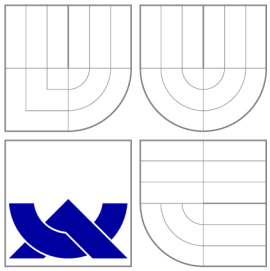
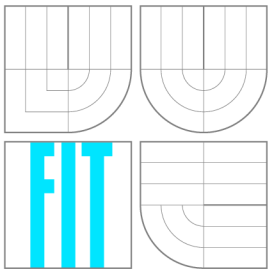


BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

A TOOL FOR DEVELOPMENT OF OVAL DEFINITIONS WITHIN OPENSCAP PROJECT

NÁSTROJ PRO TVORBU DEFINIC OVAL V PROJEKTU OPENSCAP

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE

JAN ČERNÝ

SUPERVISOR
VEDOUCÍ PRÁCE

Ing. ALEŠ SMRČKA, Ph.D.

BRNO 2016

Abstract

This thesis deals with the SCAP standard, used in area of computer security, and describes its open source implementation OpenSCAP. The analysis focuses on OVAL, a language for determining vulnerabilities and configuration issues on computer systems. Typical problems of OVAL are discussed. Based on obtained findings, an extension of the OpenSCAP project for reporting and diagnostics of OVAL interpretation has been designed. The thesis describes implementation, integration and testing of proposed extension.

Abstrakt

Tato práce se zabývá standardem SCAP používaným v oblasti počítačové bezpečnosti a popisuje jeho svobodnou implementaci OpenSCAP. V textu je analyzován jazyk OVAL sloužící pro popis zraniteností a bezpečné konfigurace systémů. Důraz je kladen na typické problémy tohoto jazyka. Na základě získaných poznatků je navrženo rozšíření projektu OpenSCAP o možnost reportování a diagnostiky průběhu interpretace jazyka OVAL. Práce následně popisuje implementaci, integraci a testování tohoto rozšíření.

Keywords

SCAP, OpenSCAP, OVAL, security audit, security standards, security policy, software analysis

Klíčová slova

SCAP, OpenSCAP, OVAL, bezpečnostní audit, bezpečnostní standardy, bezpečnostní politika, analýza softwaru

Reference

ČERNÝ, Jan. *A Tool for Development of OVAL Definitions within OpenSCAP Project*. Brno, 2016. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Smrčka Aleš.

A Tool for Development of OVAL Definitions within OpenSCAP Project

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Aleš Smrčka, Ph.D. and Mgr. Šimon Lukašík. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Jan Černý
May 18, 2016

Acknowledgements

I would like to thank my advisor Ing. Aleš Smrčka, Ph.D. and my consultant Mgr. Šimon Lukašík from Red Hat Czech for their professional help, valuable feedback and extraordinary support.

© Jan Černý, 2016.
This thesis was created as a school work at the Brno University of Technology, Faculty of Information Technology. The thesis is protected by copyright law and its use without author's explicit consent is illegal, except for cases defined by law.

Contents

1 Introduction **3**

2 Methods of Security Audit **4**
2.1 Security Compliance . . . 4
2.2 Vulnerability Assessment . . . 5
2.3 Security Guidances . . . 6
2.4 Security Content Automation Protocol . . . 6
2.4.1 Open Vulnerability and Assessment Language . . . 7
2.4.2 Extensible Configuration Checklist Description Format . . . 8
2.4.3 Script Check Engine . . . 8
2.5 OpenSCAP Project . . . 9
2.5.1 OpenSCAP Library . . . 9
2.5.2 OpenSCAP Command Line Tool . . . 10
2.5.3 SCAP Security Guide . . . 10
2.5.4 SCAP Workbench . . . 11
2.5.5 OpenSCAP Daemon . . . 12
2.5.6 OSCAP Anaconda Add-on . . . 12
2.6 Competitive Security Audit Tools . . . 12
2.7 Future of Security Audit . . . 13

3 Development of OVAL Definitions **14**
3.1 OVAL Documents . . . 14
3.2 Creating OVAL Definitions . . . 15
3.2.1 OVAL Definition . . . 15
3.2.2 OVAL Test . . . 16
3.2.3 OVAL Object . . . 16
3.2.4 OVAL State . . . 17
3.2.5 OVAL Variable . . . 18
3.3 Usage of OVAL . . . 18
3.4 Problems of OVAL . . . 19

4 Refining OVAL Development **22**
4.1 Requirements for OpenSCAP Modification . . . 22
4.2 Possible Approaches to Refining OVAL Development . . . 23
4.3 Reporting OVAL Evaluation . . . 24
4.3.1 Message Categories and Format . . . 25
4.4 Changes in OpenSCAP User Interface . . . 25
4.5 Implementation Details . . . 26

4.6 Problems Faced during Implementation 27

5 Evaluation of Extension **29**
5.1 Demonstration of Usage 29
5.2 User Testing 30
5.3 Feedback and Evaluation 31
5.4 Ideas for Further Improvements 32

6 Conclusion **33**

Bibliography **34**

Chapter 1

Introduction

Nowadays, nearly every organization uses information technology in its business. Computers control essential systems and store confidential information or personal data. With enormous progress of information technology, a risk of misuse rises significantly, so computer security becomes more and more important topic.

Systems need to be protected against attacks, intrusion, and data violence because those incidents may often lead to big damages or financial loss. Many techniques, like encryption, firewalls and others, were developed in order to prevent and mitigate those risks. One of often used technologies is the method of automated security audit. This approach is used to detect vulnerabilities and check compliance with a given security policy for computer systems. For automation of the security audit, the Security Compliance Automation Protocol (SCAP) has been standardized. Automation reduces necessary time and lowers risks of human factor.

The SCAP standard is widely used in industry and has several implementations. Currently popular is open source project OpenSCAP, that is available in various Linux distributions.

SCAP standard provides various languages to represent security policies, prepare automated checklists or store results of system assessment. One of them is The Open Vulnerability and Assessment Language (OVAL) that is mostly used to write definitions describing configuration state of some system.

However, OVAL is a difficult language with many drawbacks. Its specification is vast and ambiguous and the definitions in OVAL are hard to develop and debug. Also, the process of OVAL interpretation in OpenSCAP is not transparent. For a long time, community around OpenSCAP struggled with difficulties of OVAL and desired to improve capabilities of OpenSCAP to fasten their work.

In this thesis, we will study SCAP and OpenSCAP in depth. Then we will focus on analysing OVAL and we will describe drawbacks of OVAL. We will propose a tool supporting development of OVAL definitions and integrate it into the OpenSCAP project. This change should bring better developer experience, and as a result, faster development of OVAL definitions within OpenSCAP. Finally, we will verify the new OpenSCAP extension on example use cases and we will discuss possibilities of further improvements.

Chapter 2

Methods of Security Audit

Security audit is a process of testing, ensuring and verifying that a computer system complies with specifications for computer security. From theoretical point of view, there are two basic approaches to security audit. The first is called security compliance. It is a proactive approach striving to minimize security threats by applying certain rules by a security policy. The second approach, known as vulnerability assessment, aims at detecting and classifying known vulnerabilities on a system. The process of securing a computer system is always a combination of those two approaches.

Traditionally, security audit was very long and expensive manual process. Current trend is to perform security audit in an automated way. It is possible by implementing standardized specifications as The Security Content Automation protocol (SCAP). Many companies offer software tools that can perform automated security audit of a computer or whole IT infrastructure. Thanks to these tools the average time of security audit of a computer has shortened significantly.

In this chapter, we will briefly explain the key concepts of security audit and describe technologies and common practices that are being used in this area. First, we will discuss principles of security compliance and vulnerability assessment. Then we will outline concepts of security guidances and describe components of SCAP. Finally, we will describe the OpenSCAP project and also other implementations of SCAP.

2.1 Security Compliance

Security compliance is a state where computer systems are in line with a specific security policy [11]. It is a proactive approach to information security. Its purpose is to prevent security issues and reduce impact of security flaws. This goal is achieved by applying available system protection mechanisms and enforcing their configuration.

Computer system configuration is very important with regard to security protection. Correct configuration can protect the system against most of the attacks. By contrast, wrong system configuration can leave the doors opened. The default configuration often does not provide a high level of security. We will demonstrate some practices often required by security policies.

For example, some old network protocols as telnet or FTP transport data in an unencrypted form. They are not considered secure, because plain text communication can be easily eavesdropped. A security policy may require to forbid those protocols in order to prevent possible attacks. Modern secure protocols as SSH should be used instead.

Another example could be passwords used to access the system. The techniques used by attackers to guess the password are more successful on short or dictionary based passwords. A security policy may require criteria for passwords used on the system such as minimum length or character variety of a password in order to avoid the possibility of breaking the password.

Many other techniques and tools are known to improve the security of a operating system, for example running SELinux in enforcing mode, restricting root logins, configuring the firewall, using verified software and updating software regularly or running only necessary system services. Together these settings can greatly improve the overall system protection.

Security compliance is rather a continuous process than a single action because the area of computer security is quickly changing. Another reason is that it is not likely to have the configuration intact for whole period of system lifetime. Each setup of new software or the simplest change of configuration can change the system state from compliant to uncompliant. Therefore security audit should be performed periodically.

2.2 Vulnerability Assessment

Second basic area of security audit is vulnerability assessment. It is a process that indentifies and classifies vulnerabilities on a computer system. Similar to security compliance, the process is continuous, because the amount of vulnerabilities affecting every system is quickly growing over time.

Security specialists around the world concentrate their efforts on discovering security flaws in various products. Each weakness is then classified and identified by a CVE number. CVE stands for Common Vulnerabilities and Exposures and its specification is part of SCAP standard. Information about publicly known flaws is collected in the National Vulnerability Database, maintained by US National Institute of Standards and Technology (NIST).

Some of the serious vulnerabilities are also given a name to popularize them in public. For example, an OpenSSL vulnerability with CVE 2014-0160 is called Heartbleed, because it is a flaw in TLS heartbeat mechanism [1]. Or the vulnerability with CVE 2015-0235 found in GNU C library is famous as Ghost, because it is a flaw in `gethostbyname*()` functions [14].

From the system administrator point of view, a vulnerability can be usually fixed by installing updated version of package in question. Information about versions impacted by vulnerabilities is needed to decide which software is necessary to update. Moreover, it is necessary to know the exact version number of fixed version. Some vendors provide their customers information about vulnerabilities affecting their products. This is usually done either in a form of RSS feeds or better in a form of SCAP documents, which can be easily used in automated vulnerability assessment. For example, Red Hat provides descriptions of all vulnerabilities on their enterprise systems as a part of Red Hat Security Advisories. A SCAP document containing information about all the vulnerabilities affecting all the Red Hat systems can be downloaded from their web.

Knowledge about vulnerabilities is beneficial also for software developers who want to avoid writing vulnerable code and produce more secure applications. The best practices in writing secure applications are shared within The Open Web Application Security Project (OWASP) [12].

2.3 Security Guidances

Security guidance is a document describing requirements to be met to secure a computer system. Various security guidances have been published by different authorities. They are usually industry standards or may be enforced by government regulations. Choice of a security guidance depends on the type of an organization. Size of the organization, business strategy, partners exchanging the data, law requirements and other demands have to be considered.

Moreover, each security guidance aims at a specific target. For systems processing payment card information the Payment Card Industry Data Security Standard (PCI DSS) must be followed. In the United States, government computers must comply to the Security Technical Implementation Guides (STIG) which was prepared by The United States Department of Defense. Another example of a security guidance is the Health Insurance Portability and Accountability Act (HIPAA), which specifies requirements for security of medical records and personal information. Many organizations have also their internal security guidances which reflect the company policy.

Security guidances are written in natural language. They are not suitable for computer processing. To express the requirements in a machine readable form, languages and protocols have been standardized. One of those standards is SCAP which is discussed in the following section.

2.4 Security Content Automation Protocol

The Security Content Automation Protocol (SCAP) is a suite of specifications for describing software flaws and secure configuration of your systems. The SCAP is an U.S. standard.

The specification is managed by the National Institute of Standards and Technology (NIST). The specification is written in NIST's special publication 800-126, revision 2 [15]. NIST is responsible for developing and maintaining the SCAP specification. Current version (in 2016) is SCAP 1.2. This version was published in September 2011. The new version 1.3 is currently being prepared.

The SCAP standard consists of multiple components. It contains XML based languages, reporting formats, enumerations, measurement and scoring systems and integrity.

The first part of the SCAP standard are SCAP languages. They are used basically to write a security policy or describe the desired configuration of software systems. Each of these three languages has a different purpose, but they cooperate together. The following languages are defined in the standard:

1. Extensible Configuration Checklist Description Format (XCCDF) is a language for creating security checklists or benchmarks and is used to organize and manage security policies and to aggregate them into one complex unit.

2. Open Vulnerability and Assessment Language (OVAL) is a language representing system configuration information and reporting assessment results.

3. Open Checklist Interactive Language (OCIL) is a language used to collect information from people.

Each language specification supplies also a XML schema for given language. XML schema is a document which describes syntax of all constructions allowed in the language.

The documents can be validated against that XML schema to check whether the document follows the specification and does not contain not allowed constructions.

More SCAP documents written in different languages can be merged in a single file called Data Stream. The Data Stream may for example contain a XCCDF checklist and OVAL definitions together. The main advantage of Data Streams is that it is easier to ship online a single file than more files.

Second important part of the SCAP standard are the reporting formats. The standard contains two reporting formats.

1. Asset Reporting Format (ARF) is a format that expresses the transport format of information about assets, and the relationships between assets and reports. It is also often called Result DataStream because it is complementary to Source DataStream.

2. Asset Identification 1.1, a format for uniquely identifying assets based on known identifiers and/or known information about the assets.

Third part of the SCAP standard are various enumerations which are both nomenclatures and dictionaries of enumerated objects.

1. Common Platform Enumeration (CPE) describes hardware, operating systems, and applications.

2. Common Configuration Enumeration (CCE) describes software security configurations.

3. Common Vulnerabilities and Exposures (CVE) describes security-related software flaws.

Additionally, the standard specifies the Common Vulnerability Scoring System (CVSS) and Common Configuration Scoring System (CCSS). They both allows to measure severity of issues.

In following subsections we will study specific SCAP components more deeply. Especially we will discuss the most widely used SCAP components.

2.4.1 Open Vulnerability and Assessment Language

The Open Vulnerability and Assessment Language (OVAL) is the most essential part of the SCAP standard. According to its web page [5], OVAL is a community-developed language for determining vulnerability and configuration issues on computer systems.

OVAL serves mainly for describing configuration of computer systems. Documents written in OVAL language are comprised from one or more declarative definitions, that describe objects in the system and express their required state.

Declarative character of OVAL means that OVAL definitions cannot be executed directly. A special interpreter, usually called scanner, is needed to evaluate the definitions. If a certified or trusted scanner is used, this approach practically minimizes the risk of damage of a system by badly written scripts. It also fulfills the requirement to run the scan with root privileges without compromising the security policy itself.

OVAL is also used to express the actual system state. Once the system has been analyzed to evaluate given OVAL definitions, the results are reported also in a form of OVAL document. The OVAL results can be processed automatically or reused in future.

One of main ideas of OVAL is interoperability among different security products, which means that the security policies written in OVAL are independent on used auditing software. OVAL definitions are consumed by tools from variety of vendors on numerous operating systems. Therefore the language is designed in an expandable and robust way. The common core of the language defines its overall structure allowed constructions and element types. On the top of common part there are many platform-specific extensions, which define objects and states applicable for respective platforms. These extensions are strongly dependent on the platform specifics and implementation details. Currently more than 10 platforms are supported, including Linux, Windows, Android, iOS and Mac OS X.

The current version of the OVAL Language is 5.11.1. In past OVAL and its specification were developed and maintained by the MITRE corporation. In 2015, the OVAL maintainership was transformed to the Center for Internet Security (CIS). XML schema is available on GitHub and everyone can contribute. In the time of writing this thesis there are no information available about the next upcoming version of the language.

2.4.2 Extensible Configuration Checklist Description Format

The eXtensible Configuration Checklist Description Format (XCCDF) is used to create security benchmarks. Unlike OVAL, it cannot be used to describe a configuration or state of any system. The main purpose of this language is to create a platform independent checklist. As the name „checklist“ suggests, an XCCDF document is basically a set of rules to be fulfilled. But technical implementation of these rules is not present in the document.

The root element of an XCCDF document is called Benchmark. Benchmark can contain more groups which can be optionally nested. A benchmark or a group contain multiple rules. A rule is basic element of the XCCDF checklist. It usually has a short description, references to guidances and a link to external (most often OVAL) check.

A single XCCDF document may describe more than one security policy using profiles. A profile is a set of rules that should be evaluated. A rule can be member of multiple profiles. Profiles are useful when different security policies for one software product contain same rules so they can be shared between the security policies. For example, the SCAP Security Guide for Red Hat Enterprise Linux 7 has only one XCCDF file, common for all security policies. A specific security policy (eg. PCI DSS or USGCB) is then chosen by selecting a specific profile.

For case that a particular rule has not been satisfied, the rule may supply a remediation script. It is a code that should be run to fix the system in order to satisfy the previously failing rule. Using XCCDF benchmark with allowed remediation scripts offer the highest level of automation. However, it is recommended to consider all impacts of applying those scripts because they are run with root privileges and therefore may harm the system.

2.4.3 Script Check Engine

OVAL definitions are not the only possible source of rule implementation. OpenSCAP offers a special extension called The Script Check Engine (SCE) as another option. SCE allows to write checks in any scripting language (eg. Bash or Python) and link those scripts from an XCCDF document. The SCE extension facilitates transition from script based solution to using SCAP. SCE is not a part of SCAP standard, however other projects like Joval implement it as well.

2.5 OpenSCAP Project

The OpenSCAP project is an ecosystem of open-source tools implementing the SCAP standard.

The development of the OpenSCAP project was started in November 2008 within Red Hat, Inc. OpenSCAP is developed as an open source software from the very beginning. Nowadays software engineers from other companies like Oracle Corporation or SUSE Linux GmbH are contributing to the project, but Red Hat still has the leader position. Also security specialists and auditors are involved in the community. Currently OpenSCAP is widely used by many businesses and government organisations.

The fact that OpenSCAP is an open source project brings many benefits. Sharing knowledge and ideas across the community enables the progress. Open source brings everyone the opportunity to learn from existing code and allows others to contribute and use the software freely. Publicly accessible source code also means that users do not have to trust manufacturers proclamations, but they may verify the software independently [16].

OpenSCAP supports SCAP standard version 1.2 and is backward compatible with SCAP 1.1 and 1.0. It supports the OVAL language in current version 5.11.1 and the XCCDF in current version 1.2.

The OpenSCAP project consists of many security auditing tools and SCAP content. These are used in both vulnerability assessment and security compliance areas.

Most important part of the ecosystem is the OpenSCAP shared library. On the top of the library is built the OpenSCAP scanner. It is a command line tool with plenty of features. Graphical user interface for OpenSCAP is called SCAP Workbench. The security policies are developed under the name of SCAP Security Guide. Other parts of the OpenSCAP ecosystem are a plugin to Red Hat Enterprise Linux installer called OSCAP Anaconda Add-on, storage server SCAPTimony, OpenSCAP daemon, and small utilities to scan virtual machines, containers and remote servers.

OpenSCAP ecosystem is integrated with systems management applications Spacewalk (Red Hat Satellite 5) and Foreman (Red Hat Satellite 6). Currently, integration with other products such as Ansible and ManageIQ is developed.

All the complex information about OpenSCAP ecosystem and its usage for the security compliance and vulnerability assessment can be found in detail on OpenSCAP portal [3]. In following subsections we will introduce briefly each part of the OpenSCAP ecosystem.

2.5.1 OpenSCAP Library

The OpenSCAP library is the basic stone of the whole project. It provides functionality in publicly accessible Application Programming Interface (API). The library implements parsing and validating SCAP documents, scanning the system, evaluating OVAL definitions and XCCDF rules, exporting the results, creating reports and other related capabilities. The library is written in C and uses the most advanced techniques of this programming language. It is a shared library loaded as `libopenscap.so`.

The system scanning is performed by separate binaries called probes. Each probe implements one of the OVAL tests according to the OVAL specification. Probes are executed on request by the library. The probes communicate via `AF_UNIX` socket using serial expressions (SEXP).

OpenSCAP API can be used in any external program. An application may use OpenSCAP library by including respective header files, eg. `#include <openscap/oscap.h>`.

To make the implementation easier, all the data types and API functions behavior are documented online in a detailed way. The documentation is automatically generated from annotations in the library source code.

To use the OpenSCAP capabilities in a program written in other programming language than C, conversion interfaces called bindings are provided. OpenSCAP offers bindings for Perl, Python 2, Python 3 and Ruby. The bindings are also used within the OpenSCAP project, for example in SCAP database and storage server SCAPTimony.

2.5.2 OpenSCAP Command Line Tool

OpenSCAP project offers a command line tool called `oscap`. Its main goal is to perform configuration and vulnerability scans of a local system. The tool is targeting mostly experienced users who want to use the advanced features.

The tool can evaluate both XCCDF checklists and OVAL definitions and generate results either in a form of machine readable SCAP documents or in a form of nice human readable HTML reports. Also it works with DataStreams and other SCAP components.

The SCAP capabilities are categorized to sub commands called modules. The modules are named after SCAP component that they implement. It has OVAL, XCCDF, DS, CVSS, CPE, CVE modules and also a special info module. The modules contain submodules which are identified by operation they perform, for example evaluate, analyze, validate, etc. Each module has its own set of options and switches and also its own help.

More information about all the modules or the `oscap` tool in general are described in detail in the manual page or can be found the user manual.

The tool is implemented in C programming language. It is built upon the OpenSCAP library mentioned in previous section and provides a user front-end to that library. The tool itself has rather short code, all the functionality is provided by the library.

2.5.3 SCAP Security Guide

Having a SCAP scanner (like `oscap`) is not enough to perform security audit. A security policy in a form of SCAP files must be provided to scanner. As mentioned, SCAP documents, particularly OVAL definitions, are platform dependent. Originally, there was lack of full-fledged open source security policies for Red Hat Enterprise Linux. SCAP security guide project was started to create some benchmarks that can be consumed by tools within OpenSCAP project.

The aim of SCAP Security Guide is to offer open source implementation of popular security guidances as PCI DSS, DISA STIG or USGCB. Security policies from SCAP Security Guide are applicable to various operating systems, namely Red Hat Enterprise Linux, Fedora and Debian GNU/Linux, and also to asses software like Mozilla Firefox, Chromium or Java Runtime Environment.

Besides the OVAL checks and XCCDF rules, SCAP Security Guide contains detailed description and rationale in each rule. A well aranged HTML document, which explains policy requirements in a human readable way, can be generated for each profile.

SCAP Security Guide is available on many Linux distributions. After installing respective package, DataStreams, XCCDF and OVAL files and human readable HTML guides will be available on the system. For example, on Fedora, these are installed in `/usr/share/xml/scap/ssg/content` directory.

2.5.4 SCAP Workbench

SCAP Workbench is a graphical user interface (GUI) of the OpenSCAP project. It provides an easy way to perform common tasks. The tool is preferred by beginners, because the basic scanning can be performed in 3 clicks. It can be used to scan either local systems or remote systems using the SSH protocol.

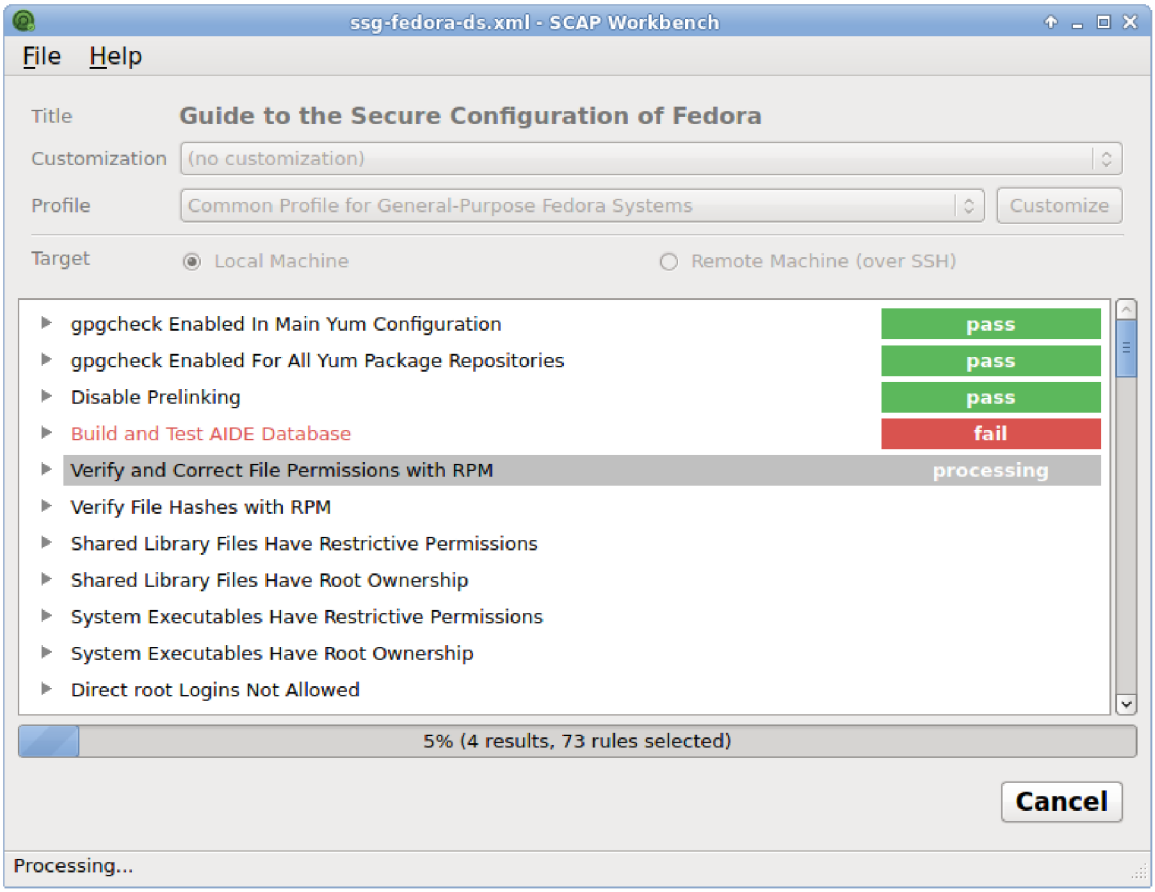


Figure 2.1: SCAP Workbench performing a scan of Fedora 23.

When started, SCAP Workbench automatically offers available security policies from SCAP Security Guide. The user selects desired profile and clicks the `Scan` button. The tool asks for root user privileges and performs the scan of this machine. After the scan result is displayed colourfully. Moreover, a detailed report can be generated and displayed in a web browser. Also the automatic remediation of the scanned system can be performed during the scan.

SCAP Workbench is implemented on the top of the OpenSCAP library using the Qt toolkit and the C++ programming language. The remote scanning feature requires SSH access to the remote machine and `openscap-scanner` package installed on it.

SCAP Workbench can be also used to customize the security policies. This process is also sometimes called XCCDF tailoring. The customized policy is saved to the so-called tailoring file. This file describes only changes from the original security policy, so when a security policy updates, users do not need to perform the customization again, because they can easily apply the tailoring file against the new version of the policy. Moreover, the

file is very small and can be easily distributed.

SCAP Workbench is available in various Linux distributions. Versions for Microsoft Windows and Mac OS X are also available to download. However, on the non-Linux systems it can be used only for remote scanning and security policies customization. That is because underlying OpenSCAP library does not implement probes for those systems, only Linux probes are implemented. Nevertheless SCAP Workbench is useful on non-Linux systems, because many users want to audit remote Linux servers from their laptops without a need to install Linux on their laptops.

2.5.5 OpenSCAP Daemon

OpenSCAP daemon is a system service which allows scheduling and performing security scans on a regular basis according to a given schedule. It comprises of two components - a `oscapd` daemon running permanently in background, and a command-line client `oscapd-cli`. These two parts communicate with each other using dbus interface. Both of them are implemented in Python.

The client provides an interactive mode to plan scanning and display results. Usage of the client is much easier than interface of the `oscap` tool, because the client offers options only for the most common tasks. The OpenSCAP daemon can scan local, remote and virtual machines leveraging the OpenSCAP library. It is also used in project Atomic to implement offline scanning of Linux container images.

2.5.6 OSCAP Anaconda Add-on

OSCAP Anaconda Add-on is a plug-in for Anaconda, the Red Hat Enterprise Linux installer. The plug-in applies a security policy while the operating system is being installed [13]. Before the installation, user is notified if his installation configuration that he selected is not compliant with selected security policy. This is especially important when a security policy requires settings that cannot be changed after the installation is finished. Example of such configuration might be an attempt to install system on an unencrypted disk partition.

All the settings required by selected policy are adjusted during the installation. An initial period when system is not configured is avoided, which means that the operating system is compliant from the very first boot. This feature saves the time of system administrators and is particularly useful for large environments or for deploying virtual machines.

OSCAP Anaconda Add-on was introduced to Red Hat Enterprise Linux in version 7.2. It supports both graphical and text mode of installation. Unattended installation of compliant system is also possible with a Kickstart file.

2.6 Competitive Security Audit Tools

OpenSCAP is not the only project implementing the SCAP standard. Other tools, either open source or commercial, are available. In this section we will point out main differences between them and OpenSCAP.

Joval Continuous Monitoring is a multi-platform implementation of SCAP written in Java. The project was started in 2011 and is led by only two developers. It can scan Windows, Linux, Solaris, and many other systems, because it implements all the tests specified in OVAL language. It fully implements the SCAP standard and also SCE extension. Core part of source code is developed open source. Unlike OpenSCAP, to perform

remote scanning Joval does not require to have an agent installed on a remote system. Joval also offers wider range of output formats than OpenSCAP—HTML report, CSV files, SQL commands, JSON, diagnostics reports. On the other hand, Joval requires JVM to run, but system requirements of OpenSCAP are minimal. Another drawback from customer point of view is that Joval is not certified by NIST.

McAfee Policy Auditor is a security auditing software fully implementing the SCAP. It is developed by McAfee which is now part of Intel Security. Unlike OpenSCAP it is not an open source project. On the other hand it can scan multiple operating systems, namely Windows and also various Linux distributions. It ships benchmark templates for PCI DSS, HIPAA, FISMA and other security guidances. To allow infrastructure management and deployment, it is integrated with McAfee ePolicy Orchestrator. It has received SCAP 1.2 certification from NIST.

OVAL Interpreter (OVALdi) is a reference implementation of OVAL language. The program is open source and is licensed under a BSD license. It serves to demonstrate abilities of the OVAL language, it is not an enterprise scanning tool.

2.7 Future of Security Audit

Current effort is to offer tools than can perform security audit of virtual machines and also containers.

Containers are now the easiest and the most scalable way for deploying software applications. But to use them in production, their security aspects must be also considered. Each container contains its own bundle of software packages and libraries. This fact implies that each container can be affected with various different vulnerabilities.

A typical computing node can host thousands of running containers based on different operating systems. It is not desirable to install security audit software inside each container, because that would considerably decrease performance, cost a lot of time, storage and network capacity. Instead, a scanner can be installed on the host or even better, deployed in a super privileged container (SPC) and scan the containers from outside.

For example, the OpenSCAP project started to integrated with Atomic, a tool for managing containers. Also a OpenSCAP SPC that can scan containers for vulnerabilities is available to pull from Docker hub [17].

Also configuration scan of containers, which means enforcing compliance of containers with some policy, is very desirable. However, security policies and the best practices for containers are still a subject of research.

On the field of standardization, the SCAP version 1.3 is currently prepared [2]. This new version is expected to include OVAL 5.11.1 and other updated specifications.

We can expect that security audit will be integrated in more system management and administrator tools since the importance of computer security will be continuously growing.

Chapter 3

Development of OVAL Definitions

The definitions written using The Open Vulnerability and Assessment Language (OVAL) are used for automation of both areas of security audit—security compliance and vulnerability assessment.

In this chapter we will describe concepts and features of the OVAL. We will learn how OVAL definitions are developed and demonstrate their usage. Finally, we will discuss several weak points or disadvantages of the language and identify main problems of OVAL.

3.1 OVAL Documents

As mentioned in the previous chapter, OVAL is core part of the SCAP standard. OVAL is domain specific language designed exclusively for purpose of security auditing, so its abilities are limited. It covers three major domains of system assessment [5]:

1. Describing desired configuration of a system.
2. Analyzing the system for the presence of the specified machine state.
3. Reporting the results of assessment performed on a system.

The OVAL specification [6] describes syntax and semantics which are used for all these three areas. The specification defines more document formats with different purpose.

- *OVAL Definitions* document describe desired configuration of a system and is an input for a scanner (eg. OpenSCAP).
- *OVAL Variables* document supplies external variables for *OVAL Definitions*.
- *OVAL System Characteristics* document contains information collected on assessed system during evaluation of given OVAL definitions and is generated as output of a scanner.
- *OVAL Results* document contains detailed results of the system evaluation computed from comparison of *OVAL Definitions* and *OVAL System Characteristics* and is generated as output of a scanner.
- *OVAL Directives* document amends information aggregated in *OVAL Results* by specifying its level of detail. This format is rarely used.

All the document formats have similar structure. To understand the OVAL Results and OVAL System Characteristics, full understanding of respective OVAL definitions is needed.

3.2 Creating OVAL Definitions

To write a definition in OVAL it is necessary to know the format of *OVAL Definitions* XML document. The root element (`oval_definition`) contains children elements that represent major parts of the document: generator, definitions, tests, objects, states, and variables. A generator contains only metadata about origin of the file. Definitions are the most high-level elements. Tests represent criteria that have to be fulfilled. Objects describe entities under examination. States specify requirements on those entities. Variables serve to parametrize objects and states. We will discuss the most interesting of these elements in following subsections.

3.2.1 OVAL Definition

Definition is the most high level logical unit of the OVAL language. Single file can contain multiple definitions.

A definition consists of one or more criteria that needs to be fulfilled to satisfy the definition. The criteria create a logical expression using operators AND, ONE, OR and XOR and can be nested. Each criterion points to a *test* which shall be performed to get the resulting value of the criterion.

Instead of a using a criterion, another definition which must be evaluated at first, can be referenced.

Each definition also must contain an unique identifier (ID), a title and short textual description. The ID can be also used to reference given definition from an XCCDF Checklist. Further, a definition may contain information about which platforms are affected by a definition and it may also provide references to external sources (eg. CVE identifier). These metadata have only informational character and do not affect the results of evaluation.

Example of an OVAL definition that checks whether the network time synchronization is enabled can be seen in Listing 3.1.

Listing 3.1: Example of OVAL definition.

```
<definition class="compliance" id="oval:x:def:1" version="1">
  <metadata>
    <title>Service chronyd enabled</title>
    <description>
          The chronyd service should be enabled if possible.
    </description>
  </metadata>
  <criteria comment="package chrony installed and service chronyd is
      configured to start" operator="AND">
    <criterion comment="multi-user.target wants chronyd"
      test_ref="oval:x:tst:1"/>
    <extend_definition comment="chrony installed"
      definition_ref="oval:x:def:2"/>
  </criteria>
</definition>
```

As can be seen from the example, the definition does not actually say what should be assessed on the system to get the result. To understand which actions need to be performed it is necessary to look at the referenced test or tests.

In the step of writing `definition` element, a developer has to realise what is purpose of the definition from a high-level point of view and express the description.

3.2.2 OVAL Test

In OVAL terms, a *test* is implementation of definition criteria. In simple words, the test binds together object and states and their relation. A developer of OVAL file must start thinking here what actually needs to be examined and what exact type of object will be examined.

For the purpose of configuration checking, OVAL specifies many kinds of tests for various operating system objects. For every test the language specifies also a corresponding OVAL object and OVAL state. An example of OVAL test can be seen in Listing 3.2.

Listing 3.2: Example of OVAL test.

```
<linux:rpminfo_test check="all" check_existence="all_exist"
  id="oval:com.example:tst:1" version="1"
  comment="package chrony is installed">
    <linux:object object_ref="oval:com.example:obj:1"/>
    <linux:state object_ref="oval:com.example:ste:1"/>
</linux:rpminfo_test>
```

Some of the tests are platform independent, for example `textfilecontent54_test`, which can examine data in text files. On the other hand, tests applicable only to one specific platform also exist. For example, `systemdunitproperty_test` can be applied on certain Linux distributions that use systemd as init system, but it cannot be applied on Microsoft Windows systems at all.

To evaluate the test, it is needed to collect from the system all the items specified by given object element and compare the resulting set with given state element. The test has `check` attribute, which specifies how many objects must satisfy the state, and optional `check_existence` attribute, which specifies how many objects must exist on the system.

3.2.3 OVAL Object

OVAL object is an element describing particular object that exists on a system—a file, a process, an environment variable, an RPM package, a SELinux boolean, a systemd unit, a kernel parameter, a value in a configuration file, an entry in SQL database, an entry in LDAP directory and many others. Each type of object has a name (eg. `rpminfo_object` or `file_object`) and specific set of child elements and attributes. They are different depending on the object purpose.

For example, a `textfilecontent54_object` serves to describe a text string in a file. Its declaration can be seen in Listing 3.3.

Listing 3.3: Example of OVAL object.

```
<ind:textfilecontent54_object id="obj_password_pam_pwquality_minlen"
    version="1">
  <ind:filepath>/etc/security/pwquality.conf</ind:filepath>
  <ind:pattern operation="pattern match">^minlen[\s]*=[\s]*(-?\d+)
     (?:[\s]|$)</ind:pattern>
  <ind:instance datatype="int" operation="greater than or equal">
  1</ind:instance>
</ind:textfilecontent54_object>
```

In area of security compliance of Linux systems, the objects representing values from configuration files are the most popular, as can be seen in Table 3.1.

OVAL object	Count
textfilecontent54_object	119
systemdunitdependency_object	56
file_object	32
rpminfo_object	32
variable_object	12
partition_object	11
sysctl_object	2
selinuxsecuritycontext_object	2
rpmverifyfile_object	2
password_object	1
symlink_object	1

Table 3.1: Objects used in SCAP Security Guide for Red Hat Enterprise Linux 7.

Basically, to write a correct object definition, it is necessary to read carefully the specification and well understand the meaning of all elements and attributes of the object. It is also important that the object declaration covers all possible situations. For example, some applications may have configuration files stored in multiple locations (users and system-wide configuration files).

To evaluate the object, the scanner has to find every item on the system that corresponds to the object declaration. More than one item can be collected. The collected items are stored into *OVAL system characteristics* model. Some items can be filtered out by optional `filter` child element of object and therefore they will not be contained in the resulting set. After collecting will finish, collected items will be compared with corresponding OVAL state.

3.2.4 OVAL State

OVAL state specifies features of an object that the object has to conform to fulfill the requirements of respective test. States are optional elements. In OVAL documents can be found a lot of tests that do not contain any state, they only check whether particular object exists or does not exist.

After all the objects are collected from a system they are compared with the state and based on the comparison result, the test result is determined. Same as with the OVAL objects, the OVAL states corresponding to each OVAL test are defined in specification. States usually have more child elements than objects, but usually they are all optional.

The example in Listing 3.4 is a declaration of an OVAL state corresponding to example of OVAL object in Listing 3.3 in previous subsection.

Listing 3.4: Example of OVAL state.

```
<ind:textfilecontent54_state
  id="state_password_pam_pwquality_minlen" version="1">
  <ind:instance datatype="int">1</ind:instance>
  <ind:subexpression datatype="int"
    operation="greater than or equal">8</ind:subexpression>
</ind:textfilecontent54_state>
```

3.2.5 OVAL Variable

Instead of hard-coding the values to objects and states directly, they can be parametrized by variables. Variables may be constant, but usually their values are provided by referencing and evaluating other OVAL objects. It basically means that before using them, their value must be determined by evaluating another objects and collecting additional items from the system.

This is used often when we need to use information from more sources in one test. For example when we want to write a test that verifies whether some file are is not owned by not existing user group, we have to first get list of user groups on the system from `/etc/groups`, store them in a variable and then use the variable in OVAL `file_state`.

Another important feature related to OVAL variables are functions that are used to process values of variables. Apart from simple functions like concatenation of strings or counting numbers, there exist also quite advanced functions in the specification.

One of them is the `glob_to_regex` function that converts shell glob to a regular expression. It is used for example to check correct configuration of the `rsyslog` log processing system. The `/etc/rsyslog.conf` can contain directive `$IncludeConfig`, where a shell glob is used to specify path to other configuration files to include [9]. But in OVAL file paths must be described by a regular expression. Then the `glob_to_regex` function comes handy to get pattern for all the configuration files, because we need to browse all of them to check whether particular setting is present or not.

An example of using the `glob_to_regex` function inside an OVAL variable, taken from the SCAP Security Guide project, can be seen in Listing 3.5.

Listing 3.5: Example of function within OVAL variable.

```
<!-- First obtain rsyslogs IncludeConfig directive value -->
<ind:textfilecontent54_object id="oval:ssg:obj:1" comment="rsyslogs
    IncludeConfig directive value" version="1">
  <ind:filepath>/etc/rsyslog.conf</ind:filepath>
  <ind:pattern operation="pattern match">
    ^\$IncludeConfig[\s]+([^\s;]+)</ind:pattern>
  <ind:instance datatype="int">1</ind:instance>
</ind:textfilecontent54_object>

<!-- Turn that glob value into Perls regex so it can be used as
    filepath pattern below -->
<local_variable id="oval:ssg:var:1" datatype="string" version="1"
    comment="IncludeConfig value converted to regex">
  <glob_to_regex>
    <object_component item_field="subexpression"
      object_ref="oval:ssg:obj:1"/>
  </glob_to_regex>
</local_variable>
```

3.3 Usage of OVAL

OVAL definitions are a part of security policies. The OVAL files are either implementing checks for security policies or containing a list of vulnerabilities. They are usually referenced from a XCCDF checklist or are part of SCAP source datastream. They can be used separately, too.

To evaluate OVAL definitions using OpenSCAP, a command like this can be run from a command line:

```
# oscap oval eval --results results.xml --report report.html oval_file.xml
```

Two output files will be generated—human readable HTML report and an OVAL Results XML document suitable for machine processing. On standard output it will print only the results of all the definitions. Possible errors will be reported on standard error output.

3.4 Problems of OVAL

When a developer wants to create a simple OVAL definition, he must specify an OVAL object and a corresponding state, then connect them in OVAL test, and use it as a criterion of the definition. This task is complicated per se. But to write a security policy that reflects security requirements for real systems, he needs to use advanced OVAL constructs and more sophisticated definitions.

While analysing OVAL definitions used in practical security audit, we can identify several problems in the language and in the interpreter. Those obstacles often complicate development and debugging of definitions. We will describe some of the most significant of those problems:

Problem 1: Nested OVAL definitions are not transparent. Definitions usually consist of one or more criteria. But instead of a criterion, the definition can be extended by referencing another definition. The referenced definition must be evaluated at first, because its result must be known to compute result of current definition. Nested definitions bring more complexity to the evaluation process and lower its transparency, because during evaluation of single definition, more other definitions may be evaluated. Moreover, OVAL allows to create whole chains of references.

The nested definitions are a great way to avoid repeating code. On the other hand, they must be used carefully. When a developer oversights or makes a mistake in ID, it may lead to very complicated definition dependencies. Unfortunately, such mistake does not have to be identified from output results. It is hard to reveal what was processed and evaluated to get results of given definition.

Problem 2: Computation of results involves solving complex logical expressions. To get final result of definition, all its criteria must be evaluated [6]. Then a logical operator is applied on the criteria result to compute the final result. However, OVAL logical operators are not same as mathematical operators, because OVAL does not use Boolean logic. The result can be not only true or false, but also unknown, error, not evaluated and not applicable.

Operands of results computing operation can be afterwards found in OVAL results document, but they are poorly arranged for purpose of debugging.

The operation is also similar for OVAL tests that create the definition. Again, evaluating the test is not only a simple comparison of objects and state that would result in a boolean value. Possible results of a test are: true, false, unknown, error, not evaluated and not applicable. When developing an OVAL definition, it would be useful to have explained why those values were returned.

Moreover, the test includes `check` attribute, which specifies how many of collected objects must satisfy the state requirements. Also, the optional `check_existence` attribute might specify how many objects must exist on the system. Less experienced developers often confuse these values. A test is evaluated to true when requirements of both `check` and `check_existence` attributes are satisfied. The result is computed according to truth tables from OVAL specification. Unfortunately OpenSCAP scanner does not inform user which table has it used and why, whether values of `check` and `check_existence` were used and it also does not explain meaning of used values. Although these values are defined in specification, it is often hard to figure out which of them applies in concrete situation, because the wording is very generic.

Problem 3: Excessively large amounts of objects can be collected from the system. If an OVAL object is declared in a wrong way, even thousands of items can be collected. But often to determine results of the definition only a small amount of this set is necessary to collect.

For example a declaration of `file_object` where a regular expression is used to specify the path could match thousands of files. A filepath described by regular expression `'^.*$'` means that to evaluate the object a scanner needs to browse also filesystem mounted in `/proc` directory. Most likely it was not intention of the author to search this filesystem, because it does not contain files useful for security compliance. But the scanner will collect all of the files there and may run out of memory before it collects the object that was in author's mind and it may finally give a false negative result. Another fact is that comparing a very large amount of collected objects with a OVAL state will slow down the evaluation significantly.

A developer should keep in mind that wrongly written regular expression not only leads to incorrect results but can also affect the performance of scan. A slight amendment of the object declaration can avoid this situation and the scanner will collect only object interesting for the security rule in question.

Problem 4: Filters affects results in a nontransparent way. Resulting set of collected items does not necessary correspond to OVAL object declaration, because filtering can be involved.

The OVAL specification allows that OVAL objects may contain optional `filter` element. Filter element references some OVAL state and excludes or includes items matching that OVAL state from resulting set of collected objects.

The drawback of filters is that items filtered out will not be used for object/state comparison and mainly they are not displayed in *OVAL Results* document. The document contains only final item set after filtering. Therefore it is difficult to debug objects that contain filters, because we do not know which items were added or removed by the filter.

Problem 5: OVAL specification sometimes does not follow conventions. In some cases, behavior of tests in OVAL specification differs from usual behavior of standard system tools and therefore this behavior is not expected by the developer.

For example, the `rpmverifyfile_object` can collect all files installed by RPM packaging system and verify them against the RPM database. But the set of collected items will contain more RPM packages than output of the standard `rpm` system tool, because `rpm` tool skips ghost files, whereas OVAL object does not skip them unless special `behaviors` element is added.

Problem 6: The evaluation is not a straightforward process. OVAL objects or states can reference OVAL variables. Variable values can be set dynamically by evaluating another OVAL objects referenced from the variable. It means that there is not always single object for single test. Theoretically, during evaluation of a test, unlimited amount of OVAL objects can be queried.

These relationships mean that the scanner often does not follow a simple evaluation tree, but more likely the interpretation goes trough a very branched complicated graph. Sometimes, there are very sophisticated relationships between definitions, objects, states and other elements. Definition can be created from more tests and extended definitions, then a test can have object that combines using variables, variable functions, multiple filters, values from other objects, and so on. It is hard for a developer to visualise such graph in his mind and imagine how his definition will be evaluated.

Using variables and possibility of chain references makes it even hard to understand. It is important to realize that a single definition can contain everything from OVAL features. We see that OVAL definitions can be very complicated.

Problem 7: Parameters and return values of variable functions are not presented to the user. As mentioned in subsection 3.2.5, OVAL contains built-in function that are able to process values of variables. Although functions change values of the variables, their usage is not reported in output of scanner.

Also, variables have to specify their data type, but it may lead to false negative results when trying to compare integers with strings. Moreover, OVAL specifies so-called external variables that are used to parametrize definitions. It is a way of passing arguments from external sources, most often from a XCCDF benchmark. Again, external variables can be processed by functions.

Problem 8: OVAL allows constructs that may lead to no operation or to infinite loop. It is possible to write an OVAL definition with no effect although the definition is fully valid and in line with the specification. An easy example could be a `file_object` where absolute file path is defined by regular expression that forbids containing `'/'` character.

The OVAL even allows to write definitions that are meaningless or even harmful. For example, there can be created a definition that recursively references itself. This construct is allowed by language schema, but does not produce any sensible results. OpenSCAP does not have implemented limit for depth of references, so this kind of definition may lead to crash of the application.

All of aforementioned problems are difficult to identify when they happen. User may not notice that they occured, the worse if user wants to debug.

It may be objected that the results of evaluation are presented as an HTML report or in *OVAL results document*. However, how were the input processed, what was done on the system and how were the results computed is not presented to the user.

Moreover, the specification is very long and sometimes ambiguous. It requires a long time of studying to be able to write useful and correct definitions. Also there is lack of resources from which people can learn.

Apparently, the development of OVAL definitions is a time consuming and difficult process. A tool that would help to simplify it and make the developer's life easier would be beneficial.

Chapter 4

Refining OVAL Development

In this chapter, we will propose a solution that provides support for OVAL development within the OpenSCAP project.

We will firstly analyse requirements of users and developers. Then, we will discuss options considered during the design phase and their advantages and disadvantages. We will propose and justify a solution that fits the requirements. Finally, we will describe implementation of the solution and discuss problems faced during the implementation.

4.1 Requirements for OpenSCAP Modification

Requirements for OpenSCAP modification were based on discussions with OpenSCAP team and security policy developers. Some of them were also expressed in project ticketing systems [8] [4]. We will mention the most interesting of the requirements.

The ability to debug steps performed by the scanner was mentioned frequently. Main reason is that OVAL results document contains only final results of evaluation, but no information about the evaluation process. Developers of security policies would like to know intermediate steps—partial results, values of variables or also parameters and return values of functions. For simpler definitions, the steps of the process can be guessed from the OVAL results document. But in typical cases, it is almost impossible because those files often have thousands of lines.

On the other hand, a special editor or Integrated Development Environment (IDE) for OVAL was also requested. For users with weak knowledge of OVAL it is not comfortable to study the specification and write definitions from scratch in their text editor. They would welcome some templates and a context help.

Users also often report they expected OVAL definition being evaluated somehow, but OpenSCAP gave them different results than expected. Their issue needs to be precisely located at first. Basically, following sources of the problem are possible:

- Invalid OVAL definition file (syntax error).
- Logical mistakes in OVAL definitions or using wrong constructions in OVAL (semantic error).
- A bug in OpenSCAP implementation.
- A problem in other libraries that OpenSCAP is linked with.

- A problem in underlying operating system.

All of those possibilities have to be considered to resolve a bug. To identify problems related to OVAL it requires a very deep knowledge of implementation of OVAL in OpenSCAP to discover them using The GNU Debugger (GDB), and therefore this option is not aimed for users.

From the point of view of OVAL definitions, OpenSCAP internal implementation details are not interesting. We rather would like to know objects that were assessed, or more importantly, find the reasons why some objects were not found (e.g. due to denied permissions or stopped services). Basically, the most important requirement is to cope with problems mentioned in Section 3.4 and help with identifying them.

The developers who implement and maintain OpenSCAP mostly complaint that the bug fixing and maintenance is difficult and expensive. They would welcome a tool that helps with identifying and fixing bugs in OVAL. Easily adding debugging for newly developed features is a must. They also need to know whether the base library communicates with probes correctly and be informed when a probe accidentally dies.

4.2 Possible Approaches to Refining OVAL Development

A proposal to create an interactive debugger for OVAL was considered. The idea was that a tool similar to The GNU Debugger (GDB) would be created. Since the OVAL is rather complex standard it would be very difficult to implement an interactive debugger that covers all the requirements. Such work is beyond scope of bachelor's thesis. Finally it means to develop a lot of code similar to already existing code. The debugger would have to mimic the behavior of OpenSCAP and implement OVAL parsing and validation. Furthermore, OVAL is a declarative language, so a typical approach of debuggers of imperative languages is not applicable for OVAL.

Other possibility is to create a tool for static analysis of OVAL definitions, so-called lint. The lint could identify possible problems in OVAL definitions based on defined rules and suggest ways of improving definitions or fixing these problems. Although a static analysis tool would be surely beneficial, this method does not cover the requirement for reporting evaluation process.

Another considered option was to create a developer tool with graphical user interface (GUI). The application would allow to create OVAL definitions by clicking or selecting items from menu. The proposal of GUI application was rejected for various reasons.

Firstly, same as in case of lint this solution cannot report evaluation process and also does not involve debugging. Secondly, graphical applications for SCAP content authoring (both OVAL and XCCDF) already exist. We should mention for example Enhanced SCAP Editor (eSCAPe), Benchmark Editor, or Recommendation Tracker. These applications are poorly designed from perspective of user experience and difficult to use [10]. They are not developed or used actively. Lesson learned is that creating a GUI for OVAL authoring is extremely difficult and would require rich expertise in both security audit and user experience areas. A risk of similar fail convinces not to create graphical application of such a kind.

For security policies contributors, it is usually more comfortable to derive their OVAL definition from some existing definition. To edit them they would prefer using their favourite text editor. OVAL development in text editors could be made more pleasant by improving syntax highlighting and code completing in text editors.

To satisfy the most of the requirements we need a tool that would either work with or be integrated with OpenSCAP and could operate on run time. This way, we are able to identify run time problems. In other words, it will mean a diagnostics of OVAL evaluation. Both of those goals can be achieved by *adding reporting capabilities and diagnostics abilities into existing code of OpenSCAP.*

This seems to be the most effective solution for the following reasons:

- Integrated solution could maximally leverage existing features and code of OpenSCAP.
- Behavior of the helper tool will not differ from a behavior of production software and also that it will not differ in the future when OpenSCAP will be updated.
- The solution will be a part of already existing project, which brings several benefits—better visibility, mentioning in documentation, website, and tutorials and firmer user trust.
- When the solution will be integrated directly into library that is already packaged in Linux distributions, after a new upstream release, the work will be easily propagated into distributions. Proposing a new package to distributions is a long, difficult and also a bit bureaucratic process.

4.3 Reporting OVAL Evaluation

From its early versions OpenSCAP contained some debugging messages and offered possibility to generate a log. But the logs were not much useful for multiple reasons. Firstly, they were not available for normal user, but only for developers who compiled the program with a special option. Moreover, the messages contained in the log did not cover the process of security policy evaluation and scanning, they were focused mostly on implementation details. The log contained hexadecimal values of pointers, long dumps of data in internal communication protocol or short fragmentary messages. The OVAL interpretation could not be recognized at all. OpenSCAP is multi process program and so a separate log was created for each process. It was very difficult to find a relationship between the logs and understand them. Due to these reasons, the logs were very rarely used. Even the OpenSCAP developers did not use them much often.

Naturally, an idea to remove the disadvantages and leverage existing logging features was considered at first. Many design decisions had to be made before starting the implementation.

The first question is a way of reporting. Linux systems contain mechanisms of logging known as syslog or journal. Using these are typical for system services and daemons, but OpenSCAP (`oscap` command) is a common user application. The syslog offers proven and standardized solution. On the other hand, logging each step into syslog will slow down the evaluation, cannot be turned on only on user's demand and will pollute the system log significantly. Therefore it was decided to create own method that suits needs of the OpenSCAP project.

Regarding the format of the log, basically two possibilities were considered. The first of them is a HTML report. This choice will provide nice user experience, emphasise important parts with colors or implement sorting and searching in JavaScript. Second possibility is a plain text report that lacks those user-friendly features. However, OpenSCAP is often

used on virtual machines or servers, where is no web browser installed and users control the machine via SSH connection. In these cases user will have to download or copy the report to its workstation and then display it, which is slower, less practical and more complicated. Since OpenSCAP is more often used in a text terminal, it was decided to implement the report in a plain text form.

Another problem that had to be solved during design phase is architecture of reporting system. One of the possible solutions is to create a complex system with a defined protocol, hooks, readers, writers and postprocessing routines. However, less complicated solution that tries to solve all necessary tasks in place can also fit the requirements.

4.3.1 Message Categories and Format

The report will consist of individual messages. We will classify these messages into 4 categories according to their meaning.

1. *Error messages* inform about serious failures which OpenSCAP cannot recover from, for example a probe killed by a signal or invalid XML input.

2. *Warning messages* inform that something went wrong during interpretation, but OpenSCAP does not have to terminate, it can continue by evaluating next definition, eg. denied permissions, invalid regular expression, etc.

3. *Info messages* inform user about process of evaluating, which test is processed, what objects are queried, what are values of variables, how will the result affected etc. This category is the most interesting from the point of view of OVAL.

4. *Developer messages* show implementation details, mostly of internal character. Majority of original messages fits into this category.

Users choose verbosity level by selecting one of those categories. More verbose levels include all messages from less detailed levels. Each message is on a new line. The line starts with category of message and name of the process that created the message. Then a text of message follows.

If developer level is activated, the message continues with additional information inside brackets—PID and name of the process, thread ID and name of the thread that produced the message, source file, line and name of the function which emitted the message. In other levels these data are not included because it is a lot of distracting information that reduces readability and orientation in the log.

The format and classification designed in this thesis makes the log readable and reflects different needs of users.

4.4 Changes in OpenSCAP User Interface

To enable the functionality introduced by this thesis it was required to change command line interface of the `oscap` tool. The command line interface consists of more modules, each of them has a specific set of options. First question that needs to be answered is in what modules the functionality needs to be added.

Since this thesis is focused primarily on OVAL, the functionality was introduced firstly into `oval eval` submodule which is used for OVAL evaluation. In other submodules, `oval collect` and `oval analyse`, the reporting was also made available.

The functionality was also added for XCCDF evaluation module, because OVAL definitions are often referenced from XCCDF checklists. This extension of requirements involved studying XCCDF evaluation and covering XCCDF evaluation by log messages.

In each changed submodule, two new command line options were added.

First option, `--verbose`, turns on the verbose mode and requires an argument specifying the verbosity level. The verbosity level can be one of `ERROR`, `WARNING`, `INFO` and `DEVEL`. Those values correspond to message categories outlined before.

Second option, `--verbose-log-file`, specifies name of the file that the log will be written into. This option is optional. When the option is omitted, the logging messages will be written on standard error output.

Originally, the `--verbose-log-file` was obligatory and user always had to specify file name. This restriction was implemented because before clean up of legacy debugging messages the output was too long, so user would redirect it to a file anyway. However, after numerous improvements of the verbose mode, a feedback was provided that standard error output should be used if this option is not specified, because typing another option takes some time and using the option within the test environment was more difficult.

4.5 Implementation Details

The implemented functionality is not a standalone application. It is a set of changes integrated as an extension of the OpenSCAP project.

The work was based on OpenSCAP 1.2 branch, which is current feature branch. The goal of this thesis was achieved by continuous contribution to the project. Those changes were proposed, reviewed and merged in numerous pull requests against upstream repository. The source code of the changes merged into the OpenSCAP project is saved on attached CD in a form of individual patches exported from Git version control system. The whole set of changes consists of 109 patches. Authenticity of those patches can be verified on public GitHub repository [7].

In OpenSCAP, same as in many other open-source projects nowadays, pull requests are typical development workflow. Firstly, a developer creates a new branch in his own local fork of the git repository. Then he develops a new feature by changing that branch. After, he creates small commits, each with a detailed description explaining changes proposed by the commit. When finished, he pushes the changes to his repository fork on GitHub. He creates a pull requests containing the new commits, and attaches message explaining the purpose of the new changes. The commits are reviewed by other developers involved in the project. Usually, the author has to reflect their comments and improve his code. After the commits are accepted by the reviewer, they are merged into a public branch. This workflow was also used by author during implementation phase.

Changing already existing and well established project is very hard. Every single change involves not only detailed understanding of the code but also a good knowledge of all use cases that are implemented by the code. Moreover, in OpenSCAP project, all the SCAP standards are strictly followed, so we must be very careful to not broke compliance with those standards. Also, the OpenSCAP project is a shared library. Other applications rely on its stable API. The API cannot be broken by changing already existing symbols. The fact that symbols will be part of API for many years must considered in depth when creating new functions. The commits must have detailed commit messages and pull requests must be well documented and justified. All patches must follow the coding style and other conventions followed in the project. If some of aforementioned requirements are not met, the patches

are rejected. All these facts made the implementation phase much more complicated and time-consuming than when the project is written from scratch.

Due to aforementioned difficulties, the implementation was decomposed into smaller steps. Every step made a small improvement of the current state that aimed to lead to the goal of this thesis. The work started by leveraging existing features of OpenSCAP.

The implementation itself begun by enabling the logging messages in the build system. Firstly, the current debugging messages were made available for the normal user. New options, described in previous chapter, were added. Then all the messages were redirected into a single file. This required to synchronize threads and processes by a file lock to avoid inconsistencies in the output. For our purposes, is not necessary to implement a central collector.

In next phase, work focused on adding new messages into the source code. Based on problems of OVAL mentioned in previous chapter, sample OVAL definitions that containing particular tricky constructions were created. These definitions were used for analysis of OVAL interpretation of OVAL within OpenSCAP project. The `oscap` process was traced and the source code were to find the states that were reached and possibilities that can happen. This work involved in depth study of source code in C language and full understanding of whole architecture. After identifying important paths in the code, reporting of each important step was added in a form of a human readable log message.

The debug messages from previous OpenSCAP versions are still present in the log. However, most of them were put into the *Developer* category. The new messages were added with focus on usefulness for OVAL developers. Main purpose of the messages is to help identify problems described in Section 3.4.

It was necessary to convert some entities to human readable form before inclusion in verbose log. For example, values of OVAL variables were serialized. A string buffer and other helper datatypes were implemented for this purpose. They can be reused in other code.

4.6 Problems Faced during Implementation

During the implementation phase, many difficulties raised. Fixing some of them involved changes in other parts of the OpenSCAP library.

First problem that appeared was to synchronize the threads and processes writing to output correctly. It was found that some messages are missing in the output file on random occasions. To fix the output it was necessary to declare file descriptor of output file as external global variable and open the file always in append mode. Synchronization and right order of messages in output was guaranteed by using a file lock and mutexes.

Although the synchronization was correct, from the moment when all main steps of OVAL evaluation were covered by logging messages, it was apparent that sometimes the messages are not in expectable logical order. For example, if a definition consisted of multiple criteria, then log firstly showed that evaluation of all tests have started, and after querying for all objects was performed, results of all tests were reported, but in a different order. This gave an impression that the test evaluation was proceeded in parallel, which is not true. This confusion was caused by an issue in design of OpenSCAP. Firstly it collected all the items for all the tests in a definition. Results of tests and consequently the result of the definition were computed in second pass. In other words, all tests were examined twice—once to collect data for them system and once to get definition result.

To solve the problem of confusing output these two steps of evaluation were merged into a single one. The changes included a very detailed analysis of the source code, major refactoring and testing. Although it was a large change, it was successfully implemented. This change allows to remove a lot of duplicate code in next major version of OpenSCAP. Also, implementation of shortcut evaluation of logical expression became possible.

Chapter 5

Evaluation of Extension

The extension of OpenSCAP project that provides reporting and diagnostic of OVAL evaluation and partially also XCCDF evaluation was successfully designed and implemented.

In this chapter, we will demonstrate usage of the implemented feature and verify the functionality on some typical problematic OVAL definitions. Next, we will discuss possibilities of user testing. Then we will focus on feedback and suggestions for improvements. Finally, we will outline possible future extensions of this work.

5.1 Demonstration of Usage

In Section 3.4, we dealt with several problems of OVAL definitions. The evaluation should prove whether those problems are covered and can be identified using features introduced into OpenSCAP during work on this thesis. To verify this, some *OVAL definitions* documents were created. Then all of those definitions were evaluated by OpenSCAP 1.2.9 with verbose mode turned on, both in developer and info verbosity level.

To generate the log, the following command can be used:

```
# oscap oval eval --verbose INFO --verbose-log-file log.txt definition.xml
```

Listing 5.1 shows a snippet of output generated by the `oscap` tool for sample definition `service_chronyd_enabled`. This definition is extended by another definition, which can be easily noticed from the output.

Listing 5.1: Output generated by OpenSCAP in verbose mode.

```
.................
I: oscap: Identified document type: oval_definitions
I: oscap: Created a new OVAL session from input file 'service_chronyd_enabled.xml
    '.
I: oscap: No external OVAL variables provided.
I: oscap: Started new OVAL agent.
I: oscap: Querying system information.
I: oscap: Starting probe on URI 'pipe:///home/jcerny/openscap/src/OVAL/probes/
    probe_system_info'.
I: oscap: OVAL agent started to evaluate OVAL definitions on your system.
I: oscap: Evaluating definition 'oval:x:def:2': Service chronyd enabled.
I: oscap:  Evaluating systemdunitdependency test 'oval:x:tst:1': systemd test.
I: oscap:    Querying systemdunitdependency object 'oval:x:obj:1', flags: 0.
I: oscap:    Creating new syschar for systemdunitdependency_object 'oval:x:obj:1'.
```

```
I: oscap:    Starting probe on URI 'pipe:///home/jcerny/openscap/src/OVAL/probes/
    probe_systemdunitdependency'.
I: oscap:    Test 'oval:x:tst:1' requires that zero or more objects defined by '
    oval:x:obj:1' exist on the system.
I: oscap:    1 objects defined by 'oval:x:obj:1' exist on the system.
I: oscap:    All items matching object 'oval:x:obj:1' were collected. (flag=
    complete)
I: oscap:    In test 'oval:x:tst:1' all of the collected items must satisfy these
    states: 'oval:x:ste:1'.
I: oscap:    Entity 'dependency'='chronyd.service' of item '1312211' matches
    corresponding entity in state 'oval:x:ste:1'.
I: oscap:    Item '1312211' compared to state 'oval:x:ste:1' with result true.
I: oscap:  Test 'oval:x:tst:1' evaluated as true.
I: oscap:  Criteria are extended by definition 'oval:x:def:1'.
I: oscap:  Evaluating definition 'oval:x:def:1': Package chrony Installed.
I: oscap:    Evaluating rpminfo test 'oval:x:tst:2': package chrony is installed.
I: oscap:      Querying rpminfo object 'oval:x:obj:2', flags: 0.
I: oscap:      Creating new syschar for rpminfo_object 'oval:x:obj:2'.
I: oscap:      Starting probe on URI 'pipe:///home/jcerny/openscap/src/OVAL/probes
    /probe_rpminfo'.
I: oscap:      Test 'oval:x:tst:2' requires that every object defined by 'oval:x:
    obj:2' exists on the system.
I: oscap:      1 objects defined by 'oval:x:obj:2' exist on the system.
I: oscap:      Test 'oval:x:tst:2' does not contain any state to compare object
    with.
I: oscap:      All items matching object 'oval:x:obj:2' were collected. (flag=
    complete)
I: oscap:    Test 'oval:x:tst:2' evaluated as true.
I: oscap:  Definition 'oval:x:def:1' evaluated as true.
I: oscap: Definition 'oval:x:def:2' evaluated as true.
I: oscap: Evaluating definition 'oval:x:def:1': Package chrony Installed.
I: oscap: Definition 'oval:x:def:1' evaluated as true.
I: oscap: OVAL agent finished evaluation.
I: oscap: OVAL evaluation successfully finished.
...................
```

Results produced by the tool were compared with aforementioned problems with focus on how the tool helps to identify or solve them. In order to demonstrate the extension more, the attached data medium contains sample OVAL definitions and also corresponding output generated by OpenSCAP 1.2.9.

5.2 User Testing

User testing should be a final part of every software project and therefore it will be done also in this thesis.

However, the topic of this thesis requires specific approach to user testing. For example, we cannot use frequently used method of Hallway testing. This method is a quick method of testing in which randomly selected people are asked to try using the product. But OpenSCAP is sophisticated product. Average user would need at least training before starting using OpenSCAP and much more longer to create his own OVAL definitions. Therefore, testing on randomly picked people would not verify whether the solution fulfils the requirements.

The target audience of the tool are creators of security policies. There are not many of them and they are spread around the world. Organizing a usability testing session that will give useful evidence would be very complicated. However, it would be easy to address a few questions to contributors of OpenSCAP and mainly SCAP Security Guide projects, who initiated the idea of refining OVAL development within OpenSCAP project.

Finally, it was decided to ask the security policy developers directly and ask them for their feedback in a form of a questionnaire.

5.3 Feedback and Evaluation

User testing in a form of survey was done after OpenSCAP 1.2.9 was released in April 2016. This version was the first version that contained usable OVAL evaluation reporting. In the survey, 5 regular contributors of OpenSCAP and SCAP Security Guide projects were briefed about the new feature of reporting OVAL evaluation and were requested to try this feature out on an arbitrary content. Then they were were asked the following questions:

1. Which of information presented by the verbose mode do you consider useful?
2. Are there some messages that are redundant or are not meaningful?
3. What do you miss in the output?
4. How should I improve readability and orientation?
5. Should also other standards than OVAL be covered?
6. Have you encountered any bugs or crashes?
7. Would you appreciate if I continue improving it?

The feedback was mostly positive, but also critical. From received answers we will point out the most interesting facts now.

1. Overall, most of messages presented in the verbose mode were considered really useful. For example the fact that values of variables can be seen in the log was highly appreciated.
2. On the other hand, one contributor reported that for his OVAL definition which served to verify if all suid and sgid binaries are audited, the output of the verbose mode was thousands of lines long. The output mostly consisted of warning messages reporting that OpenSCAP failed to get extended access control list (ACL) for each file. It was suggested to remove this warning message or at least change its category to *Developer*.
3. Any specific information has not been missed, but they would welcome to go more into depth in some cases.
4. Although the messages in verbose log are indented, it was often required to use colors in output to emphasize the most important information, and insert some empty lines between each test evaluation.

5. Most of the people would welcome covering other standards, but it should be done after focusing on OVAL more in depth.

6. Nobody encountered any bugs or crashes. However, this does not mean that there are not any.

7. Everybody wanted that work on the extension introduced in this thesis continued in the future.

5.4 Ideas for Further Improvements

This work can be extended easily, because a possibility of future expansion has been kept in mind from the very beginning. For example, every OpenSCAP developer or maintainer can add new messages for the reporting log in any place of the code base by using simple macros now.

In this section, we will outline some of possibilities that can be done after submitting this thesis. The feedback aggregated in the previous section confirms that future improvements are highly desirable.

Firstly, due to a limited time, only the most painful parts of OVAL evaluation were examined, analysed and covered. Further improvements should begin by covering other parts of OVAL evaluation and identifying more corner cases.

Secondly, OpenSCAP also supports other SCAP standards, but they are not covered now by the verbose mode. Therefore similar work could be done also for other standards than OVAL. For example, we currently do not report evaluation of SCAP Data Streams. It would be also beneficial to alert user if SCE content is being evaluated, because it means that OpenSCAP will invoke arbitrary script or binary program. To cover other standards the same approach that was used in this thesis can be reused.

Furthermore, the format of output could be improved. To create a more user friendly reports, eg. in PDF or HTML formats would be possible, although it would very likely require adding another layer on top of the current OpenSCAP architecture. We started just with a text format because it brings a benefit quickly without an overhead. More likely, the text output should be improved by using colors.

In addition, the reporting and diagnostics can be reused in SCAP Workbench because currently any information regarding the process are missed. SCAP Workbench only shows progress bar and final results. A most simple solution would be a button in the Scan Results window, that would open a window containing the same content as produced by the `oscap` verbose mode. More advanced solutions will be also possible because Qt toolkit used in Workbench provides various capabilities for building advanced user interfaces.

Since the majority of changes was done in OpenSCAP library which is a basic stone of the whole OpenSCAP ecosystem, other projects under OpenSCAP umbrella will be able to leverage them.

Chapter 6

Conclusion

The goal of this thesis was to get familiar with contemporary security audit technologies and project OpenSCAP, and design and implement a solution supporting development of security policies.

The core part of the work lies in the analysis of the OpenSCAP project, inspects and considers its internal implementation.

The thesis starts with explaining key concepts of security audit. Second chapter describes the main technology used for automation of security audit—the SCAP standard and its open source implementation OpenSCAP. Third chapter discusses the development of security policies using OVAL and identifies problems and difficulties of this language. The author drew from his more than year experience from working on the OpenSCAP project.

In fourth chapter, the author builds on discovered facts to design a diagnostic tool for reporting process of OVAL evaluation. Fifth chapter describes its implementation and problems faced during the implementation.

Author successfully introduced a new verbose mode to the OpenSCAP command line tool. This mode reports performed actions and serves to diagnose possible bugs in OVAL definitions or in OpenSCAP implementation. The solution was extended to also cover XCCDF evaluation. The patches changed the original source code significantly.

The test cases described in last chapter have shown that the solution is able to help with debugging OVAL definitions and that the tool reports the process of OVAL interpretation in a detailed but easily understandable way. All defined problems of OVAL and OpenSCAP have been successfully faced.

The author himself used the tool in his daily work on the OpenSCAP project, and so did other community members.

The work has been continuously accepted by OpenSCAP upstream and was released in OpenSCAP 1.2.9 in April 2016 and therefore will become available in major Linux distibutions soon.

Bibliography

[1] The heartbleed bug. [online], 2014. http://heartbleed.com/.

[2] NIST solicits comments on the security content automation protocol (SCAP). [online], 2015. http://csrc.nist.gov/publications/drafts/800-126/sp800-126r3_call-for-comments.html.

[3] The OpenSCAP portal. [online], 2015. https://www.open-scap.org.

[4] OpenSCAP tickets. [online], 2015. https://fedorahosted.org/openscap/.

[5] OVAL - the open vulnerability and assessment language. [online], 2015. https://oval.mitre.org/index.html.

[6] OVAL specification. [online], 2015. https://oval.mitre.org/language.

[7] OpenSCAP GitHub source code repository. [online], 2016. https://github.com/OpenSCAP/openscap.

[8] OpenSCAP issues. [online], 2016. https://github.com/OpenSCAP/openscap/issues.

[9] Rsyslog documentation. [online], 2016. http://www.rsyslog.com/doc/master/index.html.

[10] Petr Beňas. Design better content development process for SCAP standards. Diploma thesis, Brno University of Technology, Faculty of Information Technology, 2013. http://www.fit.vutbr.cz/study/DP/DP.php.cs?id=15619&file=t.

[11] Rich Murphy. The practical guide to security compliance. [online], 2014. http://www.blackstratus.com/practical-guide-security-compliance/.

[12] The Open Web Application Security Project OWASP. The Ten Most Critical Web Application Security Risks. [online], 2013. https://www.owasp.org/index.php/Top_10_2013-Main.

[13] Vratislav Podzimek. SCAP policy compliance configuration in Linux installations [online]. Diploma thesis, Masaryk University, Faculty of Informatics, Brno, 2013. http://is.muni.cz/th/324874/fi_m/.

[14] Amol Sarwate. The GHOST vulnerability. [online], 2014. https://community.qualys.com/blogs/laws-of-vulnerabilities/2015/01/27/the-ghost-vulnerability.

[15] David Waltermire, Stephen Quinn, Karen Scarfone, and Adam Halbardier. *The Technical Specification for the Security Content Automation Protocol (SCAP): SCAP Version 1.2*. National Institute of Standards and Technology, Gaithersburg, Maryland, 2011.

[16] Jim Whitehurst. *The open organization: igniting passion and performance*. Harvard Business Review Press, Boston, Massachusetts, 2015.

[17] Jan Černý. Scanning containers for vulnerabilities. [online], 2016. http://www.jan-cerny.cz/2016/02/18/scanning-containers-for-vulnerabilities/.