MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

SCAP policy compliance configuration in Linux installations

MASTER THESIS

Vratislav Podzimek

Brno, spring 2013

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: Ing. Mgr. Zdeněk Říha, Ph.D.

Acknowledgement

Hereby I would like to thank to my parents for their love and everlasting support and my girlfriend for her love and kind words as well as a lot of patience when things didn't go the way I wanted them to go. Big thanks also goes to the advisers Ing. Mgr. Zdeněk Říha, Ph.D. and Ing. Peter Vrabec (the technical adviser) and my colleague Ing. Martin Sivák for all valuable pieces of advice they had been giving me during my work on this thesis and to Bc. Šimon Lukašík for his work on the **oscap** tool's support for remediation and for extracting fix data needed by the addon. Last but not least, I would like to thank to Ing. Marie Pospíšilová and Mgr. Miroslav Špetla for the first sparks of computer science knowledge that has become a burning interest I now have, and resulted, among the other things, in this thesis.

Abstract

One of the biggest problems of IT security is that even there are many known principles and rules which should be followed to prevent attacks and misuse, they are hardly ever deployed leaving systems with vulnerabilities caused by wrong configuration. This thesis focuses on the Security Content Automation Protocol (SCAP) and the possibilities of (semi-)automated evaluation and remediation of systems according to rules provided as a content following that specification. As a result of analyzing the SCAP and examples of SCAP content an approach based on two-phase application of the content during the installation process is suggested and implemented as an extension for the Fedora and RHEL GNU/Linux distributions' installer, Anaconda, with the goal to provide administrators an easy way to choose and apply the right security profile for their newly installed systems.

Keywords

SCAP, GNU/Linux, Fedora, RHEL, Anaconda, OS installation, Python, hardening, IT security, OpenSCAP

Contents

Introduction

We live in a world where computers drive basically all areas of human interest and allow them grow more rapidly than ever before [1]. But wherever computers bring many advantages, there they also bring a big potential for problems caused by their misconfiguration, failures or misuse by an adversary. To mitigate these risks it is very important to keep in mind that these causes are mutually related to each other. Wrong configuration may result in a failure, but similarly a failure may result in a wrong configuration[1]. The same applies to the relation between wrong configuration and misuse by an adversary and to the relation between a failure and misuse. That means all these risks has to be fought against to get a safe and secure system.

Obviously, there are two groups of people who are responsible for mitigation of such risks – the producers (of software or hardware) and the customers. But with each kind of risk there are different possibilities of actions these two groups could take. Failures are caused by flaws in the software or hardware design or its implementation and construction, respectively, or by not enough redundancy that would cover the cases of expected failures due to quite short lifetime of some hardware components. Thus only a little can be done to fight the risks on the customer's side. But with misconfiguration and misuse prevention the burden lies on the customers because only they can define which services the system will be providing and then configure it in a right way and only they can set up the right policies preventing misuse. With both configuration and policies being very broad areas it is hard to achieve the state where the system is set up in a best possible way[2].

---

1. e.g. when a disk with logs, audit information or network traffic data fails
2. In fact it is impossible because there are always new types of attacks appearing that require some counteractions in the configuration and policies.

Because only a little can be done on the customer's side to mitigate the risk of failures and there are well-known principles [2] that should be followed to do so, we focus on the area of system configuration and policies. While the right configuration of the system to make it work in a way it is supposed to work depends on the individual case and purpose of the system, in the area of the configuration responsible for preventing attacks and misuse there are lots of general recommendations, rules and principles that can be followed to improve it. However, the problem is that even though such guidelines are available, only few administrators know where to get them and even less administrators actually apply them. Where to get these recommendations, rules and principles, how to apply them and how to encourage administrators to apply them are the areas this thesis focuses on.

The first chapter of the thesis describes the *Security Content Automation Protocol* (SCAP) and its components. One of the problems with application of *security content*[3] is that there are many formats used by the different producers of the content which complicates the already hard task to achieve and maintain secure configuration. The SCAP is a standard that tries to resolve this problem by defining a machine-readable format for the documents holding security content. The main advantage of the format is that the evaluation of the content on a particular system can be done in an automated way.

Once there is some content, the other questions are how to apply the content on a system and when to do it. The application of some content on a particular system typically has two phases – *evaluation* and *remediation*. *Evaluation* is the phase where the rules defined by the content are evaluated on the system to find out if they are followed or not. In case some rules are not followed (the check does not pass), *remediation* takes place to fix the system's configuration so that the rules are followed (their checks pass). Both of these phases can be done manually, automatically or semi-automatically. Manual checking if the rules defined in the content are met on a particular system would be possible, but, as it was already mentioned, the standardized format (defined by the SCAP) is machine-readable

3. which is a term used for recommendations, rules, principles, examples, etc.

and thus can be applied (semi-)automatically[4]. One of the tools that allow such (semi-)automated evaluation of the SCAP content as well as remediation is part of the *OpenSCAP* project [3] that, together with a number of related projects, focuses on open-source [4] implementations of tools and libraries that facilitate the work with SCAP content. The second chapter of this thesis describes the *OpenSCAP* project and some other open projects related to it.

The other question is when to do the evaluation and remediation. For sure there first needs to be a system to apply the content on. But on the other hand using the system with a wrong configuration and thus in a vulnerable state may result in an undetected attack that may ruin any following attempts to improve system's security and configuration (e.g. by letting an attacker to create an undetected backdoor to the system). Thus, at the first glance, a best time to apply the security content on a system seems to be right after its installation. But as will be described in the fourth chapter focused on security content (policy) compliance configuration, there are reasons for a different, better approach – doing evaluation and remediation during the operating system (OS) installation.

To implement that better approach is the main goal of this thesis. Because it requires extending an OS installer, we will focus on open-source projects that provide much easier way to understand and extend their functionality. In particular we will focus on the GNU/Linux[5] distributions *Fedora* and *Red Hat Enterprise Linux* (RHEL) and their installer, *Anaconda*, that provides a support for extensions (called addons). Moreover there is a lot of security content available for these two systems (in particular for the RHEL). The fifth chapter of this thesis focuses on the *Anaconda installer* and the sixth chapter describes an addon that implements the evaluation and remediation in the OS installation process. The last chapter then summarizes the goals of the thesis and describes which of them and how have been achieved.

---

4. A full automation is in many cases impossible, because somebody has to be responsible for the configuration of a particular system and at the same time somebody has to decide which rules has to be followed and required.
5. which is an operating system using the Linux kernel [5] and tools from the GNU project [6]

Chapter 2

Security Content Automation Protocol

There are many institutions, organizations and projects producing and providing security content in many different forms. From one or two-page pamphlets with just basic and most important rules like:

> "**Encrypt all data transmitted over the network.**
>
> Encrypting authentication information (such as passwords) is particularly important." [7]

that can be found in the *Hardening Tips for the Red Enterprise Linux 5 pamphlet* [1] to hundreds of pages long documents like *Guide to the Secure Configuration of Red Hat Enterprise Linux 5* [8]. Both these documents are provided by National Security Agency (NSA) which is together with National Institute of Standards and Technology (NIST) one of the most important and reliable providers of such recommendations and rules. NIST, for example, provides the *United States Government Configuration Baseline* (USGCB) which is a set of security configuration baselines that are required or recommended to be followed in federal institutions in the United States.

Usage of many different formats for the security content complicates the interoperability of the tools used to process it and practically blocks attempts on a collaborative development of such content. Generally, in order to achieve interoperability between multiple tools processing some data as well as with the producers creating such data there needs to be a protocol or standard specifying the format. In the area of security content such protocol is the *Security Content Automation Protocol* (usually referred as SCAP) with its most

1. the word *hardening* is a term used for application of the security content to make the system less vulnerable to attacks and misuse

recent version specified in the NIST Special Publication 800-126 – *The Technical Specification for the Security Content Automation Protocol (SCAP): SCAP Version 1.2* [9] starting with the following definition:

> "The Security Content Automation Protocol (SCAP) is a suite of specifications that standardize the format and nomenclature by which software flaw and security configuration information is communicated, both to machines and humans." [9]

As the definition describes, SCAP is not a single specification, but a set of multiple component specifications. They can be divided in the following five categories (with overview of the components):

- languages – XCCDF, OVAL, OCIL,
- reporting formats – ARF, Asset Identification
- enumerations – CPE, CCE, CVE,
- measurement and scoring systems – CVSS, CCSS and
- integrity – TMSAD.

Many of these components use XML as the underlying technology and a typical SCAP content then consists of multiple pieces of XML with format defined by the (standalone) specifications of the components listed above either in separate, but mutually linked, files or in a single file with the pieces encapsulated in *data stream collection* defined in the aforementioned NIST Special Publication 800-126.

Every well-formed [10] XML file must have one root element. In case of the SCAP content encapsulated in a single file it is the `data-stream-collection` element that defines XML namespaces and is composed of one or more `data-streams`, `components` and optionally `Signatures`. `Signature` elements (which are required to be valid XML digital signatures [11]) can be used to ensure integrity and authenticity of the data. `component` elements then hold the content, as defined by the SCAP components' specifications, that are referenced by the `data-streams`. `data-stream` elements are composed (among the other things) of `dictionaries` (links to CPE

dictionaries), `checklists` (links to XCCDF content) and `checks` (links to OVAL or OCIL checks). Since the intention was to facilitate the creation of data streams and data stream collections from an existing content using separate files, all these links are realized as `component-ref` elements that provide IDs for what used to be separate files (so that all the references from an existing content are still valid) and contain `catalogs` that do basically the same for the referenced component – i.e. `catalogs` define mappings between what used to be separate files used by the component to IDs of the `component-refs` for the content from those files.

The rest of the chapter is dedicated to descriptions of the SCAP components, that are the basic building blocks of the SCAP content, with the amounts of text proportional to the importance of the components in connection with this thesis.

2.1 SCAP languages

The languages category contains the two probably most important components of SCAP – Extensible Configuration Checklist Description Format (XCCDF) and Open Vulnerability and Assessment Language (OVAL) – and the Open Checklist Interactive Language (OCIL). In short, the Extensible Configuration Checklist Description Format is a language that can be used to define rules and cases in which particular rules should be applied. As it is the most important component of SCAP (or at least from the perspective of this thesis) and as it directly or indirectly relates to all the other components, a separate section at the end of this chapter is dedicated for its broader description.

The XCCDF provides a way how to define rules, but only on the level of their titles, descriptions and IDs. While for the human reader just a description (or even a title) like *"Create a separate partition or logical volume for /tmp"* could be enough to get the meaning of the rule and to perform a check, for a computer anything like that is practically impossible. And since one of the main goals of the SCAP is automation, it needs to provide a way how to include some machine-readable data to the rules on basis of which an automatic evaluation could be done. That is where the *Open Vulnerability and*

Assessment Language (OVAL) comes into play with its three major components – *OVAL System Characteristics*, *OVAL Definitions* and *OVAL Results* as defined by *The OVAL Language Specification* [12]. The first component is used for representing the configuration information of systems, the second one is for expressing a specific machine state and the last one is for reporting the results of an assessment. Because system characteristics and definitions of the desired state depend on the particular system, the OVAL specification defines just a basic, but extendable framework. Other specifications focused on particular systems then give the OVAL content the ability to describe the system characteristics and the desired state of real systems. Examples of such extending specifications are *The OVAL Language UNIX Component Model Specification* [13] for UNIX-like systems [14] and *The OVAL Language Windows Component Model Specification* [15] for Windows systems, but these are definitely not the only ones (in total there are currently 15 extensions for *OVAL Definitions*). An interesting fact about *The OVAL Language Specification* and the specifications of the extensions is that they use the Unified Modeling Language (UML) [16] to describe XML elements. On one hand it is quite unusual, but on the other hand it perfectly fits the model the OVAL language uses where new elements are in many cases defined as being inherited[2] from some other elements and elements often use (contain) the other elements.

From the perspective of this thesis the most important part of the OVAL language are *Oval Definitions* that are, among the other things, used in XCCDF rules' checks. Example of OVAL definitions can be seen in the attachments. It contains metadata about the content generation (`product_name`, `schema_version` and `timestamp` elements) followed by the `definitions`, `tests` and `objects`. Each definition has some metadata (`title`, `description`) and `criteria` where each `criterion` references one or more `tests` that describe what should hold for some `objects`. The basic framework covered by the OVAL language specification consists of all the elements but `tests` and `objects` that are system-specific (and thus specified in the extensions that apply to a particular system).

2. in the same way as it is known from the object oriented programming

As it was already mentioned above, an OVAL content is, among the other things, usually used to define checks whether XCCDF rules are followed or not. That means there has to be an interpreter that actually checks the system according to the OVAL content and provides back data showing if the check failed of passed. There is a freely available[3] referential implementation of such tool – **The OVAL Interpreter** [18], but since it doesn't cover all the functionality needed in many cases (especially the extensions), some other interpreters are usually being used. For example there is the *jOVAL* [19] project focused on implementing a more comprehensive OVAL interpreter (covering more extensions) that is licensed under the Affero GPL license [20].

Another important part of the OVAL framework is the *OVAL Repository* described on its home page as follows:

> "The OVAL Repository is the central meeting place for the OVAL Community to discuss, analyze, store, and disseminate OVAL Definitions. Members of the community contribute definitions by posting them to the OVAL Repository Forum, where the OVAL Team and other members of the community review and discuss them." [21]

which plays an important role in reusability of the OVAL content.

Though it may seem that the OVAL covers everything needed to define checks for XCCDF rules, there are cases when it cannot be used because of the fact that it is oriented only on systems and their states and configurations and not on the users of such systems. This allows OVAL content to be evaluated automatically, but on the other hand doesn't cover the rules like *"All users must have passed the basic security training."* that may appear in the SCAP content. For cases like that elements of the *Open Checklist Interactive Language* are used. The most recent format of the language is defined by the NIST Interagency Report 7692 – *Specification for the Open Checklist Interactive Language (OCIL) Version 2.0*. [22] The OCIL provides a unified and general framework for creating questionnaires that can be used, among the other places[4], in a SCAP con-

3. licensed under the BSD licence [17]
4. e.g. school exams

tent, especially as the interactive checks of some types of rules. To be compatible with OVAL, the OCIL also uses XML as an underlying technology. The root element of the OCIL data is `ocil` which contains metadata[5], `questionnaires`, `task_actions`, `questions`, `results` and other elements. The `questionnaires` reference one or more `task_actions` that describe which `questions` should be used and what should happen if the questions cannot be answered. The results of the interactive checks (i.e. `task_actions`) are then stored in the `results` element.

2.2 SCAP reporting formats

Another category of the SCAP components contains the SCAP reporting formats – *Asset Reporting Format* (ARF) and *Asset Identification*. As their names suggest these two components are closely related, concretely the assets[6] the ARF data reports about can be identified the way the Asset Identification format defines. The specification of the Asset Identification is covered in the NIST Interagency Report 7693 – *Specification for Asset Identification 1.1* [23]. The Asset Identification format provides a standardized, general and machine-readable way of identification. It uses XML and leverages, among the other standards, *extensible Address Language* (xAL) and *extensible Name Language* (xNL) by *Organization for the Advancement of Structured Information Standards* (OASIS) to reuse existing standards instead of "re-inventing the wheel". The core specification provides data model for eleven types of assets – Person, Organization, System, Software, Database, Network, Service, Data, Computing Device, Circuit and Website, but extensions can define additional types by inheriting from any of those types. The particular assets are then identified using a set of so called *literal attributes* (classical identifiers like Media Access Control (MAC) address of a device etc.) and *relationships* to other assets.

Asset Reporting Format specified in the NIST Interagency Report 7694 – *Specification for the Asset Reporting Format 1.1* [24] defines a

5. like information about the data generation and the `document` itself – its `title`, `description` and `notices` – that can be shown to the user
6. defined as *"anything that has value to an organization"* [24]

versatile data model for interoperable and machine-readable reports about assets. The relation to the other components of the SCAP (except the Asset Identification) may be not obvious, but due to defining only a common structure the ARF can be used to create a report about e.g. OVAL or XCCDF results (which are, by definition, assets). The root (XML) element of the ARF content is `asset-report-collection` that contains `report-requests`, `assets`, `reports` and `relationships`. `assets` are the assets identified using the Asset Identification for which `reports` are generated on `report-requests`' behalf. The `relationships` then provide subject–predicate–object relations between the other elements (e.g. which `report` was generated for which `asset` on which `report-request`'s behalf.

2.3 SCAP enumerations

One of the goals of every standard or specification defining a data format is to provide an interoperable and platform independent way to represent some kind of data. The same applies to SCAP and all of its components which means that SCAP content can be created, stored and processed on any platform (of course some computing and storage resources are required). But on the other hand, as we have already seen in the section about the OVAL, pieces of SCAP content may be targeting a particular platform. Just a simple example – there may be a rule saying that system firewall has to be turned on, but a check for that rule will definitely use different steps on some Windows machine than on e.g. a GNU/Linux machine. So in a high-quality and versatile content there would be at least two checks related to such rule, each for a particular platform, and the content processor[7] would have to decide and make sure that the right sequence of steps (the right check) is used on any machine. But what may seem trivial for a human is basically impossible for a computer. Well, without any additional data. The data allowing computer (or more precisely the content processor) to decide which check to use on which

7. a tool that evaluates the content on a machine

machine is the *Common Platform*[8] *Enumeration* (CPE) defined by four NIST Interagency Reports: 7695 – *Common Platform Enumeration: Naming Specification* [25], 7696 – *Common Platform Enumeration: Name Matching Specification* [26], 7697 – *Common Platform Enumeration: Dictionary Specification* [27] and 7698 – *Common Platform Enumeration: Applicability Language Specification* [28], all, in the time of this thesis being written, in version 2.3. As the number of specifications shows, there are four components of CPE. *CPE Naming* defines how valid CPE names should be created, *CPE Name Matching* defines how the evaluation of patterns (will be described later) in the CPE identifiers should be done, *CPE Applicability Language* is a format of XML data that allows creation of more complex expressions with *AND* and *OR* logical operators (to express for example "Mozilla Firefox 3.6 on Windows 7").

CPE Dictionary is a format enveloping multiple CPE items in one component (file or XML subtree) that could be used in connection with other SCAP content. The CPE dictionary (the `cpe-list element` is composed of `cpe-items` linking CPE name and check(s) specifying how to recognize the platform (usually OVAL checks). In a SCAP content the CPE names or IDs of more complex expressions are used in places where a platform specification is needed (e.g. in case of multiple platform-dependent checks for a rule) and the content processor is responsible for searching for the checks and evaluating them. Since there is only a limited number of commonly used platforms (the CPE names), the goal is to create a single repository (CPE Dictionary) of CPE items that would be used by all SCAP content. Such official CPE Dictionary is hosted by NIST [29] and all creators of SCAP content are encouraged to share their CPE data there if possible. CPE names consist of multiple parts that specify e.g. the type, name and version of the platform. Since some content may apply to multiple versions of a platform, CPE specification allows usage of some simple patterns in identifiers[9] that can be then matched with multiple CPE names.

8. where the *platform* means a class of applications, operating systems or hardware devices

9. the '*' and '?' characters with common meaning known from e.g. file path patterns and the keyword *ANY* with obvious meaning plus missing item having the same meaning as *ANY*

The other two enumeration components of SCAP are *Common Vulnerabilities and Exposures* (CVE) and *Common Configuration Enumeration* (CCE) that reflect the fact that there are well-known vulnerabilities as well as well-known recommended configurations. The major parts of these components are freely available lists – the *CVE List* and the *CCE List* – providing data gathered from many sources. The ways how these lists are being populated are described on web pages of the projects:

> "The process begins with the discovery of a potential security vulnerability or exposure. The information is then assigned a CVE Identifier by a CVE Numbering Authority (CNA) and posted on the CVE Web site. As part of its management of CVE, The MITRE Corporation functions as Editor and Primary CNA. The CVE Editorial Board oversees this process." [30]
>
> "CCE entries are currently assigned to configuration issues by members of the CCE Content Team and posted on the public CCE Web site." [31]

2.4 SCAP measurement and scoring systems

One of the long standing problems of the security-related configuration and policy management has always been prioritization. With a lot of known vulnerabilities (and new ones always emerging) together with known configuration patterns and rules that should be followed to mitigate the risks of their misuse it is important to first fix the issues that bring the highest risk. But it is hard to asses the vulnerabilities and the configuration rules correctly to find out what these "hot issues" are. To mitigate this, two standardized and versatile mechanisms for scoring were created – *Common Vulnerability Scoring System* (CVSS) and *Common Configuration Scoring System* (CCSS) – specified in NIST Interagency Reports 7435 – *The Common Vulnerability Scoring System (CVSS) and Its Applicability to Federal Agency Systems* [32] and 7502 – *The Common Configuration Scoring System (CCSS): Metrics for Software Security Configuration Vulnerabilities* [33]. These two scoring mechanisms are very similar to

each other. Products of both are values between 0.0 and 10.0 and both use the same three metric groups – *Basic*, *Temporal* and *Environmental* – where the first one *"represents the intrinsic and fundamental characteristics of a vulnerability that are constant over time and user environments"* [32], the second one is for characteristics that vary with time but not with the user environment and the last one covers the characteristic that are specific for the user environment. This also determines who is responsible for performing the scoring in particular metric groups.

The computation of the resulting score is then defined by equations combining weighted "subscores" of these metric groups to result in a number between 0.0 and 10.0 as the final score. Computations of "subscores" are defined by equations and coefficients weighting numeric values corresponding to the semantic values of the elementary metrics (like e.g. *Access Vector*, *Access Complexity*, and *Authentication metrics* that are parts of the *Basic metric group* of CVSS). The resulting score is then assigned an identifier which can be used in for example SCAP content to allow prioritization. One of the biggest advantages of CVSS and CCSS is that the outputs are not only the score values but also all the values the resulting score values were computed from.

2.5 Trust Model for Security Automation Data

Evaluation of the SCAP content needs to run under administrator's privileges because it often needs to check and modify system's configuration. This means that should the content contain some malicious data, it could for example ruin the system or create backdoor when being evaluated. In the same time a typical SCAP content has tens of thousands of lines of XML data [34] that can hardly be checked for malicious content either manually or automatically. Thus the content consumer has to trust the content producer that the provided content is secure and will not cause any unexpected changes in the system. Once such trust is established, the consumer still has to verify the authenticity (whether it was really created by the trusted producer) and integrity (whether it was not modified after the producer created it) of the content. The most common way

for the producer of some data to allow consumer to verify its authenticity and integrity is to create a digital signature which is also a method used by the SCAP. And since SCAP uses XML as an underlying technology, the *Trust Model for Security Automation Data* (TMSAD) component, that specifies the format of digital signatures of the SCAP content, leverages the *XML Signature Syntax and Processing* (XMLDSig) standard[10] and just specifies which algorithms and protocols can be used for digital signatures of security content [36]. More about the XMLDSig can be found e.g. at [11].

2.6 Extensible Configuration Checklist Format

So far the SCAP components that allow definitions of automated or manual checks (OVAL and OCIL, respectively), reporting (ARF and Asset Identification), platform, vulnerabilities and configurations enumeration (CPE, CVE and CCE), scoring (CVSS and CCSS) and integrity verification (TMSAD) have been described. But as it was described in the introduction, the configuration patterns and recommendations (the security content) are usually given as a set of rules that the targeted system must follow to be in compliance with the content. Also systems serving different purposes will need to follow different sets of rules as well as some tweaks to the rules may be needed on the content consumer's side. Finally, when the rules are evaluated on a particular system, the results of the evaluation has to be available to check if the rules are followed or not. All these cases are covered by the *Extensible Configuration Checklist Format* (XCCDF) with it's most recent version specified in the NIST Interagency Report 7275 – *Specification for the Extensible Configuration Checklist Description Format (XCCDF) Version 1.2* [37].

As the majority of the SCAP components, XCCDF also uses XML as the underlying technology. But since it covers a wide area of the content data, its specification defines three types of XCCDF documents each with a different root (top-level) element:

- `Benchmark` – that can contain rules,
- `TestResult` – for results of `Benchmark` evaluation and

10. defined by the World Wide Web Consortium [35]

- `Tailoring` – for modifications of some `Benchmark`.

As it can be seen from the list above, the most important document type (and root element) is the `Benchmark`[11] that consists of `Profile` elements or so called *items* (or both) where an *item* can be either `Value`, `Group` or `Rule`. `Rule` elements (which define the configuration rules that should be followed) may use `Values` and may be (together with `Values`) grouped into `Groups` or `Profiles`. The `Profiles` may contain either `Groups` or `Rules` and the same applies to `Groups` that can be nested this way. Every *item* can then extend another item of the same type if it is in its scope (where a scope of an item is very similar to a scope of a class attribute in object-oriented programming[12]). Other than that the `Benchmark` element also holds metadata describing it. For example the `status` that *"SHOULD indicate the level of maturity or consensus for the benchmark"* [37] (and optionally also the `dc-status` that *"Holds additional status information using the Dublin Core format"* [37]), the `title`, the `description` and `version`. If the benchmark is platform specific, it should also contain the `platform` element specifying the platform with a CPE identifier as described in the SCAP enumerations section (2.3).

Every *item* (group, rule or value) should contain the same metadata elements as listed above for the `Benchmark`[13] and possibly also one or more `warning` and `question` elements where `warnings` should inform e.g. about the serious impacts of the item's application (like *"remote connections to the machine will not work if this rule is followed"*) and `questions` should help with the decisions about the *item* during tailoring (will be described later). In addition to groups the items may also be grouped into clusters by multiple of them having the `@cluster-id` attribute set to the same value. On top of that rules and groups may contain the `selected`, `requires` and `conflicts` elements where the the first one means whether the rule

11. the elements listed in the following paragraphs don't cover all elements that are allowed or required in particular context (all can be found in the XCCDF specification [37]
12. the precise definition of the scope of an item can be found in the XCCDF specification [37]
13. with `platform` allowed only in `Group` and `Rule` elements

or group should be evaluated or not and `requires` (`conflicts`) elements contain references to other items that have to (cannot) be selected in the same time.

The most important building block of the SCAP content is probably the `Rule` element as it holds data that specify a rule that should be followed to remediate some vulnerability or generally improve the system's configuration. In addition to the elements it has common with `Groups` and `Values` it may (and in most cases should) contain other child elements holding data that allow (semi-)automatic evaluation of the rule on a particular system and further describe or identify the rule. Those elements are the parts of the XCCDF that make use of the other SCAP components. To follow the ordering from the XCCDF specification, we will start with the `ident` element, that is expected to hold an assigned name of the issue the rule instantiates, implements or remediates. The data in the `ident` element should be a name assigned in the system (e.g. CPE, CVE, CCE and others) specified by the `@system` attribute's value. Then there may be an `impact-metric` element[14] that should contain a value as defined by the CVSS specification. Among the other possible child elements there are three of them that are crucial for the evaluation and remediation – `check`, `complex-check`, `fix` and `fixtext`.

`check` and `complex-check` are mutually exclusive and either a one or more `checks` can be used or one `complex-check` that defines a boolean expression on multiple checks. Each `check` should then contain or reference data that can be used to check if the system is in compliance with the rule or not and its `@system` attribute's value should specify, by a Universal Resource Identifier (URI), the system the check data should be interpreted with. Typically this will be either the URI of the OVAL system[15] or the URI of the OCIL system[16], but any other value understood by the processing tool can be used as well (we will see an example in the chapter 6). Another important attribute is `@selector` that can be used to specify in which cases the check should be used by setting the same value for the check's and rule's `@selector` attributes. If there are multiple checks

14. although it is recommended to be moved under the `metadata` element
15. http://oval.mitre.org/XMLSchema/oval-definitions-5
16. http://scap.nist.gov/schema/ocil/2.0

with the same value of `@selector` used by the rule, then each check should use a different system. The OVAL checks take precedence over OCIL checks because the former system supports automatic evaluation.

Every rule has its check(s) associated so that it can be decided whether a particular system is in compliance with the rule or not. If it is, then nothing more needs to be done except of reporting that the rule is followed (or that its check passed). But if the check fails there are two options of what can be done – either the fail is just reported, so that the administrator of the system knows the rule isn't followed or an attempt can be made to fix the system's state or configuration so that the rule is followed and subsequent run of the check passes. The second case is covered by the `fixtext` and `fix` elements. While the `fixtext` element is expected to contain plain text that describes how to fix the system's state or configuration, the `fix` element is supposed to contain a sequence of commands that can be automatically interpreted by a tool specified by its `@system` attribute. `fixtext` may also contain a reference to a `fix` which allows pairing fix procedures with their explanations. In addition to that, both `fixtext` and `fix` may also specify whether the system should be rebooted or not (the `@reboot` attribute) after their application, what is the so called *strategy* of the fix (the `@strategy` attribute with one of predefined values[17]), what is the level of disruption (`@disruption`) the application of the fix would cause and what is the complexity or difficulty (`@complexity` of the application. On top of those attributes, the `fix` element may also specify the platform (the `@platform` attribute with a CPE name or CPE applicability language expression's id as the value) it can be applied to and the already mentioned system (`@system`) it should be interpreted with. There are some predefined values (URIs) of the `@system` attribute for most common systems, such as `urn:xccdf:fix:urls` for a list of URLs to the resources that should be applied and multiple `urn:xccdf:fix:script:LANGUAGE` values where the `LANGUAGE` is substituted by a script language identifier (e.g. `sh`, `perl`, `vbscript`,

17. for example `configure` for configuration changes, `disable` for turning off or uninstalling system's component of `patch` for an application of a patch, update, etc.

etc.), but the content may use any value that will be understood by the processing tool.

The `Value` elements are useful for giving names to parameters used in the content and also for tailoring where the real value they hold can be easily changed in multiple places with only one change done, actually.

So far we have described the `Benchmark` that can hold `Rules` (or `Groups` of `Rules`), that can be selected or not, `Values` and some additional data. While this would be enough as a format for data that can be evaluated on a particular system, there would have to be a lot of documents each focused on a particular use case of a system and all these documents would redundantly contain a lot of rules that could have been shared in one place. The refinement of the content so that it fits better some use case is called *tailoring*. The basic way to do tailoring is to use `Profile` elements that may (in addition to the metadata and various elements shared with `Rule` and `Group` elements) select (the `select` element) rules and groups or clusters rules and refine values and rules (the `refine-value` and `refine-rule` elements) by using so called *selectors*[18]. The `Tailoring` elements and documents then allow tailoring to be done outside of the `Benchmark` element.

The third document type (and in the same time root element) of the XCCDF is the `TestResult` which can be, as its name suggests, used to store information about a single evaluation of a (possibly tailored) benchmark. Because the test results only make sense in connection with the benchmark and tailoring that were used to produce the results, it needs to reference both of these elements (or documents). In addition to that, the `TestResult` may contain a name of the `organization` applying the benchmark, the `identity` of the user applying the benchmark, the `profile` that was used and information about the target system the benchmark was applied to (using the *Asset identification* component). The `TestResult`, of course, has to contain the actual results of the test (benchmark application). This can be done in two ways – by an overall `score` (with `@system` attribute specifying the scoring system used) which is compulsory and optionally with `rule-result` elements representing the results

18. more about the selectors can be found in the XCCDF specification [37]

of rule evaluations (as metadata and optionally results from the check system plus one of the predefined values such as `pass`, `fail`, `fixed`, etc.).

As can be seen from the length of this chapter providing a rather brief than exhaustive description of the components, the *Secure Content Automation Protocol* is a very complex and broad specification and every tool that wants to be in conformance with it has to follow a lot of rules and restrictions.

Chapter 3

Open SCAP Projects

An important feature of the SCAP, also pointed out in it's name, is that it allows automation which means that the evaluation of a SCAP content on a particular system can be done by some tool instead of manual verification of the rules and manual fixing of the system's state and configuration. On the other hand the creation of SCAP content cannot be done automatically (as today's computers are not able to detect a vulnerability, provide a fix for it and compose an XML file will all the data needed), but still can be done in a "computer-aided" way. There are many projects focused on both SCAP content creation and evaluation, some of them focusing on separate particular components and some of them trying to cover the whole specification. Fortunately, many projects are focused on producing open-source [4] tools and content that is publicly available with all the source code for usage, modifications and redistribution which means that they could be used as part of the *Fedora* GNU/Linux distribution strictly requiring all its components to be open-source and publicly available. We have already seen examples of such projects in the section describing the OVAL (2.1) – *OVAL Interpreter* and *jOVAL* – both focused on the evaluation of the OVAL content.

An example of a project that tries to cover all components of the SCAP is the *OpenSCAP* project. Although still being under development it already provides powerful tools to process the SCAP content and creates an important "crossroad" for many other projects focused on particular components. The OpenSCAP per se provides [3]:

- a library that can be used for SCAP content processing and evaluation,

- a scanner that utilizes the library and provides local scanning capabilities,
- a number of XSLT[1] [38] transformations that can be used to transform an XML content to more human-readable HTML format and
- SCAP content intended for testing and experimental purposes.

The library is written in C, but there are bindings for Perl and Python languages, so that it can be used also from these high-level languages. Probably the most important part of the *OpenSCAP* project is the scanner (called **oscap**) which can be used for a wide range of actions done with a SCAP content starting from validation and basic information extraction going through various transformations and ending with complete evaluation. As can be seen in the chapter 6, the Anaconda addon implemented as part of this thesis utilizes the **oscap** tool to perform many operations. All parts of the OpenSCAP project are distributed as source codes that can be compiled and installed to the system. For the Fedora GNU/Linux distribution, Red Hat Enterprise Linux and their derivatives there are also software packages using the RPM[2] format. These packages contain the compiled and linked binaries together with additional files and can be easily installed on a system with the following command:

```
$ yum install openscap openscap-utils openscap-content
```

that also downloads and installs the dependencies (other packages that are needed for **openscap** packages to work properly) which simplifies the installation a lot and allows a proper uninstallation that removes the contents of the packages.

By providing a tool for evaluation of SCAP content and basically all of its components, the *OpenSCAP* project acts like some kind of a "central meeting point" for various projects that focus only on some particular components or separate procedure that can be done with the content. The rest of this chapter is dedicated to brief descriptions of these related projects.

1. Extensible Stylesheet Language Transformations
2. originally Red Hat Package Manager, now RPM Package Manager

3.1 scap-workbench

Although XML is a text format and can be quite easily read and understood not only by computers, but also by humans. But still, the creation of a valid XML file following some specification is not an easy task. Especially when there are many such specifications and every one of them is that complicated as SCAP components are. Also it is not so common and comfortable to use command-line tools (as the **oscap** is) for a work, especially when they produce a lot of output which can be hard to read in the textual interface. To facilitate the creation and modifications of the SCAP content and to provide a better user experience when doing evaluation the *scap-workbench* project [40] was started with the goal to create a GUI tool that could be used for scanning, tailoring and editing SCAP content together with validation. While still being under development, the tool with the same name as the whole project already provides a basic functionality in all areas it is has been being created for. It is written in the Python programming language and uses the *openscap* library (the one created as part of the *OpenSCAP* project) through its Python bindings. As a graphical framework for the GUI it uses the *Gtk3* library [64].

3.2 Security State

When applying some SCAP content on a system, the first goal is to get the system to the correct configuration and state so that it follows the rules defined in the content (or a particular profile, group of rules, etc.). However, once achieved, this only means that the system is in the desired state at the moment when the rules are being evaluated. Since the overall (long-term) goal is to improve the system's configuration and state to make it is less vulnerable to various attacks and mitigate the risks, there needs to be a mechanism created to keep the system in compliance with the content and if something gets wrong, to fix it as soon as possible and in an automatic way. This is the area the *Security State* [41] (or *SecState*) focuses on. It provides a tool called **secstate**, that uses the *openscap* library[3] to

3. again by its Python bindings because it is also written in Python

do evaluation of the SCAP content on a system. The evaluation is something that can be done also with the **oscap** tool, but the **oscap** is stateless in a way that it doesn't keep any information between the runs. So what the **secstate** allows on top of the **oscap** tool's capabilities is some long-term management of the content on a particular system. It allows the administrator to import SCAP content (registered by the **secstate**), then to choose profiles, groups, rules, etc. and in the end run the audit (evaluation) and remediation. Since it keeps the state (the imported content together with the selections) the audit and remediation can be triggered periodically with various ways of reporting issues, misconfiguration and other problems (if any) and thus keeping the system in compliance with the policies given by the chosen SCAP content.

3.3 SCC

An important component of the SCAP is the OVAL language as it allows definitions of automated checks used in various places of the SCAP content. However, creation of such checks is hard due to the complexity of the OVAL language and also its model relying on usage of indices and referencing elements that are placed in a completely different parts of the document. Another problem is that the XML format brings in a lot of "noise" (enclosing characters without any actual meaning) and with proper indentation (and line breaking) that improves readability a typical OVAL content is both very wide and long. To address these issues the *SCC* project [42] has been started with the goal to create a simpler, better-readable and clearer language that could be used to define checks. The language is called *SC* and instead of using XML it uses a very simple and clear syntax similar to the *JavaScript Object Notation* (JSON) with curly brackets enclosing complex values and the equal sign creating key-value pairs. It allows the content creator to omit the long identifiers required by the OVAL and supports locality in a way that the related definitions (tests together with objects and their states) can be written in one place next to each other. Thus the creation and reading (e.g. when doing a review) of check definitions and is easier, but in contrast to the OVAL, the SC is not a standard nor a part of the SCAP

which means that there is no support for it in the tools doing the evaluation of the content. To solve this problem, there is the **scc** (SC compiler) tool, that takes an SC content and produces a valid OVAL content that can be used by the standard tools supporting OVAL. To allow that, the SC language provides a clear mapping to the OVAL elements by using the same keywords for them.

3.4 SCE Community Content

Another problem resulting from the complexity of the OVAL language is that it is almost impossible for the system administrators to convert their scripts they have been using for a long time to check system's state and configuration all at once to OVAL checks which would allow them to have the rules and checks defined as a proper SCAP content instead of their own non-standardized and custom formats that are understood only by their custom tools. For this reason the developers of the OpenSCAP project came with the extension of the *openscap* library (and thus also the **oscap** tool) called *Script Check Engine* (SCE) [43] that allows checks referenced in the SCAP content to be any executables (i.e. scripts with the shebang[4] or even binaries). When the executable is run during the evaluation process, two special environment variables are added to its scope, `$XCCDF_RESULT_PASS` and `$XCCDF_RESULT_FAIL`, with the values that should be used as exit codes when the check performed by the executable passes or fails, respectively. In order to provide additional information about the results of the check, it is possible to include the standard output and standard error output gathered when running the executable to the results.

The SCE extension is used by the *SCE Community Content* project, that focuses on creation of a publicly available and easily understandable SCAP content that could be used on GNU/Linux systems. Either directly or after customization that is, due to the usage of easily understandable scripts instead of complex OVAL language definitions, much easier when compared to the SCAP content using the

4. special line in the beginning of the script with the following format: `#!/path/to/the/interpret` that points the OS to the tool which the script should be interpreted with

OVAL checks. The main platform targeted by the *SCE Community Content* project [44] is the Fedora GNU/Linux distribution as it is a platform both OpenSCAP and SCE Community Content projects are being developed on and also a platform that contains new versions of packages (**openscap*** packages included) soon after they are released by the developers (so called *upstream*). The goal is to provide a default content possibly for all installations of the Fedora distribution and a part of the work on this thesis is to participate in the project mainly with the rules that should be considered in the pre-installation phase (as will be discussed in the chapter 4.

3.5 Aqueduct

Development of a comprehensive SCAP content is a hard task that takes a lot of work of many people. Thus it is quite often that only rules and their checks are included in the publicly available content with the fix elements missing. While it is enough in many cases where the manual changes that would take the system into a compliant state are possible, it is a problem in case of a large number of installations that are supposed to follow certain rules. An automated remediation is crucial in such cases, but it requires the fix elements to be added to the SCAP content. The *Aqueduct* project tries to target this issue on the *RHEL 5* by developing multiple sets of fixes for many security guidelines provided by various agencies [45]. The fixes are available in the form of *Bourne Again Shell* (bash) scripts or so called *Puppet*[5] manifests and are supposed to be run independently on the content evaluation (e.g. with the **oscap** tool) which should then follow to check if the system's state and configuration actually was changed in the right way.

3.6 SCAP Security Guide

As it was mentioned in the previous section, development of a SCAP content is hard task that takes a lot of work and time. One of the results of this is that the SCAP content is usually created a long time

5. Puppet is a system for remote management [46]

after a new (major) version of a system is released. And since the main producers of security content are various agencies that often use the older and thoroughly tested versions of systems rather than newer and not so reliable versions, the recent versions of systems are not covered by a security content at all or only a little bit. These facts are reflected also on the state of the official security guidelines for the RHEL were almost all content targets the RHEL 5 (initial version released in 2007) and only a little content is available for the RHEL 6 (initial version released in 2010). That is why the *SCAP Security Guide* (SSG) project [47] was launched with a goal to provide SCAP based security guidelines for the RHEL 6 and also *JBoss Enterprise Application Server 5* (JBoss EAP 5) that could be used by the customers using those tools and that could serve as a base for the official content provided by the agencies that produce the official content. It utilizes a lot of general and often vague recommendations and bridges the gap between them and the real application that has to be based on strict and well-defined rules and actions. By bridging this gap it allows automated evaluation of the recommendations e.g. with the **oscap** tool. In fact, the SSG content is often used as a testing content for the products of the OpenSCAP and related projects. For example the Aqueduct project provides fixes for the rules that can be found in the SSG and the HTML version of the SSG content is generated with the **oscap** tool [39].

Chapter 4

Policy Compliance Configuration

A chosen set of rules (e.g. from an XCCDF profile) defines a policy on how a system should be set up and for the state of following such rules or not the term *compliance* is used. When combined together, we can talk about the *policy compliance* and a general goal is to configure a particular system to be compliant with a particular policy, given by a SCAP content in our case.

In order to do that, it is needed to:

1. get the content,
2. evaluate it on the system,
3. remediate the system and
4. check the results of another run of the evaluation,

where the last step is particularly important because the remediation may not be 100% successful or there might be rules that are missing usable[1] `fix` elements either because they were not created or it is impossible to create them for a general case.

As for the first step, there are many organizations and projects providing SCAP content either for free and publicly or under some restrictions. One of the organizations providing such content under restrictions is the *Center for Internet Security* (CIS) [49] and its *Security Benchmarks division*. Their benchmarks are being recommended as a de facto security standards and are required by some of the laws and regulations (e.g. Federal Information Security Management Act [48]). Also they claim that:

1. i.e. with a matching `@platform` attribute and other values

> "Configuring IT systems in compliance with these Benchmarks has been shown to eliminate 80-95 % of known security vulnerabilities." [50]

An example of a content provided publicly and for free is the *Security Technical Implementation Guide* (STIG) [51] created and distributed by the *Defense Information Systems Agency* (DISA) which is together with the *NSA guides* [52] a standard for the Department of Defence Information Assurance. Another examples of the available content are *The Health Insurance Portability and Accountability Act* [53] targeting health information privacy, *Payment Card Industry Data Security Standard* (PCI-DSS) [54] targeting cardholder information protection and *The United States Government Configuration Baseline* (USGCB) [55] the purpose of which is *"to create security configuration baselines for Information Technology products widely deployed across the federal agencies"* [55].

As has been described in the chapter 2, the SCAP version 1.2 comes with a definition of so called *data stream collections* and *data streams*. But since a lot of the content has been created before the most recent version of the SCAP specification has been released and widely adopted, documents with the content may be provided in variety of formats that can be divided into two types – a single XML document with the data streams holding all content from all SCAP components or multiple separate documents each for a particular component (usually a CPE dictionary, OVAL checks, etc.). To facilitate the fetching and work with the content separated in multiple documents, it is common to create a single file carrying all of the documents. A typical format used for this is the ZIP archive, but another types of archives, such as TAR archives with various compression algorithms[2], can be used. In addition to that, in GNU/Linux (or UNIX) environment the documents can be provided as a software package. On Fedora, RHEL and their derivatives it would be the RPM packaging system that on top of the archives' abilities allows also integrity and signature checks during installation (fetching content) and also a subsequent integrity checks of the installed content compared to the the original versions provided by the package. Many of the tools provide a way to use directly the archived content, some other needs

2. that are often very efficient due to the XML nature of the content

to extract the archive in advance manually which can complicate the work in some cases, e.g. when the content is evaluated on a remote system.

Steps 2. and 3. are evaluation and remediation. Both can be done with a various tools, e.g. the ones described in the chapter 2. But as it was already briefly described in the introduction, there is a question when to apply the content (do the evaluation and remediation) to a system. First idea may be to do it right after the system is installed, because it leaves basically no time for an attacker to interact with the possibly misconfigured system and on the other hand the system already exists and can be modified. But with a closer look at some rules it can be found that it is extremely hard and almost impossible to fix the system's configuration after the installation. For example this rule can be found in the *Guide to the Secure Configuration of Red Hat Enterprise Linux 5*:

> "**Create Separate Partition or Logical Volume for /tmp**
>
> The /tmp directory is a world-writable directory used for temporary file storage. Ensure that it has its own partition or logical volume." [8]

While it is possible to create a separate partition or logical volume for the **/tmp** directory even after when the system is already installed, it is much easier (and from the perspective of reliability and availability of the server also safer) to create and configure such partition or logical volume during the installation process. But in the installation process, the system is not yet installed and it is impossible to evaluate any rules on it as the configuration of the resulting system is not known and the same, of course, applies to the remediation of such system. It is obvious that proper evaluation and remediation cannot be done all at once as well as it cannot be done (even periodically) only on the installed system.

The best choice seems to be a combination of both approaches – making sure that the to be installed system will follow the rules the requirements of which are hard to be fulfilled with post-installation changes and running the content evaluation and remediation right after the installation, possibly even before the first boot of the newly installed system. The main problem of this approach is that unless

some additional data is provided by the SCAP content, the rules that need to be evaluated in the pre-installation phase are almost impossible to identify in an automated way. Because it is really hard for a computer to decide about that from a title or description of a rule, any automated identification of such rules would need to implement the OVAL interpretation and there would have to be some consensus or standard that would define how the OVAL checks would be constructed since there are often multiple ways how to write an OVAL check for a particular rule. With the OVAL relying on its extensions that are still evolving and appearing, anything like that would be practically impossible. So instead of endlessly developing such identification system and teaching content creators how to write OVAL checks in a specific way, it is much better to utilize the extensibility of SCAP specifications (namely the XCCDF) by adding and processing additional data in the SCAP content.

As it was described in the second chapter's section focused on the XCCDF (2.6, the `Rule` element may have multiple `fix` elements each for a different interpretation system defined by the value of the `@system` attribute. And while there are some predefined values of the `@system` attribute, the specification allows using any URNs that are understood by the content processing tool. If the value is understood and the interpret identified by it is available, the content of the fix element is passed to the interpret. Or the processing tool can have the ability to extract contents of all fix elements from a given profile[3] intended for a given interpret. Thus there could be fix elements added to the rules that need to be considered in the pre-installation phase with a special format and a special value of their `@system` attribute that could be pulled by the OS installer and interpreted in a way that it makes sure those rules will be followed once the system is installed.

The special value of the `@system` attribute needs to mark the fix element as being intended for the installer of a particular OS so that the installer can pull the contents of the fix elements it is able to process and understand. In the same time it should be a simple and mnemonic value to facilitate the creation and identification of such additional fix elements by the content creators. The special format of

3. identified e.g. by the data stream ID, checklist ID and profile ID

the contents of those fix elements needs to be both human-readable and computer-readable, because on one hand the content creators will be writing it and on the other hand the OS installer will be processing it to get it's semantic meaning. Since there are no restrictions for the contents of the fix elements given by the XCCDF specification[4] [37], there are many possible ways how the format could look like. However, when designing a format for some kind of data, the first step should be the identification of what the data will represent. In our case it means identification of the rules that need to be considered (i.e. evaluated) in the pre-installation phase and their attributes that will have to be passed to the OS installer. By going through various security recommendations and guidelines (provided as a SCAP content or in a different way) it can be found that there are the following types of such rules:

- rules concerning partitions or logical volumes (their existence or options they are mounted with),
- rules defining characteristics of passwords (mainly the administrator's password),
- rules concerning the boot loader[5] configuration (in particular the existence of a password protecting it against malicious modifications) and
- rules defining which packages should be or should not be installed on the system.

The last type of rules is somehow optional because it is usually easy to remove the package from the installed system, but not installing it shortens the time of the installation process and prevents any problems and configuration issues that can be caused by the mere installation of the package (if it doesn't clean up after itself correctly when being uninstalled). Thus the format of the fix elements specific for

4. other than that it needs to be valid piece of an XML document which e.g. means it has to be a text

5. which is a small program at a predefined position on a hard drive that is started when the computer is turned on and is responsible for loading operating system and giving the control to it

the pre-installation phase should provide (at least) a way to represent those types of rules.

Chapter 5

Anaconda Installer

The previous chapter describes what the steps of the application of a SCAP content on a particular system are and points out that in order to make system's configuration compliant with a set of rules, it is useful and sometimes necessary to evaluate the rules twice during the installation of the system – in the pre-installation phase and in the post-installation phase. Implementation of such approach needs a modification or extension of the OS installer that drives the installation and thus may run the right checks and actions in the right time. It is practically impossible to modify or extend a closed-source OS installer and OS installers hardly ever provide an API[1] for extensions as they are very complex and single purpose programs. Fortunately, the installer of the Fedora and RHEL GNU/Linux distributions, the *Anaconda* installer [56], is an open-source project and recently a support for third-party extensions (called *addons*) has been added to it so the approach suggested in the previous chapter can be implemented. The rest of this chapter is dedicated to the descriptions of the Anaconda installer, it's specifics and the addon API it provides.

> "The Anaconda is the operating system installer (OS) used in Fedora, RHEL and their derivatives. From a closer look it is a set of Python modules and scripts together with some additional files like Gtk widgets (written in C), systemd units and dracut libraries. Altogether they form a tool that allows user to set parameters of the resulting (target) system and then set such system up on a machine." [57]

1. Application Programming Interface

As the basic characterization above describes, the Anaconda installer has multiple components and the most important and most complex one is the set of Python modules and scripts that tie all the other components together. The overall installation process done by the Anaconda starts with booting to the installer (e.g. from a DVD, USB stick or over network) which includes hardware detection and initialization. Then the Anaconda's main process is started and based on the installation method (will be described later) and interaction with the user starts. Once the user gets all configuration options to the desired state and starts the actual OS installation, the following actions are run [57]:

- installation destination preparation (usually disk partitioning),
- software packages and data installation,
- boot loader installation and configuration and
- configuration of the newly installed system.

5.1 Installation Modes

There are three different modes of the installation or more precisely three different ways how configuration options for the target system (and in some cases also for the installation process) may be entered. These modes differ in user-friendliness and numbers of configuration options they provide. The most often used one is the Graphical User Interface (GUI) mode that covers all common use cases and tries to be clear enough even for non-advanced users. And since both Fedora and RHEL are quite often installed also on machines without monitors or displays (so called *headless machines*), the GUI mode can also be run via a *Virtual Network Computing* (VNC) session [58].

However, some systems don't support any graphical output (or sometimes the users don't like GUIs). For such cases, the Anaconda installer has also the text user interface (TUI) mode that can be run either directly on a system's display or remotely via serial console. The TUI mode operates basically in the same way as a monochromatic line printer, because it is intended to support even a bit exotic

hardware platforms[2] with consoles that don't support colors and cursor moving. The result of these restrictions is that it is generally not much user-friendly and it doesn't provide as much functionality (in the number of exposed configuration options) as the GUI mode and some of the options has to be set by the command-line options passed to the Anaconda.

The last, but definitely not least, installation mode that the Anaconda provides is the so called *kickstart* mode where the installation is driven by a simple text file with command-like syntax referred as a *kickstart file*. If a kickstart file contains all the configuration options that need to be set to install a working system, the whole installation runs automatically. If something is missing, the automatic processing stops, the installer's user interface[3] (UI) asks the user for the missing input and then the installation continues as if no kickstart file was used (except for the values from the kickstart file being used if not changed by the user in the UI). This means that one can use a partial kickstart file with a few options specified and the others left for an interactive completion. In contrast to the TUI mode that provides a way to set only the most important configuration options, the kickstart mode is the only full-featured mode. In other words, everything that Anaconda allows to be configured can be configured in a kickstart file and a common practice is that everything is first supported by the kickstart mode and some of the features are then exposed in the GUI or TUI while trying to keep the user interfaces clear and understandable. Being based on a text file with a defined syntax and providing all configuration options that are implemented, the kickstart mode is considered to be something like and expert mode. Nevertheless, if one starts with some basic backbone of a kickstart file and only does a few changes and additions, it is quite easy to automatize the installation process which is very useful especially when installing a lot of systems with the same or similar configuration. As a good input for such tailoring[4] is a kickstart file that is created as an output of GUI or TUI installation.

---

2. e.g. *s390* and *s390x*
3. graphical or textual, depending on which one is chosen with a command-line option or in the kickstart file
4. if we borrow the term from the SCAP area

5.2 Architecture

The Anaconda installer is a very complex tool and probably the most advanced commonly used OS installer. It allows configuration of a big number[5] of aspects of the target system and supports installation from various types of sources (e.g. DVD, CDs, USB stick, from network over HTTP, FTP or NFS) together with installation to various types of storage devices such as local disks with standard partitions, logical volumes and software RAIDs or non-local storage devices attached via iSCSI, FCoE or ZFCP[6] all with preexisting or newly created filesystems of various types.

Having such a complex tool as a monolithic giant would result in impossible maintainability and thus the Anaconda installer is divided into many parts itself (as was mentioned in the quoted description in the beginning of this chapter) and uses multiple external libraries and tools (some of them being integral part of the Anaconda in the past). Major components taking place in the installation process are these Python packages:

- **pykickstart** – providing parsing and storing in-memory representation of a kickstart file,
- **yum** – providing downloading and installation of RPM software packages from packages repositories,
- **blivet** – not so longer ago being part of the Anaconda as **pyanaconda.storage** and thus providing storage devices manipulation and
- **pyanaconda** – gluing all the other components together and providing some additional functionality like language, keyboard, timezone and network configuration.

Then, ideally not participating in the installation process, there is also the **python-meh** package which provides an exception handler that

5. the kickstart mode understands more than 50 types of commands specified in a kickstart file and vast majority of the commands has multiple options that can be specified with them
6. even a brief explanation of these acronyms is beyond the scope of this thesis

can gather a lot of information from the crashed installation and prepare data for the *libreport* library which then allows user to create a bug report.

From a different angle of view the Anaconda installer is a multi-thread application combining modular and object-oriented programming (OOP) performing a transactional and data-driven installation of an operating system. It needs to be multi-thread because many actions it runs take seconds or tens of seconds that would, by running all in a single thread, often render the UI unresponsive for a notable amount of time. Moreover, many actions can be effectively run in parallel[7] since e.g. probing storage devices is bounded by slow I/O operations whereas downloading packages is bounded by network bandwidth both not necessarily blocking the UI interaction with the user. Being written in Python, the Anaconda installer (as well as the other major libraries and packages used by it) utilizes the language's multi-paradigm nature that allows writing code not strictly in one programming paradigm. So for example all the UI elements are objects (i.e. instances of classes as known in the OOP), but on the other hand code that writes out keyboard configuration to the newly installed system is a function defined in the *pyanaconda.keyboard* module.

The best way to reason about why we can say that the installation process is transactional and data-driven is to describe the life cycle of the data. At the beginning, if a kickstart file is passed to the Anaconda, the file is processed (parsed and validated) by the **pykickstart** package to create and in-memory tree-like representation of it (which we will refer as the *kickstart data* in the following text). If no kickstart file is given, a default tree-like structure is created with default values (`None`, empty strings, zeros or some reasonable defaults like `en_US` as the preset language) in its leaves. Then, unless all the necessary configuration options where specified in the kickstart file, the interaction with the user follows during which the values in the kickstart data are updated based on user's input. This phase ends with the user starting the actual installation process[8] (by clicking the

7. even with Python having the Global Interpreter Lock that allows Python code to be executed only in one thread at a time [59]

8. the starting is done automatically if all required configuration options are set

right button or entering the right string in the text mode) and until that happens, the underlying machine and its system is not modified in any way. Thus the user is free to do any changes in the desired configuration as well as going back and forth with the decisions and only after giving a clear command the actual installation transaction starts to be performed. However, what is different compared to common transactions known e.g. from database systems is, that the installation transaction cannot be reverted if something goes wrong. The actual installation process consists of the steps mentioned at the beginning of this chapter and each of the steps is driven by the values stored in the kickstart data. But first of all, the runtime environment of the installation is set up to a desired state by running the `setup` methods of the nodes in the kickstart data tree. Then the installation destination is prepared (usually by disk partitioning and creation of filesystems) and a set of software packages is installed to it together with a boot loader (if not requested otherwise by the particular kickstart data value). Finally the newly installed system is configured by calling the `execute` methods of the nodes in the kickstart data. These methods usually take the values stored in their subtrees and change the target system's configuration to reflect those values. In the end the kickstart data tree is transformed (back) to a kickstart file that is written out to a predefined directory in the newly installed system. Such a file can be used to install another system with the same configuration (or reinstall the system if it breaks) or, as mentioned above, can be used for tailoring and creation of another kickstart file. So the contents of the kickstart data tree really drive the installation procedure and much of the work is done by calling nodes' methods.

5.3 Addons Support

Although the Anaconda installer is a very complex tool and allows a lot of configuration options to be set in the installation process, the resulting system has much more aspects that can be configured or set up in many ways. On one hand the runtime environment of the installation is quite specific and usually it is easier to do configu-

in the kickstart file

ration that can be done post-install after the first boot of the newly installed system (due to common tools working much better in the standard runtime environment), but on the other hand once something can be done in the installation process by the installer, it can be automatically reproduced on arbitrary number of machines via kickstart installation. Moreover such settings can be set on a pre-installed system (in case of so called *OEM*[9] *installations*) leaving the post-installation configuration for the first boot and thus the end customer. So it may be potentially very useful to extend the OS installer as much as possible, but more functionality means more lines of code and greater complexity which results in more bugs and worse maintainability that is a real problem for any such critical component. As a trade off the Anaconda provides a support for extensions called *addons*. The addons can be created and maintained by the experts in the area they cover and can be added to or removed from the installation environment based on their reliability and stability or on the target user group of the distribution or a *spin*[10].

Since a typical addon will add something to the installation UI, before we describe how such Anaconda addon can be created and how the API it has to implement looks, we first need to understand the general model used for the Anaconda's user interfaces. While traditionally tools like OS installers use the so called *wizard* model that guides the user through groups of configuration options one after another (with a possibility to go back or not), the Anaconda installer (for many reasons) implements a model usually referred as the *hub&spoke model* [57] . The figure 5.1 depicts the basic structure and possible transitions between the screens. The central points of the model are *hubs*, that provide access to the screens called *spokes* (screens number 2 to 13 in the figure) in an arbitrary order and arbitrary number of visitations usually together with descriptions or summaries for the spokes. The somehow special screens number 1 and 14 are so called *standalone spokes* that enforce the order of visiting. At the first glance those screens don't fit into the model, but they are very practical because very often some actions need to be done

9. Original Equipment Manufacturer
10. a spin is a modified version of a distribution with customized configuration, different set of software packages or both

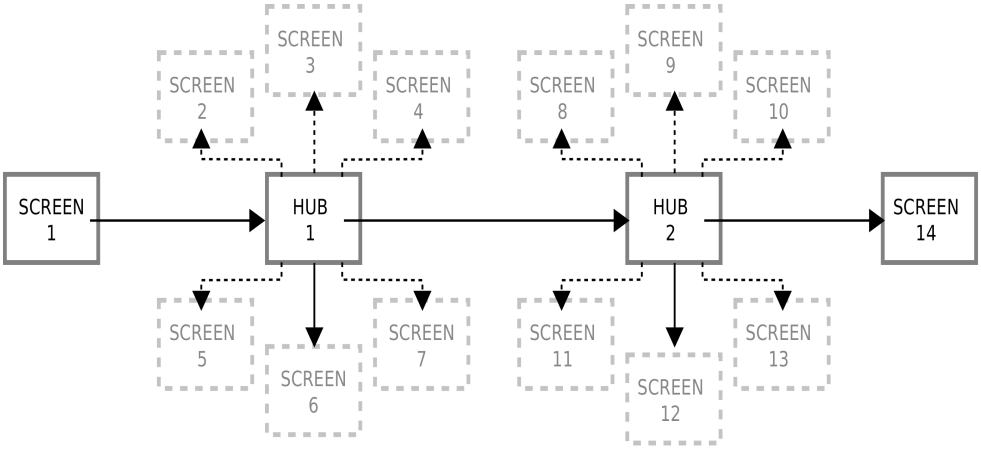

Figure 5.1: hub&spoke model diagram [57]

before (or after) some other actions (that are then accessible from the hubs). An example of such action is the language selection which is implemented as a standalone spoke in the Anaconda installer. The spokes can be grouped together by the hub they are reachable from and in addition to that the hub can group together spokes from a particular category by placing them (or more precisely their accessors) next to each other.

By the time of the work on this thesis being started there had been no Anaconda addon implemented and used. Actually, there had been no documentation of the API for addons and no referential code. To improve that state, the development of the *Anaconda Addon Development Guide* [57] was started as part of this thesis together with a trivial but well-commented *Hello world addon* [60] which could be used as a referential code or a backbone for a different, more complex and useful addon. The addon development guide is still in a draft state, but it is expected to be reviewed by the documentation editors and become a part of the official documentation for the Fedora GNU/Linux distribution.

An addon has to be a valid Python package (i.e. a directory with the **__init__.py** file) and its subpackages (subdirectories) has to contain modules with classes inherited from a specified set of classes reflecting the installation modes. The name of the addon's top-level package should start with a prefix of the project or organization the

addon comes from. So, for example the OSCAP addon described in the next chapter uses the name **org_fedora_oscap** because it was created as part of the Fedora project (the underscores replace the dots to form a valid Python identifier). The subpackages with code for the different installation modes have the predefined names – **ks**, **gui** and **tui**, for the kickstart, GUI and TUI modes respectively. As it was mentioned in the section focused on the architecture (5.2), everything in the installer should be supported first in the kickstart mode. Addons are no exceptions and without having the kickstart support implemented, they wouldn't have their part (a subtree) of the kickstart data available during the installation process. The **ks** subpackage has to contain a Python module (of an arbitrary name) with a definition of a class inherited from the *AddonData* class from the *pyanaconda.addons* module. An instance of such inherited class is then responsible for parsing lines from the kickstart file and storing the parsed values in the in-memory attributes. Then it should have the `setup` and `execute` methods as described in the previous section and the `__str__` method so that the in-memory attributes can be returned as a string and written out to the resulting kickstart file at the end of the installation process. The **gui** subpackage may have one or more subpackages each for a different group of UI elements – **spokes**, **hubs** and **categories** – where the last one is for defining a category of spokes, that should appear on the hub next to each other. All of these subpackages (if existing) should then contain modules with classes inherited from the special classes defined by the API. However, for addons only the standard spokes are recommended as they fit best in the hub&spoke model without disturbing the user experience (UX) much. The *NormalSpoke* is the special class defined by the API for the standard spokes and the addon's class inherited from it should implement the methods for initialization of the spoke, refreshing when visited, getting status of the spoke and information about its readiness and completeness and for applying changes[11] done by the user on the spoke. Moreover the inherited class should have a set of predefined attributes defining the spoke's title, icon, the file with the UI definitions[12], etc. [57]. The **tui** sub-

11. storing them in kickstart data and modifying the runtime environment's state
12. created in the Glade [61] designer

package then follows similar rules as the **gui** subpackage, only the categories of spokes are implemented in a different way – as string identifiers specified by the spokes, not special classes.

The whole package with the addon should be then placed to the installation environment to the */usr/share/anaconda/addons* directory. When the Anaconda installer starts, it iterates over the directories found there and searches for the classes inherited from one of the special classes defined by the API. This way all the addons' data handlers, spokes, etc. are collected and instantiated.

Chapter 6

OSCAP Anaconda Addon

Chapters 4 and 5 described, among the other things, that it is a good strategy to do first run of the content evaluation and remediation even in the pre-installation phase, which means that the OS installer has to be extended, and that the Anaconda installer allows such extension to be created and deployed. This chapter is dedicated to the description of the actual implementation of the extension – the *OSCAP Anaconda addon*. The name is derived from the *OpenSCAP* project's name because the addon utilizes the *openscap* library and the **oscap** tool a lot. In spite of the fact that the addon is still being developed, it already provides basic, but useful, functionality.

6.1 Architecture

As was described in the section focused on the addons support in the Anaconda installer, the only supported programming language for the addons is Python and an addon has to be a Python package with the name prefixed by the organization or project it was created under. The name of the OSCAP addon's package was thus set to the *org_fedora_oscap* because it is being created under the Fedora project. Similarly to the Anaconda, the OSCAP addon is combining the OOP and modular programming paradigms and contains a lot of functions next to definitions of classes and methods. The top-level package has two subpackages, **ks** and **gui** (with the purpose described in the section 5.3 and four helper modules providing useful functionality for working with the content. The **tui** subpackage is not implemented for now, because the GUI has higher priority (even in the Anaconda installer) and the TUI will be implemented later if it turns out there are users that cannot use the GUI mode. The four

modules in the top-level package are: *common*, *content_handling*, *data_fetch*, *rule_handling* and *utils*. The simple *utils* module provides utility functions that are useful but not provided by standard Python libraries/modules. And while the *content_handling* module defines classes for getting information (mainly lists of items) from the data stream collections, data streams and standalone XCCDF benchmarks, the *rule_handling* module provides a class for handling special rules (or more precisely the contents of fix elements) for the pre-installation phase. The *data_fetch* defines a function for fetching data from various sources and the *common* module provides constants and functions that are used by the modules for installation modes but are not specific any particular mode.

6.2 SCAP content processing

The OSCAP Anaconda addon implements the approach of SCAP content application suggested in the chapter Policy compliance configuration. In the beginning of the installation process the values from the kickstart file are read and parsed to find out from where the content should be fetched and which XCCDF profile (and in case of the data stream collection from which data stream and checklist) should be applied. When the GUI starts (as described in the previous section there is no support for the TUI mode for now), the content is fetched and if it is an archive, also extracted, then the contents of the special fix elements for the pre-installation phase[1] are gathered from the chosen profile and evaluated to check and change the configuration values set by the user in the kickstart file or in the UI. The results of the check and information about the changes are shown on the *SECURITY PROFILE* spoke[2] that is marked as finished or not depending on the results of the check. Also a profile together with data stream and checklist IDs can be changed on the spoke and in the future there may appear a field for entering a content URL (which can be done only in a kickstart file for now). If a configuration that is not in compliance with the chosen profile is detected, changes which

1. having their `@system` attribute set to `urn:redhat:anaconda:pre`
2. more information about the spoke can be found in the section 6.4 describing the **gui** and **ks** subpackages

can be done automatically are made and problems that need to be fixed manually[3] are reported on the spoke. Unless all problems are fixed, the spoke's status is set to *Misconfiguration detected* and start of the actual installation process is blocked. The user can go to the *SECURITY PROFILE* spoke to find out what is the problem with the current configuration and then go to the other spokes and change the configuration or choose a different profile. Once that happens, the installation may proceed and before the actual installation process starts one more check of the compliance is done so that even the potential problems caused by late changes in the configuration are caught. This check is performed also if the GUI is not run (e.g. in case of the TUI mode) so it prevents a system with non-compliant configuration to be installed.

At the end of the installation process, when all packages are installed to the installation destination and the newly installed system's configuration is set up according to the values requested by the user, the fetched (and extracted) content is copied to the target system and evaluation with remediation is run. The results are saved under the root's home directory of the newly installed system for later analysis by the administrator. In case there are requests for showing the results in the UI, an additional spoke for the installation screen or the *Initial Setup* tool[4] (or both) will be implemented. A screen for the Initial Setup would be the preferred way because the results of evaluation right after the installation and during the first boot may differ and the latter are more important and accurate. However, due to the Initial Setup uses the **pyanaconda** package and has the same API for addons, it may be possible to make the spoke universal and suitable for both cases.

3. for example a requested partition cannot be created (or more precisely scheduled to be created) automatically because it is practically impossible to guess the right size for it

4. the tool run during the first boot of the system using the **pyanaconda** package and having the same addons support

6.3 The helper modules

It is a good practice to split the functionality into separate modules, classes and functions with well-defined API, so that the implementations of the functions and classes can be changed liberally as long as the API is preserved and the modules can be tested as separate standalone units without the tests being affected by the rest of the code (so called *unit testing*). The OSCAP addon follows this practice by having the functionality it needs to implement split into four helper modules (apart from the **gui** and **ks** packages required by the Anaconda's API).

The *utils* module provides two simple but very useful functions – `ensure_dir_exists` and `universal_copy`. The first one takes a directory path, checks if such directory exists and if not, creates it. This is a common operation done very often, but, for some reason, the standard Python `os.makedirs` function fails if the directory already exists. Having such utility function defined means less lines of code, smaller risk of bugs caused by writing the code from it again and again and it can be tested by a unit test. The `universal_copy` function provides a universal functionality for copying data, similar to the standard **cp** utility. The *shutil* module available as part of the standard Python distribution provides functions for copying data, but there are separate functions for copying files and directories and if copying directory, the destination directory must not exist. The `universal_copy` function resolves these issues and moreover it works with globs[5].

Although the OSCAP addon uses the **oscap** tool to do content evaluation and remediation, it also needs to process the content itself to get the list of available data streams, checklists and profiles. The *openscap* library and its Python bindings can be used for that, but since it is very low-level library, it is useful to create a module providing a high-level API for things needed by the addon. This is implemented in the *content_handling* module and its *DataStreamHandler* and *BenchmarkHandler* classes. As the name suggests, the former one provides handling of data streams in a data stream collection and has public methods for getting data stream IDs, checklist

5. strings with special characters (*, ?, ...) that can match multiple items

IDs and profiles in a given data stream and checklist respectively – `get_data_streams`, `get_checklists` and `get_profiles`. The much simpler *BenchmarkHandler* class provides only the `profiles` property for getting a list of profiles in the benchmark (an XCCDF document) it is instantiated with. Both classes use the *ProfileInfo* helper class for encapsulation of the profile information (currently the ID, title and description).

When a SCAP content is evaluated in the pre-installation phase, the contents of the special fix elements have to be pulled from the content, parsed and evaluated. While getting the contents of the fix elements can be done with the **oscap** tool, the results (the special rules) has to be parsed and evaluated by the addon itself. Since it is not trivial and basically the most critical part of the addon that needs to be tested well, a separate module – *rule_handling* – has been created for it. As it was described in the chapter 4, the format for the contents of the special fix elements has to be able to express a set of rules and in the same time has to be clear and simple so that the content creators can easily add such data next to the standard fix elements. As a format meeting both these requirements the command-like syntax used also for the kickstart files has been chosen. It is a very simple and by being very similar to the kickstart files, it is familiar to people writing such files for automated installations. Currently, the following rules are supported:

- `part MOUNTPOINT` specifying that a separate partition or logical volume for some mount point has to exist, optionally with a `--mountoptions=OPT1,OPT2,...` option specifying the mount options that should be used for the mount point,

- `passwd --minlen=NUMBER` specifying the minimal length of user passwords,

- `package --add=PACKAGE --remove=PACKAGE` specifying which packages should or should not be installed, respectively, and

- `bootloader --passwd` specifying that the boot loader's configuration during boot should be protected with a password.

All these rules (or commands) are expected to be extended with additional options as more types of rules that need to be handled in the pre-installation phase are identified.

The *rule_handling* module exports[6] only the *RuleData* class that can be used for parsing, storing and evaluating the rules listed above. It has three public methods[7] – `new_rule`, `eval_rules` and `revert_changes` – that can be used, in the same order, for adding (parsing and storing) new rules, evaluating rules by doing changes to the installer's runtime configuration and reporting which changes have been done or need to be done and for reverting changes done by previous evaluations (which is needed when switching between profiles). These methods use a couple of private methods and instances of non-exported classes (child object) for the types of rules listed above. The evaluation (reverting changes) is then done recursively, by doing evaluation (reverting changes) of the child objects.

The *common* module provides constants and functions that are used by the packages and modules for installation modes but are not specific to a particular mode. For example there are constants defining the paths of the files and directories containing the content in both pre-installation and post-installation phases. Then there is a function for getting special rules (or more precisely contents of the fix elements) for the pre-installation phase by using the **oscap** tool – `get_fix_rules_pre` – which takes a profile ID, path to the file with a SCAP content and optionally data stream ID and checklist (XCCDF) ID (in case of the content in the data stream collection) and returns the concatenated contents of the special fix elements for the rules selected by the given profile. The `run_oscap_remediate` takes the same arguments plus one more that can be used to specify in which root[8] it should run the evaluation of the SCAP content and remediation by using the **oscap** tool. The `wait_and_fetch_net_data` function can be used for waiting for network connection and fetching data. It is supposed to be called in a separate thread because otherwise it would block the caller potentially for a long time. As

6. in its __all__ variable
7. the Python language doesn't have any keywords for access modifiers and classes, methods, functions, etc. that should be considered private are just prefixed with one or more underscores
8. the top-level directory of a system's directory tree

it was described in the chapter 4 focused on the policy compliance configuration, SCAP content can be provided as an archive. To allow handling of such data, there is the `extract_data` function that extracts the data from a given archive to a given output directory. The type of the archive is determined from the archive's file name. Optionally, the function can also check if some given file is included in the archive. Apart from these functions and constants the *common* module also defines the *RuleMessage* class (as a named tuple) and three message types that can be used for passing information about the pre-installation rules evaluation and remediation (done e.g. by a *RuleData* instance).

Last but not least there is the *data_fetch* module, that exports only a single function, `fetch_data`, and contains multiple constants and a helper function needed for its implementation. For now, the content data can be fetched only from network via the HTTP or HTTPS[9] protocols, but in the future some other protocols or types of sources (such as a USB stick) are expected to be supported. All by the universal `fetch_data` function that will use the non-exported functions for specific sources or protocols. Although the SCAP defines ways how to check authenticity of the content, it may be sometimes enough and much easier to just check the authenticity of the server from which the content is downloaded by verifying its SSL certificate. For cases where such server doesn't have a certificate from a well-known and trusted certification authority, the `fetch_data` function has, among the others, an argument that allows passing a path to a file containing a certificate chain in the Privacy Enhanced Mail (PEM) [63] format.

6.4 Subpackages for installation modes

The helper modules described in the previous section provide a core functionality needed to process the SCAP content from fetching it through pulling information from it to evaluating it on a newly installed system with remediation. However, to become useful, all these pieces need to be glued together and hooked up to the Anaconda installer. This is what the **ks** and **gui** subpackages do by implement-

9. HTTP over Secure Sockets Layer (SSL) [62]

ing the classes required by the Anaconda's API for addons. As it is common, the GUI code is much bigger and more complex than the code handling the text-based user input even tough it utilizes the helper modules and as well as the kickstart support code. Thus we will start with a description of the **ks** subpackage, or more precisely, its only module – *oscap*. This module exports the *OSCAPdata* class inherited from the *AddonData* class from the **pyanaconda.addons** module as the API requires. The *OSCAPdata* class provides methods for parsing and storing the contents of the kickstart file and applying such parsed values on both the installation environment and the target system (as was described in the section 5.2 focused on the Anaconda's architecture).

When the Anaconda installer processes a kickstart file, it goes line after line and parses the contents from it by using the **pykickstart** package and its classes. Lines in a kickstart file can have four forms – blank lines, comments starting with the # character, commands with options and arguments (e.g. `part /tmp --size=1024`) or lines opening (closing) sections that start with the % character. The addon support in kickstart files is implemented as special sections starting with the `%addon ADDON_NAME` line and ending with the `%end` line. When the opening line is reached, an instance of the addon's *AddonData* implementation is created[10] and all following lines are passed to its `handle_line` method one after another. At the end of the section, the `finalize` method is called. Then the `setup` and `execute` methods are called as described in the section 5.2. A typical *%addon* section may look like this:

```
%addon org_fedora_oscap
   content-type = archive
   content-url = https://example.com/scap_content.zip
   profile = xccdf_com.example_profile_my_profile
   xccdf-path = xccdf.xml
%end
```

The *OSCAPdata* class implements the methods mentioned above and provides some extra properties and attributes that can be used e.g. by the GUI code. The `handle_line` method is responsible for

10. if no such addon is found an instance of the *AddonData* class itself is created as a placeholder

parsing the key-value pairs[11] and storing them in the internal attributes. Since different actions can be needed for different keys and values, the method just looks at the key and calls the appropriate internal (private) method responsible for parsing and storing the value for that particular key. For now, the following keys are supported – `content-type`, `content-url`, `datastream-id`, `xccdf-id`, `xccdf-path` and `profile` – but more are expected to be added in the future. Values for these keys can be used for specifying which type the content is using (`datastream`, `archive`, etc.), from where it should be fetched, which data stream, checklist should be used (if the `datastream` type is used) or what is the path of the XCCDF benchmark in the archive (if the `archive` type is used) and which profile should be chosen for evaluation and remediation. The `finalize` method then performs some basic checks of the values such as that if the archived content is used, the XCCDF path is specified and so on. If some of the checks fails, an exception is raised and the installation stops. Despite this may look as a too strict policy, it is used by the Anaconda installer with the intention to prevent installation of a system with a non-working or non-requested configuration.

Then there are the `setup` and `execute` methods performing the pre-installation and post-installation changes, respectively. The former one makes sure that the configuration is in compliance with a chosen profile[12] and adds packages needed for the post-installation phase (especially the **openscap** and **openscap-utils** packages) to the list of to be installed packages. The latter one copies the SCAP content to the newly installed system and runs evaluation and remediation on it[13]. On top of these methods required by the API, there are also three properties useful for the UI code providing paths to the fetched and extracted content in the pre-installation and post-installation phases.

The **ks** package may seem to provide all the functionality that is needed for application of SCAP content in both pre-installation and post-installation phases. However, if there is some problem with the configuration that cannot be resolved automatically, the installation

11. which was decided to be a clear and simple format for information that need to be expressed
12. by running the `eval_rules` method of a *RuleData* instance
13. by calling the `run_oscap_remediate` function with a different root specified

just terminates without giving user a chance to fix the configuration and continue. Also if some required information (e.g. the profile) is not given in the kickstart file, the installation terminates. For actions that need an interaction with the user, the **gui** has been implemented. It has two subpackages – **spokes** and **categories** – containing the modules and classes implementing the Anaconda's API for addons' GUI elements. The **categories** subpackage provides only the *security* module exporting only the *SecurityCategory* class. This class simply defines a category titled as *SECURITY*, that is supposed to appear on the *SummaryHub*, the main screen of the Anaconda's user interface.

The **spokes** subpackage contains the *oscap* module exporting the *OSCAPSpoke* class which is inherited from the *NormalSpoke* class and defines a spoke that belongs to the *SecurityCategory* category and thus appears on the *SummaryHub*. Apart from that, the *oscap* module contains three helper functions for setting selections of combo boxes and tree views and getting selections from combo boxes, because these functions are not provided by the graphical elements themselves and are often needed. The Anaconda installer uses the *Gtk3* library [64] for the GUI so the *OSCAPSpoke*, that hooks into the user interface, uses it as well. The look of the spoke is defined in the **oscap.glade** file created with the **Glade** designer [61]. The *AnacondaSpokeWindow* widget is used as the main window because that's what is required for an installer spoke. The *OSCAPSpoke* class has a set of special class attributes defined by the API which tell the Anaconda how to instantiate it and how to show it. The `uiFile` attribute points to the **oscap.glade** file, the `mainWidgetName` attribute's value tells the installer the ID [14] of the spoke's main window widget, the `icon` attribute specifies which icon should be used for the spoke on the hub [15] and the `title` specifies what should appear next to the icon as a name of the spoke as well as the spoke's window title. Figure 6.1 shows how the spoke's accessor looks on the hub. As can be seen, the spoke uses a lock icon (symbolizing hardening) from the same set of icons the Anaconda's spokes use. This way it looks like an integral part of the installer and at first sight suggests what its purpose is. The spoke appears in the SECURITY category with

14. as appears in the **oscap.glade** file
15. in its *SpokeSelector* instance acting as an accessor for the spoke

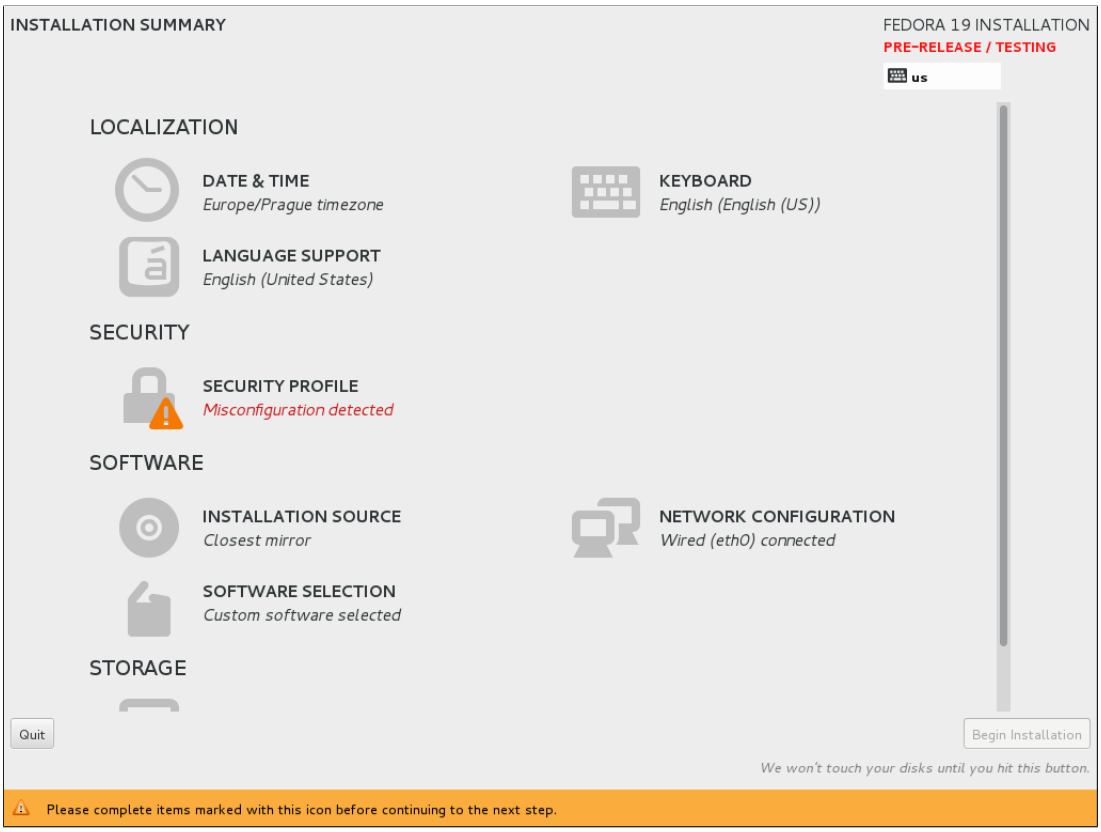

Figure 6.1: OSCAPSpoke on the Summary hub

the SECURITY PROFILE title and provides basic feedback about its state in the status (*"Misconfiguration detected"*). The orange triangle with an exclamation mark means that the spoke is not completed and user needs to take some actions to continue with the installation.

The figure 6.2 shows the spoke's screen as it appears when the accessor shown in the figure 6.1 is clicked or activated by keyboard. At the top there is a header provided by the Anaconda installer[16]. Below the header there are two combo boxes allowing the user to select data stream (the left one) and checklist from the selected data stream (the right one). These combo boxes are hidden if a standalone benchmark (not as part of a data stream collection) is used as a con-

16. or more precisely by its *SpokeWindow* widget

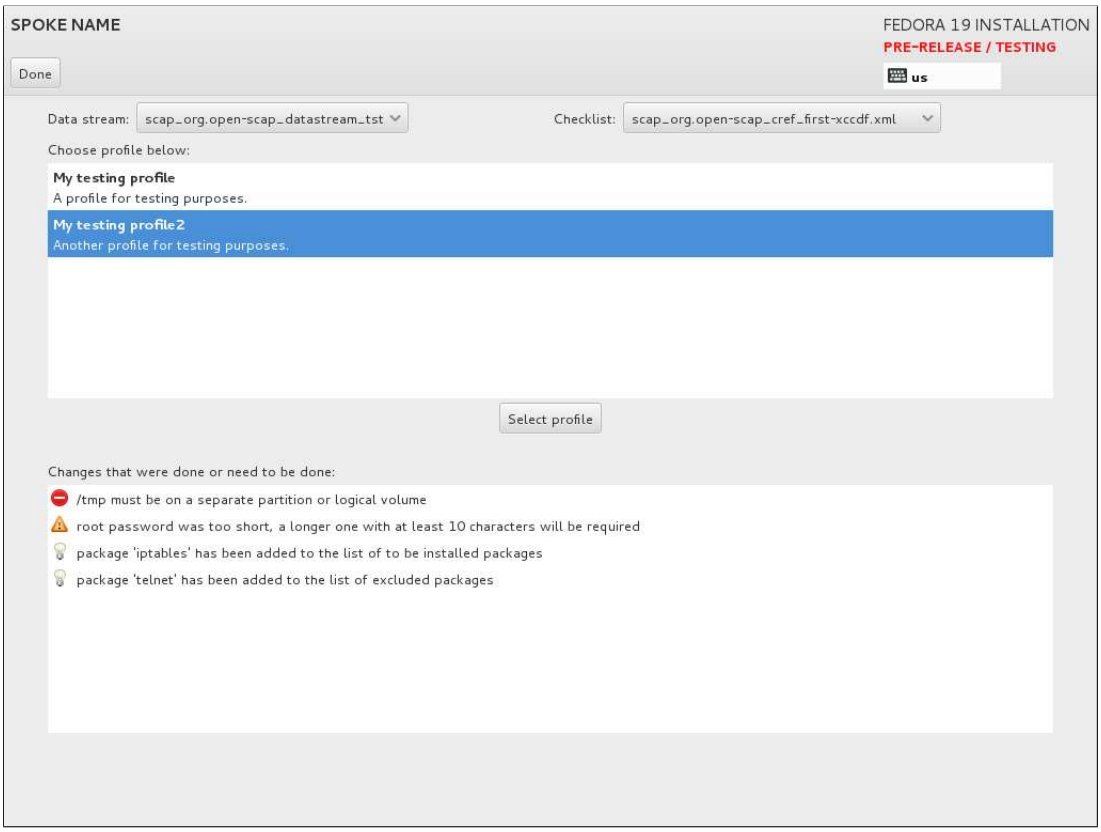

Figure 6.2: OSCAPSpoke's main screen

tent. The biggest two elements in the screen are two views listing the profiles available in the chosen checklist (the top one) and so called *rule messages* which inform the user about all changes that were or need to be done (according to the currently active profile) and warn about potential issues. The *"Select profile"* button in between these two views can be used for activation of the profile selected in the upper view. If the selected profile is the currently active one, the button is insensitive and thus cannot be clicked. Once a different profile is selected, the button becomes sensitive and if the user clicks it the selected profile is activated (evaluated) and the store with the rule messages is updated.

The screen was designed to fit in the style used by the Anaconda installer and also according to the rules described in the Eric. S. Raymond's book *The Art of UNIX Programming* [66] that emphasize on the clarity of the user interface together with visibly differentiating between more and less important pieces of information.

The *OSCAPSpoke* class defines properties and methods required by the Anaconda's addon API and a few additional properties and methods that facilitate the rest of the code. Every spoke must have the following properties:

- `ready` specifying if the spoke can be visited or not,
- `completed` telling whether all required values are set or not and
- `status` returning the short summary of the values set on the spoke that then appears on the hub.

Besides these properties, the spoke also have to implement these methods:

- `initialize` which is called when the GUI is being initialized,
- `refresh` that is called every time the spoke is visited,
- `apply` which is responsible for storing the values from the UI to the kickstart data instance[17] when the spoke is left and
- `execute` that should do all changes to the runtime environment the spoke requires.

Being a standard Python class, the spoke must also have the `__init__` method serving as a constructor. In OSCAPSpoke's case this method just initializes the instance attributes with default values and one of them, `self._storage`, with a value it gets as one of its arguments. This attribute holds reference to the Anaconda's storage configuration information and is crucial for pre-installation rule evaluation. The `initialize` method then pull's and stores references to

17. available as the `self.data` attribute

the UI elements of the spoke by querying an instance of the *GtkBuilder* class creating objects defined in the **oscap.glade** file. Then it looks at the value of the *OSCAPdata* instance's `content_url` attribute and if it is a valid URL it starts a thread for fetching the content, sends messages which say that the spoke is not ready and that it is fetching data to the hub and starts a thread that watches the fetching thread. Once fetching is finished, it initializes a *DataStreamHandler* or *BenchmarkHandler* instance with the fetched content, evaluates the special pre-installation rules from the content[18], initializes the UI elements (chosen profile, rule messages, etc.) and sends messages marking the spoke ready with a status describing the configuration's compliance with the chosen profile (will be described later). The `refresh` method just updates the UI elements with the values set in the kickstart data (data stream ID, XCCDF ID and profile) which are set either from parsing the kickstart file or by the `apply` method when the spoke is left. One interesting thing about the `refresh` method is that it is decorated with the `@gtk_action_wait` decorator (provided by the **pyanaconda** package) which makes sure that even if the decorated method is called from a non-main thread, it is run in the main thread (which is required by the Gtk library for code manipulating with the UI elements) and the caller's thread is blocked until the decorated method finishes. The returned value is then correctly returned to the caller. The reason for decorating the `refresh` method is that it is called from a non-main thread after data fetching and refreshes the UI elements. Since the spoke doesn't require any runtime changes to be done when leaving it, the `execute` method does nothing.

The `ready` property just returns the value of the `self._ready` attribute that is set during initialization and after data fetching. Because the spoke should prevent a system with non-compliant configuration to be installed, the `completed` property returns `True` or `False` depending on whether there are some problems (with the configuration) that cannot be fixed automatically or not. This way the spoke blocks the installation process by being incompleted until the configuration is fixed (e.g. a partition is scheduled to be created)

18. by initializing and populating a *RuleData* instance and calling its `eval_rules` method

or a different profile that is in compliance with the configuration is chosen. Last of the required properties is the `status` property. Its value reflects if there are any problems with the configuration (in relation to the chosen profile) or not. If there are no problems, the value is *"Everything okay"*. If there are warnings (some problems were fixed automatically by an important change of the configuration), the value is *"Warnings appeared"*. And finally, if there are some problems that cannot be fixed automatically, the *"Misconfiguration detected"* string is returned. The user is then expected to visit the spoke to find out what the problem is.

Most of the code in the *OSCAPSpoke* actually lies in the methods that are used for reacting on user's actions in the GUI. There are handlers for the signals emitted by the UI elements when user interacts with them (by changing the selected item or clicking them) and helper methods and properties that are used to update the UI after changes. For example there is the `_switch_profile` method that gets the chosen profile (plus the data stream and XCCDF IDs) from the GUI, reverts the changes done by evaluation of the previous profile and updates the rule messages store with the messages reported by the evaluation of the newly chosen profile. This method, as well as the other helper methods, is complicated by the fact, that the used SCAP content may be a standalone XCCDF benchmark or a data stream collection. Each of these two cases has slightly different code paths that need to be executed.

6.5 The code and deploying

The OSCAP addon is an open source project. The source codes are available in a public Git repository [65] so anybody can download them and since the licence used for the project is the *GNU General Public Licence* (GPL) version 2 (or later), the code can be freely used, modified and redistributed under the same licence or any later version of the GPL. To support the openness of the project, the source code is straightforward, clean and a vast majority of the functions, classes and methods have the so called *docstrings* providing a documentation about their purpose, parameters, return values, etc. In places where the coding style compatibility with some other pieces

of the code is not needed[19], the code follows the coding style described in the *Python Enhancement Proposal 8* (PEP 8) and follows the suggestions provided by the **pylint** tool. The code and its design follow the rules, suggestions and guidelines from the two famous books about UNIX/Linux programming: Eric S. Raymond's *The Art of Unix Programming* [66] and Robert Love's *Linux System Programming* [67]. And there are 56 unit tests that should be run after every change in the code and should be kept passing. New tests are expected to be added as more functionality is implemented.

For now, the OSCAP addon is just a Python package as the Anaconda's addon API requires. If somebody wants to test it, it has to be placed under the particular directory in installation environment so that the Anaconda installer finds it and instantiates it. This can be done by creating a so called *updates image* (for the installer) [68] with the addon included in it together with the dependencies the addon needs[20]. This is impractical and so for the future the plan is to create an RPM package with the addon that would have the right dependencies specified and that could be installed to the installer's environment by the **lorax** tool creating the installation images. However, it is a good practice to create a package only after the packaged software has a stable version which is not the case of the OSCAP addon yet because it is expected to be developed more before it is used by many users. The plan is to have the addon as something like a "technical preview" for the Fedora 19 release (the final release is planned on the beginning of July 2013 [69]), so that people can try using and testing it, and after fixing problems identified that way, the addon should be packaged and possibly included as part of every installation media for the Fedora 20 (the end of 2013) and the subsequent releases.

19. for example the OSCAPSpoke tries to follow the similar coding style as the Anaconda's spokes
20. the **openscap**, **openscap-utils** and **openscap-python** packages

Chapter 7

Conclusions

One of the biggest problems of IT security is that even there are often known principles and steps how to minimize attacker's possibilities and how to mitigate the risk of misuse of some particular system, they are not so often deployed and used by the administrators. Either they don't know how to apply them correctly, don't know where to get such rules and recommendations or apply them in a wrong phase of the system's life-cycle. This thesis tries to fight this problem by analyzing options provided by the Security Content Automation Protocol together with analyzing examples of the content following the format defined by the SCAP, all finalized with the implementation of an addon for the Anaconda OS installer that allows administrators to choose a security profile for the newly installed RHEL or Fedora[1] system with (semi-)automatic evaluation and remediation of the system.

The analysis of the SCAP an its components has shown that it is a very broad, versatile and extensible protocol providing enough expressivity for data that would help administrators to bring their systems in compliance with a chosen profile defined by the content they use. With the SCAP being so broad and complex it would be very difficult and time consuming to implement a whole chain of libraries and tools for processing content following the specifications. Fortunately, there is the OpenSCAP project (and the related projects) providing an open-source and freely available implementation of such chain containing a library and a tool for processing SCAP content that allow building a solution helping to fight the problem mentioned in the previous paragraph on top of them.

1. and number of their derivatives

How such solution should look like was identified by an analysis of the examples of available SCAP content and the rules they contained. We have found out that it can be hard to modify the system to be in compliance with some types of rules after the installation and thus an approach combining evaluation and remediation in the OS pre-installation and post-installation phases has been suggested as the best way to bring the system in compliance with a content. Moreover, the OS installation is a process administrators usually pay a lot of attention to so bringing up the question of security hardening during it is a nice way to notify them something like that should be and could be easily done. While OS installers are often very complex and one-purpose tools, fortunately, the Anaconda installer used by multiple GNU/Linux distributions provides a simple API for extensions (called addons). By studying the API and the capabilities it provides we have found out that it is possible to implement the suggested approach with an Anaconda addon.

However, by the time the work on the addon was being started there wasn't any working addon that would prove the Anaconda's addons support really works and show how an addon should look like. Thus as a first step on the way to the SCAP content processing addon the simple *Hello world* addon [60] has been implemented and deployed in the installation environment which helped to identify and fix bugs in the Anaconda's addon support [70]. And since there also was no documentation of the API and there were no guidelines for the Anaconda addon development, the work on the *Anaconda Addon Development Guide* [57] has been started to provide information covering that area. By the time of the work on this thesis it is still in a draft state, but it is expected to be reviewed, edited and taken as part of the official documentation for the Fedora and RHEL distributions which would simplify the work for the other people implementing an Anaconda addon. The Anaconda's addon support (with the *Hello world* addon as a proof of concept) was presented at the *Developer Conference* in February 2013 [71].

Finally the *OSCAP addon* that implements the approach of combining pre and post-installation evaluation of the SCAP content has been created. It uses the *openscap* library and the **oscap** tool provided by the *OpenSCAP* project (hence the name OSCAP addon) and hooks up to the Anaconda installer where it checks the configu-

ration of the to be installed system and remediates the system once it is installed, all according to a chosen profile from a given SCAP content. Its graphical user interface provides a way to choose a profile and review the changes in the configuration that are automatically done to match the rules defined by the profile, together with the changes that need to be done manually. And although it requires additional data to be added to the SCAP content (the special `fix` elements for the pre-installation phase), the format for this data has been designed to be very simple, clear and similar to the format of the so called *kickstart* files that is well-known to people using these files for (semi-)automatic installations. Moreover, kickstart files are sometimes parts of the security content provided by organizations [72] Thus it shouldn't be a problem to add such pieces of data to the existing and newly created content.

Even though the OSCAP addon now provides only the basic functionality and there is a lot of room for further enhancements and additional features, two videos [73] [74] that were recorded to provide a preview of its look and functionality got a very positive and constructive feedback from the members of the communities around the OpenSCAP and SCAP Security Guide projects [75] [76]. It will be available as a technical preview for the Fedora 19 release so that the interested people can try and test it, and then, after fixing reported issues and adding the most requested features it will be packaged as a software package for the Fedora distribution and provided in a easily deployable way for the Fedora 20 release together with some default SCAP content created by the community and amended with the rules for the pre-installation phase. Ideally as part of every installation media and thus every installation process which would mean more than million uses with every release[2] [77]. And since the addon API of the Anaconda installer should be preserved also by its version for the RHEL, the OSCAP addon will be also available for installations of this enterprise GNU/Linux distribution that will open a huge area of potential users with demanding (and often financial) interest in security and secure configuration of their systems.

So the two main goals now are further development of the addon and its features and working together with SCAP content creators on

2. if the user base of the distribution doesn't shrink unexpectedly

amending rules with the special fix elements understood and processed by the addon in the pre-installation phase. Since the whole project is open-source and publicly available, more contributors are expected to get involved once it receives more attention with the releases of the Fedora GNU/Linux distribution.

Bibliography

[1] GRUSKA, Jozef. *Future challenges of informatics 2013 – Chapter 4: New perception of (scientific) Informatics* [online], [cited 10.04.2013]. Available at:

<http://www.fi.muni.cz/usr/gruska/future13/>

[2] Microsoft Corporation. *Preventing Hardware Failures* [online] 2005, [cited 10.04.2013]. Available at:

<http://technet.microsoft.com/en-us/library/ee799385%28v=cs.20%29.aspx>

[3] Members of the OpenSCAP community. *Main Page – Openscap* [online] 26.04.2013, [cited 28.04.2013]. Available at:

<http://open-scap.org/wiki/index.php?title=Main_Page&action=history>

[4] The Open Source Initiative. *The Open Source Definition* [online], [cited 28.04.2013]. Available at:

<http://opensource.org/osd>

[5] Linux.org. *What is Linux* [online] 20.07.2012, [cited 28.04.2013]. Available at:

<http://www.linux.org/article/view/what-is-linux>

[6] Free Software Foundation, Inc. *The GNU Operating System* [online] 10.03.2013, [quoted 28.04.20113. Available at:

<http://www.gnu.org/gnu/gnu.html>

[7] National Security Agency. *Red Hat Linux Hardening Tips* [online] June 2012, [cited 01.05.2013]. Available at:

<http://www.nsa.gov/ia/_files/factsheets/rhel5-pamphlet-i731.pdf>

[8] Operating Systems Division Unix Team of the Systems and Network Analysis Center, National Security Agency. *Guide to the Secure Configuration of Red Hat Enterprise Linux 5* [online] 28.02.2011, [cited 01.05.2013]. Available at:

<http://www.nsa.gov/ia/_files/os/redhat/NSA_RHEL_5_GUIDE_v4.2.pdf>

[9] HALBARDIER, Adam, QUINN, Stephen, SCARFONE, Karen, WALTERMIRE, David. *The Technical Specification for the Security Content, Automation Protocol (SCAP)* [online] September 2011, [cited 01.05.2013]. Available at:

<SCAPVersion1.2>

[10] Refsnes Data. *XML Validation* [online] 2013, [cited 01.05.2013]. Available at:

<http://www.w3schools.com/xml/xml_dtd.asp>

[11] World Wide Web Consortium. *XML Signature Syntax and Processing (Second Edition)* [online] 10.06.2008, [cited 10.05.2013]. Available at:

<http://www.w3.org/TR/2008/REC-xmldsig-core-20080610/>

[12] BAKER, Jonathan, HANSBURY, Matthew, HAYNES, Daniel. *The OVAL® Language Specification Version 5.10.1* [online] 20.1.2012, [cited 02.05.2013]. Available at:

<http://oval.mitre.org/language/version5.10.1/OVAL_Language_Specification_01-20-2012.pdf>

[13] HAYNES, Danny, MELACHRINOUDIS, Stelios. *The OVAL® Language UNIX Component Model Specification Version 5.10.1* [online] 03.04.2013, [cited 02.05.2013]. Available at:

<http://oval.mitre.org/language/version5.10.1/OVAL_Unix_Component_Specification.04-04-2012.pdf>

[14] Linux Information Project. *Unix-like Definition* [online] 18.06.2006, [cited 02.05.2013]. Available at:

<http://www.linfo.org/unix-like.html>

[15] HAYNES, Danny, MELACHRINOUDIS, Stelios. *The OVAL® Language Windows Component Model Specification Version 5.10.1* [online] 19.01.2012, [cited 02.05.2013]. Available at:

<http://oval.mitre.org/language/version5.10.1/OVAL_Windows_Component_Specification_01-19-2012.pdf>

[16] Object Management Group. *OMG Unified Modeling Language (OMG UML), Infrastructure* [online] 05.08.2011, [cited 02.05.2013]. Available at:

<http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF>

[17] Open Source Initiative. *The BSD 2-Clause License* [online] , [cited 02.05.2013]. Available at:

<http://opensource.org/licenses/bsd-license.php>

[18] MITRE Corporation. *OVAL Interpreter* [online] 01.11.2012, [cited 02.05.2013]. Available at:

<http://oval.mitre.org/language/interpreter.html>

[19] Farnam Hall Ventures LLC. *jOVAL.org: OVAL implemented in Java. For free.* [online] 2013, [cited 03.05.2013]. Available at:

<http://joval.org/>

[20] Free Software Foundation, Inc.. *GNU Affero General Public License* [online] 28.02.2013, [cited 03.05.2013]. Available at:

<http://www.gnu.org/licenses/agpl.html>

[21] MITRE Corporation. *OVAL Repository* [online] 2013, [cited 03.05.2013]. Available at:

<http://oval.mitre.org/repository/>

[22] CASIPE, Maria, SCARFONE, Karen, WALTERMIRE, David. *Specification for the Open Checklist Interactive Language (OCIL) Version 2.0* [online] April 2011, [cited 08.05.2013]. Available at:

<http://csrc.nist.gov/publications/nistir/ir7692/nistir-7692.pdf>

[23] HALBARDIER, Adam, WALTERMIRE, David, WUNDER, John. *Specification for Asset Identification 1.1* [online] June 2011, [cited 08.05.2013]. Available at:

<http://csrc.nist.gov/publications/nistir/ir7693/NISTIR-7693.pdf>

[24] HALBARDIER, Adam, JOHNSON Mark, WALTERMIRE, David. *Specification for the Asset Reporting Format 1.1* [online] June 2011, [cited 08.05.2013]. Available at:

<http://csrc.nist.gov/publications/nistir/ir7694/NISTIR-7694.pdf>

[25] CHEIKES, Brant A., SCARFONE, Karen, WALTERMIRE, David. *Common Platform Enumeration: Naming Specification* [online] August 2011, [cited 08.05.2013]. Available at:

<http://csrc.nist.gov/publications/nistir/ir7695/NISTIR-7695-CPE-Naming.pdf>

[26] BOOTH, Harold, PARMELEE, Mary C., SCARFONE, Karen, WALTERMIRE, David. *Common Platform Enumeration: Name Matching Specification* [online] August 2011, [cited 08.05.2013]. Available at:

<http://csrc.nist.gov/publications/nistir/ir7696/NISTIR-7696-CPE-Matching.pdf>

[27] CICHONSKI, Paul, SCARFONE, Karen, WALTERMIRE, David. *Common Platform Enumeration: Dictionary Specification* [online] August 2011, [cited 08.05.2013]. Available at:

<http://csrc.nist.gov/publications/nistir/ir7697/NISTIR-7697-CPE-Dictionary.pdf>

[28] CICHONSKI, Paul, SCARFONE, Karen, WALTERMIRE, David. *Common Platform Enumeration: Applicability Language Specification* [online] August 2011, [cited 08.05.2013]. Available at:

<http://csrc.nist.gov/publications/nistir/ir7698/NISTIR-7698-CPE-Language.pdf>

[29] National Vulnerability Database. *Official Common Platform Enumeration (CPE) Dictionary* [online] 17.05.2013, [cited 20.05.2013]. Available at:

<http://static.nvd.nist.gov/feeds/xml/cpe/dictionary/official-cpe-dictionary_v2.2.xml>

[30] MITRE Corporation. *CVE – Frequently Asked Questions* [online] 2013, [cited 12.05.2013]. Available at:

<http://cve.mitre.org/about/faqs.html#a8>

[31] MITRE Corporation. *CCE Creation Process* [online] 22.03.2013, [cited 12.05.2013]. Available at:

<http://cce.mitre.org/lists/creation_process.html>

[32] MELL, Peter, ROMANOSKY, Sasha, SCARFONE, Karen. *The Common Vulnerability Scoring System (CVSS) and Its Applicability to Federal Agency Systems* [online] August 2007, [cited 08.05.2013]. Available at:

<http://csrc.nist.gov/publications/nistir/ir7435/NISTIR-7435.pdf>

[33] MELL, Peter, SCARFONE, Karen. *The Common Configuration Scoring System (CCSS): Metrics for Software Security Configuration Vulnerabilities* [online] December 2010, [cited 08.05.2013]. Available at:

<http://csrc.nist.gov/publications/nistir/ir7502/nistir-7502_CCSS.pdf>

[34] SCAP Security Guide community. *DRAFT Guide to the Secure Configuration of Red Hat Enterprise Linux 6* [online] 13.05.2013, [cited 15.05.2013]. Available at:

<http://people.redhat.com/swells/scap-security-guide/RHEL6/output/ssg-rhel6-xccdf.xml>

[35] World Wide Web Consortium. *About W3C* [online] 2012, [cited 10.05.2013]. Available at:

<http://www.w3.org/Consortium/>

[36] BOOTH, Harold, HALBARDIER, Adam. *Trust Model for Security Automation Data 1.0 (TMSAD)* [online] September 2011, [cited 10.05.2013]. Available at:

<http://csrc.nist.gov/publications/nistir/ir7802/NISTIR-7802.pdf>

[37] SCARFONE, Karen, SCHMIDT, Charles, WALTERMIRE, David, ZIRING, Neal. *Specification for the Extensible Configuration Checklist Description Format (XCCDF) Version 1.2* [online] September 2011, [cited 10.05.2013]. Available at:

<http://csrc.nist.gov/publications/nistir/ir7275-rev4/NISTIR-7275r4.pdf>

[38] World Wide Web Consortium. *XSL Transformations (XSLT) Version 1.0* [online] 16.11.1999, [cited 10.05.2013]. Available at:

<http://www.w3.org/TR/1999/REC-xslt-19991116>

[39] SCAP Security Guide community. *DRAFT Guide to the Secure Configuration of Red Hat Enterprise Linux 6* [online] 17.04.2013, [cited 15.05.2013]. Available at:

<http://people.redhat.com/swells/scap-security-guide/RHEL6/output/rhel6-guide.html>

[40] scap-workbench community. *scap-workbench* [online] 25.10.2012, [cited 15.05.2013]. Available at:

<https://fedorahosted.org/scap-workbench/>

[41] Security State community. *Security State (SecState)* [online] 27.11.2012, [cited 15.05.2013]. Available at:

<https://fedorahosted.org/secstate/>

[42] Tresys Technology. *SCC – Trac* [online] 07.10.2010, [cited 15.05.2013]. Available at:

<http://oss.tresys.com/projects/scc/wiki>

[43] OpenSCAP community. *Script Check Engine* [online] 14.03.2012, [cited 15.05.2013]. Available at:

<http://open-scap.org/page/SCE>

[44] SCE Community Content community. *SCE Community Content* [online] 01.06.2012, [cited 15.05.2013]. Available at:

<https://fedorahosted.org/sce-community-content/>

[45] Aqueduct community. *aqueduct – Supported Security Configuration Guidances* [online] 02.07.2012, [cited 15.05.2013]. Available at:

<https://fedorahosted.org/aqueduct/>

[46] Puppet Labs. *Puppet Labs Documentation* [online] 17.05.2013, [cited 15.05.2013]. Available at:

<http://docs.puppetlabs.com/#puppetpuppet>

[47] SCAP Security Guide community. *Welcome to scap-security-guide* [online] 01.04.2013, [cited 15.05.2013]. Available at:

<https://fedorahosted.org/scap-security-guide/>

[48] ROUSE, Margaret. *Federal Information Security Management Act (FISMA)* [online] May 2013, [cited 15.05.2013]. Available at:

<http://searchsecurity.techtarget.com/definition/Federal-Information-Security-Management-Act>

[49] Center for Internet Security. *About CIS Security Benchmarks Division* [online] 2013, [cited 15.05.2013]. Available at:

<http://benchmarks.cisecurity.org/about/>

[50] Center for Internet Security. *CIS Security Benchmarks Division Resources* [online] 2013, [cited 15.05.2013]. Available at:

<http://benchmarks.cisecurity.org/about/#resources>

[51] Defence Information Systems Agency. *Security Content Automation Protocol (SCAP) Content and Tools* [online] 27.04.2013, [cited 15.05.2013]. Available at:

<http://iase.disa.mil/stigs/scap/index.html>

[52] National Security Agency. *Security Configuration Guides* [online] 22.06.2012, [cited 15.05.2013]. Available at:

<http://www.nsa.gov/ia/mitigation_guidance/security_configuration_guides/index.shtml>

[53] U.S. Department of Health & Human Services. *HIPAA Privacy, Security, and Breach Notification Audit Program* [online], [cited 15.05.2013]. Available at:

<http://www.hhs.gov/ocr/privacy/hipaa/enforcement/audit/index.html>

[54] PCI Security Standards Council, LLC.. *Documents Library* [online] 2013, [cited 15.05.2013]. Available at: <https://www.pcisecuritystandards.org/security_standards/documents.php?document=pci_dss_v2-0#pci_dss_v2-0>

[55] National Institute of Standards and Technology, Information Technology Laboratory. *The United States Government Configuration Baseline (USGCB)* [online] 29.04.2013, [cited 15.05.2013]. Available at:

<http://usgcb.nist.gov/>

[56] Fedora project community. *Anaconda wiki* [online] 16.05.2013, [cited 20.05.2013]. Available at:

<https://fedoraproject.org/wiki/Anaconda>

[57] PODZIMEK, Vratislav. *Anaconda Addon Development Guide* [online] 20.05.2013, [cited 21.05.2013]. Available at:

<http://vpodzime.fedorapeople.org/anaconda-addon-development-guide>

[58] Wikipedia contributors. *Virtual Network Computing* [online] 05.05.2013, [cited 18.05.2013]. Available at:

<http://en.wikipedia.org/wiki/VNC>

[59] Python community. *GlobalInterpreterLock* [online] 02.08.2012, [cited 18.05.2013]. Available at:

<http://wiki.python.org/moin/GlobalInterpreterLock>

[60] PODZIMEK, Vratislav. *Hello World addon repository* [online], [cited 16.05.2013]. Available at:

<http://www.fi.muni.cz/~xpodzim/git/?p=hello-world-anaconda-addon.git>

[61] Glade Project community. *Glade - A User Interface Designer* [online] 06.03.2013, [cited 18.05.2013]. Available at:

<http://glade.gnome.org/>

[62] The Linux Documentation Project community. *What is SSL and what are Certificates?* [online], [cited 18.05.2013]. Available at:

<http://www.tldp.org/HOWTO/SSL-Certificates-HOWTO/x64.html>

[63] KENT, Stephen T.. *Internet Privacy Enhanced Mail* [online] 01.02.2006, [cited 18.05.2013]. Available at:

<http://www.acsac.org/secshelf/book001/17.pdf>

[64] GNOME Project. *GTK+ 3 Reference Manual* [online] 2012, [cited 19.05.2013]. Available at:

<https://developer.gnome.org/gtk3/stable/>

[65] PODZIMEK, Vratislav. *OSCAP addon repository* [online], [cited 18.05.2013]. Available at:

<http://www.fi.muni.cz/~xpodzim/git/?p=master-thesis.git>

[66] RAYMOND, Eric S. *The Art Of Unix Programming*. 1st Edition. Addison Wesley Professional, 2003. 547 pages. ISBN 0-13-142901-9.

[67] LOVE, Robert. *Linux System Programming*. O'Reilly Media, Inc., 2007. 368 pages. ISBN 0-596-00958-5.

[68] Fedora project community. *Anaconda Updates wiki* [online] 06.07.2012, [cited 19.05.2013]. Available at:

<https://fedoraproject.org/wiki/Anaconda/Updates>

[69] Fedora project community. *Releases/19/Schedule* [online] 13.04.2013, [cited 19.05.2013]. Available at:

<https://fedoraproject.org/wiki/Releases/19/Schedule>

[70] PODZIMEK, Vratislav. *[PATCH 1/2] Use ksdata.addons instead of ksdata.addon and add ADDON_PATHS to sys.path* [online] 12.02.2013, [cited 18.05.2013]. Available at:

<https://lists.fedorahosted.org/pipermail/anaconda-patches/2013-February/003052.html>

[71] PODZIMEK, Vratislav. *The technology beyond Anaconda NewUI and 3rd party extensions* [online] 12.03.2013, [cited 18.05.2013]. Available at:

<http://www.youtube.com/watch?v=e9bIubGmpD4>

[72] GRUBB, Steve. *USGCB Standard Desktop Baseline Kickstart* [online] 30.09.2011, [cited 18.05.2013]. Available at:

<http://usgcb.nist.gov/usgcb/content/configuration/workstation-ks.cfg>

[73] PODZIMEK, Vratislav. *SCAP content based configuration in Fedora installation* [online] 24.04.2013, [cited 19.05.2013]. Available at:

<https://vimeo.com/64702496>

[74] PODZIMEK, Vratislav. *SCAP content based configuration in Fedora installation (update1)* [online] 13.05.2013, [cited 19.05.2013]. Available at:

<https://vimeo.com/66085973>

[75] SHIMKO, Spencer. *Video preview of the OSCAP Anaconda addon* [online] 15.05.2013, [cited 19.05.2013]. Available at:

<https://lists.fedorahosted.org/pipermail/scap-security-guide/2013-May/003331.html>

[76] WELLS, Shawn. *Re: [Open-scap] Video preview of the OSCAP Anaconda addon* [online] 14.05.2013, [cited 19.05.2013]. Available at:

<https://www.redhat.com/archives/open-scap-list/2013-May/msg00014.html>

[77] Fedora project community. *Statistics – FedoraProject* [online] 10.05.2013, [cited 19.05.2013]. Available at:

<http://fedoraproject.org/wiki/Statistics>

Attachments

Examples are often much better and clearer than long descriptions and explanations. For this reason, a few directories and files should be distributed with this text as attachments, either on an attached CD or next to the file with the text. The directories have the following structure and contents:

- *examples*:
  - **data_stream_coll.xml** – a data stream collection
  - **xccdf.xml** – a standalone XCCDF benchmark file
  - **oval.xml** – an OVAL definitions file (used by the **xccdf.xml** file)
  - **sce_xccdf.xml** – an XCCDF using the SCE
  - **check.sh** – simple script used by the **sce_xccdf.xml** file
  - **ks.cfg** – a kickstart file that can be used to test the OSCAP addon
- *org_fedora_oscap* – directory with the OSCAP addon's package
- *tests* – directory containing the unit tests and the **README** file explaining how to run them

Index