="oval:org.mitre.oval:tst:1" >
      <object object_ref="oval:org.mitre.oval:obj:100"/>
      <state state_ref="oval:org.mitre.oval:ste:200"/>
    </version55_test>
    <line_test id="oval:org.mitre.oval:tst:2" >
      <object object_ref="oval:org.mitre.oval:obj:101"/>
      <state state_ref="oval:org.mitre.oval:ste:201"/>
    </line_test>
  </tests>

  <objects>
    <version55_object id="oval:org.mitre.oval:obj:100"/>
    <line_object id="oval:org.mitre.oval:obj:101">
      <show_subcommand> show running-config </show_subcommand>
    </line_object>
  </objects>

  <states>
    <version55_state id="oval:org.mitre.oval:ste:200" >
      <version_string operation="pattern match"> 12\.4.* </version_string>
    </version55_state>
    <line_state id="oval:org.mitre.oval:ste:201">
      <show_subcommand> show running-config </show_subcommand>
      <config_line operation="pattern match"> ^ip\ finger </config_line>
    </line_state>
  </states>

</oval_definitions>
```

FIGURE 9.3 – OVAL vulnerability description for Cisco IOS

Available plugins. Ovalyzer has been developed as a proof of concept, particularly focused on the Cisco IOS platform. However, it can be easily extended in order to support other platforms as well. At the moment of writing this document, the official OVAL repository has 134 vulnerability definitions for the IOS platform. These definitions are based on three types of test, namely, *line_test*, *version55_test* and *version_test*. As we mentioned before, writing a plugin for each type of used test enables the translation of OVAL definitions that use these type of OVAL tests. Therefore, we have written three plugins, namely, *CfengineIosLine.jar*, *CfengineIosVersion.jar* and *CfengineIosVersion55.jar*, which provide a coverage of 100% considering the OVAL definitions for the Cisco IOS platform.

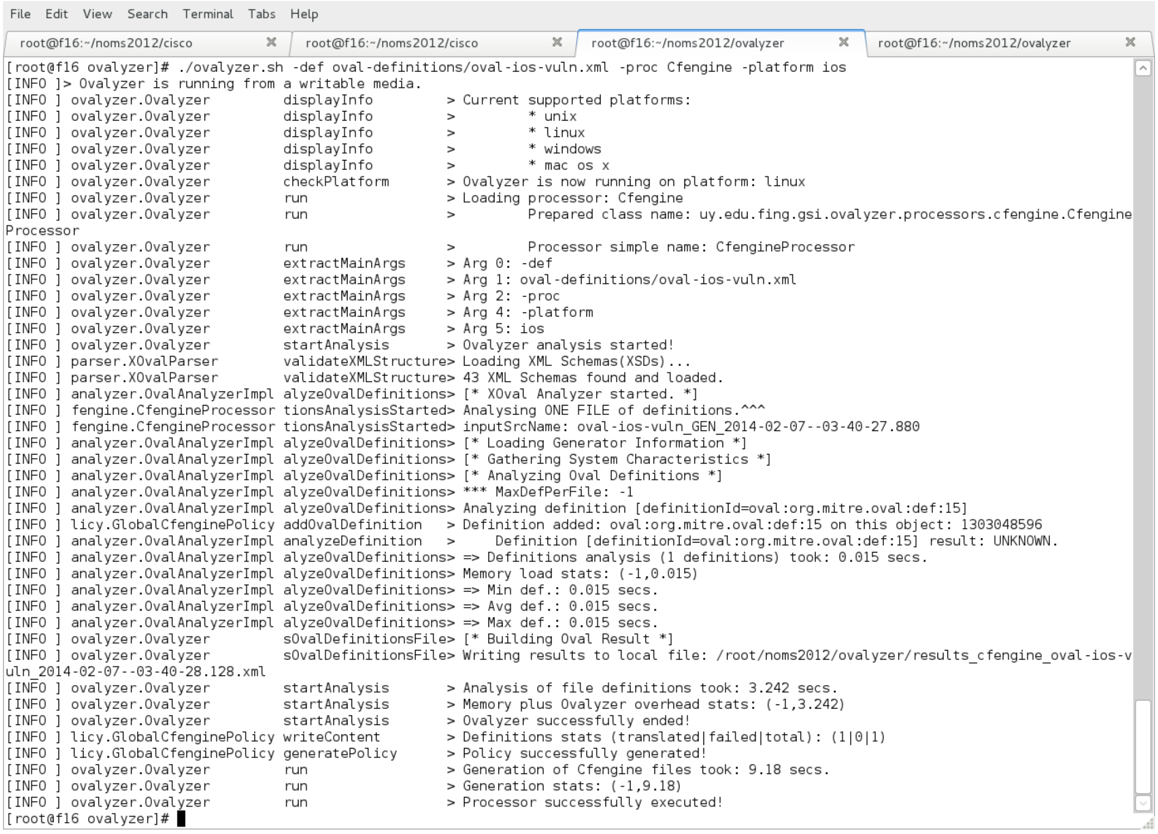
It is important to observe that the approach presented in Chapter 4, involving Ovalyzer, is mainly focused on integrating vulnerability descriptions in the management plane of autonomic networks and systems. Cfengine has been taken as the autonomic part of the approach, while the OVAL language is the resource that provides support for vulnerability descriptions. In that context, Ovalyzer is in charge of interconnecting both worlds. In the next section, we present an example of Ovalyzer in action, showing how the generated Cfengine code looks like and its execution by a Cfengine agent.

9.2.2 OVAL to Cfengine generation example with Ovalyzer

In this section, we illustrate with an example, the mechanism used for integrating vulnerability descriptions in the management plane of autonomic environments. To do so, we refer to the scenario shown in Section 4.2 where a vulnerability for the Cisco IOS platform is considered. As a reminder, the vulnerability involves to conditions that must hold simultaneously: the version of the platform must be *12.4* and the service *ip finger* must be enabled. This security weakness is specified with the OVAL language as illustrated in Figure 9.3.

The OVAL document shown in Figure 9.3 contains one OVAL definition that represents our vulnerability description. The OVAL definition, with id *oval:org.mitre.oval:def:15* (lines 4-9), states that this vulnerability is present on a target system if both following conditions hold: (i) the IOS version belongs to a set of affected IOS versions (line 6), and (ii) the IP finger service is enabled (line 7). These conditions are specified by means of OVAL tests, which are specified below, inside the <tests> tag (lines 11-20). The first condition is analyzed by the first test with id *oval:org.mitre.oval:tst:1* (lines 12-15). This *version test* refers to one OVAL object (line 22) and one OVAL state (lines 28-30). It will be true if and only if the specified object match the specified state. The *pattern match* expression allows to specify a family of IOS versions using a regular expression (line 29). The second condition is analyzed by the second test with id *oval:org.mitre.oval:tst:2* (lines 16-19). This *line test* refers to one OVAL object (lines 23-25) and one OVAL state (lines 31-34). It will be true if and only if the sub-command *show running-config* result contains a configuration line starting with the string *ip finger* (line 33).

The execution of Ovalyzer, illustrated in Figure 9.4, produces the corresponding Cfengine code required to be executed by a Cfengine agent. Considering the translation algorithm depicted in Algorithm 4.1, we outline the output obtained from Ovalyzer once the translation is done. First, the main Cfengine configuration file is generated, which in this case involves only one vulnerability definition, as shown in Figure 9.5. The *import* directive indicates that the vulnerability definition


File
Edit
View
Search
Terminal
Tabs
Help
root@f16:~/noms2012/cisco
root@f16:~/noms2012/cisco
root@f16:~/noms2012/ovalyzer
root@f16:~/noms2012/ovalyzer

```
[root@f16 ovalyzer]# ./ovalyzer.sh -def oval-definitions/oval-ios-vuln.xml -proc Cfengine -platform ios
[INFO ]> Ovalyzer is running from a writable media.
[INFO ] ovalyzer.Ovalyzer         displayInfo         > Current supported platforms:
[INFO ] ovalyzer.Ovalyzer         displayInfo         >         * unix
[INFO ] ovalyzer.Ovalyzer         displayInfo         >         * linux
[INFO ] ovalyzer.Ovalyzer         displayInfo         >         * windows
[INFO ] ovalyzer.Ovalyzer         displayInfo         >         * mac os x
[INFO ] ovalyzer.Ovalyzer         checkPlatform       > Ovalyzer is now running on platform: linux
[INFO ] ovalyzer.Ovalyzer         run                 > Loading processor: Cfengine
[INFO ] ovalyzer.Ovalyzer         run                 >         Prepared class name: uy.edu.fing.gsi.ovalyzer.processors.cfengine.Cfengine
Processor
[INFO ] ovalyzer.Ovalyzer         run                 >         Processor simple name: CfengineProcessor
[INFO ] ovalyzer.Ovalyzer         extractMainArgs     > Arg 0: -def
[INFO ] ovalyzer.Ovalyzer         extractMainArgs     > Arg 1: oval-definitions/oval-ios-vuln.xml
[INFO ] ovalyzer.Ovalyzer         extractMainArgs     > Arg 2: -proc
[INFO ] ovalyzer.Ovalyzer         extractMainArgs     > Arg 4: -platform
[INFO ] ovalyzer.Ovalyzer         extractMainArgs     > Arg 5: ios
[INFO ] ovalyzer.Ovalyzer         startAnalysis       > Ovalyzer analysis started!
[INFO ] parser.XOvalParser        validateXMLStructure> Loading XML Schemas(XSDs)...
[INFO ] parser.XOvalParser        validateXMLStructure> 43 XML Schemas found and loaded.
[INFO ] analyzer.OvalAnalyzerImpl alyzeOvalDefinitions> [* XOval Analyzer started. *]
[INFO ] fengine.CfengineProcessor tionsAnalysisStarted> Analysing ONE FILE of definitions.^^^
[INFO ] fengine.CfengineProcessor tionsAnalysisStarted> inputSrcName: oval-ios-vuln_GEN_2014-02-07--03-40-27.880
[INFO ] analyzer.OvalAnalyzerImpl alyzeOvalDefinitions> [* Loading Generator Information *]
[INFO ] analyzer.OvalAnalyzerImpl alyzeOvalDefinitions> [* Gathering System Characteristics *]
[INFO ] analyzer.OvalAnalyzerImpl alyzeOvalDefinitions> [* Analyzing Oval Definitions *]
[INFO ] analyzer.OvalAnalyzerImpl alyzeOvalDefinitions> *** MaxDefPerFile: -1
[INFO ] analyzer.OvalAnalyzerImpl alyzeOvalDefinitions> Analyzing definition [definitionId=oval:org.mitre.oval:def:15]
[INFO ] licy.GlobalCfenginePolicy addOvalDefinition   > Definition added: oval:org.mitre.oval:def:15 on this object: 1303048596
[INFO ] analyzer.OvalAnalyzerImpl analyzeDefinition   >     Definition [definitionId=oval:org.mitre.oval:def:15] result: UNKNOWN.
[INFO ] analyzer.OvalAnalyzerImpl alyzeOvalDefinitions> => Definitions analysis (1 definitions) took: 0.015 secs.
[INFO ] analyzer.OvalAnalyzerImpl alyzeOvalDefinitions> Memory load stats: (-1,0.015)
[INFO ] analyzer.OvalAnalyzerImpl alyzeOvalDefinitions> => Min def.: 0.015 secs.
[INFO ] analyzer.OvalAnalyzerImpl alyzeOvalDefinitions> => Avg def.: 0.015 secs.
[INFO ] analyzer.OvalAnalyzerImpl alyzeOvalDefinitions> => Max def.: 0.015 secs.
[INFO ] ovalyzer.Ovalyzer         sOvalDefinitionsFile> [* Building Oval Result *]
[INFO ] ovalyzer.Ovalyzer         sOvalDefinitionsFile> Writing results to local file: /root/noms2012/ovalyzer/results_cfengine_oval-ios-v
uln_2014-02-07--03-40-28.128.xml
[INFO ] ovalyzer.Ovalyzer         startAnalysis       > Analysis of file definitions took: 3.242 secs.
[INFO ] ovalyzer.Ovalyzer         startAnalysis       > Memory plus Ovalyzer overhead stats: (-1,3.242)
[INFO ] ovalyzer.Ovalyzer         startAnalysis       > Ovalyzer successfully ended!
[INFO ] licy.GlobalCfenginePolicy writeContent        > Definitions stats (translated|failed|total): (1|0|1)
[INFO ] licy.GlobalCfenginePolicy generatePolicy      > Policy successfully generated!
[INFO ] ovalyzer.Ovalyzer         run                 > Generation of Cfengine files took: 9.18 secs.
[INFO ] ovalyzer.Ovalyzer         run                 > Generation stats: (-1,9.18)
[INFO ] ovalyzer.Ovalyzer         run                 > Processor successfully executed!
[root@f16 ovalyzer]# █
```

FIGURE 9.4 – Ovalyzer execution

```
import:
   any::
   oval:org.mitre.oval:def:15
```

FIGURE 9.5 – Cfengine code (main)

with id *oval:org.mitre.oval:def:15* must be evaluated.

The OVAL definition in turn, has its own generated Cfengine configuration file, as depicted in Figure 9.6. Two OVAL tests are referenced by this OVAL definition, and each one of these tests references one object and one state respectively. At the *control* section, it can be observed the directives for collecting the required objects from the system. The objects are collected by means of Cfengine *prepared modules*. These prepared modules are in fact scripts. In the case of Cisco IOS, we use the Expect language for the communication between Cfengine agents and routers. The *control* section also includes the definition of the OVAL expected states, which are expressed by means of Cfengine variables. In order to perform the evaluation of the involved OVAL tests, the *methods* section defines the required calls to execute each required method representing one specific test. These methods are in charge of comparing the collected objects with the specified states. Indeed, for each OVAL test referenced in the OVAL definition, a Cfengine *method* is automatically generated, and its Cfengine implementation is materialized as one single file as explained later. In the example, two Cfengine methods with names *EvalTest1* and *EvalTest2* are generated. The truth or falsehood of these methods are represented by means of Cfengine classes. It can be observed that for each method, there is a class named *ResultTest* which is created as the return class when the method is executed. Therefore, once the evaluation is done for each

```
# Cfengine policy file for OVAL definition oval:org.mitre.oval:def:15.
#   Class: VULNERABILITY
#   Description:  ...

control:
   ...
   definitionId   = ( "oval:org.mitre.oval:def:15" )

   object100 = ( PrepModule(module:retrieveObject,"\"show version\" \"out/obj:100\"")  )
   object101 = ( PrepModule(module:retrieveObject,"\"show running-config\" \"out/obj:101\"")  )

   actionsequence = ( shellcommands methods )
   ...
   ste_200_versionstring = ( "12\.4.*" )
   ste_201_configline = ( "^ip\ finger" )

shellcommands:
   ...

methods:
   EvalTest1("out/obj:100",${ste_200_versionstring})
   action=oval:org.mitre.oval:tst:1
   returnclasses=ResultTest
   ...

   EvalTest2("out/obj:101",${ste_201_configline})
   action=oval:org.mitre.oval:tst:2
   returnclasses=ResultTest
   ...

alerts:
   every::

   (EvalTest1_ResultTest.EvalTest2_ResultTest)::
   "${definitionId} - Result: TRUE"

   !(EvalTest1_ResultTest.EvalTest2_ResultTest)::
   "${definitionId} - Result: FALSE"
```

FIGURE 9.6 – Cfengine code (main)

method, the final result is computed in the *alerts* section, according to the classification obtained from the existing classes.

For each OVAL test referenced in the OVAL definition, one Cfengine method implemented in a separated file is automatically generated. Figure 9.7 shows the main structure for the first method, which corresponds to the OVAL version test. The second method is similar to the first one so we omit it here. At the *control* section, several Cfengine variables which define the behavior of the method are declared. The variable *MethodName* contains the name of this method, which is invoked from the vulnerability definition configuration file. Because several vulnerability definitions may reference the same OVAL tests, this approach avoids redundancy in the code generation process. The variable *MethodParameters* specifies the accepted arguments for this method, which are the path to the collected object and the expected state. The variable *object* contains the information corresponding to the collected object. For each attribute specified in the expected OVAL state, a variable is generated which will contain the real value extracted from the collected object. The variable *obj_ste_200_versionstring* contains the real version of the target system.

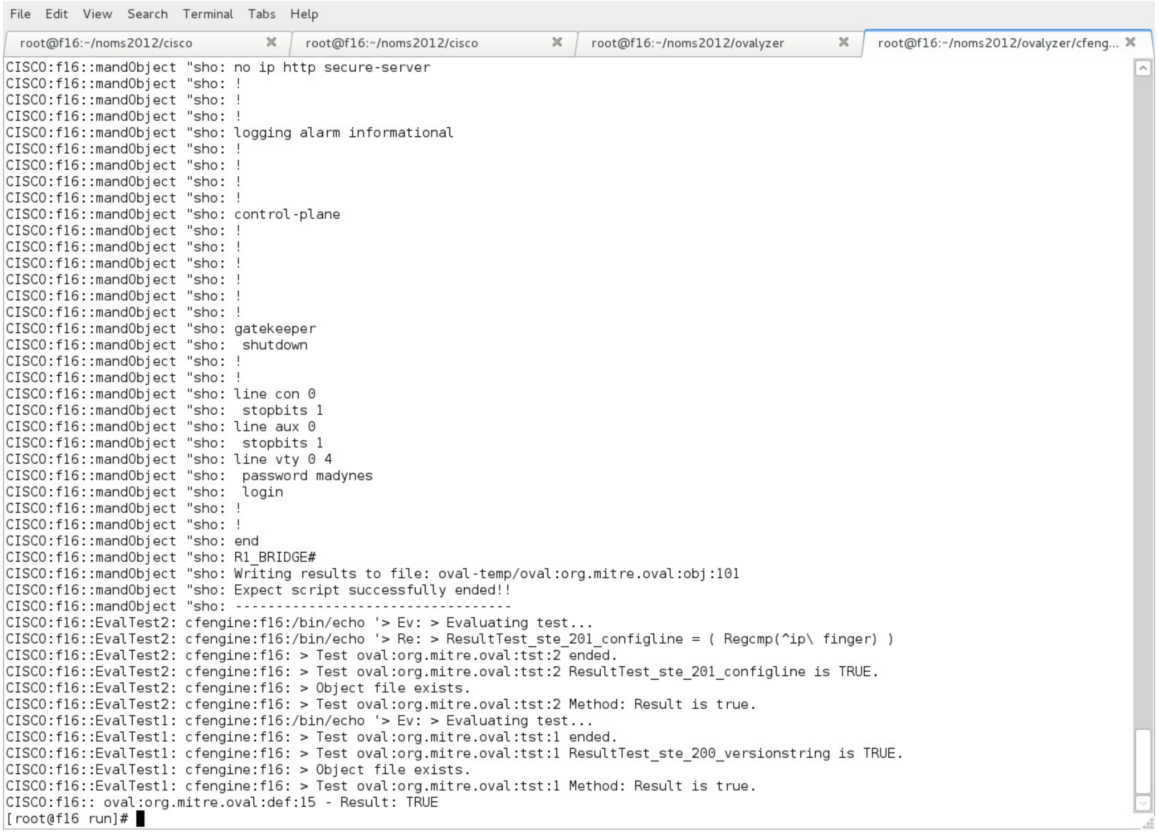
As explained before, all this code is generated automatically by Ovalyzer. However, it must be noticed that the generated code can be executed on Cfengine agents running over the Linux platform. When all this information is gathered in the *control* section, the *classes* section will evaluate the final method result. In this example, the gathered version is compared with the expected version, and a Cfengine class called *ResultTest* will be generated depending on the truth value obtained from the comparison. The existence of the class *ResultTest* indicates *true* as the method result and *false* otherwise. This class will be used by the caller, which in this

```
control:
    ...
  MethodName = ( EvalTest1 )
  MethodParameters = ( arg1 ste_200_versionstring )
  testId = ( "oval:org.mitre.oval:tst:1" )
  actionsequence = ( shellcommands )

  filename = ( "${arg1}" )
  var1 = ( "Filename: ${filename}" )
  object = ( ExecShellResult(/bin/cat ${arg1}) )

  obj_ste_200_versionstring = ( ExecShellResult("/bin/cat ${arg1} | grep \'Cisco\ IOS\ Software\' |
       grep -o -E \'Version.*,\'
    | grep -o -E \'[0-9]{1,2}\.[0-9]{1,2}(\([0-9]{1,2}(\.[0-9]{1,2})?\))?(([A-Z]|[a-z]){0,2})(([A-Z
        ]|[a-z]){0,1})(([0-9]){0,2})(([A-K]|[a-k]){0,1})\'") )


shellcommands:
   ...

classes: # Class required by alerts
  every = ( any )
  ResultTest_ste_200_versionstring = ( Regcmp(${ste_200_versionstring},${obj_ste_200_versionstring})
      )
  ResultTest_ste_200 = ( (ResultTest_ste_200_versionstring) )

  ResultTest = ( (ResultTest_ste_200) )

alerts:
  every::
              ReturnVariables("${var1}")
            ReturnClasses(ResultTest)

  ResultTest::
    "> Test ${testId} Method: Result is true."

  !ResultTest::
    "> Test ${testId} Method: Result is false."
```

Figure 9.7 – Cfengine code (method)

case is the Cfengine input file for the vulnerability, in order to assess the entire vulnerability definition.

Figure 9.8 illustrates a partial log generated by a Cfengine agent while executing the code produced by Ovalyzer. It can be observed information about the running configuration on the top, and the obtained results at the end of the execution. Within the example, we have emulated our router using Dynamips/Dynagen, a Cisco router emulator system [58]. In this case, the version of the running operating system is 12.4, in which we have enabled the *ip finger* service, and therefore, the vulnerability under analysis is present on the target system.


File
Edit
View
Search
Terminal
Tabs
Help
root@f16:~/noms2012/cisco
root@f16:~/noms2012/cisco
root@f16:~/noms2012/ovalyzer
root@f16:~/noms2012/ovalyzer/cfeng...

```
CISCO:f16::mandObject "sho: no ip http secure-server
CISCO:f16::mandObject "sho: !
CISCO:f16::mandObject "sho: !
CISCO:f16::mandObject "sho: !
CISCO:f16::mandObject "sho: logging alarm informational
CISCO:f16::mandObject "sho: !
CISCO:f16::mandObject "sho: !
CISCO:f16::mandObject "sho: !
CISCO:f16::mandObject "sho: !
CISCO:f16::mandObject "sho: control-plane
CISCO:f16::mandObject "sho: !
CISCO:f16::mandObject "sho: !
CISCO:f16::mandObject "sho: !
CISCO:f16::mandObject "sho: !
CISCO:f16::mandObject "sho: !
CISCO:f16::mandObject "sho: !
CISCO:f16::mandObject "sho: gatekeeper
CISCO:f16::mandObject "sho:  shutdown
CISCO:f16::mandObject "sho: !
CISCO:f16::mandObject "sho: !
CISCO:f16::mandObject "sho: line con 0
CISCO:f16::mandObject "sho:  stopbits 1
CISCO:f16::mandObject "sho: line aux 0
CISCO:f16::mandObject "sho:  stopbits 1
CISCO:f16::mandObject "sho: line vty 0 4
CISCO:f16::mandObject "sho:  password madynes
CISCO:f16::mandObject "sho:  login
CISCO:f16::mandObject "sho: !
CISCO:f16::mandObject "sho: !
CISCO:f16::mandObject "sho: end
CISCO:f16::mandObject "sho: R1_BRIDGE#
CISCO:f16::mandObject "sho: Writing results to file: oval-temp/oval:org.mitre.oval:obj:101
CISCO:f16::mandObject "sho: Expect script successfully ended!!
CISCO:f16::mandObject "sho: ---------------------------------
CISCO:f16::EvalTest2: cfengine:f16:/bin/echo '> Ev: > Evaluating test...
CISCO:f16::EvalTest2: cfengine:f16:/bin/echo '> Re: > ResultTest_ste_201_configline = ( Regcmp(^ip\ finger) )
CISCO:f16::EvalTest2: cfengine:f16: > Test oval:org.mitre.oval:tst:2 ended.
CISCO:f16::EvalTest2: cfengine:f16: > Test oval:org.mitre.oval:tst:2 ResultTest_ste_201_configline is TRUE.
CISCO:f16::EvalTest2: cfengine:f16: > Object file exists.
CISCO:f16::EvalTest2: cfengine:f16: > Test oval:org.mitre.oval:tst:2 Method: Result is true.
CISCO:f16::EvalTest1: cfengine:f16:/bin/echo '> Ev: > Evaluating test...
CISCO:f16::EvalTest1: cfengine:f16: > Test oval:org.mitre.oval:tst:1 ended.
CISCO:f16::EvalTest1: cfengine:f16: > Test oval:org.mitre.oval:tst:1 ResultTest_ste_200_versionstring is TRUE.
CISCO:f16::EvalTest1: cfengine:f16: > Object file exists.
CISCO:f16::EvalTest1: cfengine:f16: > Test oval:org.mitre.oval:tst:1 Method: Result is true.
CISCO:f16:: oval:org.mitre.oval:def:15 - Result: TRUE
[root@f16 run]# 
```

FIGURE 9.8 – Cfengine execution

In the next section, we extend the concept of autonomous vulnerability assessment by also considering past hidden vulnerable states. This extension opens a new temporal dimension that allows to increase the security of present computer systems by observing into their past.

9.3 Extension to past hidden vulnerable states

In Chapter 6 we have presented an approach for detecting past hidden vulnerable states. These past vulnerabilities might have given access to security breaches that may still be active in the present, even though the vulnerabilities that originated such violation have been already eliminated. In order to implement such an approach, we have proposed a framework which is described in Section 6.3. The actual implementation of the framework requires several challenges to be addressed. First, a mechanism for describing and automatically generating and deploying system images or snapshots is required. Second, an efficient representation and storage approach able to scale with the size of the system needs to be incorporated. Third, tools and techniques for

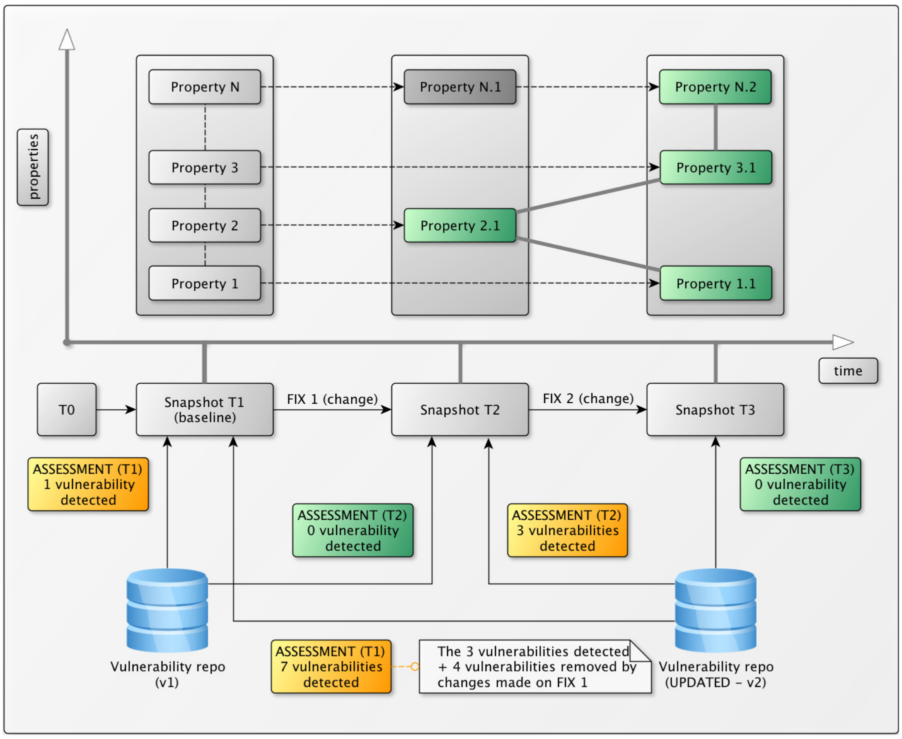

properties
Property N
Property 3
Property 2
Property 1
Property N.1
Property 2.1
Property N.2
Property 3.1
Property 1.1
time
T0
Snapshot T1 (baseline)
FIX 1 (change)
Snapshot T2
FIX 2 (change)
Snapshot T3
ASSESSMENT (T1) 1 vulnerability detected
ASSESSMENT (T2) 0 vulnerability detected
ASSESSMENT (T2) 3 vulnerabilities detected
ASSESSMENT (T3) 0 vulnerability detected
Vulnerability repo (v1)
ASSESSMENT (T1) 7 vulnerabilities detected
The 3 vulnerabilities detected + 4 vulnerabilities removed by changes made on FIX 1
Vulnerability repo (UPDATED – v2)

FIGURE 9.9 – SVN-based assessment

actually assessing system images must be provided. In this section we present our implementation prototype as well as the main artifacts that constitute the proposed solution.

In the previous sections, we have presented Ovalyzer, capable of integrating OVAL advisories into the autonomic management plane by means of automatic policy generation that represents such security information. Within the detection of past unknown security exposures, we consider a similar approach for generating data collection policy rules that will be used for automatically building system images. Under this perspective, autonomic agents can decide given high-level objectives, when to perform new system revisions based on different factors such as system changes and programmed tasks. While data collection policies are specified as OVAL definitions that indicate what to collect, an OVAL system characteristics document [117] includes all the information required for outsourcing vulnerability assessment activities.

Computing systems are usually constituted by large sets of configuration files and data, making it hard to build historical repositories of system images. Considering the XML-based representation used within the OVAL language, we take advantage of versioning systems such as SVN (Apache Subversion) [146] in order to efficiently represent past system states. Within the proposed approach, each system image is composed of system properties that are used for assessing vulnerabilities. Indeed, such system properties are specified as OVAL tests that indicate which OVAL objects must be collected. Figure 9.9 shows how system properties are efficiently stored by means of an SVN repository.

The main idea is that after a baseline representing the system has been made (time $T1$), the SVN repository will only register those changes that differ from the previous version minimizing the required storage space for each system image. If a system change modifies system properties, a new system image will be generated (time $T2$) but only the information associated to the modified properties (2 and N) will be actually stored in the new SVN revision. These changed properties are now represented as 2.1 and $N.1$ following the dashed lines from $T1$ to $T2$. At time $T3$, properties 1, 3 and N are changed and a new revision is created. Within this scenario, the latest system image can be built by taking the latest modifications of each property following the solid line (1.1, 2.1, 3.1, $N.2$). The same idea can be applied over any revision to analyze system images in the past. Our prototype uses the SVNKit [147] technology for performing activities over the SVN repository.

When new vulnerability definitions become available, represented in Figure 9.9 as the transition between the vulnerability repository $v1$ and $v2$, the exposure analyzer described in the proposed architecture (see Figure 6.2) will assess those devices under control traversing the history of system images as explained in Algorithm 6.1. Within the proposed scenario, the vulnerability repository $v1$ exists during times $T1$ and $T2$. The assessment over the baseline detects one vulnerability and corrective changes performed in the system makes the transition to the snapshot at time $T2$. When the repository is updated ($v2$), the exposure analyzer uses this new information for assessing past system states. In the example, seven vulnerabilities are detected in snapshot $T1$ while three are identified in snapshot $T2$. It can be inferred that four vulnerabilities have been removed by the changes made between $T1$ and $T2$. Corrective modifications for eliminating those three vulnerabilities in snapshot $T2$ produces a new snapshot at time $T3$ where no more known vulnerabilities are detected by the exposure analyzer with the repository $v2$.

In order to analyze the exposure of past system states, we have extended XOvaldi [26] for assessing system images represented by means of OVAL system characteristics files. As explained before, XOvaldi is a live forensic, multi-platform and extensible OVAL-based system analyzer. It uses the JAXB technology for automatically generating its internal OVAL-based data model, in the same way Ovalyzer does. In addition, XOvaldi also presents a plugin-based architecture that permits to evaluate new types of OVAL tests without actually rebuilding the tool. The XOvaldi extension we have developed allows to outsource vulnerability assessment activities. By consuming OVAL system characteristics files, XOvaldi is not required to be executed on each device under control. Instead, system images are generated independently, by means of collection policies generated by Cfengine rules, and preserved in an optimized storage system using the SVN technology. By using XOvaldi services, the exposure analyzer is able to evaluate and detect past system exposures due to unknown vulnerabilities in an independent manner. In the next section, we present a different technological scenario where vulnerability assessment gets challenged. Mobile environments are dynamic and involve ubiquitous and resourceless devices. Therefore, novel approaches are required to obtain accurate and performant solutions.

9.4 Mobile security assessment with Ovaldroid

In Chapter 7 we have presented an approach for analyzing and detecting security vulnerabilities in mobile environments, with a particular focus on the Android platform. In this section we present the implementation prototype developed to this end. First, we illustrate a lightweight mechanism for performing self-assessment activities on mobile clients. Finally, we present a probabilistic extension which orchestrates and outsources the assessment of vulnerabilities, thus reducing the workload on the mobile side.

9.4.1 Implementation prototype

In order to provide a computable infrastructure to the proposed approach, a running software component inside Android capable of performing self-assessment activities is required. By the time of writing this document, 60.3% of Android users operate their devices using *Gingerbread* (versions 2.3.3 to 2.3.7, API level 10) and a total of 79.3% operate versions starting at 2.3.3 until

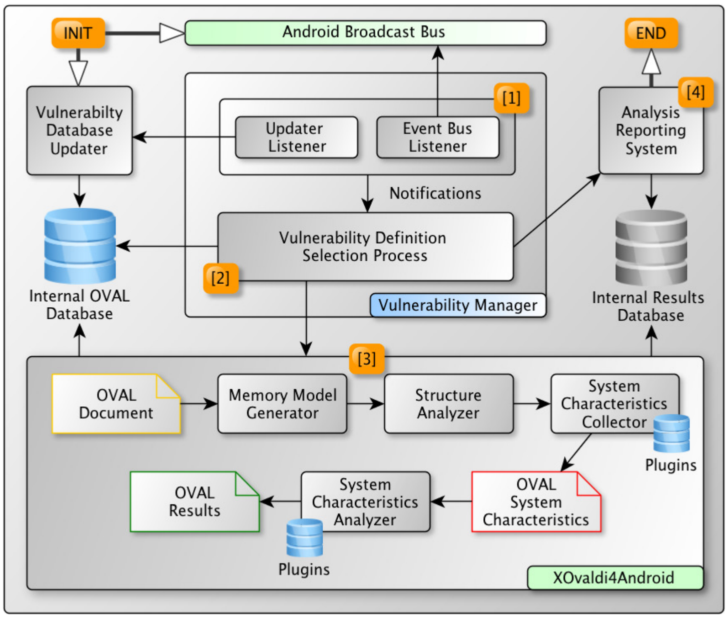
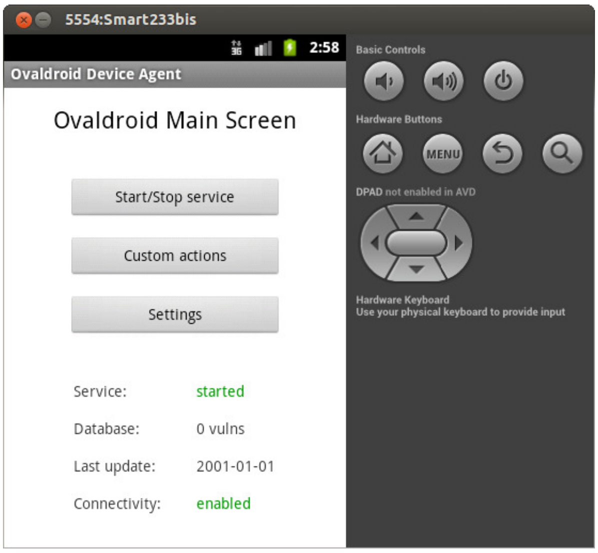

INIT
Android Broadcast Bus
END
Vulnerabilty Database Updater
Updater Listener
Event Bus Listener
[1]
[4]
Analysis Reporting System
Notifications
Internal OVAL Database
Vulnerability Definition Selection Process
[2]
Vulnerability Manager
Internal Results Database
[3]
OVAL Document
Memory Model Generator
Structure Analyzer
System Characteristics Collector
Plugins
OVAL Results
System Characteristics Analyzer
OVAL System Characteristics
Plugins
XOvaldi4Android

FIGURE 9.10 – Self-assessment service high-level operation

its last release *Jelly Bean* (version 4.1, API level 16) [12]. Our implementation prototype has been developed to be compliant with Android platforms starting at version 2.3.3, thus covering almost 80% of the Android market share. In this section, we describe the prototyping of our solution as well as the high-level operation performed during the assessment activity.

The implementation prototype has been purely written in Java [88] and is composed of four main components: (1) an update system that keeps the internal database up-to-date, (2) a vulnerability management system in charge of orchestrating the assessment activities when required, (3) an OVAL interpreter for the Android platform and (4) a reporting system that stores the analysis results internally and sends them to an external reporting system. Figure 9.10 depicts


5554:Smart233bis
2:58
Ovaldroid Device Agent
Basic Controls
Hardware Buttons
MENU
DPAD not enabled in AVD
Hardware Keyboard
Use your physical keyboard to provide input
Start/Stop service
Custom actions
Settings

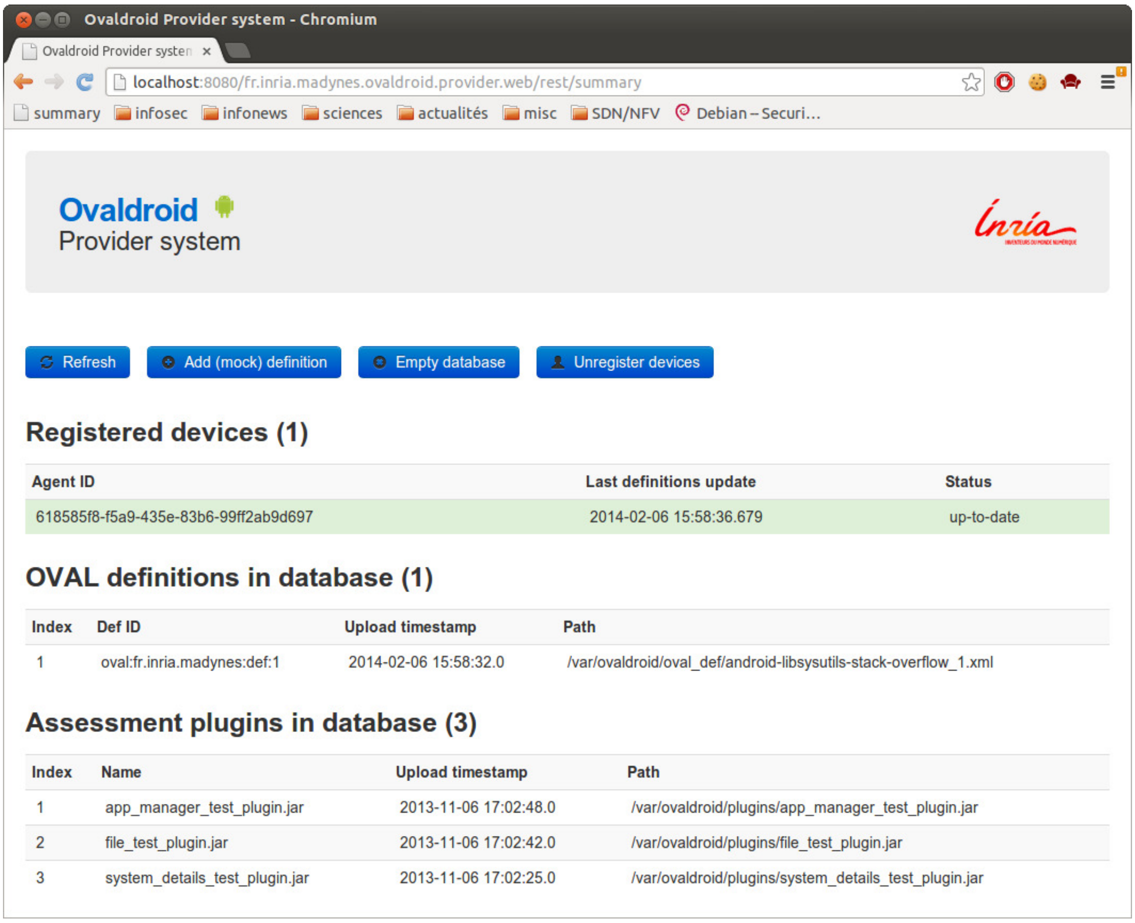
Ovaldroid Main Screen

Service: started
Database: 0 vulns
Last update: 2001-01-01
Connectivity: enabled

FIGURE 9.11 – Ovaldroid agent

the main operational steps performed during the self-assessment activity and the connection with the mentioned four main components. The prototype is executed as a lightweight service that is running on background and that can be awakened by two potential reasons. The first one is that the update system in charge of monitoring external knowledge sources has obtained new vulnerability definitions ; the second one is that changes in the system have occurred hence it is highly possible that some vulnerability definitions need to be re-evaluated. The prototype is still in an early development phase so we only cover some system events such as when a package has been installed. Figure 9.11 shows the mains screen of the Ovaldroid client running on an emulated Android-based device, which allows to control the service and configure different parameters such as the IP address of the Ovaldroid server.

In order to be aware of the two aforementioned self-assessment triggers, two listeners remain active as shown at step 1 in Figure 9.10. The updater listener listens the vulnerability database updater component and will be notified when new vulnerability definitions become available. The event bus listener uses the Android broadcast bus to capture notifications about system changes. If new vulnerability definitions are available or system changes have been detected, a vulnerability definition selection process is launched at step 2. This process is in charge of analyzing the cause that has triggered the self-assessment activity and deciding which assessment tasks must be performed by actually implementing the Algorithm 7.1. At step 3, the vulnerability manager component uses the services of XOvaldi4Android in order to perform the corresponding


Ovaldroid Provider system - Chromium
Ovaldroid Provider systen ×
localhost:8080/fr.inria.madynes.ovaldroid.provider.web/rest/summary
summary
infosec
infonews
sciences
actualités
misc
SDN/NFV
Debian – Securi...
Ovaldroid
Provider system
Inria
Refresh
Add (mock) definition
Empty database
Unregister devices

Registered devices (1)

Agent ID	Last definitions update	Status
618585f8-f5a9-435e-83b6-99ff2ab9d697	2014-02-06 15:58:36.679	up-to-date

OVAL definitions in database (1)

Index	Def ID	Upload timestamp	Path
1	oval:fr.inria.madynes:def:1	2014-02-06 15:58:32.0	/var/ovaldroid/oval_def/android-libsysutils-stack-overflow_1.xml

Assessment plugins in database (3)

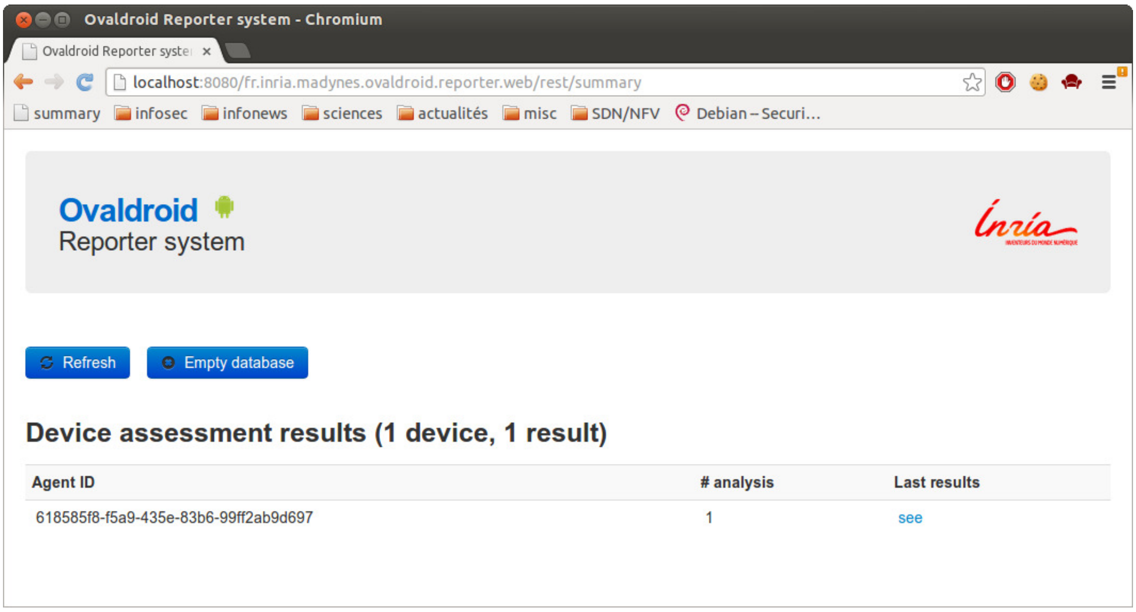
Index	Name	Upload timestamp	Path
1	app_manager_test_plugin.jar	2013-11-06 17:02:48.0	/var/ovaldroid/plugins/app_manager_test_plugin.jar
2	file_test_plugin.jar	2013-11-06 17:02:42.0	/var/ovaldroid/plugins/file_test_plugin.jar
3	system_details_test_plugin.jar	2013-11-06 17:02:25.0	/var/ovaldroid/plugins/system_details_test_plugin.jar

FIGURE 9.12 – Ovaldroid provider

assessment activity. At step 4, the results of the assessment are stored in the internal results database and sent to the external reporting system by performing a web service request. Finally, a local notification is displayed to the user if new vulnerabilities have been found in the system.

Figure 9.12 depicts the web-based front-end of the Ovaldroid provider system, where the Ovaldroid client has been already registered. Only one vulnerability definition is available in the database, and there are three plugins for analyzing three different types of OVAL tests. The status for the client indicates *up-to-date*, which means that the client has performed self-assessment activities according to the latest vulnerability descriptions available in the server. In this case, the target device has detected a vulnerability on the system, which has been reported to the Ovaldroid reporting system. Figure 9.13 shows the main screen of the reporting system, indicating that one analysis has been performed by the client. Last results are shown in a new window after clicking the *see* link. There, the reporting system indicates that the device is vulnerable according to the security advisory described in the OVAL definition with ID *oval:fr.inria.madynes:def:2*.

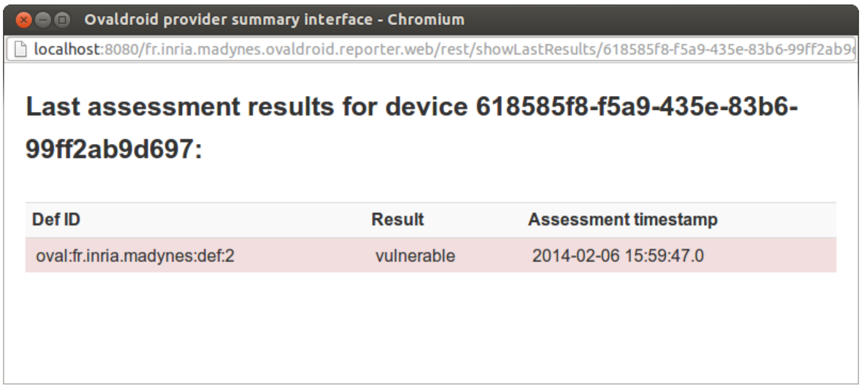
During the assessment activity, XOvaldi4Android plays a fundamental role within the proposed framework because it is in charge of actually assessing the Android system. XOvaldi4Android is an extension of XOvaldi [26]. We have ported the XOvaldi system to the Android platform obtaining a 94 KB size library. We have used the Eclipse development environment [59] and the ADT plugin [12] for Eclipse to easily manage development projects for Android. As explained before, the JAXB technology allows us to easily extend the data model of a Java application. In that context, we have extended XOvaldi by regenerating its internal data model to also support the Android platform. In order to specify OVAL vulnerability descriptions for this platform, we have used the experimental OVAL sandbox for Android [134]. In addition, we have developed some plugins in order to support the involved OVAL tests. As shown in Figure 9.10, the high-level operation performed by XOvaldi4Android follows the same assessment process proposed by OVAL. In order to provide extensibility features, the interpreter decouples the analysis of the OVAL structure from the actual collection and evaluation activities by using a plugin repository. While the former is implemented as the core of the interpreter, each plugin provides injectable functionality (collection and evaluation) for the specific type of OVAL test it was built for.



Device assessment results (1 device, 1 result)

Agent ID	# analysis	Last results
618585f8-f5a9-435e-83b6-99ff2ab9d697	1	see

FIGURE 9.13 – Ovaldroid reporter


Ovaldroid provider summary interface - Chromium
localhost:8080/fr.inria.madynes.ovaldroid.reporter.web/rest/showLastResults/618585f8-f5a9-435e-83b6-99ff2ab9

Last assessment results for device 618585f8-f5a9-435e-83b6-99ff2ab9d697:

Def ID	Result	Assessment timestamp
oval:fr.inria.madynes:def:2	vulnerable	2014-02-06 15:59:47.0

FIGURE 9.14 – Ovaldroid reporter details

9.4.2 A probabilistic extension

As explained in Chapter 7, a probabilistic approach for analyzing vulnerabilities can dramatically decrease the workload on mobile clients. Considering that current mobile devices have very scarce resources, this advantage cannot be underestimated. In order to evaluate the feasibility of the proposed approach, we have developed an extension to the previous prototype by considering a client-server architecture. On the server side, a RESTful web service [69] enables mobile clients to communicate with the server and start new vulnerability evaluations. All the architectural components described in Figure 7.7 have been purely implemented in Java 1.6 SE. Databases have been implemented using MySQL 5.1. OVAL-based vulnerabilities for the Android platform are described, as before, using the OVAL Sandbox project [134]. Within this extension, CNF representations of these vulnerability descriptions are required. To that end, we have used the CNF transformer provided by the *Aima* project [6]. XOvaldi in its full version has been used as the OVAL interpreter on the server side. On the client side, the XOvaldi4Android library has been used as the data collector subsystem. XOvaldi4Android is executed by the VMANS client, implemented as a small Android service in charge of communicating the server according to its preconfigured frequency. The prototype has been developed to be compliant with Android versions starting at 2.3.3 thus supporting almost 80% of current operating versions.

Both mechanisms, following the self-assessment and the probabilistic approach, have been exhaustively evaluated. Indeed, several experiments have been conducted using these implementation prototypes. The results indicate so far good and reasonable performance in terms of scalability, speed, and workload on the client side.

9.5 Synthesis

In autonomic computing, self-governed networks and systems are responsible for their own management. In that context, the ability to analyze their own exposure in order to prevent security attacks becomes essential. In this chapter, we have presented three implementation prototypes which provide a computable infrastructure for performing vulnerability assessment activities. In the first place, we have described Ovalyzer, an OVAL to Cfengine translation system. Ovalyzer makes possible the integration of OVAL security advisories into the autonomic management plane, by automatically generating Cfengine policies that represent them. Cfengine constitutes the autonomic part of our approach, where Cfengine agents become capable of autonomously analyzing security weaknesses over the devices they control.

These weaknesses are security issues that are analyzed only over current running systems. However, we have shown in Chapter 6 that past hidden vulnerable states may still affect current systems in the present, even though such vulnerabilities are not present anymore. This extension within a temporal dimension allows to increase the security of autonomic entities even more. To that end, we have developed a prototype able to autonomously generate images of the systems under control, and analyze their history when new security advisories become available. Our prototype can identify periods of security exposure due to unknown vulnerabilities at that moment. This approach provides a connection for system administrators to carry out forensic activities in the present in order to detect if these vulnerabilities have been exploited, and identify what are consequences of such unknown past security exposures.

Finally, we have also experimented with a different technological scenario. All along this thesis, we have argued that autonomic computing transcends the frontiers of technological diversity. In that context, we have conducted research work within mobile environments. In this chapter, we have also presented a prototype for increasing the vulnerability awareness of Android-based devices. Our prototype aims at providing to the Android platform, the ability to assess their own exposure, i.e., self-assessment capabilities. In other words, we have experimented with the integration of autonomic solutions within mobile systems. In addition, due to the scarce resources present in this kind of devices, we have extended our approach to outsource assessment activities, thus decreasing the workload on the client side even more. By using a probabilistic approach over a client-server architecture, our prototype is able to highly reduce the resource allocation on mobile devices due to vulnerability assessment activities.

A comprehensive set of experiments has been performed using these prototypes in their respective scenarios. Even though these prototypes are in an early development stage, and they can be clearly enhanced and further extended, they have provided a strong support to prove the scientific approaches presented in the previous chapters. Indeed, the obtained results are very promising. Therefore, we believe that it is worth to further investigate in the research axes we have presented in this chapter, as well as autonomic solutions for distributed vulnerabilities and remediation activities in an consistent and unified manner.

Conclusion

Chapter 10

General conclusion

Contents

10.1 Contributions summary 145

10.1.1 Autonomic vulnerability management 146

10.1.2 Implementation prototypes 147

10.2 Perspectives 148

10.2.1 Proactive autonomic defense by anticipating future vulnerable states 148

10.2.2 Unified autonomic management platform 148

10.2.3 Autonomic security for current and emerging technologies 148

10.3 List of publications 149

10.1 Contributions summary

The large-scale deployment of disparate computing devices over evolving dynamic networks has profusely augmented the complexity of network management. Indeed, computing technologies spread fast, the number of end-users increases rapidly, and there is a constant demand for more and better services. Underneath, computer networks constitute the platform of this convoluted digital world. This accelerated evolution has tested the boundaries of traditional network management approaches, which do not scale properly with today's network requirements. In other words, network management tasks have become so intensive and diversified, that the model, *one administrator per n computers*, is not effective anymore. Therefore, a need for new management approaches became evident. In light of this, the autonomic computing paradigm has emerged in order to cope with this new and challenging landscape. By specifying high-level objectives, autonomic computing aims at delegating management activities to the networks themselves. In this manner, human administrators avoid the execution of heavy and error-prone tasks, becoming able to focus on higher levels of management issues, with a simpler and cleaner view of the network.

Autonomic computing has become a very important research field within the scientific community, featuring strong foundations and promising perspectives. However, two main points require special attention. First, security issues have been poorly discussed in autonomic environments, particularly, vulnerability management mechanisms. Second, the expertise obtained from autonomic approaches has been barely experimented in non-autonomic environments. In this thesis, we have pursued both goals ; to investigate and develop novel vulnerability management approaches for autonomic environments, and to transfer autonomic principles to non-autonomic scenarios. In this chapter, we provide general conclusions about our research work as well as our technical implementations. Finally, we present research perspectives and further work.

10.1.1 Autonomic vulnerability management

Our contributions can be classified in two main categories according to the vulnerability management process, namely, vulnerability assessment and vulnerability remediation. We describe both in the following subsections.

Autonomic vulnerability assessment

Vulnerability assessment constitutes the first step within the vulnerability management process. It is a critical activity that enables computer systems to increase their awareness about security threats. In this thesis, we have presented several approaches for autonomously assessing vulnerabilities in different scenarios. First, we have proposed an approach that integrates OVAL vulnerability descriptions into the autonomic management plane. By translating these security advisories into Cfengine policy rules, autonomic agents deployed across the network become able to analyze their own security exposure. This approach targets the autonomic assessment of device-based vulnerabilities, with a particular focus on the Cisco IOS platform. However, there exist other scenarios where vulnerability assessment techniques are required as well. In that context, we have proposed three dimensional research axes which involves distributed vulnerabilities (**spatial dimension**), past hidden vulnerable states (**temporal dimension**), and mobile security assessment (**technological dimension**).

Our first research axis involves distributed vulnerabilities, which constitute an extension to the concept of device-based vulnerabilities within the spatial dimension. The concept of distributed vulnerabilities considers situations where two or more devices under specific conditions may present safe states, but when combined across the network, a vulnerable state arises. These scenarios are real and must be taken into account by vulnerability assessment approaches. Our second research axis involves the time dimension, where we have presented an approach for autonomously increasing the security of present computer systems by analyzing past hidden vulnerable states. Security advisories may be available late in time, but the vulnerabilities expressed by them can have been unknowingly active in the past. By autonomously building an historical image repository of the systems under surveillance, our approach is able to identify periods of security exposure due to unknown vulnerabilities at that time, where malicious activities may have taken place. This feature may allow forensic activities to be performed in order to identify current security breaches. Our third and final research axis in the context of vulnerability assessment involves the technological dimension, where we have presented an approach for autonomously assessing vulnerabilities in mobile environments. Indeed, we have proposed two complimentary approaches that deal with the assessment activity over resource-constrained devices. First, we have introduced a lightweight autonomous vulnerability assessment service that permits Android devices to assess their own exposure. Then, we have extended this approach by considering a probabilistic framework where assessment activities are outsourced to an external server which controls the overall assessment process, thus decreasing the workload of mobile clients even more.

Autonomic vulnerability remediation

In order to close the vulnerability management control loop, vulnerability remediation mechanisms are required. In that context, we have proposed two autonomic vulnerability remediation approaches focused on device-based and distributed vulnerabilities respectively. These activities however, are particularly challenging because they should not generate new vulnerable states when perform their operations. In order to remediate device-based vulnerabilities, we have modeled the set of all known vulnerability descriptions as a conjunction of propositional logical formulas. A vulnerable device will therefore make this formula evaluate to *true*. Because we look for safe configurations, we need to find which properties must be modified to make this formula evaluate to *false*, or which amounts to the same, make its negation evaluate to *true*. To this end, we have encoded our problem as a SAT problem, where properties that cannot be changed are fixed, and those for which correction actions exist are freed. A solution provided by a SAT solver describes a safe configuration. Within our experiments, we have used the NETCONF protocol for performing effective management operations over the Cisco IOS platform. The obtained results confirm the feasibility of our approach. Within our second approach, we have proposed a collaborative mechanism for describing and performing treatments of distributed vulnerabilities in autonomic networks and systems. This approach also considers correction advisories that are taken into account by our framework, which is able to remediate vulnerable states found across the network. To do this, a distributed algorithm is executed over the Cfengine system. There, each Cfengine agent involved in the vulnerability under analysis will report the activities it can perform to eradicate the threat as well as the cost to do this. This information is collected from all the involved nodes and analyzed at the Cfengine server. The Cfengine server then selects a node to apply corrective actions based on the reported costs. Even though there is not a complete prototype implementation of our second approach, we have performed an analytical evaluation of its performance, obtaining successful linear costs when it is integrated into the vulnerability management process.

10.1.2 Implementation prototypes

With the objective of technically proving the feasibility of our previous contributions, we have developed three implementation prototypes which correspond to three different vulnerability assessment scenarios. First, we have developed Ovalyzer, an OVAL to Cfengine translation system. Ovalyzer enables the generation of Cfengine policy rules that represent OVAL vulnerability descriptions. In this manner, these policies are deployed to autonomic agents which become able to perform self-assessment activities. Second, we have implemented a prototype for identifying past unknown security exposures. We have reused the idea behind Ovalyzer, but this time for autonomously generating XML-based images of the states of the systems being monitored. These images are stored in a cost-efficient manner by using an SVN repository. When new vulnerability descriptions become available, our prototype is able to analyze the history of system images looking for vulnerable periods according to this new information. Therefore, forensic activities can be performed over those identified exposure periods in order to analyze security breaches that may still compromise the security policies in the present. Finally, we have developed Ovaldroid, an OVAL-based vulnerability assessment framework for Android. Indeed, our first approach considers a lightweight self-assessment service running inside the mobile device. In order to further reduce the load on the mobile side, we have implemented our probabilistic approach that free mobile devices from performing assessment activities themselves. Instead, mobile devices periodically notify their availability for being analyzed, and receive directives from the server indicating which data must be collected and reported. Vulnerability assessment activities are outsourced in the server and then notified to the mobile client. In addition, the server conducts special algorithms that allows to decrease the communication with mobile devices even more. All these prototypes have served as a computational infrastructure to prove the feasibility and scalability of our autonomic approaches.

10.2 Perspectives

10.2.1 Proactive autonomic defense by anticipating future vulnerable states

During our research work, we have analyzed different research dimensions for vulnerability management. In particular, we have proposed approaches for analyzing vulnerabilities in the present and the past, which in turn are complemented with vulnerability remediation approaches. However, what if we could anticipate the trajectory of a system, considering its dynamic state, and avoid changes that will lead the system to known vulnerable spaces ? In other words, we have shown in this thesis how system states can be characterized by the properties they present. We also know how to model vulnerabilities by characterizing the properties expected to be observed on a target system. Considering that we have n properties we can model, a target system could be graphically located on a single point of an n-dimensional space. In the same manner, known vulnerabilities would have their corresponding points in such space. Similar vulnerabilities would probably conform clusters or vulnerable subspaces. Our idea is that observing the movement of a target system, its trend could be monitored and determined on this space. If such a trajectory indicates high closeness levels to vulnerable states, it could be deviated by averting changes that may get the system closer or even fall into these vulnerable subspaces. This approach could provide autonomic systems with a continuous metric of vulnerability awareness that can be taken into account when management operations are performed, and therefore enabling systems to anticipate and avoid vulnerable configuration states.

10.2.2 Unified autonomic management platform

In this thesis we have presented several approaches for tackling different needs of vulnerability management in the context of autonomic environments. However, these approaches need to be unified, over a common and consistent platform, able to provide all these features in a seamless manner. Changes that can lead a system to secure states may contradict existing operational requirements. This issue poses a hard problem that should be addressed. Therefore, a main challenge is to provide mechanisms able to coexist with other policy-based systems, maintaining coherency at all levels, including operational and security perspectives. The approaches proposed in this work reinforce the security of a network from different perspectives, making it more reliable and stronger. However, our models should be extended and unified so as to cover these activities as a whole. In particular, our approach for managing distributed vulnerabilities requires more technical work, as well as further investigation on the metrics required to collaboratively perform forensic and remediation tasks. Therefore, the construction of a standard model and a system able to contemplate all these aspects under a single view, would be extremely useful for the community of autonomic computing, as a basis for autonomously managing vulnerabilities.

10.2.3 Autonomic security for current and emerging technologies

The security enhancement of current paradigms such as cloud computing, and emerging models like software-defined networks (SDN) and Internet of Things (IoT), is also extremely challenging. Briefly, cloud computing tackles availability and processing power by decoupling services from the underlying hardware. More recently, SDNs also separate the management (control plane) from the hardware that actually implement network functionalities (data plane). Both approaches aim at providing reliable and scalable services while decreasing the complexity of their management and accomplishment. This is where the autonomic perspective can be extremely helpful. By providing self-configuration and self-protection mechanisms, these operational management models can also become scalable and resilient in the security plane, which is essential to achieve reliability. IoT on the other hand, gives rise to a tremendous and vertiginous growth of disparate interconnected devices. This trend clearly states a need for scalable and adaptive management mechanisms, where their security must be also as much autonomous as these mechanisms will be. In that context, autonomic security solutions might be a key element in the evolution of this new challenging landscape.

10.3 List of publications

International peer-reviewed journals

◇ **Martín Barrère**, Rémi Badonnel, and Olivier Festor. *Vulnerability Assessment in Autonomic Networks and Services: a Survey.* **IEEE Communications Surveys & Tutorials**, 16(2):988-1004, May 2014. (Impact factor at acceptance date: 6.311).

Book chapters

◇ **Martín Barrère**, Gaëtan Hurel, Rémi Badonnel, and Olivier Festor. *Increasing Android Security using a Lightweight OVAL-based Vulnerability Assessment Framework.* In Automated Security Management, E. Al-Shaer et al, Eds. Springer International Publishing, 2013, ch. 3, pp. 41-58, ISBN: 978-3-319-01432-6. Book chapter based on our paper selected from the 5th International Symposium on Configuration Analytics and Automation (**SafeConfig'12**), October 3-4, 2012, Baltimore, USA.

International peer-reviewed conferences

◇ **Martín Barrère**, Rémi Badonnel, and Olivier Festor. *A SAT-based Autonomous Strategy for Security Vulnerability Management.* In Proceedings of the IEEE/IFIP Network Operations and Management Symposium (**NOMS'14**), Mini-Conference, May 5-9, 2014, Krakow, Poland.

◇ **Martín Barrère**, Gaëtan Hurel, Rémi Badonnel, and Olivier Festor. *A Probabilistic Cost-efficient Approach for Mobile Security Assessment.* In Proceedings of the 9th IEEE International Conference on Network and Service Management (**CNSM'13**), October 14-18, 2013, Zürich, Switzerland. (Acceptance rate 18.1%, 21 out of 116 papers).

◇ **Martín Barrère**, Rémi Badonnel, and Olivier Festor. *Improving Present Security through the Detection of Past Hidden Vulnerable States.* In Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (**IM'13**), Mini-Conference, May 27-31, 2013, Ghent, Belgium.

◇ **Martín Barrère**, Rémi Badonnel, and Olivier Festor. *Collaborative Remediation of Configuration Vulnerabilities in Autonomic Networks and Systems.* In Proceedings of the 8th IEEE International Conference on Network and Service Management (**CNSM'12**), Mini-Conference, October 22-26, 2012, Las Vegas, USA.

- **Martín Barrère**, Rémi Badonnel, and Olivier Festor. *Towards the Assessment of Distributed Vulnerabilities in Autonomic Networks and Systems.* In Proceedings of the IEEE/IFIP International Network Operations and Management Symposium (**NOMS'12**), April 16-20, 2012, Maui, Hawaii, USA. (Acceptance rate 26.2%, 55 out of 210 papers).

- **Martín Barrère**, Rémi Badonnel, and Olivier Festor. *Supporting Vulnerability Awareness in Autonomic Networks and Systems with OVAL.* In Proceedings of the 7th IEEE International Conference on Network and Service Management (**CNSM'11**), October 24-28, 2011, Paris, France. (Acceptance rate 14.6%, 24 out of 164 papers).

- **Martín Barrère**, Gustavo Betarte, and Marcelo Rodríguez. *Towards Machine-assisted Formal Procedures for the Collection of Digital Evidence.* In Proceedings of the 9th IEEE Annual International Conference on Privacy, Security and Trust (**PST'11**), July 19-21, 2011, Montreal, Canada.

- **Martín Barrère**, Rémi Badonnel, and Olivier Festor. *Towards Vulnerability Prevention in Autonomic Networks and Systems.* In Proceedings of the 5th International Conference on Autonomous Infrastructure, Management and Security (**AIMS'11**), Ph.D. Symposium, Springer, June 13-17, 2011, Nancy, France.

Demonstrations and seminars

- **Martín Barrère**, Gaëtan Hurel, Rémi Badonnel, and Olivier Festor. *Ovaldroid: an OVAL-based Vulnerability Assessment Framework for Android.* Demonstration Sessions of the IFIP/IEEE International Symposium on Integrated Network Management (**IM'13**), May 27-31, 2013, Ghent, Belgium.

- **Martín Barrère**. *Vulnerability Management for Safe Configurations in Autonomic Networks and Systems.* Ph.D. Seminar, **NSS Department of Loria**, March 28, 2013, Nancy, France.

- **Martín Barrère**, Rémi Badonnel, and Olivier Festor. *Ovalyzer: an OVAL to Cfengine Translator.* Ph.D. Student Demo Contest of the IEEE/IFIP International Network Operations and Management Symposium (**NOMS'12**), April 16-20, 2012, Maui, Hawaii, USA.

Bibliography

[1] M. Abedin, S. Nessa, E. Al-Shaer, and L. Khan. Vulnerability Analysis for Evaluating Quality of Protection of Security Policies. *In Proceedings of the 2nd ACM Workshop on Quality of Protection (QoP'06)*, 2006.

[2] H. Achi, A. Hellany, and M. Nagrial. Network Security Approach for Digital Forensics Analysis. *In Proceedings of the International Conference on Computer Engineering and Systems (CCES'08)*, pages 263–267, November 2008.

[3] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman. A Survey of Information-Centric Networking. *IEEE Communications Magazine*, 50(7):26–36, 2012.

[4] M. S. Ahmed, E. Al-Shaer, and L. Khan. A Novel Quantitative Approach For Measuring Network Security. In *Proceedings of the 27th IEEE Conference on Computer Communications (INFOCOM'08)*, pages 1957 –1965, April 2008.

[5] M. S. Ahmed, E. Al-Shaer, M. M. Taibah, M. Abedin, and L. Khan. Towards Autonomic Risk-aware Security Configuration. *Proceedings of the IEEE Network Operations and Management Symposium (NOMS'08)*, pages 722–725, April 2008.

[6] CNF Transformer. `https://code.google.com/p/aima-java/`. Last visited on November, 2013.

[7] Akamai Technologies, Inc. `http://www.akamai.com/`. Last visited on November, 2013.

[8] L. Akue, E. Lavinal, T. Desprats, and M. Sibilla. Runtime Configuration Validation for Self-Configurable Systems. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IM'13)*, pages 712–715, 2013.

[9] L. Akue, E. Lavinal, and M. Sibilla. Towards a Validation Framework for Dynamic Reconfiguration. In *Proceedings of the 6th IEEE International Conference on Network and Service Management (CNSM'10)*, pages 314–317, 2010.

[10] M. Albanese, S. Jajodia, and S. Noel. Time-Efficient and Cost-Effective Network Hardening Using Attack Graphs. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'12)*, pages 1–12, 2012.

[11] Android. `http://www.android.com/`. Last visited on November, 2013.

[12] Android Developers. `http://developer.android.com/`. Last visited on November, 2013.

[13] Apple iOS. `http://www.apple.com/ios/`. Last visited on November, 2013.

[14] Th. Arampatzis, J. Lygeros, and S. Manesis. A Survey of Applications of Wireless Sensors and Wireless Sensor Networks. In *Proceedings of the 13th Mediterrean Conference on Control and Automation*, pages 719–724, 2005.

[15] The Advanced Research Projects Agency Network (ARPANET). `http://en.wikipedia.org/wiki/ARPANET`. Last visited on November, 2013.

[16] E. Asmare, A. Gopalan, M. Sloman, N. Dulay, and E. Lupu. Self-Management Framework for Mobile Autonomous Systems. *J. Network Syst. Manage.*, 20(2):244–275, 2012.

[17] K. J. Astrom and R. M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, April 2008.

[18] L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, October 2010.

[19] AVDL. `http://www.oasis-open.org/`. Last visited on November, 2013.

[20] R. E. Ball. *The Fundamentals of Aircraft Combat Survivability Analysis and Design, 2nd Edition*. AIAA Education Series. American Institute of Aeronautics and Astronautics, Inc., 2003.

[21] J. Banghart and C. Johnson. The Technical Specification for the Security Content Automation Protocol (SCAP). Nist Special Publication. `http://scap.nist.gov/revision/`, 2011. Last visited on January, 2013.

[22] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani. Data Center Network Virtualization: A Survey. *IEEE Communications Surveys & Tutorials*, 15(2):909–928, 2013.

[23] N. Bari, G. Mani, and S. Berkovich. Internet of Things as a Methodological Concept. In *Proceedings of the Fourth International Conference on Computing for Geospatial Research and Application (COM.Geo)*, pages 48–55, 2013.

[24] M. Barrère, R. Badonnel, and O. Festor. Supporting Vulnerability Awareness in Autonomic Networks and Systems with OVAL. *In Proceedings of the 7th IEEE International Conference on Network and Service Management (CNSM'11)*, October 2011.

[25] M. Barrère, R. Badonnel, and O. Festor. Towards the Assessment of Distributed Vulnerabilities in Autonomic Networks and Systems. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS'12)*, pages 335–342, April 2012.

[26] M. Barrère, G. Betarte, and M. Rodríguez. Towards Machine-assisted Formal Procedures for the Collection of Digital Evidence. In *Proceedings of the 9th Annual International Conference on Privacy, Security and Trust (PST'11)*, pages 32 –35, July 2011.

[27] A. Bartel, J. Klein, M. Monperrus, and Y. L. Traon. Automatically Securing Permission-Based Software by Reducing the Attack Surface: An Application to Android. *CoRR*, abs/1206.5829, 2012.

[28] BitTorrent. `http://www.bittorrent.com/`. Last visited on November, 2013.

[29] R. Bohme. Vulnerability Markets. What is the Economic Value of a Zero-Day Exploit ? In *Proceedings of the 22nd Chaos Communication Congress*, December 2005.

[30] 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.

[31] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging {IT} platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599 – 616, 2009.

[32] J. Caballero, Z. Liang, P. Poosankam, and D. Song. Towards Generating High Coverage Vulnerability-Based Signatures with Protocol-Level Constraint-Guided Exploration. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID'09)*, pages 161–181. Springer-Verlag, 2009.

[33] CAPEC, Common Attack Pattern Enumeration and Classification. `http://capec.mitre.org/`. Last visited on November, 2013.

[34] M. Castillo, F. Farina, A. Cordoba, and J. Villadangos. A Modified O(n) Leader Election Algorithm for Complete Networks. In *Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP '07, pages 189–198, Washington, DC, USA, 2007. IEEE Computer Society.

[35] CCE, Common Configuration Enumeration. `http://cce.mitre.org/`. Last visited on November, 2013.

[36] The Content-Centric Network Project. `http://www.ccnx.org/`. Last visited on November, 2013.

[37] CEE, Common Event Expression. `http://cee.mitre.org/`. Last visited on November, 2013.

[38] Cfengine. `http://www.cfengine.com/`. Last visited on November, 2013.

[39] Chef. `http://www.getchef.com/chef/`. Last visited on November, 2013.

[40] F. Chiang, J. Agbinya, and R. Braun. Risk and Vulnerability Assessment of Secure Autonomic Communication Networks. In *Proceedings of the 2nd International Conference on Wireless Broadband and Ultra Wideband Communications (AusWireless 2007)*, pages 40–40. IEEE, August 2007.

[41] 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, 2008.

[42] Cisco Visual Networking Index. `http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.html`. Last visited on November, 2013.

[43] S. A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[44] V. Corey, C. Peterman, S. Shearin, M.S. Greenberg, and J. Van Bokkelen. Network Forensics Analysis. *Internet Computing, IEEE*, 6(6):60 – 66, Nov 2002.

[45] CPE, Common Platform Enumeration. `http://cpe.mitre.org/`. Last visited on November, 2013.

[46] CVE, Common Vulnerabilities and Exposures. `http://cve.mitre.org/`. Last visited on November, 2013.

[47] CVSS, Common Vulnerability Scoring System. `http://www.first.org/cvss/`. Last visited on November, 2013.

[48] CybOX, Cyber Observable eXpression. `http://cybox.mitre.org/`. Last visited on November, 2013.

[49] O. Dabbebi, R. Badonnel, and O. Festor. Dynamic Exposure Control in P2PSIP Networks. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS'12)*, pages 261–268, April 2012.

[50] H. Dai, C. Murphy, and G. Kaiser. Configuration Fuzzing for Software Vulnerability Detection. *2010 International Conference on Availability, Reliability and Security*, pages 525–530, February 2010.

[51] Dalvik Virtual Machine. `http://www.dalvikvm.com/`. Last visited on November, 2013.

[52] Datalog User Manual. `http://www.ccs.neu.edu/home/ramsdell/tools/datalog/datalog.html`. Last visited on November, 2013.

[53] J. Demott. The Evolving Art of Fuzzing. Software Testing. `http://vdalabs.com/tools/The_Evolving_Art_of_Fuzzing.pdf`, 2006. Last visited on January, 2013.

[54] A Road Map for Digital Forensic Research. In Report From the First Digital Forensic Research Workshop (DFRWS). `http://www.dfrws.org/2001/dfrws-rm-final.pdf`, August 2001.

[55] DHS, Department of Homeland Security. `http://www.dhs.gov/`. Last visited on November, 2013.

[56] Y. Diao, A. Keller, S. Parekh, and V. V. Marinov. Predicting Labor Cost through IT Management Complexity Metrics. In *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management (IM'07)*, pages 274–283. IEEE, May 2007.

[57] S. Dobson, F. Zambonelli, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, and N. Schmidt. A Survey of Autonomic Communications. *ACM Transactions on Autonomous and Adaptive Systems*, 1(2):223–259, December 2006.

[58] Dynamips/Dynagen Cisco Router Emulator. `http://www.dynagen.org/`. Last visited on November, 2013.

[59] Eclipse. `http://www.eclipse.org/`. Last visited on November, 2013.

[60] eMule. `http://www.emule-project.net/`. Last visited on November, 2013.

[61] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11. USENIX Association, 2011.

[62] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *Security Privacy, IEEE*, 7(1):50–57, January-February 2009.

[63] R. Enns, M. Bjorklund, J. Schönwälder, and A. Bierman. RFC 6241, Network Configuration Protocol (NETCONF). `http://tools.ietf.org/html/rfc6241`, June 2011.

[64] R. P. Esteves, L. Z. Granville, and R. Boutaba. On the Management of Virtual Networks. *IEEE Communications Magazine*, 51(7):80–88, 2013.

[65] European Telecommunications Standards Institute (ETSI). Network Functions Virtualisation: Introductory White Paper. `http://portal.etsi.org/NFV/NFV_White_Paper.pdf`. Last visited on November, 2013.

[66] Expect, NIST. `http://www.nist.gov/el/msid/expect.cfm`. Last visited on November, 2013.

[67] Facebook. `http://www.facebook.com/`. Last visited on November, 2013.

[68] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A Survey of Mobile Malware in the Wild. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 3–14, New York, NY, USA, 2011. ACM.

[69] R. Fielding. Architectural Styles and the Design of Network-based Software Architectures, PhD. Dissertation, 2000. `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`. Last visited on November, 2013.

[70] Flickr. `http://www.flickr.com/`. Last visited on November, 2013.

[71] P. Foreman. *Vulnerability Management*. Information Security. CRC Press, 2009.

[72] S. Frei, D. Schatzmann, B. Plattner, and B. Trammel. Modelling the Security Ecosystem - The Dynamics of (In)Security. In *Proceedings of the Workshop on the Economics of Information Security (WEIS'09)*, June 2009.

[73] H. Garcia-Molina. Elections in a Distributed Computing System. *IEEE Trans. Comput.*, 31(1):48–59, January 1982.

[74] Gartner. `http://www.gartner.com`. Last visited on November, 2013.

[75] GNS3. `http://www.gns3.net/`. Last visited on November, 2013.

[76] The Greenery Project. `http://qntm.org/greenery`. Last visited on November, 2013.

[77] D. J. Hand, P. Smyth, and H. Mannila. *Principles of Data Mining*. MIT Press, Cambridge, MA, USA, 2001.

[78] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[79] M. C. Huebscher and J. A. McCann. A Survey of Autonomic Computing–Degrees, Models, and Applications. *ACM Comput. Surv.*, 40:7:1–7:28, August 2008.

[80] IBM. `http://www.ibm.com/`. Last visited on November, 2013.

[81] IBM. An Architectural Blueprint for Autonomic Computing. *IBM White Paper*, 2006.

[82] IBM Autonomic Computing Deployment Model. `http://www-03.ibm.com/press/us/en/pressrelease/464.wss`. Last visited on November, 2013.

[83] Milestones:Inception of the ARPANET, 1969. `http://www.ieeeghn.org/wiki/index.php/Milestones:Inception_of_the_ARPANET,_1969`. Last visited on November, 2013.

[84] The Internet Engineering Task Force (IETF). `http://www.ietf.org/`. Last visited on November, 2013.

[85] V. Igure and R. Williams. Taxonomies of Attacks and Vulnerabilities in Computer Systems. *IEEE Communications Surveys & Tutorials*, 10(1):6–19, January 2008.

[86] Cisco IOS. `http://www.cisco.com/`. Last visited on November, 2013.

[87] ITSM - IT Service Management. `http://www.itsm.info/`. Last visited on November, 2013.

[88] Java technology. `http://www.oracle.com/technetwork/java/`. Last visited on November, 2013.

[89] Java Architecture for XML Binding. `http://java.sun.com/developer/technicalArticles/WebServices/jaxb/`. Last visited on November, 2013.

[90] G. P. Joshi, S. Y. Nam, and S. W. Kim. Cognitive Radio Wireless Sensor Networks: Applications, Challenges and Research Trends. *Sensors*, 13(9):11196–11228, 2013.

[91] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, January 2003.

[92] M. J. Khan, M. M. Awais, and S. Shamail. Enabling Self-Configuration in Autonomic Systems Using Case-Based Reasoning with Improved Efficiency. *In Proceedings of the 4th International Conference on Autonomic and Autonomous Systems (ICAS'08)*, pages 112–117, March 2008.

[93] J. Ko, A. Terzis, S. Dawson-Haggerty, D. E. Culler, J. W. Hui, and P. Levis. Connecting Low-Power and Lossy Networks to the Internet. *IEEE Communications Magazine*, 49(4):96–101, April 2011.

[94] Y. Kwon, H. J. Lee, and G. Lee. A Vulnerability Assessment Tool Based on OVAL in Linux System. *Network and Parallel Computing*, pages 653–660, 2004.

[95] D. Lazer, A. Pentland, L. Adamic, S. Aral, A. L. Barabási, D. Brewer, N. Christakis, N. Contractor, J. Fowler, M. Gutmann, T. Jebara, G. King, M. Macy, D. Roy, and M. Van Alstyne. Life in the Network: The Coming Age of Computational Social. *Science*, 323(5915):721–723, 2009.

[96] S. Li. Juxtapp and DStruct: Detection of Similarity Among Android Applications. Master's thesis, EECS Department, University of California, Berkeley, May 2012.

[97] Limelight Networks. `http://www.limelight.com/`. Last visited on November, 2013.

[98] LinkedIn. `http://www.linkedin.com/`. Last visited on November, 2013.

[99] R.P. Lippmann, K.W. Ingols, and Lincoln Laboratory. *An Annotated Review of Past Papers on Attack Graphs*. Project report IA. Massachusetts Institute of Technology, Lincoln Laboratory, 2005.

[100] Lookout Mobile Security. `https://www.mylookout.com/mobile-threat-report`. Last visited on November, 2013.

[101] G. F. Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, USA, 2009.

[102] MAEC, Malware Attribute Enumeration and Characterization. `http://maec.mitre.org/`. Last visited on November, 2013.

[103] P. Mell, T. Bergeron, and D. Henning. Creating a Patch and Vulnerability Management Program. *NIST*, November 2005.

[104] MITRE Corporation. `http://www.mitre.org/`. Last visited on November, 2013.

[105] Z. Movahedi, M. Ayari, R. Langar, and G. Pujolle. A Survey of Autonomic Network Architectures and Evaluation Criteria. *IEEE Communications Surveys & Tutorials*, PP:1–27, May 2011.

[106] Nessus. `http://www.tenable.com/products/nessus`. Last visited on November, 2013.

[107] Netconf4J. `https://github.com/dana-i2cat/netconf4j`. Last visited on November, 2013.

[108] NIST, National Institute of Standards and Technology. `http://www.nist.gov/`. Last visited on November, 2013.

[109] Nmap. `http://nmap.org/`. Last visited on November, 2013.

[110] Norton Mobile Security. `http://us.norton.com/norton-mobile-security/`. Last visited on November, 2013.

[111] NVD, National Vulnerability Database. `http://nvd.nist.gov/`. Last visited on November, 2013.

[112] Open Handset Alliance. `http://www.openhandsetalliance.com/`. Last visited on November, 2013.

[113] OpenVAS. `http://www.openvas.org/`. Last visited on November, 2013.

[114] OSVDB, The Open Source Vulnerability Database. `http://osvdb.org/`. Last visited on November, 2013.

[115] X. Ou, W. F. Boyer, and M. A. McQueen. A Scalable Approach to Attack Graph Generation. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS'06)*, pages 336–345. ACM Press, 2006.

[116] X. Ou, S. Govindavajhala, and A. W. Appel. MulVAL: A Logic-based Network Security Analyzer. *on USENIX Security*, 2005.

[117] The OVAL Language. `http://oval.mitre.org/`. Last visited on November, 2013.

[118] Ovaldi, the OVAL Interpreter reference implementation. `http://oval.mitre.org/language/interpreter.html`. Last visited on November, 2013.

[119] N. K. Pandey, S. K. Gupta, S. Leekha, and J. Zhou. ACML: Capability Based Attack Modeling Language. *In Proceedings of the 4th International Conference on Information Assurance and Security*, pages 147–154, September 2008.

[120] R. Patton. *Software Testing (2nd Edition)*. Sams, 2005.

[121] Picasa. `http://picasa.google.com/`. Last visited on November, 2013.

[122] N. Poolsappasit, R. Dewri, and I. Ray. Dynamic Security Risk Management Using Bayesian Attack Graphs. *IEEE Transactions on Dependable and Secure Computing*, 9(1):61–74, 2012.

[123] M.R. Prasad, A. Biere, and A. Gupta. A Survey of Recent Advances in SAT-Based Formal Verification. *STTT*, 7(2):156–173, 2005.

[124] Puppet. `http://www.puppetlabs.com/`. Last visited on November, 2013.

[125] RFC 2460, Internet Protocol (IPv6). `http://www.ietf.org/rfc/rfc2460.txt`. Last visited on November, 2013.

[126] RFC 4765. `http://www.ietf.org/rfc/rfc4765.txt`. Last visited on November, 2013.

[127] RFC 675, Internet Transmission Control Program. `http://tools.ietf.org/html/rfc675`. Last visited on November, 2013.

[128] RFC 791, Internet Protocol (IPv4). `http://tools.ietf.org/html/rfc791`. Last visited on November, 2013.

[129] RFC 793, Transmission Control Protocol. `http://tools.ietf.org/html/rfc793`. Last visited on November, 2013.

[130] D. Saha. Extending Logical Attack Graphs for Efficient Vulnerability Analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, pages 63–74, New York, NY, USA, 2008. ACM.

[131] J. Sahoo, S. Mohapatra, and R. Lath. Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues. In *Proceedings of the Second International Conference on Computer and Network Technology (ICCNT'10)*, pages 222–226. IEEE, April 2010.

[132] Saint. `http://www.saintcorporation.com/`. Last visited on November, 2013.

[133] N. Samaan and A. Karmouch. Towards Autonomic Network Management: an Analysis of Current and Future Research Directions. *IEEE Communications Surveys & Tutorials*, 11(3):22–36, July 2009.

[134] OVAL Language Sandbox. `http://oval.mitre.org/language/sandbox.html`. Last visited on November, 2013.

[135] Vulnerability Naming Schemas and Description Languages: CVE, Bugtraq, AVDL and VulnXML. The SANS Institute. `http://www.sans.org/`. Last visited on November, 2013.

[136] 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.

[137] K. Scarfone and T. Grance. A Framework for Measuring the Vulnerability of Hosts. *In Proceedings of the 1st International Conference on Information Technology (ICIT'08)*, pages 1–4, May 2008.

[138] T. Setzer, K. Bhattacharya, and H. Ludwig. Decision Support for Service Transition Management - Enforce Change Scheduling by Performing Change Risk and Business Impact Analysis. *In Proceedings of the IEEE Network Operations and Management Symposium (NOMS'08)*, pages 200–207, April 2008.

[139] C. Severance. Van Jacobson: Content-Centric Networking. *Computer*, 46(1):11–13, 2013.

[140] S. Sezer, S. Scott-Hayward, P.K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao. Are We Ready for SDN ? Implementation Challenges for Software-Defined Networks. *IEEE Communications Magazine*, 51(7):36–43, 2013.

[141] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google Android: A Comprehensive Security Assessment. *Security Privacy, IEEE*, 8(2):35–44, March-April 2010.

[142] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated Generation and Analysis of Attack Graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, SP '02, pages 273–, Washington, DC, USA, 2002. IEEE Computer Society.

[143] M. Sloman and E. Lupu. Engineering Policy-Based Ubiquitous Systems. *The Computer Journal*, 53(7):1113–1127, 2010.

[144] R. Sterritt, G. Garrity, E. Hanna, and P. O'Hagan. Survivable Security Systems Through Autonomicity. In *Proceedings of the Second international Conference on Radical Agent Concepts: innovative Concepts for Autonomic and Agent-Based Systems*, WRAC'05, pages 379–389, Berlin, Heidelberg, 2006. Springer-Verlag.

[145] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969. 10.1007/BF02165411.

[146] Apache Subversion. `http://subversion.apache.org/`. Last visited on November, 2013.

[147] SVNKit. `http://svnkit.com/`. Last visited on November, 2013.

[148] A. Tang, A. Nicholson, Y. Jin, and J. Han. Using Bayesian Belief Networks for Change Impact Analysis in Architecture Design. *Journal of Systems and Software*, 80(1):127–148, January 2007.

[149] Technology definition. `http://en.wikipedia.org/wiki/Technology`. Last visited on November, 2013.

[150] S. J. Templeton and K. Levitt. A Requires/Provides Model for Computer Attacks. *In Proceedings of the Workshop on New Security Paradigms (NSPW'00)*, pages 31–38, 2000.

[151] H. M. Tran and J. Schönwälder. Distributed Case-Based Reasoning for Fault Management. In *Proceedings of the 1st international conference on Autonomous Infrastructure, Management and Security: Inter-Domain Management (AIMS'07)*, pages 200–203, Berlin, Heidelberg, 2007. Springer-Verlag.

[152] H. M. Tran, I. Tumar, and J. Schönwälder. NETCONF Interoperability Testing. In *Proceedings of the Third International Conference on Autonomous Infrastructure, Management and Security (AIMS'09)*, pages 83–94, 2009.

[153] W. Truszkowski, H. Hallock, C. Rouff, J. Karlin, J. Rash, M. Hinchey, and R. Sterritt. *Autonomous and Autonomic Systems: With Applications to NASA Intelligent Spacecraft Operations and Exploration Systems.* Springer, 2009.

[154] The UniverSelf Project. `http://www.univerself-project.eu/`. Last visited on November, 2013.

[155] T. Vidas, D. Votipka, and N. Christin. All Your Droid Are Belong To Us: A Survey of Current Android Attacks. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[156] World Wide Web Consortium (W3C). `http://www.w3.org/`. Last visited on November, 2013.

[157] S. Wallin and C. Wikström. Automating Network and Service Configuration using NETCONF and YANG. In *Proceedings of the 25th International Conference on Large Installation System Administration*, LISA'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[158] J. A. Wang and M. Guo. OVM: An Ontology for Vulnerability Management. In *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies (CSIIRW'09)*, pages 34:1–34:4, New York, NY, USA, 2009. ACM.

[159] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'10)*, pages 497 –512, May 2010.

[160] W. Wang and T. E. Daniels. A Graph-based Approach Toward Network Forensics Analysis. *ACM Transactions on Information and System Security (TISSEC)*, 12(1), 2008.

[161] J. A. Wickboldt, L. A. Bianchin, and R. C. Lunardi. Improving IT Change Management Processes with Automated Risk Assessment. *In Proceedings of the IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'09)*, pages 71–84, 2009.

[162] A. Williams and M. Nicolett. Improve IT Security with Vulnerability Management. `http://www.gartner.com/id=480703`, 2005. Last visited on November, 2013.

[163] X-Ray for Android. `http://www.xray.io/`. Last visited on November, 2013.

[164] G. Xylomenos, C. Ververidis, V. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K. Katsaros, and G. Polyzos. A Survey of Information-Centric Networking Research. *IEEE Communications Surveys & Tutorials*, PP(99):1–26, 2013.

[165] YouTube. `http://www.youtube.com/`. Last visited on November, 2013.

[166] G. Zhang, S. Ehlert, T. Magedanz, and D. Sisalem. Denial of Service Attack and Prevention on SIP VoIP Infrastructures using DNS Flooding. In *Proceedings of the 1st International Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm'07)*, pages 57–66, New York, NY, USA, 2007. ACM.

[167] N. Ziring and S. D. Quinn. Specification for the Extensible Configuration Checklist Description Format (XCCDF). NIST (National Institute of Standards and Technology). `http://scap.nist.gov/specifications/xccdf/`. Last visited on January, 2013.

Annexes

Annexe A

CAS, a Configuration Assessment Service for UMF

Contents

A.1 Introduction and problem statement 163
A.2 Background 164
A.3 Configuration modeling for UMF 164
A.4 Configuration assessment service architecture 168
A.5 UMF, conclusions and perspectives 168

A.1 Introduction and problem statement

The UMF framework has been created with the purpose of providing a single and simple platform where autonomic elements (NEMs) can be deployed and activated in order to perform the actions and achieve the objectives the were built for [154]. However, each NEM has particular requirements and specific configurations in order to work properly. In addition, the interconnections between hundreds of NEMs and the services provided by them increase the complexity of their configuration. In that context, management problems such as configuration errors, conflicts between services and inconsistencies can occur leading to severe operational problems as well as security issues within the framework itself. Even though operating systems where NEMs are deployed and also the NEMs themselves may have security solutions to be protected, such fact does not ensure the security of the whole framework. Unexpected behaviour, failures, service unavailability and many other problems may occur if the UMF framework is not configured properly. In light of this, we propose in this document a configuration assessment service (CAS) for the UMF framework that provides the ability to both identify configuration errors and inconstancies, and at the same time, it can be used for evaluating whether recommendations and best practices are being considered as well. We propose the use of the OVAL [117] and DOVAL [25] languages for expressing expected or wrong configurations within standalone and distributed scenarios respectively. In this document we show how these languages can be used in the context of UMF and we also present an architecture that illustrates how the configuration assessment service can be integrated into the UMF framework.

A.2 Background

The OVAL language has been developed mostly thinking about describing vulnerabilities. However, its potential goes beyond such objective, it allows to describe machine states that depending on the context, these states can be understood as a bad thing (vulnerabilities) or a good thing (best practices). At the end, machine configurations and states are what OVAL is all about, and this adequately fits to the UMF requirements for analysing potential configuration problems such as service dependency inconsistencies, combinations of wrong NEM versions, incorrect values within configuration parameters and also known vulnerable NEMS. It also can be used for controlling the application of best practices. The usual or intuitive way to think about a vulnerability is to consider it as a combination of conditions or properties that if hold on a target system, the potential exploitable 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. In the OVAL language, a specific vulnerability is described using an OVAL definition. An OVAL definition specifies criteria that logically combine 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. The OVAL language provides a powerful basis for expressing configuration conditions of NEMs loaded in the framework that can be used for increasing their control and ensuring secure configurations. However, under a logical perspective, the OVAL language only allows to predicate over one component at a time, i.e., only unary predicates are possible [24]. Being NEMs the individuals of the universe over we are predicating on, such issue restricts the ability to define a predicate between two or more NEMs, i.e., n-ary predicates. In light of this, the DOVAL language has been proposed for covering distributed scenarios where the involvement of two or more devices is necessary to describe a distributed vulnerability. As explained before, the OVAL and DOVAL approach can be used not only for vulnerabilities but also for describing general configurations and states. The DOVAL language extends OVAL by permitting to express relationships between identified objects thus providing the ability to simultaneously predicate over a set of network devices more than standalone machines or systems.

A.3 Configuration modeling for UMF

In order to explain the proposed approach, we put forward three different scenarios using OVAL and DOVAL in the context of the UMF framework. Within these examples, we show features and limitations that need to be addressed in order to capture the characteristics of these scenarios. The first scenario uses the OVAL language and it is depicted in Figure A.1. Within such scenario, we aim at describing the configuration of two NEMs where one of them requires the other to be active. The first NEM acts as a SIP management NEM where audio calls can be performed through it. The second NEM provides load-balancing services including the ability to guarantee a minimum bandwidth for specific clients. In order to ensure quality in audio calls, the SIP management NEM requires the load-balancing service provided by the second NEM to be active. In that context, we need to specify two OVAL objects representing the NEMs (lines 21-28), on which two OVAL tests will be applied (lines 11-20), one test for each object (lines 12-15 and 16-19). In order to identify the required NEMs we use the entry <nem_name> to specify the NEM name, sip_management for the first NEM (line 23) and load_balancing for the second one (line 26). Within the tests, the conditions we expect to see over the objects are represented by OVAL states (lines 29-44). It is important to notice that the connection between one object and one state is defined by the test structure and the references used in it (lines 11-20). The first OVAL state (lines 30-36) applies over the first NEM, the SIP management service. In the proposed state, it is being declared that the SIP management NEM must have any version in the family 1.4. In addition, it is specified that the property active of this NEM must have the value 1. Usually, state's attributes used in the OVAL language are atomic. The structured attribute called nem_property with children nem_property_key and nem_property_value is an extension in terms of its normal use. With this OVAL state specifying a specific version and that the NEM must be active, we are expressing one of the two conditions (line 6) for our OVAL definition representing a configuration error (line 4). The second condition follows the same idea where the load_balancing NEM must be inactive no matter what the version is. If both conditions hold (line 5), the specified configuration error is present in the UMF framework.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<oval_definitions ...>

  <definitions>
    <definition id="oval:umf:def:1" class="configuration error">
      <criteria operator="AND">
        <criterion test_ref="oval:umf:tst:1"/>
        <criterion test_ref="oval:umf:tst:2"/>
      </criteria>
    </definition>
  </definitions>

  <tests>
    <umf_nem_test id="oval:umf:tst:1">
      <object object_ref="oval:umf:obj:101"/>
      <state state_ref="oval:umf:ste:201"/>
    </umf_nem_test>

    <umf_nem_test id="oval:umf:tst:2">
      <object object_ref="oval:umf:obj:102"/>
      <state state_ref="oval:umf:ste:202"/>
    </umf_nem_test>
  </tests>

  <objects>
    <umf_nem_object id="oval:umf:obj:101">
      <nem_name>sip_server</nem_name>
    </umf_nem_object>

    <umf_nem_object id="oval:umf:obj:102">
      <nem_name>load_balancer</nem_name >
    </umf_nem_object>
  </objects>

  <states>
    <umf_nem_state id="oval:umf:ste:201">
      <nem_version operation="pattern match"> 1\.4\.* </nem_version>
      <nem_property operation="map">
        <nem_property_key> active </nem_property_key>
        <nem_property_value> 1 </nem_property_value>
      </nem_property>
    </umf_nem_state>

    <umf_nem_state id="oval:umf:ste:202">
      <nem_version operation="pattern match"> .* </nem_version>
      <nem_property operation="map">
        <nem_property_key> active </nem_property_key>
        <nem_property_value> 0 </nem_property_value>
      </nem_property>
    </umf_nem_state>
  </states>

</oval_definitions>
```

FIGURE A.1 – UMF configuration error description with OVAL

Even though the OVAL language can be very useful for describing NEM configurations as shown in the previous example, the relationship between them cannot be formally represented. In that context, we now present the same scenario using the DOVAL language, depicted in Figure A.2. The proposed DOVAL specification reuses existing OVAL tests and objects (NEMs) and provides the ability to describe relationships between those NEMs. The structure of a DOVAL document follows the same philosophy as an OVAL document, i.e., tests over objects expecting specific states. The main difference is that DOVAL allows referring more than one object within the DOVAL test section thus permitting to analyse states over such objects simultaneously. Within this scenario, only one test is defined (line 11) involving the two NEMs with their specific characteristics previously defined in the OVAL document. NEMs are referenced by using their OVAL ids (lines 12-13) so there is no need to redefine these objects in the DOVAL document. The expected state between these two objects is referenced in the test section (line 14) and specified in the state section (lines 26-30). The service state specifies that the service with name bandwidth_assurance (line 27) is set between the SIP management NEM as its consumer (line 28) and the load-balancing NEM as its provider (line 29). Such DOVAL document specifies the same situation shown in the first scenario but now the relationship between both NEMs is formally declared.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<doval_document>
  <doval_definitions>
    <doval_definition id="doval:fr.inria.doval:def:1" class="distributed_configuration_error">
      <criteria>
        <criterion test_ref="doval:fr.inria.doval:tst:4141"/>
      </criteria>
    </doval_definition>
  </doval_definitions>

  <tests>
    <doval_test id="doval:fr.inria.doval:tst:4141" comment="DOVAL test specifying service
         provisioning between 2 NEMs." check_existence="at_least_one_exists" check="at least one">
      <object device_ref="doval:fr.inria.doval:dev:222"/>
      <object device_ref="doval:fr.inria.doval:dev:256"/>
      <state state_ref="doval:fr.inria.doval:ste:7777"/>
    </doval_test>
  </tests>

  <objects> <!-- devices -->
    <device_object id="doval:fr.inria.doval:dev:222">
      <prop ovaltest_ref="oval:umf:tst:1"/> <!--SIP server NEM running-->
    </device_object>

    <device_object id="doval:fr.inria.doval:dev:256">
      <prop ovaltest_ref="oval:umf:tst:2"/> <!--Load balancer NEM inactive-->
    </device_object>
  </objects>

  <states> <!-- relationships -->
    <service_state id="doval:fr.inria.doval:ste:7777"> <!--Service config-->
      <name operation="equals"> bandwidth_assurance </name>
      <consumer device_ref="doval:fr.inria.doval:dev:222"/>
      <provider device_ref="doval:fr.inria.doval:dev:256"/>
    </service_state>
  </states>

</doval_definitions>
```

Figure A.2 – UMF distributed configuration error description with DOVAL

Both OVAL and DOVAL can be used for specifying best practices as well. We now present a more complex scenario illustrated in Figure A.3 using the DOVAL language and involving the same two NEMs, the SIP management service and the load balancer linked by a consumer-provider relationship, but now both of them are active. In addition, we want to express a condition between attributes of both NEMs that should always hold. Let us suppose that the SIP management service uses G.711 for encoding its phone calls, i.e., 64 Kb/s for each call. In addition, let us assume that the maximum number of clients while assuring certain quality threshold is set to 100. In its limit, the SIP management NEM would require at least 64*100=6400Kb=800KB assured bandwidth for effectively ensuring the expected quality. This means that the NEM in charge of ensuring the minimum bandwidth level, the load balancer NEM in this case, should be configured to provide at least this capacity. In order to specify this context, we use the same test reference for the SIP management NEM (line 20) though the test for the load-balancing NEM has been changed because it has to be active now (line 23). If these two NEMs are active and related (lines 27-31), we need to ensure that the capacity provided by the load-balancing NEM is sufficient enough for the SIP management NEM to ensure its quality service. To do so, a configuration state is defined (lines 32-40) stating that the bit rate used by the SIP service (line 35) multiplied by its maximum number of clients (line 36) should be less than or equals to the minimum assured bandwidth by the load-balancing NEM (line 38). Such description provides a formal specification of a configuration that should be taken into account (best practices) in order to avoid unexpected behaviour.

```
<?xml version="1.0" encoding="UTF-8"?>
<doval_document>
  <doval_definitions>
    <doval_definition id="doval:fr.inria.doval:def:2" class="distributed_configuration_best_practices
        ">
      <criteria>
        <criterion test_ref="doval:fr.inria.doval:tst:6262"/>
      </criteria>
    </doval_definition>
  </doval_definitions>

  <tests>
    <doval_test id="doval:fr.inria.doval:tst:6262" comment="DOVAL test specifying best practices
        between 2 NEMS. " check_existence="at_least_one_exists" check="at least one">
      <object device_ref="doval:fr.inria.doval:dev:222"/>
      <object device_ref="doval:fr.inria.doval:dev:300"/>
      <state state_ref="doval:fr.inria.doval:ste:4444"/>
    </doval_test>
  </tests>

  <objects> <!-- devices -->
    <device_object id="doval:fr.inria.doval:dev:222">
      <prop ovaltest_ref="oval:umf:tst:1"/> <!--SIP server NEM running-->
    </device_object>

    <device_object id="doval:fr.inria.doval:dev:300">
      <prop ovaltest_ref="oval:umf:tst:3"/> <!--Load balancer NEM active-->
    </device_object>
  </objects>

  <states> <!-- relationships -->
    <config_state id="doval:fr.inria.doval:ste:4444"> <!--Config. requirements-->
      <operation type="comparison" name="greater_than">
        <operation type="arithmetic" name="multiplication">
          <attribute name="bit_rate" device_ref="doval:fr.inria.doval:dev:222"/>
          <attribute name="number_of_clients" device_ref="doval:fr.inria.doval:dev:222"/>
        </operation>
        <attribute name="min_assured_bandwidth" device_ref="doval:fr.inria.doval:dev:300"/>
      </operation>
    </config_state>
  </states>

</doval_definitions>
```

FIGURE A.3 – DOVAL scenario for best practices

A.4 Configuration assessment service architecture

The ability to automatically assess configuration errors, vulnerabilities and also best practices provides a strong support for increasing the security awareness of the UMF framework. By consuming security advisories and configuration descriptions from a database, the CAS service gathers the required information from the UMF framework and analyses the configuration of components (typically NEMs) loaded in the framework in order to detect configuration inconsistencies that may lead to operational and security problems. The assessment results become then available for the UMF framework thus corrective actions can be performed if necessary.

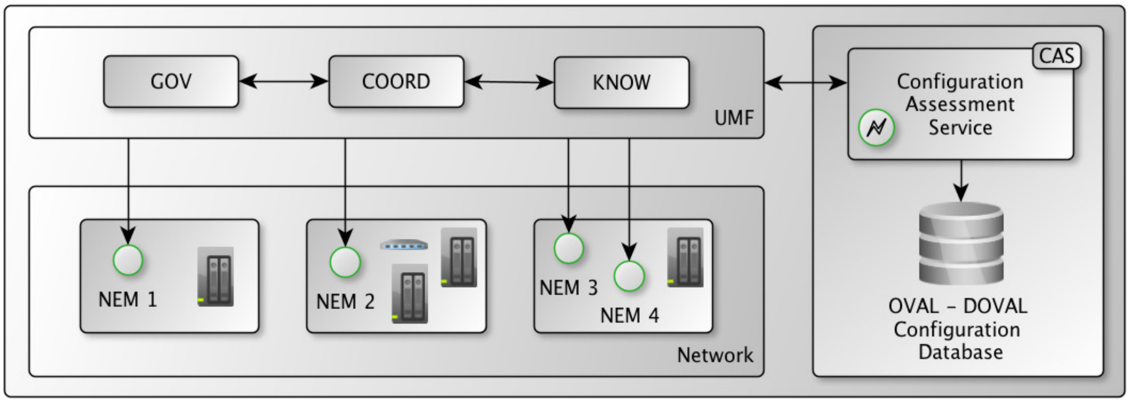

GOV
COORD
KNOW
UMF
CAS
Configuration Assessment Service
NEM 1
NEM 2
NEM 3
NEM 4
Network
OVAL - DOVAL Configuration Database

Figure A.4 – Configuration assessment service architecture

The positioning of the configuration assessment service is illustrated in Figure A.4. Within this architecture, the CAS service provides the UMF framework with the ability to analyse the configuration of NEMs loaded in the framework and to identify potential configuration and security problems. In order to do this, the UMF framework should provide through a web service for instance, information about the NEMs under control while the CAS service is in charge of correlating such information with security advisories and configuration descriptions present in the configuration database. These descriptions are specified with the OVAL and DOVAL languages for capturing standalone and distributed scenarios respectively. The integration of CAS within the UMF framework leverages the latter by providing self-assessment capabilities and increasing its overall security as well.

A.5 UMF, conclusions and perspectives

The UMF framework provides a unified platform for embedding autonomic solutions targeted on specific objectives called NEMs that together can provide autonomic management solutions for different needs and contexts. The integration of a configuration assessment service into the UMF framework can highly increase its security and stability by ensuring safe configurations among NEMs loaded in the platform. In this proposal, we have presented an approach for integrating such a service that provides the UMF framework with the ability to assess its own internal configuration. Our approach relies on the use of the OVAL and DOVAL languages for specifying configuration errors or situations that should not happen as well as best practices that should be integrated into the NEM management plane. Both OVAL and DOVAL require extensions to be integrated into the UMF framework so as interpreters, though it is feasible. The DOVAL language is currently under development though its applicability has already been shown in [25].

In order to integrate our solution into UMF it is also required to define interfaces, web services for instance, for exchanging required information between the UMF framework and the configuration assessment service. The formalization of the data required to analyse NEMs configurations has to be performed in order to extend the DOVAL language as to cover such requirements. In this document we have proposed some scenarios that exemplify and illustrate how our approach can be used for expressing configuration problems and best practices in the context of UMF. The integration of such service may highly enhance the overall stability of the UMF framework itself by increasing its configuration and security awareness.

UNIVERSITÉ DE LORRAINE

Loria
Laboratoire lorrain de recherche en informatique et ses applications

Inria
INVENTEURS DU MONDE NUMÉRIQUE

Résumé / Abstract

Le déploiement d'équipements informatiques à large échelle, sur les multiples infrastructures interconnectées de l'Internet, a eu un impact considérable sur la complexité de la tâche de gestion. L'informatique autonome permet de faire face à cet enjeu en spécifiant des objectifs de haut niveau et en déléguant autant que possible les activités de gestion aux réseaux et systèmes eux-mêmes. Cependant, lorsque des changements sont opérés par les administrateurs ou directement par les équipements autonomes, des configurations vulnérables peuvent être involontairement introduites, même si celles-ci sont correctes d'un point de vue opérationnel. Ces vulnérabilités offrent un point d'entrée pour des attaques de sécurité. Les environnements autonomes doivent être capables de se protéger pour éviter leur compromission et la perte de leur autonomie. À cet égard, les mécanismes de gestion des vulnérabilités sont essentiels pour assurer une configuration sûre de ces environnements.

Cette thèse porte sur la conception et le développement de nouvelles méthodes et techniques pour la gestion des vulnérabilités dans les réseaux et systèmes autonomes, afin de leur permettre de détecter, d'évaluer et de corriger leurs propres expositions aux failles de sécurité. Nous présenterons tout d'abord un état de l'art sur l'informatique autonome et la gestion de vulnérabilités, en mettant en relief les défis importants qui doivent être relevés dans ce cadre. Nous décrirons ensuite notre approche d'intégration du processus de gestion des vulnérabilités dans ces environnements, et en détaillerons les différentes facettes, notamment : extension de l'approche dans le cas de vulnérabilités distribuées, prise en compte du facteur temps en considérant une historisation des paramètres de configuration, et application en environnements contraints en utilisant des techniques probabilistes. Nous présenterons également les prototypes et les résultats expérimentaux qui ont permis d'évaluer ces différentes contributions.

Mots clés: sécurité, gestion de réseaux, informatique autonome, gestion de vulnérabilités.

Over the last years, the massive deployment of computing devices over disparate interconnected infrastructures has dramatically increased the complexity of network management. Autonomic computing has emerged as a novel paradigm to cope with this challenging reality. By specifying high-level objectives, autonomic computing aims at delegating management activities to the networks themselves. However, when changes are performed by administrators and self-governed entities, vulnerable configurations may be unknowingly introduced. Vulnerabilities constitute the main entry point for security attacks. Hence, self-governed entities unable to protect themselves will eventually get compromised and consequently, they will lose their own autonomic nature. In that context, vulnerability management mechanisms are vital to ensure safe configurations, and with them, the survivability of any autonomic environment.

This thesis targets the design and development of novel autonomous mechanisms for dealing with vulnerabilities, in order to increase the security of autonomic networks and systems. We first present a comprehensive state of the art in autonomic computing and vulnerability management, and point out important challenges that should be faced in order to fully integrate the vulnerability management process into the autonomic management plane. Afterwards, we present our contributions which include autonomic assessment strategies for device-based vulnerabilities and extensions in several dimensions, namely, distributed vulnerabilities (spatial), past hidden vulnerable states (temporal), and mobile security assessment (technological). In addition, we present vulnerability remediation approaches able to autonomously bring networks and systems into secure states. The scientific approaches presented in this thesis have been largely validated by an extensive set of experiments which are also discussed in this manuscript.

Keywords: security, network management, autonomic computing, vulnerability management.