Processing engine for security health checks

Christofer Huynh, Jesper Gustafsson

LUND
UNIVERSITY

Department of Computer Science

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

Printed in Sweden.
Lunds University
Lund 2017

Abstract

Computer security is still an often neglected field even though the IT industry is huge and still growing. Making sure that systems are secure is something that is very important but can take a lot of man hours.

This thesis contains the research and documentation for an auditing tool created on behalf of a company that specialises in computer security, TrueSec. The audit tool uses the log files that are built from the scripts that TrueSec uses to runs various unix commands as a base, but it has been designed in such a way to allow the addition of other types of logs. The tool was designed with future development in mind. It uses configuration files that are seperated from the code itself to declare rules which the log files are checked against, allowing the tool to evolve with time by adding new rules.

At the request of TrueSec a simple web service as a proof of concept for an online service with an authentication model with log in and user roles for uploading log files and storing the output from the audit tool was also developed.

Keywords: audit tool, computer security, security audit, analysis engine, security health check

Sammanfattning

Datasäkerhet är fotfarande ofta ett misskött fält även fast IT- sektorn är stor och fortfarande växer. Att se till att system är säkra är någonting som är mycket viktigt men kräver flera mantimmar.

Det här examensarbetet innehåller undersökning och dokumentation för ett säkerhetsgranskningsverktyg(analysmotor) skapad i samarbete med ett företag som specialiserar sig inom datasäkerhet, TrueSec. Analysmotorn använder sig av loggfiler som är byggda från de skript som TrueSec använder. Dessa skripts har använts för att ge analysmotorn en bas, men analysmotorn är byggt på så sätt att man kan lägga till stöd för andra typer av loggar i framtiden. Analysmotorn är byggd med åtanke att den ska vidareutvecklas. Den använder konfigurationsfiler som är separerade från själva koden. Dessa konfigurationsfiler används för att skapa regler som loggarna skall jämföras med.

Även en enkel webbtjänst som ett proof of concept för en online service har utvecklats. Denna service innehåller en autentiseringsmetod med inloggning och registrering samt tillåter användare att ladda upp log filer och ladda ner rapporten skapad av motorn.

Nyckelord: säkerhetsgranskning, datasäkerhet, analysmotor, security health check

Acknowledgements

First of all we would like to thank Stefan Ivarsson and the rest of TrueSec for giving us the opportunity to work with them as well as helping us with questions about computer security. We would also like to thank our supervisor Martin Hell and examinator Christian Nyberg for guidance and helping us scope the thesis.

Contents

1

Introduction

1.1 Background

The IT industry is still growing and even with the growth, computer security is still often a neglected field. Many companies lack the time and resources to ensure their computer system have adequate security and data protection. This thesis will dive into the neglected field of computer security working with TrueSec which is a company that does continuous security health checks on customers systems and infrastructure.

Security health check consists of e.g. how the system and the surrounding infrastructure is administrated and maintained. For example, analysing the routines for patch management and disaster recovery but the most comprehensive part of the work consists of gathering information about the configuration of the OS, middleware, cloud infrastructure and other components used in the system. The gathering of information is mostly automatic via two types of open source scripts and one collection of scripts written by TrueSec themselves but the analysis of the said information is done manually which is time consuming. The tools or scripts used for gathering information usually provide log file(s) with the information.

The log files are then analysed by TrueSec specialists. In general TrueSec usually uses a baseline that is either derived from earlier security checks or by running the scripts on a clean installation of the system/service being investigated.

The results of the security health check is compiled into a report with potential risks and severities as well as action proposals. After the customer has adjusted the system a new security health check is often done where the customer can see an improvement.

This bachelor thesis will be the product of a collaboration between the authors and TrueSec. The thesis will help TrueSec create a prototype of an analsyis engine for analysing and working with the data from security health checks to improve the security of their customers systems. This analysis engine will make the analysis process as automatic as possible and will let the company more effectively help their customers.

While the analysis engine is what the thesis focuses on, a prototype web service has also been developed. This web service's main purpose is to allow users to upload log files to be put through the analysis engine and then download the resulting report.

1.2 Purpose

This thesis will assist the evaluation of a system by trying to automate some of the manual labor. This is desired because the manual labor is time consuming. Automating some of the manual labor should improve the speed of the analysis.

1.3 Formulation Of Goal

The end goal is a report with documented results and conclusions from the analysis by the analysis engine. The analysis engine should be able to output this report by evaluating log files created by certain preexisting scripts and audit tools (Chapter 3). The log files are to be compared in the same way they are compared manually, by comparing them to a baseline. The analysis engine is designed with the intention of being easily modified for further development.

The thesis product consists of the following two parts.

Analysis engine

The first part is an analysis engine which takes in log files generated via the audit systems used by TrueSec and outputs a text file with all security flaws it has found and possible suggestions to combat them. This engine is where the main focus of the work has been.

Web Service

The second part is a simple web service which is used to present the report generated via the analysis engine. The web service has an authentication model with login and user roles. A user should be able to upload log files to the web service which will be analysed using the analysis engine. The user can then download the resulting report from the engine. This thesis is about the analysis engine and the web service is more of an addon that will not be researched or explained in-depth.

1.4 Presentation Of The Problem

The problem right now is the time consuming manual work of the analysis phase. Thus the primary and more general objectives of this thesis are following:

- What language or languages will be used to develop the prototypes?

- How can we automate the manual labor?
- How much of the current manual labor can we make automatic?
- How much faster does the process become with our implementation?

1.5 Thesis Ambition

The reason for choosing this thesis is that computer security research is something that can be very valuable not only because of the knowledge it would give us but also for future job opportunities.

This thesis will provide prototypes that will hopefully help companies such as TrueSec to quicken the analysing process of their customers' system. Additionally, the thesis might illuminate the security problems that are common within current systems and how you can fix them.

1.6 Limitations

For this thesis the goal is for our prototypes to work with RedHat and Debian based distributions and the Amazon AMI, a CentOS distribution running on Amazons cloud service.

Because it is not realistically achievable to create a fully automated analysis engine for the time span of this thesis, the analysis engine is written with the intent of being easily modifiable. This will allow future developers to automate it further, eventually getting close to full automation.

2

Technical Background

This section presents the technical details in order to understand this thesis work.

2.1 Flask

The web service will be made in Flask[1], which is a web framework for Python. Flask provides tools, libraries and technologies that are used for building web applications.

Flask is categorized as a *microframework*. Microframeworks are frameworks with almost no dependencies to external libraries. The pros with a microframework are that they are light and there is little dependency to update and watch for security bugs. The cons are that the user has to do more work or increase the list of dependencies by adding plugins. Flask has the following two dependencies:

- Werkzeug, a WSGI[2] utility library
- jinja2, a template engine (See chapter 2.4)

There are also extensions for Flask such as, FlaskSQLAlchemy, Flask-WTForms, Flask-Login, Flask-Bcrypt and Flask-Assets, which are all used in this thesis work.

The FlaskSQLAlchemy[3] extension adds support for SQLAlchemy[4] and other SQL databases. It aims to simplify the usage of SQLAlchemy with Flask by providing useful defaults and extra helpers which makes it easier to accomplish common tasks.

[1] http://pymbook.readthedocs.io/en/latest/flask.html
[2] A protocol defined so that Python application can communicate with a web server and be used as a web application outside of the Common Gateway Interface (CGI)
[3] http://flask-sqlalchemy.pocoo.org/2.1/
[4] https://www.sqlalchemy.org/

One of the advantages of SQLAlchemy is the powerful common statements and types that ensures that the SQL queries are properly and efficiently crafted for each database type and vendors without the user having to think about it[5].

Flask applications uses the Flask-Migrate[6] extension to handle database migrations using Alembic[7]. By using the following command the user can create a migration repository:

```
$ flask db init
```

This command will add a migration folder to the application. The contents of the folder needs to be version controlled along with other source files. Now the user can generate an initial migration:

```
$ flask db migrate
```

To apply the migration to the database:

```
$ flask db upgrade
```

Each time the database model changes, the migrate and upgrade commands must be repeated.

Flask-WTForms[8] is used for form input handling and validation. WTForms generate the form field HTML and also allows customizations of templates. This allows the user to manage the seperation of code and presentation, and to keep the disordered parameters out of the python code. WTForms strives for loose coupling and allows the user to do it in any templating engine they choose.

Flask-Login[9] is a user session management extension for Flask. It handles the common tasks such as logging in, logging out and remembering the users' session over an extended period of time. Flask-Login stores the active user's ID in the session and this will be used to log in and out easily. This also restricts views to logged in and logged out users. From a security perspective this extension protects the users' sessions of the web service from being stolen by cookie thieves.

Flask-Bcrypt[10] is a extension that provides bcrypt[11] hashing utilities for the Flask application.

Flask-Assets integrates webassets[12] into the Flask application. A webasset is in general, a dependency-independency library used to manage a web application's assets. It can compress files like CSS and JavaScript files.

[5] Jason Myers & Rick Copelannd, *Essential SQLAlchemy*, O'Reilly Media, California, 2016 p xiii
[6] https://flask-migrate.readthedocs.io/en/latest/
[7] Alemic is a lightweight database migration tool
[8] https://wtforms.readthedocs.io/en/latest/crash_course.html
[9] https://flask-login.readthedocs.io/en/latest/
[10] https://flask-bcrypt.readthedocs.io/en/latest/
[11] Bcrypt is a password hashing function designed by Niels Provos and David Mazières, based on the Blowfish cipher, and presented at USENIX in 1999.
[12] https://webassets.readthedocs.io/en/latest/index.html#index

2.2 MySQL Database

MySQL[13] is one of the most popular Open Source Relational SQL database management system. The development, distribution and support of this database management system is done by Oracle.

The SQL which stands for "Structured Query Language" is the most common standardized language used to access databases. A database is a structured collection of data which has a wide range of uses. It can be something simple such as a simple shopping list to a vast amounts of information in a corporate network.

Steve Suehring (2002) mentioned that MySQL offers a best of all worlds scenario because it runs on many platforms and is cheap and stable[14]. The documentation is excellent, MySQL AB has the reference material on their website and also offers a high quality support for their products, for example a service that allows MySQL developers to log into the server of a web application to e.g. correct the problems or optimize the server.

A Relational DataBase Management System (RDBMS) is a software that enables the implementation of tables, columns and indexes to databases, which guarantees the Referential Integrity between rows of various tables.

For the web service, MySQL database is used to integrated the Flask framework with the help of the FlaskSQLAlchemy extension.

2.3 Bower

Bower is a front end package manager made by Twitter. The concept of package management is also known as dependency management[15]. Package manager is not a new idea. Ambler and Cloud explained that this practice has only recently gotten a widespread adoption because of the management of front end web assets, such as JavaScript libraries, stylesheets, fonts, icons and images that all serve as building blocks for a modern web application. The authors mean that the need for a good structure became obvious as the foundation on the modern web became more complex.

Bower is a simple command line utility that help managing some of the tedious tasks with front end assets. One thing that differs Bower from other well known packet managers is that it was not made for handling some specific needs of some specific platform or language. It was designed to create a simple tool for managing just code but also the wide variety of the front end assets such as stylesheets, font, images and other unforeseen future dependencies.

[13] https://dev.mysql.com/doc/refman/5.7/en/what-is-mysql.html

[14] S.Suehring, *MySQL Bible*, Willey Publishing, Inc, New York, 2002, p. 11

[15] Tim Ambler and Nicholas Cloud, *JavaScript: Frameworks for Modern Web Dev*, Spring Science+Business Media New York, New York, 2015, pp 1

Some developers may think Bower is not needed when they are developing a trivial web application with few dependencies, but as the process goes on the trivial web applications have a tendency to become more complex. Which leads to developers often appreciating Bower.

> *"As bitter experience has taught us—the project itself. Err on the side of too little structure, and you risk creating an ever increasing burden of "technical debt" for which you must eventually pay a price."*
> *-Tim Ambler & Nicholas Cloud, 2015, p 9*

The web service uses Bower to manage packages such as Bootstrap3 (see section 2.5), fonts, and jQuery.

2.4 jinja2

Jinja2 is a full featured template engine for Python which has a full unicode support and an optional sandboxmode environment, a library for python that is designed to be fast, flexible and secure[16]. Jinja is also the most used template engine for Python. It is very important to distinguish the HTML code from the Jinja code so the Jinja engine knows what code is HTML and which is Jinja, therefore Jinja code is surrounded by "{%" and "%}" as seen in the example below.

```
{% extends "layout.html" %}
{% block body %}
  <ul>
  {% for user in users %}
    <li><a href="{{user.url}}">
    {{user.username}}</a></li>
  {% endfor %}
  </ul>
{% endblock %}
```

Some other features of Jinja2 are:

- Template inheritance, making it possible to use the same or similar layout for all templates
- Easy debugging, line numbers of exceptions pointing to the corresponding line in the template
- A powerful automatic HTML escaping system which prevents cross site scripting (XSS).

The web serivce uses Jinja2 in its HTML templates to generate the web pages.

[16] http://jinja.pocoo.org/

2.5 Bootstrap3

The framework was created by Mark Otto and Jacob Thornton at Twitter. Bootstrap is one of the most popular HTML, CSS and JavaScript framework for faster and easier web development. It also scales websites and applications with a single code base, from phone to website[17].

The perks for using Bootstrap3 is:

- **It is simple:** It doesn't require advanced knowledge of HTML and CSS, anyone can get started. It also have a official site with great documentation.
- **Responsive design:** The framework adjusts phones, tablets and websites.
- **Components:** Bootstrap has many reusable components such as navigations, dropdowns and alerts which is customizable.
- **Browser support:** It is compatible with all popular browsers.

An example of Bootstrap responsive design, is that it has a own grid system:

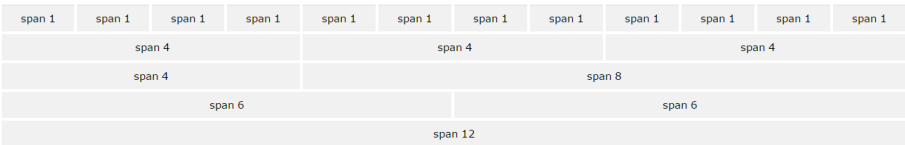

Figure 2.1 Bootstrap grid system

This grid system allows up to 12 columns across the page and is responsive which means it will adjust automatically depending on the screen size. Additional it has Grid classes such as xs (phones), sm (tablets), md (desktops), lg (large desktops) which can be combined in order to create more dynamic and flexible layouts.

The Bootstrap framework is applied by the web service, using its design and components such as navigation system. Additionally it uses the responsive design to adjust the web service to phones and tablets.

[17] https://www.tutorialspoint.com/bootstrap/index.htm

2.6 Auditing Linux/Unix Systems

A security audit may focus on any number of areas. While some of these areas are generic and intrinsic to nearly all Linux distributions, other areas and security policies can be very company specific. In other words, a security audit is very different depending on the company. Computer security is a delicate field, what is a major security issue at one company may be a standard policy in another. However there are important audit areas such as[18]:

- Physical Security
- Logical Security
- Security policy and administration

Limiting physical access to a system will keep accidents and trespassers away. Auditors must also ensure that PC booting from CD/DVD, external devices, floppy disk are turned off and that the password is enabled in the BIOS which protects the GRUB (GRand Unified Bootloader) to ensure the restriction of physical access of the server[19].

The logical security is the bigger part of the security audit which includes topics such as:

Password Policies The password should contain a minimum number of characters, including numerical, non-alpha and special characters. It should also avoid using words that are typically found in a dictionary.

The cryptographic hash functions should be the strongest afforded by the system, such as SHA512. The passwords should be stored in a file not readable by users by using a "shadow" password file which separates the user information from the actual hashed passwords.

Controlling the root password The root password enables unlimited access to a system which could potentially allow intruders to bridge the system and compromise the security of a group of systems easily. Usually root should be only logged in from the terminal (physical or virtual). The password should be complex e.g. 24 random characters. The password could also be stored in a form of a keystore.

Securing SSH During the audit the remote connectivity services of the server and the Secure Shell (SSH) protocol which uses an encryption technology during the communication should be checked. The auditors can check the SSH configurations by accessing the */etc/ssh/sshd_config* file. It is necessary to check if the root login is

[18] Richard Williams, 2003, *UNIX Audit: Performing a Successful Unix Audit*, Computer Fraud & Security, vol. 2003, pp. 11-12

[19] Muhammad Mushfiqur Rahman, *Auditing Linux/Unix Server Operating Systems*, ISACA Journal Volume 4, 2015 pp 1

disabled. Attackers can log in using root@ipaddress and try to brute force the password. This means if root login is disabled, the attacker has to first guess a username before they can brute force the password.

Sudo "Sudo" is a utility for a system administrator to restrict commands that can be run by users[20]. It is not necessary to give a user all the privileges, but just some of them. A user that types *sudo <command>* will be asked to authenticate using a password. If the command is listed in the sudoers sudo configuration file, then user will be able to run the command with root privileges.

Users can be sorted into groups to make system administration easier. The system administrator can assign rights and privileges to the entire group by changing the configuration in the */etc/group* file. This will also make it easier to grant or revoke permissions to groups instead of multiple users.

Cron and at Moohkey and Burghate mentioned that the *cron* and *at* utilites are the favorite place for the hackers to insert their own processes. The cron and at utilities are used to execute certain tasks at a predefined time[21]. The tasks are mostly system maintenance tasks such as making backup files and cleaning up log files. A security measure is to configure the */etc/cron.allow* and */etc/cron.deny* files by defining users who have privileges to schedule such tasks. Thus it is recommended to explicitly allow only the root to be able to schedule tasks by restricting privileges to the /etc/cron.allow file.

Deleting files Files and objects exists in logical and in physical memory. If a user removes a file from the filesystem it will still exist in some form. In Unix, a user can use the commands *link* and *ln* to create a new link with a pointer to the original file. If the user uses the *rm* or *rmdir* command to remove the original file, it will disappear from its parent directory, however the links to the file and its content still exists. This means the user may think the file is deleted but it still exists in the system[22]. Since these files still exists, a recommended security measure is to delete all the links to that file and wipe the files content by overwriting with e.g. all zeroes or other content, so that malicious users cannot access the content.

Environment Variables Environment variables are normally used for configuration of the behaviour of utility programs. A process inherits the environment variables from its parent process, which means if a program X executes program Y, then program X can set the environment variables for program Y[23].

A security problem with this is that the invoker of SUID/SGID programs has control over the environment variables these programs are given. Thus an attacker

[20] Mookhey, K.K. and Burghate, Nilesh, *Linux: Security, Audit and Control Features*, Information Systems Audit and Control Association, 2005, pp 56
[21] Ibid pp 33
[22] Dieter Gollmann, *Computer Security*, John Wiley & Sons, Inc, New York, NY, USA, 1999, pp 120
[23] Ibid pp 122-123

can try to take control of the execution by setting the environment variables to dangerous values. A countermeasure is to erase the entire environment with a SUID/SGID program and then reset a small set of environment variables to safe values.

The environment variables file contains a list and and some of the variable looks like following:

PATH	searchpath for shell commands
HOME	path for the home directory
HOSTNAME	name of the Unix host
IFS	character separating command line arguments

The values on the right hand side have security implications and the auditors must make sure that the values are set to appropiate values. The value on the right hand side can look something like this:

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:
/usr/bin:/sbin:/bin:/snap/bin
```

The value colon (:) separates directories on the Linux system. The operating system will search within these directories for a binary or script that can be executed. For example, a program can be started by just typing its name without specifying its location. Thus auditors should ensure a string like (:.:) does not exist within this path[24]. The period within the two colons marks the current directory. This means that the system will search for the command within the directory, in addition to others directories. The system will stop searching at the first location where the program with the specified name is found.

It is therefore possible to insert a trojan by giving it the same name as an existing program in a directory that will be searched earlier than the original program's directory[25]. The security measure is to call the program by its full path and the auditors should make sure that the current directory is not in the search path of programs executed by root.

The security features of an operating system are useless if they are not properly used. The installation and configuration of the system is important. An operating systems is a continually evolving software system, for that reason there is always a chance for new vulnerabilities that accidentally gets introduced in a new release[26].

A way to keep track of a user's action is necessary to be able to investigate security breaches and trace attempted security attacks, hence the system must keep an audit log of security relevant events.

This section is not a complete introduction to the Linux/Unix security system but some of the basic security features that are relevant to the thesis.

[24] Mookhey, K.K. and Burghate, Nilesh, Linux: Security, Audit and Control Features, Information Systems Audit and Control Association, 2005, pp 66

[25] Dieter Gollmann,Computer Security, John Wiley & Sons, Inc, New York, NY, USA, 1999, pp 124

[26] Ibid pp 108

2.7 YAML

YAML[27] is a language used for data serialization commonly used for configuration files. It is possible to use YAML files to create a dictionary[28] in Python. The following is an example code used in the analysis engine:

```
'LoginGraceTime':
  'nlt':
    'value': 60
    'severity': YELLOW
    'msg': LoginGraceTime is less than 60
  'ngr':
    'value': 120
    'severity': RED
    'msg': LoginGraceTime is greater than 120
```

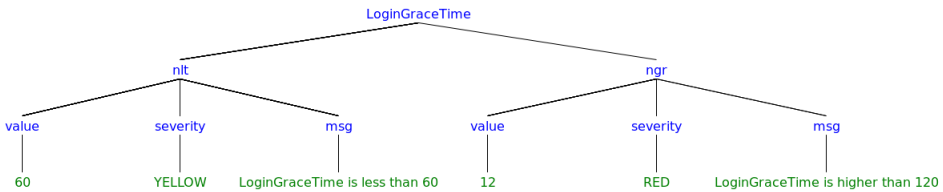

Figure 2.2 YAML key-value tree

Figure 2.2 shows a tree view of the dictionary that would be loaded using the above YAML code. The text in blue are keys and then text in green are values. By traversing through the keys one can find the associated values. The analysis engine uses YAML files to store values that are then compared to the logs given by the scripts, these are easily modifiable edit or add new values with their own warning messages and severity label.

[27] http://www.yaml.org/spec/1.2/spec.html
[28] https://www.tutorialspoint.com/python/python_dictionary.htm

3

Environmental Analysis

In order to create a time efficient analysis engine that is equal or faster than manual labor, an environmental analysis has been done. This section includes the analysation of existing methods (e.g. using earlier security check reports and data), API and frameworks for evaluating data from security health checks.

3.1 Tiger Analytical Research Assistant (TARA)

Tiger is a security tool used for security audit and intrusion detection system written entirely in shell language[1]. The security tool was developed to provide checks for UNIX systems on the Texas A&M campus. However the development stopped after version 2.2.4 in 1994[2].

This tool has now been resurrected is currently developed by Advanced Research Computing, making it useful for newer UNIX operating systems.

Example of some of files the script checks are:[3]

- **/etc/passwd file:** Make sure there are no entries with unset password or duplicate UIDs in.
- **/etc/shadow file:** Make sure there are no entries with unset password or duplicate UIDs in.
- **/etc/group file:** Check if UNIX groups are available in the system and looks for conflicts and improper entries.
- **home directories:** Confirm that the home directories only are readable by the correct UID/GID.

[1] http://www.nongnu.org/tiger/index.html#history.html
[2] http://savannah.nongnu.org/projects/tiger
[3] Steve Kemp, *Common Security check for a base installation - package reviewed*, 2002, https://lists.debian.org/debian-devel/2002/12/msg01566.html

Tiger security checks 40 different files[4]. After the check the output file is put in /var/log/tiger/security.report.ubuntu.YYMMDD-HH:MM, where it is only accessible by root.

3.2 Lynis

Lynis is an open source security auditing tool. Used by e.g. system administrators and security professionals to evaluate the security defense of their Linux and UNIX based systems[5]. Lynis runs on the host itself so it can perform more extensive security scans than normal scanners. The following system areas may be checked[6]:

- Boot loader files
- Configuration files
- Software packages
- Directories and files related to logging and auditing

The security scan will grade the system's weaknesses and strengths and summarize it to something called a "hardening score". For each weakness detected, it will present a warning and a suggestion of what action to take.

Lynis is lightweight and can be executed on production machines without heavy penalties. The tool is executed via the following command with optional extensions:

```
./lynis system audit
```

The tool runs various tests as bash commands and stores the output in a report file named lynis-report.dat. There is also a log file stored named lynis.log which contains more information regarding the tests such as if something was or was not found.

The lynis.log file is more readable unlike the report file which just stores the output from bash commands. Lynis runs various tests and determines whether to issue an OK, suggestion or warning which is sent to the report and log file. It will also present the information in the terminal while the scan is running and show the user if something has triggered an OK, a suggestion or a warning. It will then list the warnings and suggestions.

[4] See Appendix A for more detailed checks by Tiger
[5] https://cisofy.com/lynis/
[6] See Appendix B for more details about the shell scripts used in the security scan

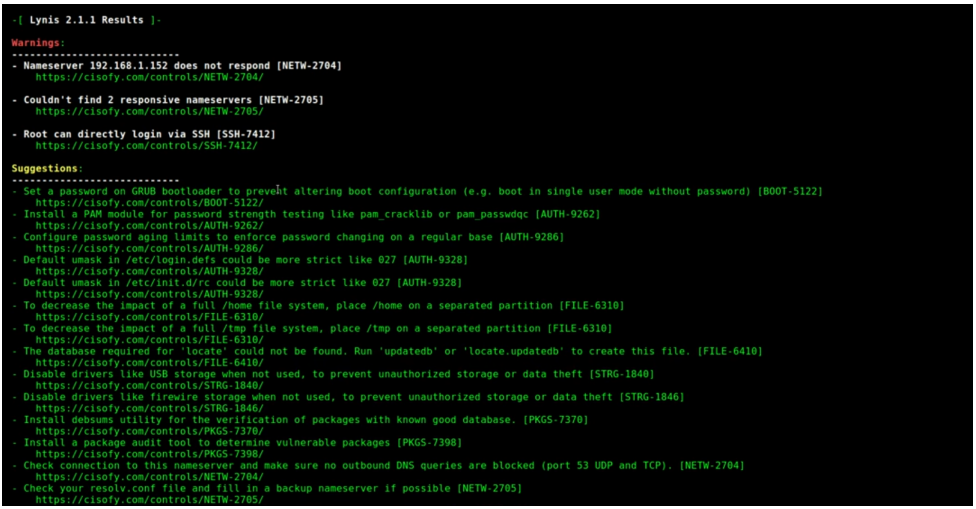

Figure 3.1 Example of warnings and suggestions from a Lynis scan

3.3 OpenSCAP

The standardised compliance checking solution for enterprise level Linux infrastructure is called SCAP. SCAP is implemented by the OpenSCAP application. This autiding tool utilizes the Extensible Configuration Checklist Description Format (XDCCDF[7]). OpenSCAP also combines with other specifications e.g. Common Platform Enumeration (CPE), Common Configuration Enumeration (CCE) and Open Vulnerability and Assessment Language (OVAL)[8].

3.4 Unix Privesc Check (UPC)

Unix Privesc Check (UPC) is an open source shell script for Unix systems that focuses on finding ways for a user to elevate itself to higher security clearance. It checks for misconfiguration that could allow local unprivileged users to escalate privileges to other users or to access local apps (e.g. databases)[9].

UPC is written as a single shell script so it can be uploaded and run without being installed. This tool was made to complement other security audit tools. It focus on finding apparent attack vectors.

This script is run using the one of the following commands in the terminal:

[7] XDCCDF is a specification language for writing security checklists, benchmarks, and related kinds of documents as defined by The National Institute of Standards and Technology.

[8] https://access.redhat.com/documentation/en-US/Red_Hat_Network_Satellite/5.5/html/User_Guide/chap-Red_Hat_Network_Satellite-User_Guide-OpenSCAP.html

[9] https://github.com/pentestmonkey/unix-privesc-check

```
./unix-privesc-check standard > output.txt

./unix-privesc-check detailed > output.txt
```

The standard mode is speed optimised but still checks for a lot of security settings.

Running it as detailed takes a lot longer as it checks more files. It also has a tendency to report false positives, i.e false warnings. Both of the scans save their findings in the output file. The script runs various bash commands depending on which mode it was launched in and reports its findings in the output file. If a security risk has been found it can be found by searching "WARNING" in the output file.

3.5 Linux Security Auditing Tool (LSAT)

Linux Security Auditing Tool is a post install security audit tool. The modular design in this audit tool was made so that new features can be added quickly. LSAT scans many system configurations and local network settings on the system for errors. It also scans for unneeded packages[10].

To handle the large reports (approximately 95 pages[11]) generated by the security scans, one of the modules creates MD5 sums of the scanned files. These sums will be used with following tests which makes the reports more manageable sizes by exclude unchanged files from the results.

3.6 Computer Oracle and Password System (COPS)

COPS is a collection of security tools designed to help UNIX system administrator, programmers and auditors working in the computer security area. Each of the security tools checks different problem areas of the UNIX security[12]. The program checks problems such as:

- File, directory, and device permissions/modes
- Poor passwords
- The content, format and security of the group and password files
- Program and files that runs in *etc/rc** and *cron/crontab* files
- CRC check against important binaries or key files
- Write permissions of user home directories and startup files

[10] http://usat.sourceforge.net/

[11] Michael Haughland & Magnus Wecksten (2014) linux Hoist Review: En undersökning av automatiserade auditverktyg, ss 7

[12] http://ftp.cerias.purdue.edu/pub/tools/unix/scanners/cops/cops.1.02.README

- Anonymous FTP setup

Like most security audit tools, it only warns the administrator about the potential problem of their system, it does not fix the problem. COPS should be used to tighten the security of a system, not as a weapon to find security flaws. It should be able to protect its users from their own ignorance and carelessness.

3.7 TrueSec's Data Gathering Tools

TrueSec employs two types of open source scripts and one collection of scripts that they have written themselves. The two open source scripts are Lynis and UPC which are mentioned above.

remote_job_linux_osx

remote_job_linux_osx is a collection of TrueSec's own data gathering tools. These scripts are used on running systems and gathers information such as running tasks, current users and last logins. This collection of scripts is not a security audit itself as it does not evaluate the retrieved information. The scripts can be executed on production machines without heavy penalties.

The script is run by executing the file **gathering_information.bash** using the following command in the terminal with optional extensions, to run as a root user you have to run it as a sudo command:

```
gathering_information.bash
```

These extensions are the following:

-h	help	Gives a list of these commands
-v	verbose	Produces output while the script is running
-d	debug	Outputs additional information to see which functions are run etc
-k	keep	Produces a directory with the results after run. (Otherwise it is tarballed).
-f	fork	Fork the process
-r	remote	Used when running the scripts on remote hosts

The script runs various bash commands and saves the output in corresponding folders. The folders are saved in a tarball named *result*[13]. Inside every folder there is a .log file where the output from each command is stored.

Evaluation of data

TrueSec's security health checks are usually primarily used for operational systems and services, internal networks and upon customer requests TrueSec will perform security health checks for lesser system management/maintenance.

[13] See Appendix C for more details about the folders

The TrueSec guidelines derives from field experience, selected parts from published security initiatives which includes, but not limited to: OpenSCAP and NIST SP 800. The guidelines may vary depending on the customer's environment and due to restrictions agreed upon between TrueSec and customers.

TrueSec grades each finding in order to assist in the analysis of the results in their document. Their grade system is primarily focused on the security. The aspects of general code quality and maintainability are assessed secondarily. The grade system has three different levels:

- Red: Severe security and maintainability issue. Should be addressed immediately.
- Yellow: Finding that does not affect the security or the maintainability but should be addressed as soon as possible.
- Blue: No need for immediate action. Could be viewed as a beauty flaw, that may be addressed if time permits.

3.8 The Web Service

Once the final report has been created this needs to be presented to the customer. TrueSec want this to be shown or downloadable through a web service. For starters this web service will just show the report itself but eventually it will be extended with an authentication model with log in and user roles so that it can be used for multiple reports. The web service will be independent of the analysis engine and therefore can be developed in parallel.

3.9 Conclusion Of Environmental Analysis

Since all of the mentioned auditing tools are open source these can be used for inspiration. TrueSec uses Lynis, UPC along with their own customizable scripts for their health checks, because of this the analysis engine will be developed with this in mind. It should be able to handle these audits but at the same time leave the possibility open to incorporate other audits in case of further development.

The solution for converting the logs to readable warnings and proposed actions will therefore assume the logs are from these tools. In order to correctly assess the severity of a potential security risk, comparisons with the TrueSec guidelines and previous reports will be made. To convert the reports and logs from the scripts to readable warnings the analysis engine will read and evaluate the given reports and logs to produce warnings and suggestions.

The different scripts differ, both in what they test and how their report and log files looks. This means that there may have to be different solutions for every file.

UPC and Lynis logs produces warnings easily found by searching the logs. Finding the issues found by these programs will therefore be simple. The Lynis logs also contains suggestions which can be used in the analysis engine.

Developing the analysis of the remote_job output however will be much more time consuming as it will have find the flaws itself without assistance from a security audit. For every flaw found a warning and possible suggestion should have to be printed to the *final report* produced by the analysis engine.

The contents of the final report will be presented on the web service.

4

Methodology

4.1 Workflow

At the beginning of the thesis a Gantt chart was made to estimate the time of development for the project (see Figure 4.1 below).

#	Activity	Dependency	Week
			5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
A1	Initial description		
G1	Initial description done		
A2	Environmental analysis (API and framework)	A1	
A3	Searching for literatures		
G2	Environmental analysis and literature searching done		
A4	Analysis and implementation of PoC	A2, A3	
G3	Analysis of PoC done		
G4	Implementation of PoC done		
A5	Webservice		
G5	Integration with webservice done	A4, A5	
A6	Thesis	A2,A3	
G6	Thesis done		
A7	Poster	A6	
G7	Poster done		
A8	Preperations for presentation	A6	
G8	Preperations for presentation done		
A9	Preperation for opposition		
G9	Preperation for opposition done		
A10	Report to supervisor		
A11	Report to examinator		
A12	Presentation		
A13	Opposition		

Figure 4.1 Preliminary Gantt chart

During the first phase of development the focus was on an *Environmental Analysis* and *searching for literature* (See chapter 3). In the environmental analysis other security audit systems such as, OpenSCAP, Linux Security Auditing Tool were analysed. However the ones chosen as a base for the analysis engine were the main security audit systems TrueSec were using, which are Lynis and UPC, as well as TrueSec's own remote_job_linux script collection. This was planned for week 7-9, however the environmental analysis needed modifications after the first handout to the supervisor and examinator. The modifications took longer time than expected

because of exams (not shown in the Gantt chart), thus overlapped and delayed the *Analysis and implementation of PoC*.

The writing process of the thesis began at the beginning of the project (see Figure 4.1). The writing process was parallel with the Implementation of PoC and the web service.

During the second phase of development the work was split. The setting up of the web service and the java implementation of the analysis engine was being worked in parallel. In this phase the learning process was also included. The security audit system scans Linux operating system, which meant learning a whole new operating system. In addition, the web service was developed using the Flask framework for front end and the python language for the back end.

During the third phase of development the focus was on further developing the web service and implementation of the analysis engine separately. During this phase the following was implemented: a login system, register system and the design of the web service.

During the fourth phase of development the decision was made to translate the code from Java to Python due to problems with integrating the java implementation with the rest of the system. The code translation was done simultaneously with the implementation of the web service's upload and download functions. After the code translation, YAML was integrated to make the code more modular and easier to make configurations without touching the python code.

During the fifth phase the analysis engine was integrated into the web service. The integration enabled users to upload log files from a web page and then run them through the analysis engine. The result from the analysis is then available as a download for the user.

Finally, during the last phase, documentation for the code and this thesis was completed.

4.2 Analysis Engine

The analysis engine uses output from three different scripts, *UPC, Lynis and remote_job_linux*. The output of this will be parsed and analysed by the analysis engine (see figure 4.2). A user can upload the log files to the web service, which

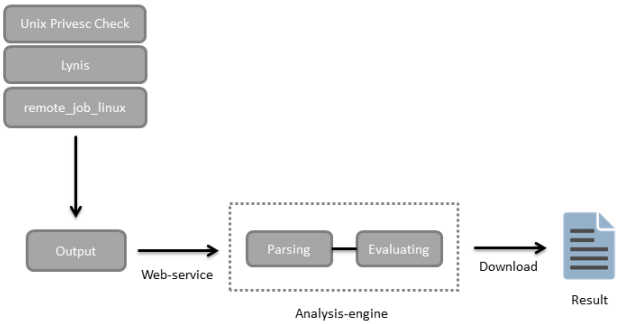

Figure 4.2 Prototype overview

sends them through the analysis engine and allows them to download the result of the analysis.

4.3 Parsing The Files

A class for each script(remote_job, UPC and Lynis) was developed, these classes will be referenced as parsers. The parsers were all based on the same idea. One read- and one evaluate function for each log file. The read function would take in a log file as an argument and output a dictionary. The evaluate function would then take in this dictionary as an argument and evaluate according to its contents and output a String which would then be appended to the final report.

The reason for using a dictionary is because of its key-value relationship. While it may have been better to implement it in a different way for some logs, the homogeneous structure was valued more, even more so as it may not have been worth the effort to figure out a different solution for each different log.

In case there is a desire to incorporate another audit into this analysis engine, it can easily be added by simply creating another class, how the analysis engine works and evaluates the log file is up for discussion as long as it outputs a string for the output.

remote_job

As expected the parser for remote_job required the most time to implement. For each of the log files given by remote job a function for reading the file and extracting

the important data was created as well as a corresponding evaluate function.

The following is an example of the *diskvolume_info* log and a short description on how the parser works.

```
Filesystem      Size  Used Avail Use% Mounted on
udev            3,9G     0  3,9G   0% /dev
tmpfs           788M  9,7M  779M   2% /run
/dev/sda1       450G  214G  214G  82% /
tmpfs           3,9G  4,5M  3,9G   1% /dev/shm
tmpfs           5,0M  4,0K  5,0M   1% /run/lock
tmpfs           3,9G     0  3,9G   93% /sys/fs/cgroup
tmpfs           788M  124K  788M   92% /run/user/1000
nfs             3,2G  2,8G  400M   91% /nfs
```

For this log, each mount point is considered a key. Each key has their own set of keys and values. The key being the column and the value being the value in the corresponding column. Parsing this results in a dictionary where you can traverse through each mount point and column to find the value, which later can be evaluated.

Lynis

The lynis log already provides warnings and suggestion, meaning we can simply just search for these. The following is a snippet from a Lynis log:

```
2017-05-16 21:34:05
 ↪  ===-----------------------------------===
2017-05-16 21:34:05 Performing test ID NETW-2705 (Check
 ↪  availability two nameservers)
2017-05-16 21:34:05 Result: less than 2 responsive
 ↪  nameservers found
2017-05-16 21:34:05 Warning: Couldn't find 2 responsive
 ↪  nameservers [test:NETW-2705] [details:-] [solution:-]
2017-05-16 21:34:05 Note: Non responsive nameservers can give
 ↪  problems for your system(s). Like the lack of recursive
 ↪  lookups, bad connectivity to update servers etc.
2017-05-16 21:34:05 Suggestion: Check your resolv.conf file
 ↪  and fill in a backup nameserver if possible
 ↪  [test:NETW-2705] [details:-] [solution:-]
2017-05-16 21:34:05 Hardening: assigned partial number of
 ↪  hardening points (1 of 2). Currently having 77 points
 ↪  (out of 109)
2017-05-16 21:34:05 ===-----------------------------------===
2017-05-16 21:34:05 Performing test ID NETW-3001 (Find
 ↪  default gateway (route))
```

```
2017-05-16 21:34:05 Test: Searching default gateway(s)
2017-05-16 21:34:05 Result: Found default gateway 192.168.2.1
2017-05-16 21:34:05
 ↪ ===----------------------------------------===
```

Unix Privesc Check

The UPC log already contains warnings, meaning we can simply search for these. The following is a snippet from a UPC log:

```
    Checking if anyone except root can change /t09
    Checking if anyone except root can change /tmp
WARNING: /usr/lib/xorg/Xorg is currently running as root.
 ↪ /usr/lib/xorg/Xorg contains the string /tmp. World write
 ↪ is set for /tmp (but sticky bit set)
    Checking if anyone except root can change /tmpf
    Checking if anyone except root can change /tmp/launch
WARNING: /usr/lib/xorg/Xorg is currently running as root.
 ↪ /usr/lib/xorg/Xorg contains the string /tmp/launch. World
 ↪ write is set for /tmp (but sticky bit set)
    Checking if anyone except root can change /u-
    Checking if anyone except root can change /udev
    Checking if anyone except root can change /usr
```

4.4 Evaluation

remote_job

In order to evaluate the logs we have decided to simply compare the values given with values that would be deemed unacceptable. If the given value is one of these a warning and possible solution should be provided. The top priority was find a way to make the design modular. To find out if there could be a risk and presenting a possible solution was a secondary to this. The reason for a modular design is that it is near impossible to create a reliable security audit tool in the time we had, along with the fact that modular design means the possibility of easily exanding upon the analysis engine in the future. Using YAML to store these unacceptable values along with their corresponding warning messages and possible solutions means that a future developer can add more unacceptable values without interacting with the code. Another reason for using YAML files is that they can be loaded into a dictionary which can be compared with the log files information, which is also stored in a dictionary.

crontab_at_info This log is generated via the command *ls -la /etc/cron.allow ; ls -la /etc/at.allow* that prints out the permissions for editing the files */etc/cron.allow* and */etc/at.allow*.

In the case where the files are not found the analysis engine spits out a string that tells the user that these files have not been set up.

In the case that one of these files can be edited by anyone the analysis engine spits out a string that tells the user that it would be a good idea to make sure that only the owner and group can write to this file.

crontab_info This log is generated via a shellscript that prints out all the users who have no crontab set up. The analysis engine spits out a string saying that there is no crontab set up for those users that are expected to have it. It also spits out a string saying that there is a crontab set up for users that are not expected to have it.

diskvolume_info This log is generated via the command *df -h*. This command lists all of the file systems and their disk usage. The analysis engine checks for high usage, to make sure the system has enough space. It also checks for and warns if a volume is a nfs system.

encrypted_disk_info This log is generated via the command *blkid /dev/sd**. This command lists all file systems and their UUID(Universally Unique Identifier). The analysis engine checks for shared UUIDs and spits out a warning message if found.

An UUID is a 128-bit that is meant to be unique and having shared UUIDs can lead to unwanted consequences. It is extremely highly unlikely to generate two identical UUIDs if generated appropriatly. Creating 100 billion UUIDs per second for 100 years puts the chance of duplicate UUIDs at around 50%. It is still important to check as they can be edited by malicious attackers.

environment_info This log is generated via the command *printenv*. This command lists all environmental variables and their values. Because of the simple layout(key=value), it was very easy to save it to the dictionary. The analysis engine simply searches each key in the corresponding YAML file and if the key is in the log file it will report a warning if certain values in the YAML file is found.

firewall_info This log is generated via the command */sbin/iptables -L -n*. This command lists the default policy(Accept, Drop, Reject) for each source of traffic(Incoming, Forwarding and Outgoing). For each source it also lists the IPs that goes against the default policy. The analysis engine reports which source of traffic has the default policy of Accept because it can be a clear security risk to accept all traffic.

groups_info This log is generated via the command *cat /etc/group*. This command lists all of the groups on the system and four associated fields per line.

1. **Groupname:** The name of the group

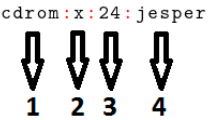

Figure 4.3 groups_info entry example

2. **Password:** The character x indicates that the user has a encrypted password-stored in the file/etc/shadow/ file. The character ! indicates that the user's password is stored unencrypted in /etc/security/passwd. The character * indicates that the user has an invalid password.

3. **Group ID:** The ID of the group.

4. A list of all the users in the group.

The analysis engine returns warnings if the second field (Password field) is set as "!" or a "*". It also warns if root has any group members.

lastlog_info This log is generated via the commands *last* and *lastlog*. The first command goes through a file */var/log/wtmp* and displays a list of all log in and log outs since the creation of that file. The second command shows a list of each users last login into tty. The analysis engine warns if predetermined users have logged in.

modprobe_info This log is generated via the command *ls /etc/modprobe.d ; lsmod*. The first command lists all the files in the */etc/modeprobe.d* folder while the second command lists the modules installed on the system. The analysis engine checks for important files in the */etc/modeprobe.d* folder and returns a warning if something is missing. It also warns if some of the found modules are predetermined as blacklisted, or if some predetermined modules are not found.

networkvolume_info This log is generated via the commands *mount* and *cat /etc/fstab*. The first command lists all the filesystems mounted according to the file */etc/mtab*. This output is in the pattern of:

<filesystem> on <mountpoint> type <filesystem_type> (<options>)

Example: ysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)

The second command lists the filesystems included in the */etc/fstab* file. This file is used to determine where filesystems should automatically be mounted. Each filesystem has its own line in the pattern of:

<file system> <mount point> <type> <options> <dump> <pass>.

Where each field means the following:

- **filesystem:** The device. Either the /dev location or the UUID for the device.

- **mount point**: Where the device should be mounted.

- **type**: The filesystem type.
- **options**: A list of options enabled for the filesystem
- **dump:** Weather to backup or not. 1 = backup, 0 = do not backup.
- **pass**: Controls the order in which fsck checks the device/partition for errors at boot time. The root device should be 1. Other partitions should be 2, or 0 to disable checking.

The analysis engine returns a warning if non-unique UUIDs has been found, if backing up is disabled, unusual options and making sure the error checking is set up according to the above.

open_connections_info This log is generated via the commands *ss -tulpan* and *lsof -i*. These commands provide an output of processes and how they are using certain ports. The connections to these ports can be set to different states including CLOSED, LISTEN and ESTABLISHED. The evaluation for this log has not been implemented.

passwdpolicy_info This log is generated via the command *cat /etc/login.defs*. The file *login.defs* defines the site specific configuration for the shadow password suite. To evaluate this log, the analysis engine can search for configuration items and evaluate them. The current engine searches for the following configuration items

- *ENCRYPT_METHOD*

If this item's value is set to MD5, the analysis engine will recommend the user to consider changing the encrypting method to SHA256 or SHA512.

- *PASS_MIN_DAYS*

The PASS_MIN_DAYS item defines the minimum of days allowed between password changes. This could be a security risk in case of a accidental password change, which is why a warning will appear and tell the user if PASS_MIN_DAYS has a greater value than zero.

- *PASS_MAX_DAYS*

PASS_MAX_DAYS defines the maximum days before users have to change their passwords. If a user fails to change the password the user will be disable. This will also disable inactive users. The recommend value for maximum days is 90 days. If the PASS_MAX_DAYS has a greater value than 90, the analysis engine spits out a warning suggesting changing the value to the recommended.

processes_info This log is generated via the command *ps aux*. This command lists proccesses for all users. Each process has its own line with the following fields:

- **USER** = User owning the process
- **PID** = Process ID of the process
- **%CPU** = It is the CPU time used divided by the time the process has been running.
- **%MEM** = Ratio of the process's resident set size to the physical memory on the machine
- **VSZ** = Virtual memory usage of entire process (in KiB)
- **RSS** = Resident set size, the non-swapped physical memory that a task has used (in KiB)
- **TTY** = Controlling tty (terminal)
- **STAT** = Multicharacter process state
- **START** = Starting time or date of the process
- **TIME** = Cumulative CPU time
- **COMMAND** = Command with all its arguments

The analysis engine checks for overwhelming CPU or memory usage. It also spits out a warning if it can or cannot find predetermined processes, either because of the process itself being deemed dangerous or if it is run by a user that should not run the process. It also tries to find out if a password can be found in the command and in that case it spits out a warning.

samba_info This log is generated via the command *egrep -v ' ^*#|^$|^;' /etc/samba/smb.conf*. Samba runs on Unix platforms, but it can speak to windows platforms. This allows Unix system to move into the Windows network territory without making a lot of confusion. Windows system can access file and print services without knowing that it is being offered by a Unix system.

To be at least moderately secure, the analysis engine must look at the perimeter firewall and the configuration of the host server that is running Samba and Samba itself. First off, the analysis engine will search for following lines:

- **hosts allow**
- **hosts deny**

The analysis engine checks if the hosts are allowing internal network e.g. localhost. Samba accepts connections from any hosts by default, which means the server is on a host that is directly connected to Internet. To prevent this, the analysis engine will give a warning if it does not find a line with *host deny = 0.0.0.0/0*.

It is also a good idea to restrict access to the Samba server by using:

- valid users = @lth, jesper

This line will restrict all server access either to the group members of lth or the user jesper.

By default the Samba accepts connection on any network interface that it finds on the system, which means Samba will accept connections on those links if the system uses a ISDN line or a PPP connection. The analysis engine will search for following lines:

- interfaces = eth* lo
- bind interfaces only = yes

If the lines does not exist the analysis engine will give a warning and suggestion to add those lines.

sshd_info This log is generated via the command *cat /etc/ssh/sshd_config*. *sshd_config* is a systemwide configuration file for OpenSSH which allows the user to set options that modify the daemon. The file contains keyword-value pairs each line. By default user can SSH to the server as root. This is not very secure, it is recommended to log in to the system using a non-root user and then do '-su' to log in as root. This is why the analysis engine searches for following line:

- ***PermitRootLogin***

if this key-value is set to 'yes' then the analysis engine gives a warning as this will allow multiple sysadmins to log in to the server as root and the system might not know which sysadmins are logged in as root. If the key-value is set to 'no' the sysadmins have to log in to the system first using their accounts before they can do '-su'. The analysis engine also checks the port variable:

- ***Port***

The default port is 22, which most attackers will check when they are trying to brute force log into the server using several username and password combinations. The analysis engine suggests that the system should use another port to log in to the server. A drawback with changing the port is that everyone in the team have to know the both the port and IP address to be able to log in to the server.

- ***LoginGraceTime***

This key-value specifies how long the server will have to wait in *seconds* before disconnecting after a unsuccessful login connect request. The default value of this is usually 600 seconds, which is a very long time. The analysis engine suggests the user to change it to be between 60 and 120 seconds.

- ***ListenAddress***

An entry address like 0.0.0.0 means it listen to all interfaces, even external ones. The analysis engine suggests changing the address to a internal one, so that the server cannot be accessed from the Internet unless the system has port forwarded on the system routing.

- ***StrictModes***
- ***RSAAuthentication***
- ***PasswordAuthentication***

The above three key-values must be set to 'yes' for obvious security reasons. The *StrictModes* specifies if whether the server should check the users permission home directory and rhost before they can log in because some users may accidentally leave their directory or files world writable.

RSAuthentication should be set to yes to be able to use public and private key pairs created by the ssh-keygen1utility for authentication purposes.

PasswordAuthentication specifies if the system should use password based authentication, which is a very strong security and should always be set to 'yes'.

There are three key-values that should explicityly be set to "no".

- ***AllowTcpForwarding***
- ***X11Forwarding***
- ***AllowAgentForwarding***

When the key-values are set to "no", *AllowTcpForwarding* will disable attackers using SSH as a VPN like tunnel. *X11Forwarding* disable a remote X window or XShell to be used by an attacker. *AllowAgentForwarding* will disable the option to jump from one system to another, using only SSH.

startup_info This log is generated via the command *ls -alR /etc/init.d**. The output contains a number of start/stop scripts for various services on a system.

The services and servers that runs constantly, have to write, read data and interact with other unknown or untrusted data owned by somebody than root should have its own user. The analysis engine goes through the permissions of all the scripts and if it finds that a script can be written to as non-root it spits out a warning.

sudoers_info This log is generated via the command *cat /etc/sudoers*. The file *sudoers* contains rules that a user must follow when using the command -sudo, that means in order to use sudo, one must configure the **/etc/sudoers/** file first. The first lines in the file:

Defaults env_reset
Defaults secure_path="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"

This is a safety measure to make sure the terminal environment remove any user variables and clear potentially harmful environmental variables from the sudo session. The analysis engine checks if this variable is set to env_reset every time it evaluates this file.

It will also check if the *secure_path* is correct. *secure_path* specifies the place with applications in the file system the operating system have to find, that will be used in the sudo operation. This will prevent harmful user pathings.

There is also a line like this:

root ALL=(ALL:ALL) ALL

The first field indicates that the rule will apply to *root*. The first ALL indicates that this rules applies for all hosts. The second ALL indicates that the root user can run commands as all users. Third ALL indicates the root user can run commands as all groups. The last ALL means that all rules applies to all commands. In other words, this means that the root user can run any command using sudo as long as the correct password is provided.

To evaluate this, the analysis engine returns a warning if it finds other users or groups with unrestricted privileges.

system_info This log is generated via the command *cat /proc/version*. The file *version* gives specific information about the current version of Linux and the version of GCC compiler used to build the system. The log generally looks like following:

Linux version 4.8.0-41-generic (buildd@lgw01-18) (gcc version 6.2.0 20161005 (Ubuntu 6.2.0-5ubuntu12)) #44-Ubuntu SMP Fri Mar 3 15:27:17 UTC 2017

The information given here is:

1. Exact version of the Linux kernel: **Linux version 4.8.0-41-generic**
2. Name of the user and host name who compiled the kernel **buildd@lgw01-18**
3. Version of the GCC compiler used for building the kernel: **gcc version 6.2.0 20161005**
4. Kernel type: **SMP**[1]

[1] Symmetric MultiProcessing kernel

5. Date and time when the kernel was built: **Fri Mar 3 15:27:17 UTC 2017**

The analysis engine returns a warning if the customer's kernel version is outdated.

users_info This log is generated via the command *cat /etc/passwd/*. The file *passwd* stores the essential information required during login such as user account information. It lists the system's accounts and information such as user ID, group ID etc. Generally a entry in the passwd file looks as following:

Figure 4.4 users_info entry example

1. **Username**: This is used when user logs in.

2. **Password**: The character x indicates that the user has a encrypted password stored in the file */etc/shadow/ file*.

3. **User ID**: Each user are assigned a user ID.

4. **Group ID**: The group which the user is assigned to.

5. **User ID info**: The comment field for extra information about the user, such as full name etc.

6. **Home directory**: User's home directory.

7. **Command/Shell**: This does not have to be a shell, but it is the absolute path of a command/shell.

The evaluation will be focused on the third field (User ID field). The analysis engine checks if the field is set to 0, which means that the user have super user rights and that could be a security exploit.

The analysis engine also gives warnings if the second field (Password field) is set with a "!" or a "*". The "!" indicates that the user's password is stored in */etc/security/passwd* and is not encrypted. The asterisk (*) indicates that the user has an invalid password.

The analysis engine also spits out a warning if predetermined users do not have either */bin/false* or */usr/sbin/nologin* in the Command/Shell field, in the case that they should not have terminal access.

Lynis

Lynis already provides possible solutions for their warnings and can therefore be easily added to our solution.

Unix Privesc Check

UPC provides a warning and a reason as to why it could be a security risk, however it does not provide a solution.

5

Implementation

5.1 Parsing & Analysis Engine

The analysis engine is started using the python module *evaluate.py* function *evaluate* by the web service, this integration is explained in more detail in the subsection 5.2. This module first runs all the log files through their read- and evaluate functions. The read function takes a log file as an argument and returns a dictionary. The evaluate function takes this dictionary and evaluates it and then returns a string including all issues it has found, which is then added to the string that will be written to the output file.

In order to explain how the engine works, the path of a few log files to its final destination in output.txt will be explained. Because there are a total of 20 log files from the remote-job audit only a few will be explained as they are quite similar anyway.

cron_at

```
1 ls: cannot access '/etc/cron.allow': No such file or
   ↪ directory
2 -rw-r--rw- 1 jesper jesper  244 maj  6 20:02 /etc/at.allow
```

Looking at this log one can see that the file *cron.allow* could not be found. The permissions for the file *at.allow* is also something that should be evaluated.

```
1 def read(file):
2     info_dict = dict()
3
4     next_line = file.readline()
5
6     while next_line:
7
8         inner_values = next_line.split()
```

```
9
10          if "No such file or directory" in next_line:
11              info_dict[inner_values[3][1:-2]] = ["No such file
                ↪  or directory"]
12
13          else:
14                  info_dict[inner_values[8]] = inner_values[0]
15
16          next_line = file.readline()
17
18      return info_dict
```

In this function a dictionary(info-dictionary) is declared. Each line of the log is then read. If a line contains "No such file or directory", a tuple will be added into the dictionary where the filename is the key and "No such file or directory" is the value.

If instead the file was found, the dictionary instead adds the permissions as the value to the dictionary under the filename as a key.

```
1  '/etc/cron.allow':
2    'neq':
3      'values':
4        'No such file or directory':
5          'severity': YELLOW
6          'msg': No /etc/cron.allow has been set up.
7
8    'permissions':
9      'other':
10       'w':
11         'severity': RED
12         'msg': Any user can edit /etc/cron.allow.
13
14
15 '/etc/at.allow':
16   'neq':
17     'values':
18       'No such file or directory':
19         'severity': YELLOW
20         'msg': No /etc/at.allow has been set up.
21
22   'permissions':
23     'other':
24       'w': #in code: checks if w is in the last three
         ↪  permissions [rwx]
```

```
25          'severity': RED
26          'msg': Any user can edit /etc/at.allow.
```

In order to explain the evaluate function, the corresponding YAML file will first be explained.

The YAML file contains several predetermined values. The keys under each filename-key reflects how their values should be compared to the value found in the log. The values under "neq" contains the severity and message that should be returned if one of the values are found in the dictionary created in the read function(info-dictionary). As for "permissions", it contains up to three sub-keys(permission groups): user, group and other. In this case it only contains "other" because the analysis engine only cares if users other than the group or owner can write to the file. Under the "other" key we find the values that should trigger a warning message, in this case "w".

```
1  def evaluate(info_dict, yaml_path):
2      return_string = ""
3
4
5      with open(yaml_path, "r") as stream:
6          yaml_dict = yaml.load(stream)
7
8
9      for file_name in yaml_dict:
10         if info_dict.has_key(file_name):
11             info_value = info_dict[file_name]
12
13             for comparison in yaml_dict[file_name]:
14                 yaml_values =
                       ↪ yaml_dict[file_name][comparison]
15                 message = compare(info_value, yaml_values,
                       ↪ comparison)
16                 if message is not None: return_string +=
                       ↪ message + "\n"
17
18     return return_string
```

The first thing the evaluate function does is creating a dictionary(YAML dictionary) from the corresponding YAML file. The function then loops through each key in the YAML dictionary, in this case corresponding to the files. If the info-dictionary contains a key it will then set the *customer_value* to its value; *"No such file or directory"* or the permissions for the file. For each comparison in the YAML dictionary it

then compares the customer_value to the declared values in the YAML file according to which comparison it is, and then returns the message returned by the compare function explained below.

```
def compare(customer_value, values, comparison):

    if comparison == "neq":
        values = values["values"]

        if customer_value in values.keys():
            message = values[customer_value]["msg"]
            severity = values[customer_value]["severity"]
            return message


    if comparison == "permissions":
        for permission_group in values:
            if permission_group == "other":
                other_rwx = customer_value[7:]
                for permission in values[permission_group]:
                    if permission in other_rwx:
                        message = values[permission_group]
                         ↪  [permission]["msg"]
                        return message

    pass
```

As for comparing "neq": If the value from the info-dictionary under the key "/etc/cron.allow"(customer_value) is found in the YAML keys within the key "values", which in this example is just "No such file or directory", the corresponding warning message will be returned ("No /etc/cron.allow set up.").

As for comparing "permissions": The function loops through each permission group in the YAML file, in this example only being "other". It then loops through each permission(read, write, execute) in the YAML file, in this example only being "w". If it turns out that the 3 last digits(permissions of others) of the original permissions value contains "w" it will return the corresponding warning message.

This code snippet only showed the code that is needed to explain this example. There are more types of comparisons that have been implemented, namely:

neq	returns warning if customer_value is specified in the YAML file.
permissions	returns warning if customer_value contains the permissions specified in the YAML file.
ngr	returns warning if customer_value is greater than specified in the YAML file.
nlt	returns warning if customer_value is lesser than specified in the YAML file.
nbtwn	returns warning if customer_value is between values specified in the YAML file.
eq	returns warning if customer_value is NOT specified in the YAML file.

The string returned to the *evaluate* function in *evaluate.py* is now:

```
No /etc/cron.allow has been set up.
Any user can edit /etc/at.allow.
```

This log file has now been evaluated and the *evaluate.py* now runs the next log file.

group

```
root:x:0:john
daemon:x:1:
bin:x:2:
sys:x:3:
adm:x:4:syslog,jesper
tty:x:5:
...
fax:x:21:
voice:*:22:
cdrom:!:24:jesper
floppy:x:25:
```

Looking at this log one can see three security issues. First of all, the root group has a member called "john", the root group should not have any members. The second issue is that the password field for the cdrom group contains "!", which indicates that the password is unencrypted. The final issue is that the password field for the voice group contains "*", which indicates that the password is invalid.

```
def read(file):
    info_dict = dict()
```

```
3
4       next_line = file.readline()[:-1]
5
6       while next_line:
7           inner_dict = dict()
8           inner_values = next_line.split(":")
9           inner_dict["group"] = inner_values[0]
10          inner_dict["password"] = inner_values[1]
11          inner_dict["id"] = inner_values[2]
12          inner_dict["users"] = inner_values[3]
13
14          info_dict[inner_dict["group"]] = inner_dict
15          next_line = file.readline()[:-1]
16
17
18      return info_dict
```

In this function a dictionary(info-dictionary) is first declared. The log is then read line by line. For each line another dictionary is declared(inner-dictionary). The line is then split between each colon and the inner-dictionary is filled with each value for each field(key). The info-dictionary then saves this inner-dictionary within itself as a value using the group name as a key. The read function then returns the info-dictionary.

```
1   'default':
2     'password':
3       'neq':
4         'values':
5           '!':
6             'severity': RED
7             'msg': The password for /group/ is invalid.
8           '*':
9             'severity': RED
10            'msg': The password for /group/ is not encrypted
              ↪  and is stored in /etc/security/passwd
11
12  'root':
13    'users':
14      'eq':
15        '':
16          'severity': RED
17          'msg': The users /users/ are in the root group.
```

```
18
19    'password':
20      'neq':
21        'values':
22          '!':
23            'severity': RED
24            'msg': The password for root is invalid.
25          '*':
26            'severity': RED
27            'msg': The password for root is not encrypted and
                ↪  is stored in /etc/security/passwd
```

In order to explain the evaluate function, the corresponding YAML file will first be explained.

This YAML file is built very similar to the previous YAML file, one difference being an additional key before the comparison key. This key represents each field(group name, password, group id, users). The comparisons are in this case "neq" and "eq". Another thing that is new is the "default" key. The comparisons and values under the "default" key are used to evaluate all the keys(groups) not listed. One interesting thing are the "/group/" and "/users/" sub-strings which will be explained shortly.

```
1   def evaluate(info_dict, yaml_path):
2       return_string = ""
3
4       with open(yaml_path, "r") as stream:
5           yaml_dict = yaml.load(stream)
6
7       default_dict = yaml_dict.pop("default")
8
9       for key in yaml_dict:
10          if info_dict.has_key(key):
11              for column in yaml_dict[key]:
12                  info_value = info_dict[key][column]
13                  for comparison in yaml_dict[key][column]:
14                      yaml_values =
                            ↪  yaml_dict[key][column][comparison]
15                      message = compare(info_value,
                            ↪  yaml_values, comparison)
16                      if message is not None:
17                          message = message.replace("/users/",
                                ↪  info_dict[key]["users"])
```

```
                        message = message.replace("/group/",
                         ↪  info_dict[key]["group"])
                        return_string += message + "\n"

    for key in info_dict:
        for column in default_dict:
            info_value = info_dict[key][column]
            for comparison in default_dict[column]:
                yaml_values =
                 ↪  default_dict[column][comparison]
                message = compare(info_value, yaml_values,
                 ↪  comparison)
                if message is not None:
                    message = message.replace("/users/",
                     ↪  info_dict[key]["users"])
                    message = message.replace("/group/",
                     ↪  info_dict[key]["group"])
                    return_string += message + "\n"

    return return_string
```

The first thing the evaluate function does is creating a dictionary (YAML dictionary) from the corresponding YAML file. The YAML dictionary is then split using the *pop* function. The new dictionary(defaults-dictionary) is created using the function *pop* which returns the dictionary inside the "defaults" key and removes it from the YAML dictionary.

The function now goes through each key in the YAML emulator and goes through each field(group name, password, group id and users). It then compares the info-dictionary's field values with the YAML dictionary's field values. The function then replaces the keywords in the returned warning message with the correct value. It then appends the returned warning message to the return_string.

Once it has gone through all the keys in the YAML dictionary it compares info-dictionary with the defaults-dictionary in the same fashion.

Finally the function returns the return_string with all the warnings to *evaluate.py*.

sshd

```
# This is ssh server systemwide configuration file.

Port 22
ListenAddress 0.0.0.0
HostKey /etc/ssh/ssh_host_key
```

```
ServerKeyBits 1024
LoginGraceTime 35
KeyRegenerationInterval 3600
PermitRootLogin yes
IgnoreRhosts yes
IgnoreUserKnownHosts yes
StrictModes no
X11Forwarding yes
PrintMotd yes
SyslogFacility AUTH
LogLevel INFO
RhostsAuthentication no
RhostsRSAAuthentication no
RSAAuthentication n
PasswordAuthentication yes
PermitEmptyPasswords no
AllowUsers admin
```

Looking at this log one can see the simple key-value relationship structure, with multiple values not being considered therefore the read function is very simple.

```
def read(file):
    info_dict = dict()

    next_line = file.readline()

    while (next_line):

        if "No such file or directory" in next_line:
            info_dict["/etc/ssh/sshd_config"] = "No such file
             ↪  or directory"

        if "#" in next_line or next_line.isspace():
            next_line = file.readline()
            continue

        next_values = next_line.split()

        info_dict[next_values[0]] = next_values[1]

        next_line = file.readline()
```

```
20
21      return info_dict
```

In this function a dictionary(info-dictionary) is first declared. The log is then read line by line with each key-value relationship being added to the info-dictionary. The lines containing a "#" characters are considered comments and are therefore ignored.

```
1  'LoginGraceTime':
2    'nlt':
3      'value': 60
4      'severity': YELLOW
5      'msg': LoginGraceTime is less than 60
6    'ngr':
7      'value': 120
8      'severity': RED
9      'msg': LoginGraceTime is greater than 120
10
11
12 'PermitRootLogin':
13   'neq':
14     'values':
15       'yes':
16         'severity': RED
17         'msg': PermitRootLogin is set to "yes" this will
           ↪ allow multiple sysadmins to login to the server
           ↪ as root and the system might not know which
           ↪ sysadmins are logged in as root. You should
           ↪ change PermitRootLogin to "no" so the sysadmins
           ↪ have to login to the system first using their
           ↪ accounts before they can do "-su".
18
19 'Port':
20   'neq':
21     'values':
22       '22':
23         'severity': YELLOW
24         'msg': The default port is set to 22, which the most
           ↪ attackers will check when they are trying to
           ↪ brute force login to the server using several
           ↪ username and password combinations. You should
           ↪ consider using another port to login to the
           ↪ server to reduce noise.
```

```
25
26    'nbtwn':
27        'values':
28            'Do not use the ports 30-60 or 2-5 for ssh':
              ↪  #MESSAGE FOR BELOW RANGES
29              'ranges':
30                - [30, 60]
31                - [2, 5]
32              'severity': RED
33
34            'Do not use the ports 65-68 or 7-10 for ssh':
35              'ranges':
36                - [65, 68]
37                - [7, 10]
38              'severity': YELLOW
```

In order to explain the evaluate function, the corresponding YAML file will be explained first.

This YAML file contains a high number of predetermined values and their associated warning messages. As for the "nbtwn"(not between) comparison, the way to define which ranges should trigger which warning message is much more different and less intuitive than the others, this is because the key can not have multiple values. The keys under *values* are the messages that should be returned in case the info-dictionary had a value within the ranges inside the message-key. The first number being the minimum in the range and the second being the maximum.

```
1  def evaluate(info_dict, yaml_path):
2      return_string = ""
3
4      with open(yaml_path, "r") as stream:
5          yaml_dict = yaml.load(stream)
6
7      for key in yaml_dict:
8          if info_dict.has_key(key):
9              info_value = info_dict[key]
10             yaml_values = yaml_dict[key]
11
12             for comparison in yaml_values:
13                 yaml_value = yaml_values[comparison]
14                 message = compare(info_value, yaml_value,
                   ↪  comparison)
15                 if message is not None:
```

```
16                      return_string += message + "\n"
17
18      return return_string
```

The first thing the evaluate function does is creating a dictionary(YAML dictionary) from the corresponding YAML file. The function then goes through each variable present in each of the dictionaries and compares them accordingly. The error messages are then sent back to *evaluate.py*

Lynis

Because the Lynis audit only output a single log file with a very simple way to find errors it was a much more simple task to implement. Only two functions were required, one for reading and extracting the warnings and suggestions and one for evaluating and returning a string to the output. The following is a snippet of the log and a description on how we parse it.

```
1  2017-05-16 21:34:05
    ↪  ===----------------------------------------===
2  2017-05-16 21:34:05 Performing test ID NETW-2705 (Check
    ↪  availability two nameservers)
3  2017-05-16 21:34:05 Result: less than 2 responsive
    ↪  nameservers found
4  2017-05-16 21:34:05 Warning: Couldn't find 2 responsive
    ↪  nameservers [test:NETW-2705] [details:-] [solution:-]
5  2017-05-16 21:34:05 Note: Non responsive nameservers can give
    ↪  problems for your system(s). Like the lack of recursive
    ↪  lookups, bad connectivity to update servers etc.
6  2017-05-16 21:34:05 Suggestion: Check your resolv.conf file
    ↪  and fill in a backup nameserver if possible
    ↪  [test:NETW-2705] [details:-] [solution:-]
7  2017-05-16 21:34:05 Hardening: assigned partial number of
    ↪  hardening points (1 of 2). Currently having 77 points
    ↪  (out of 109)
8  2017-05-16 21:34:05 ===-----------------------------------===
9  2017-05-16 21:34:05 Performing test ID NETW-3001 (Find
    ↪  default gateway (route))
10 2017-05-16 21:34:05 Test: Searching default gateway(s)
11 2017-05-16 21:34:05 Result: Found default gateway 192.168.2.1
12 2017-05-16 21:34:05
    ↪  ===----------------------------------------===
```

Looking at this log one can see all the tests and their results, warnings, notes, suggestions and hardening score as well as possible solutions and details of the warnings and suggestions. Our implementation only takes into account the tests warnings and suggestions as well as their associated details and solutions.

```
def read(file):
    warning_dict = dict()
    suggestion_dict = dict()

    next_line = file.readline()

    while next_line:

        if "Warning:" in next_line:
            start_index = next_line.find("W")
            end_index = next_line.find("[")
            warning = next_line[start_index:end_index-1]
             ↪  .replace("Warning: ", "")

            start_index = next_line.find("[test:")
            end_index = next_line.find("]")
            test = next_line[start_index +
             ↪  len("[test:"):end_index]

            next_line = next_line[end_index+2:]
            start_index = next_line.find("[details:")
            end_index = next_line.find("]")
            details = next_line[start_index +
             ↪  len("[details:"):end_index]

            next_line = next_line[end_index+2:]
            start_index = next_line.find("[solution:")
            end_index = next_line.find("]")
            solution = next_line[start_index +
             ↪  len("[solution:"):end_index]

            inner_dict = dict()
            inner_dict["warning"] = warning
            inner_dict["details"] = details
            inner_dict["solution"] = solution

            warning_dict[test] = inner_dict

```

```
35          elif "Suggestion:" in next_line:
36              start_index = next_line.find("S")
37              end_index = next_line.find("[")
38              suggestion = next_line[start_index:end_index-1]
                ↪  .replace("Suggestion: ", "")
39
40              start_index = next_line.find("[test:")
41              end_index = next_line.find("]")
42              test = next_line[start_index +
                ↪  len("[test:"):end_index]
43
44              next_line = next_line[end_index+2:]
45              start_index = next_line.find("[details:")
46              end_index = next_line.find("]")
47              details = next_line[start_index +
                ↪  len("[details:"):end_index]
48
49              next_line = next_line[end_index+2:]
50              start_index = next_line.find("[solution:")
51              end_index = next_line.find("]")
52              solution = next_line[start_index +
                ↪  len("[solution:"):end_index]
53
54              inner_dict = dict()
55              inner_dict["suggestion"] = suggestion
56              inner_dict["details"] = details
57              inner_dict["solution"] = solution
58
59              suggestion_dict[test] = inner_dict
60
61          next_line = file.readline()
62
63      info_dict = dict()
64      info_dict["warnings"] = warning_dict
65      info_dict["suggestions"] = suggestion_dict
66      return info_dict
```

From this log, two dictionaries are created. One for warnings and one for suggestions. Going through each line in the log the parser searches for the keywords "Warning: " and "Suggestion: ". The line is then made into a dictionary(inner_dict) and put into the warning- or suggestion-dictionary. In the case of the fourth line, the line is made into a dictionary with the keys and values:

warning	Could not find 2 responsive nameservers
details	None
solution	None

The two warning- and suggestion-dictionary are then combined into the info-dictionary which gets returned.

```
def evaluate(info_dict, yaml_path):

    return_string = ""

    if info_dict.has_key("warnings"):
        return_string += "The Lynis audit has found the
         ↪  following warnings: \n\n"
        for test in info_dict["warnings"]:
            inner_dict = info_dict["warnings"][test]
            warning = inner_dict["warning"]
            details = inner_dict["details"]
            solution = inner_dict["solution"]

            return_string +=  "The test " + test + " has
             ↪  found the following warning:\n" + warning +
             ↪  "\n" + "details: " + details + "\n" +
             ↪  "solution: " + solution + "\n\n"

    return_string += "\n##########################\n\n\n"

    if info_dict.has_key("suggestions"):
        return_string += "The Lynis audit has found the
         ↪  following suggestions: \n\n"
        for test in info_dict["suggestions"]:
            inner_dict = info_dict["suggestions"][test]
            suggestion = inner_dict["suggestion"]
            details = inner_dict["details"]
            solution = inner_dict["solution"]

            return_string +=  "The test " + test + " has
             ↪  found the following suggestion:\n" +
             ↪  suggestion + "\n" + "details: " + details +
             ↪  "\n" + "solution: " + solution + "\n\n"

    return return_string
```

Using this dictionary, the evaluate function simply adds a warning or suggestion message for each test and their information in the info-dictionary to the string later returned to *evaluate.py*

Unix Privesc Check

The implementation of the parser for the UPC was the least time consuming and most simple since simply searching the log for "Warning:" will provide a line with the warning. Only two functions were required, one for reading and extracting the warnings and suggestions and one for evaluating and returning a string to the output.

```
    Checking if anyone except root can change /t09
    Checking if anyone except root can change /tmp
WARNING: /usr/lib/xorg/Xorg is currently running as root.
 ↪ /usr/lib/xorg/Xorg contains the string /tmp. World write
 ↪ is set for /tmp (but sticky bit set)
    Checking if anyone except root can change /tmpf
    Checking if anyone except root can change /tmp/launch
WARNING: /usr/lib/xorg/Xorg is currently running as root.
 ↪ /usr/lib/xorg/Xorg contains the string /tmp/launch. World
 ↪ write is set for /tmp (but sticky bit set)
    Checking if anyone except root can change /u-
    Checking if anyone except root can change /udev
    Checking if anyone except root can change /usr
```

Looking at this log one can see all the tests and their possible warnings. Our implementation simply returns back the warning messages.

```
def read(file):
    info_dict = dict()

    next_line = file.readline()

    while next_line:
        if "WARNING:" in next_line:
            warning = next_line.replace("WARNING: ", "")[:-1]
            if info_dict.has_key("warnings"):
                info_dict["warnings"].append(warning)
            else:
                info_dict["warnings"] = [warning]

        next_line = file.readline()
    return info_dict
```

Similar to the Lynis parser, this parser searches for the keyword "WARNING: ". For each line containing "WARNING: " it adds the rest of the line as both key and value to a warning-dictionary. It would seem possible to just simply add all the warnings into a list instead, but we chose to use a dictionary because as previously stated we wanted the analysis engine to be as homogenous as possible, as well as the fact that in the future one might want to split up the warning message to a dictionary. It also effectively removes possible duplicate warnings as there can not be two keys with the same key name.

```
1  def evaluate(info_dict, yaml_path):
2
3      return_string = ""
4
5      if info_dict.has_key("warnings"):
6          return_string += "The unix audit has found the
           ↪  following warnings:\n\n"
7
8          for warning in info_dict["warnings"]:
9              return_string += warning + "\n"
10
11     return return_string
```

The evaluate function then takes this info-dictionary and appends each warning to the string that gets returned to *evaluate.app*.

5.2 Web Service

The web service was made with the web framework Flask. Flask is a microframework that uses python as its programming language. This service imports the analysis engine which also was written in python. The web service has a fully working authentication system with login and roles. It also has a storage system for uploading the log files and downloading the output file.

As the main focus of this thesis is on the analysis engine, the implementation of the web service itself will not be very detailed and most of the section will be about the integration with the analysis engine.

The MVC Model

The microframework Flask does not give the user any patterns and APIs to do MVC like applications, the user will have to make the routing with controllers to make actions work. Apart from the components found in MVC(*model, view, controller*) the web service also incorporates *routes*.

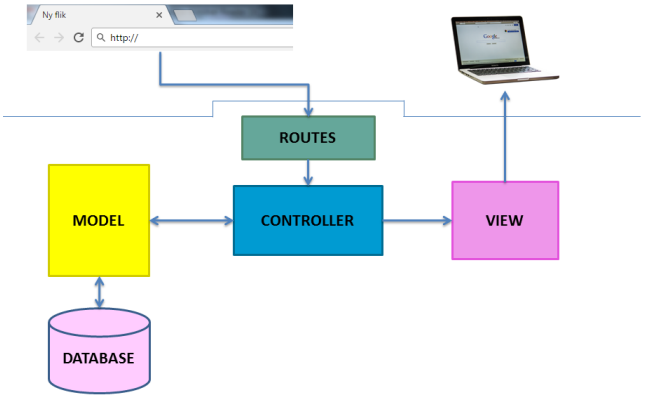

Figure 5.1 Model-View-Controller

The web service user requests to view a page by entering a URL. The application will attempt to search for a routing that matches and if it is successful it calls the action associated with that route. For example, if a user enters the URL localhost:5000/about/, Flask will search for a route for "/about/". It then runs the associated function.

```
@blueprint.route('/about/')
def about():
    """About page."""

    return render_template('public/about.html')
```

The code above is an example of routing in the web service. The route '/about/' is associated with the *about* function, which renders the template 'about.html'.

```
@blueprint.route('/', methods=['POST'])
def home_post():
    """Home page."""
    login_form = LoginForm(request.form)
    register_form = RegisterForm(request.form)
    # Handle logging in
    if request.method == 'POST':
        if login_form.validate_on_submit():
            login_user(login_form.user)
            flash('You are logged in.', 'success')
            redirect_url = request.args.get('next') or
             ↪  url_for('user.members')
```

```
            return redirect(redirect_url)
        elif register_form.validate_on_submit():
            User.create(
                username=register_form.username.data,
                email=register_form.email.data,
                password=register_form.password.data,
                active=True
            )
            flash('Thank you for registering. You can now log
             ↪  in.', 'success')
            return redirect(url_for('public.home_get'))
        else:
            flash_errors(login_form)
    return render_template('public/home.html',
     ↪  login_form=login_form, register_form=register_form)
```

The code above is an example of how a controller and model works together in the web service. The routing of this code is '/'(the front page), which is associated with the *home_post* view function. Another optional parameter of the route is "*method=['POST']*" which means that this function will make a "POST" request to send data to the database. Depending on whether a user tried to log in or register the controller will tell the model(the *User* class) to do different things. In case of a login it simply calls the login function with the information entered in the login form. In the case of a registration, the controller tells the model to create a user with the information entered in the register form. If the route is used without the "POST" request, the controller instead returns a view where the user can log in or register.

```
@blueprint.route('/', methods=['POST'])
def home_post():
    """Home page."""
    login_form = LoginForm(request.form)
    register_form = RegisterForm(request.form)
    # Handle logging in
    if request.method == 'POST':
        if login_form.validate_on_submit():
            login_user(login_form.user)
            flash('You are logged in.', 'success')
            redirect_url = request.args.get('next') or
             ↪  url_for('user.members')
            return redirect(redirect_url)
        elif register_form.validate_on_submit():
            User.create(
```

```
15                  username=register_form.username.data,
16                  email=register_form.email.data,
17                  password=register_form.password.data,
18                  active=True
19              )
20              flash('Thank you for registering. You can now log
                ↪  in.', 'success')
21              return redirect(url_for('public.home_get'))
22          else:
23              flash_errors(login_form)
24      return render_template('public/home.html',
        ↪  login_form=login_form, register_form=register_form)
```

This code is an example of a view in the web service (A view in the web service is HTML code together with Jinja2). This is the view of the front page of the web service where a user that is not logged in will be presented with, this view presents a form where a user can either log in or register. Jinja2 is used to extend the website with other HTML code located in other files. When a user has entered their information and clicks either "Sign in" or "Create an account" they are once again redirected to the front page, this time with a POST request.

Server and client

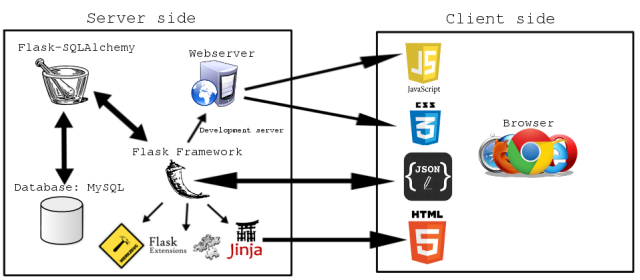

Figure 5.2 Server-Client Relation

The figure 5.2 illustrates the foundation of the web service. MySQL database was used for the web service. With the help of Flask-SQLAlchemy and Flask Framework the server side can render the data into a viewable format. However the framework and database cannot deliver the rendered website to the reader, it goes through a webserver. The webserver's job is to send data to the client side. Jinja2 embeds the data into other formats that can be used by a browser to display the website.

The Flask framework uses JSON to send data between the browser on the client side and the server. On the server side, Flask takes requests and queries the database using Flask-SQLAlchemy to retrieve data that sends back to the client.

MySQL Database

The database contains two tables. A *users* and *roles* table. A user can only have one role and a role can have zero to many users, thus a zero to many relation (see Figure 5.3).

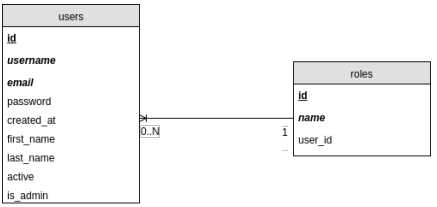

Figure 5.3 Entity Relation Diagram for web service

users table The primary key for this table is *id*. There are two foreign keys: *username* and *email*.

Field	Type	Key
id	int(11)	PRIMARY
username	varchar(80)	FOREIGN
email	varchar(80)	FOREIGN
password	blob	
created_at	datetime	
first_name	varchar(30)	
last_name	varchar(30)	
active	tinyint(1)	
is_admin	tinyint(1)	

roles table The primary key for this table is *id*. The only foreign key in this table is *name*. *user_id* is a MUL, which means the key allows multiple rows to have same value.

Field	Type	Key
id	int(11)	PRIMARY
name	varchar(80)	FOREIGN
user_id	int(11)	MUL

Integration with Analysis Engine

The file structure for the analysis engine integration part looks like following:

```
web
├── config
│   ├── rules
│   └── test_config.yaml
├── evaluate
│   ├── Parsers
│   │   ├── RJParser.py
│   │   ├── LynisParser.py
│   │   └── UnixParser.py
│   └── evaluate.py
├── logs
├── public
├── static
├── templates
├── uploads
└── user
```

The config folder contains a folder called *rules* and a YAML file called *test_config.yaml*. Inside the *rules* folder there are the multiple configuration files (YAML files) for the *Parsers*.

```
config
    ├── rules
    │   ├── cron_at.yaml
    │   ├── crontab.yaml
    │   └── ...
    └── test_config.yaml
```

The *test_config.yaml* contains all the paths for all the YAML files and is used to send the correct YAML file to the evaluate functions.

Moving on to the evaluate folder. This folder contains a *Parsers* folder and a main class *evaluate.py*. *Parsers* contains the three parsers, *LynisParser.py*, *RJParser.py* and *UnixParser.py*. The *evaluate.py* module is the main module of the analysis engine. The module runs all the associated read- and evaluate functions for the logs that are uploaded. The module contains a string that is appended to by evaluate function. Once all the logs have been analysed, the module finally writes out all the warnings to a file that is then downloadable from the server.

The *logs* contains the logs which have been unpacked from a uploaded tar.gz file. This folder exists mainly to store unpacked log files, so that the analysis engine can analyse them. After the analysation, the log files will be removed from the folder to avoid taking too much disk space.

The file structure is split into 'public' and 'user'. In *template* folder there is *public* folder and a *user* folder. The *public* folder contains HTML files which can be viewed by any user visiting the homepage. On the contrary, only logged in users are able to view the HTML files under *user* such as *members.html* and *upload.html*.

```
public
    ├── forms.py
    ├── views.py
├── templates
    ├── public
    │   ├── home.html
    │   └── about.html
    ├── user
    │   ├── members.html
    │   └── upload.html
├── user
    ├── forms.py
    ├── views.py
    └── model.py
```

The *public* and *user* under *templates* have its corresponding *forms.py* and *views.py* under *public* and *user* one directory up. For example, The HTML files in templates/user uses the user/views.py for routing and user/forms.py for forms.

To integrate the analysis engine to the web service, a 'view' has been created as *upload.html*

upload.html

```
1  {% extends "layout.html" %}
2  {% block content %}
3      <h1 class="h1-upload">Upload log files</h1>
4  <form action="{{ url_for('user.upload_get') }}" method="post"
    ↪  enctype="multipart/form-data">
5      <input type="hidden" name="csrf_token" value="{{
        ↪  csrf_token() }}"/>
6      Select logs to upload:
7      <input type="file" name="file" id="fileToUpload"> <br>
8      <select name="parsers" multiple>
9        <option value="RJParser">Remote Job</option>
10       <option value="LynisParser">Lynis</option>
11       <option value="UnixParser">Unix</option>
12     </select>
13     <input type="submit" value="Upload File" name="submit">
14 </form>
15 <hr>
```

```
<h3>List of logs available on the storage:</h3>

<table>
    <thead>
        <th>Name</th>
        <th>Size</th>
    </thead>
    <tbody>
        {% for obj in storage %}
        {% set download_url = obj.download_url() %}
        <tr>
            <td><a href="{{ download_url }}">{{ obj.name }}</a></td>
            <td>{{ obj.size }} bytes</td>
        </tr>
        {% endfor %}
    </tbody>

</table>
{% endblock %}
```

The main function of the *upload.html* is to upload and download a file. In line 7-13, is where a user can choose a file to upload. When uploading a file, the user has to choose one of the three parsers, Remote Job, Lynis and Unix. After the submit button has been clicked, the file log will be saved to the object "file" in line 7 and the parser type will be saved to the object "parsers" in line 8. Those objects will be routed to the upload_post function in /user/views.py which will be explained later.

This code also has a for loop in line 24. This for loop will go through all objects in a storage object and show a download URL for the object. The objects in the storage are the analysis engine's result of the uploaded log files.

views.py

```
@blueprint.route("/upload/", methods=["GET"])
@login_required
def upload_get():
    return render_template('users/upload.html', storage=storage)

@blueprint.route("/upload/", methods=["POST"])
@login_required
def upload_post():
    file = request.files.get("file")
```

```
10      parser_type = request.form.get('parsers')
11      parser =  getattr(Parsers, parser_type)
12
13      my_upload = storage.upload(file)
14      name = my_upload.name
15      upload_folder = os.getcwd() + '/uploads/'
16      upload_file = os.path.join(upload_folder, name)
17      log_folder = os.path.abspath('../web/logs/')
18
19      if parser_type == 'RJParser':
20          tar = tarfile.open(upload_file)
21          tar.extractall(log_folder)
22          tar.close()
23          os.remove(upload_file)
24      else:
25          os.rename(upload_file, log_folder + '/' + name)
26      evaluation = evaluate.evaluate(parser)
27      output_file = os.getcwd() + '/uploads/' + name + '.out'
28      evaluate.write_to(output_file, evaluation)
29      remove_files(log_folder)
30      return render_template('users/upload.html',
         ↪  storage=storage)
```

For *upload.html* to work, it needs to have its own routing and functionality described in the user/views.py.

Whenever a POST request is sent to the server from the *upload.html*, in this case when a user clicks on the submit button, it will route to the *upload_post* function at line 8 of *views.py*. The function then requests the "file" object and the parser type which were selected in the *upload.html* when the user uploaded the file. It will then upload the log file to the storage and evaluate the log file depending on parser type by importing and calling on the analysis engine.

The remote_job logs are tarballed into a tar.gz file and therefore the program has to unpack it to the *log* folder where the log files will be evaluated by the analysis engine.

Finally, the output of the engine is saved in storage in the function *write_to*. The logs are then removed and the function returns the upload template with updated storage, which now contains the new result files from the analysis engine.

6

Results

In this chapter, answers to the problems presented in chapter 1.4 are answered and the result of the prototype is presented.

6.1 Answers To The Presented Problems

What language or languages will be used to develop the prototypes?

The analysis engine prototype uses Python as its programming language and incorporates the data serializing language YAML for configuration files. The web service prototype uses the tools Flask, Jinja2, Python, JavaScript, CSS, JSON to generate the HTML for the browser. It also incorporates a database using MySQL.

How can we automate the manual labor?

The analysis engine automates the manual labor by reading through the logs, extracting the interesting information and then evaluating that information. The evaluation is done by comparing the information with YAML files containing rules for how the information can look. The YAML files also contain the warning messages that should be sent whenever one of the rules are met. These YAML files are easily editable without knowing the code inside the analysis engine.

How much of the current manual labor can we make automatic?

Something that became clear quickly was that making a reliable fully automatic analysis engine was unrealistic in our time frame. Because of this, the engine was designed in a way that allowed future developers to easily expand upon the engine. The YAML files provide a simple way to add more rules, as long as they follow the format of the YAML file. Creating entirely new rules may require understanding the code or maybe even editing the code itself. The current code is fairly simple and this thesis provides an in depth documentation.

How much faster does the process become with our implementation?

How fast the analysis engine analyses the log files depends on a lot of factors such as: How big the log files are, how many rules have been implemented and how fast the hardware is. Because the analysis engine only goes through simple log files and outputs a simple text file without having to do any complex calculations, the process is going to be fast. While we have very few rules, the time it takes from starting the engine to receiving a output file is less than a second on a mid-range laptop from 2014. While the engine is fast at producing an output, what is more interesting is how scalable it is. The engine can be used to check some of the more easily found security risks on hundreds of logs in the time it takes a human to check them for a single log.

Comparing the process to manual labor is difficult considering the fact that there are not many rules declared and therefore relying on this analysis engine to give an accurate security assessment in its current state is very much not advised. However by adding more rules as time goes on it will become much more reliable. The result from the engine should be seen as a complement to the complete analysis, not as a substitute for a security audit. By reading the output one can see which flaws have been detected but it would be a mistake to assume that it has passed everything else.

6.2 The Prototypes

The sitemap for the web service can be seen below:

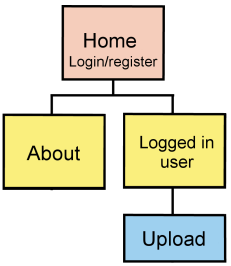

Figure 6.1 Sitemap for Web Service

Home is the page non-logged in users see when they visit the website (see Figure 6.2). The user can enter the username and password to log in to the web service. If the credentials were invalid, a message is shown telling the user there was an error in logging in and to reenter the credentials.

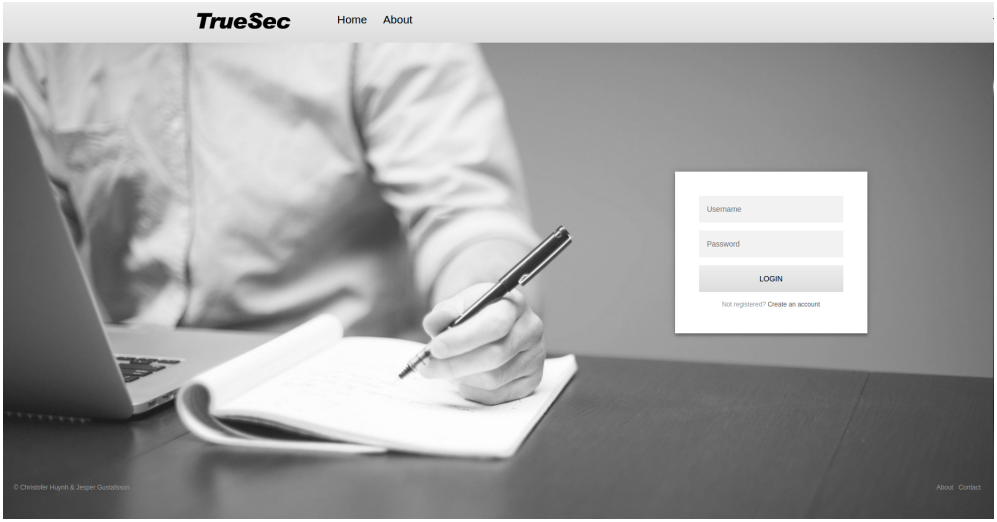

Figure 6.2 Home

If the user wants to register a new account, they can simply click on the "Create an account" link under the "LOGIN" button. This will transform the "Login-window" to a "Register-window" (See Figure 6.3). The registration needs a Username, Email, Password for the user to complete the registration. If the username or email already exists, an error message will be shown.

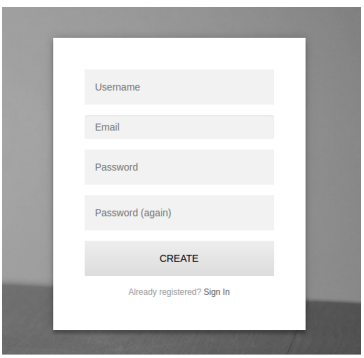

Figure 6.3 Register-window

Logged in user page is what the users always see when they logs in. The web browser saves the session which means if the user has logged in before, they will always be redirected to the members only page (see Figure 6.4).

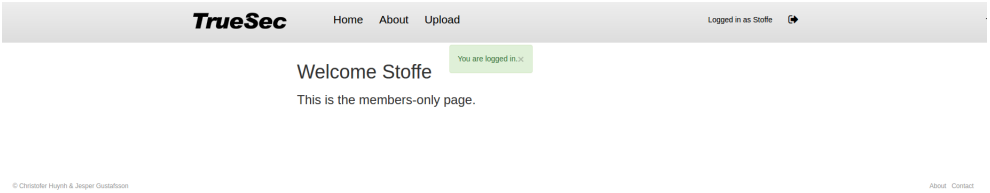

Figure 6.4 Members area after login

The *Upload* area is where the analysis engine is used. The user has to choose a log file to upload (see Figure 6.5). The currently allowed file extension for the web service are: .log, .txt, .gz, .bz2, .zip, .tar, .tgz, .txz, and .7z.

After the user has chosen a file to upload, they have to choose what type of log file to evaluate by choosing either remote_job, Lynis or Unix Privesc Check and then click on submit. One important thing to note is that the remote_job file must always be a tarballed file to be able to go through the analysis engine.

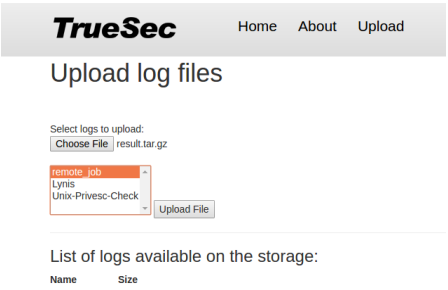

Figure 6.5 Upload

When the "Upload File" button is pressed, the uploaded log file(s) will now be analysed by the analysis engine depending on which parser type the user has chosen. In Figure 6.6, the user has uploaded a remote_job file (*result-tar.gz)*. The remote_job file goes through the analysis engine and a URL to download the results of the

analysis will be available. The output will be renamed *filename.file extension*.out, in this case it will be *result-tar.gz.out*

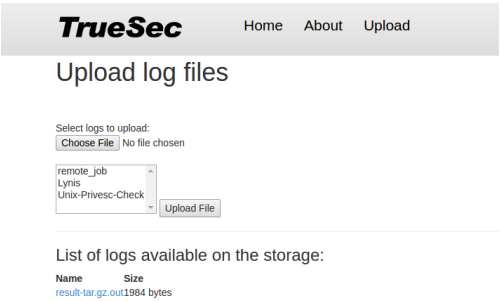

Figure 6.6 Download link

The analysed result of the remote_job log files looks like Figure 6.7 below. The formatting of the remote_job output is kept simple. The log name will be shown as "### *folder name* ###". The suggestions and warnings will be shown under the correspondent log name. The analysis of the folder ends with "#########".

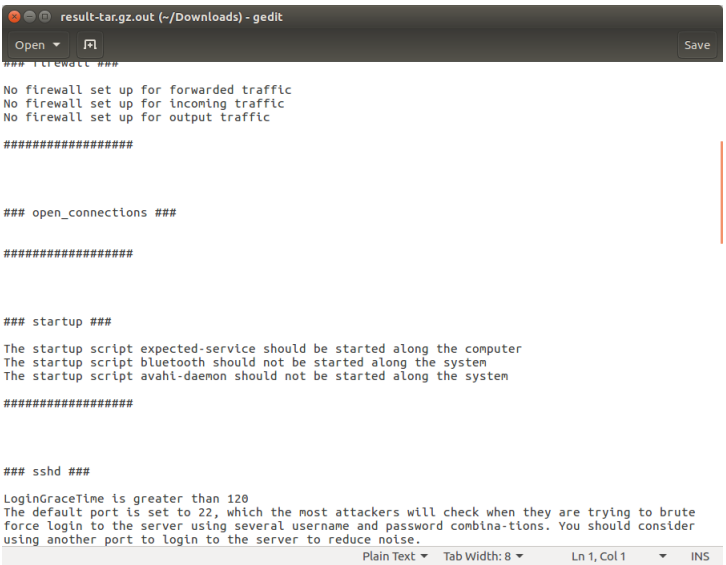

Figure 6.7 remote_job results

The Lynis result looks a little different from the remote_job's result. The formatting here is also very simple, first it will print out the warnings and then it will print out

suggestions.

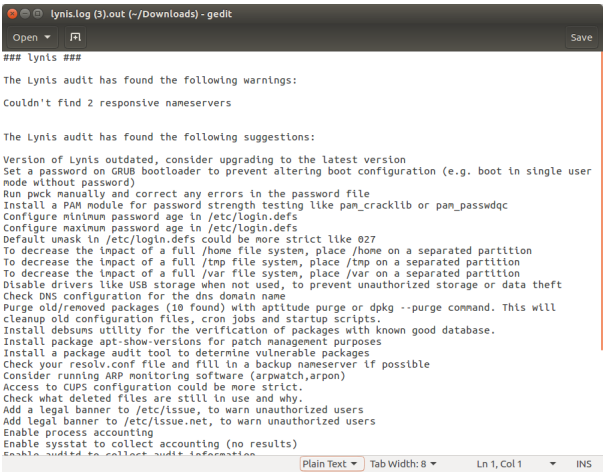

Figure 6.8 Lynis results

Unix results only prints out the warnings it finds (see Figure 6.9).

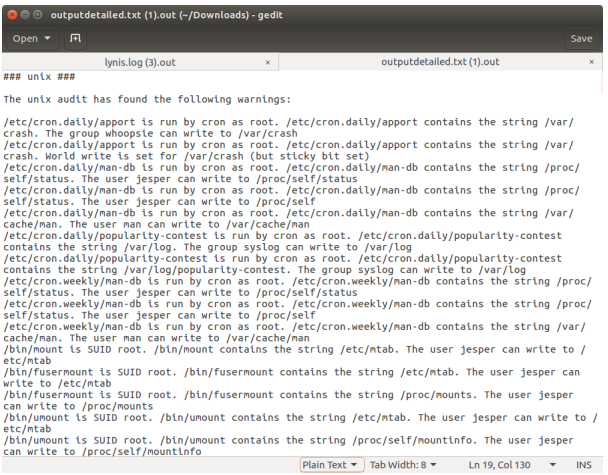

Figure 6.9 Unix results

6.3 Functionality

Amazons EC2 service was used to set up free instances of Debian, Redhat and Amazon Linux AMI remote servers. The servers then ran the scripts(remote_job, Lynis, UPC) using SSH from a local computer. The logs were then analysed with the analysis engine and no errors occurred during the execution for any of the servers.

7

Conclusion

7.1 Conclusion

An analysis engine prototype for assisting security health checks has been developed. The purpose of the analysis engine is to try to automate some of the manual labor the security auditors has to do.

Something we had to think about for each new flaw that we found was whether it could be automated and how to automate it and whether it was worth the time and effort to implement the automation. Something to keep in mind is that the analysis can probably never be fully automated, and there would certainly be more things that could be automated given more time to test configurations to find security flaws.

Because of this, the analysis engine was designed to be easily modifiable. The YAML files provide a simple way to edit or add more security checks without even having to worry about the code, giving the analysis engine a huge potential to be a very effective audit tool in the long run. The read- and evaluate functions can be written in whatever way a future developer wishes, the only requirement being that it returns a string back to *evaluate.py*, this together with the fact that one can just add a new parser also allows the analysis engine to go beyond the current scripts.

Whether the analysis engine has made the security audit process faster can be debated and will depend on the future use of the analysis engine. The current engine may have sped up the process by saving time to go through the Lynis and Unix logs with over 1000 lines by printing out the warnings and suggestions.

TrueSec has planned to have a web service for security health checks which is why a simple web service was created. In the future, the web service will have users that can perform a security checks themselves. For this reason, the web service has an authentication model with login and user roles for uploading log files and storing the result from the analysis engine. This makes it easier for further development for creating users with e.g. admin status.

8

Further Development

8.1 Analysis Engine

While we tried to make the analysis engines read- and evaluate functions as homogeneous as possible, we often found a simpler, better way to do something. This also extends to the YAML files. When finding a better way to do something we quickly found out that modifying every other function or YAML file was a waste of time as we eventually found an even better way of doing it. Another issue that could arise by doing this is making certain functions way more complex than they need to be, for example: The *cron_at* and the *open_connections* files are very different in how complex the analysis engine needs to handle them. But we do have a few suggestions on how to further develop the analysis engine, keep in mind that the read- and evaluating function as well as the layout of the YAML file are highly dependent on each other. If you change the layout of the YAML file, the evaluate function will fail. If you change how the read function is built, the evaluate function will fail. This means that you will have to edit them in parallel, you cannot just decide to change a single one of them.

Reading

Something that is probably a good idea no matter what is to build the dictionaries from the info file similar to the corresponding YAML file.

Evaluating

Assuming the parsing function has been modified according to the above, it should now be much more simple to compare the info files values with the corresponding YAML files values.

YAML files

As with the functions, the YAML file structure also changed a lot with each log file. It might be a good idea to try and make them more homogeneous as it will make it easier to add more security checks if they all follow the same rules. The downside being that it could lead to unneeded complexity.

Object oriented

Because of the constant modifying of functions, we did not think it was a good idea to apply object oriented ideas such as reducing duplicated code by making functions or using design patterns.

Once the code is more stable, it might be a wise decision to try and implement some of these object oriented ideas.

Output

The current output just consists of warnings or suggestions. One might want to use this output file to create a prettier report. Another function could be added to read the output file and organise it appropriately. This could use the for now unused *severity* variable to highlight more severe security flaws by e.g. positioning them first or using different colors.

Something else that might be useful is to provide a string if an issue is not found. That is to say, the output file should perhaps mention that it has passed certain tests. This can save more time as the auditors do not have to search for security risks where there are none. This could potentially massively speed up the analysis process, it does however introduce the risk of false positives, which can be very detrimental.

8.2 Web Service

The current web service is very simple, it has a login and register function and a storage system used to upload and download log files to and from the web service. For this thesis, everything is done locally but it is built in a way not limited to only MySQL database, you can use other databases for the web service. The storage system could be further developed, it has functions to access, upload, download and delete files. This could be done on a cloud service such as AWS S3, Google storage and Microsoft Azure.

The authentication model can be further developed to allow users to have their own log storage. The user can upload their own log files of their systems generated by e.g. remote_job. Additionally, there should also be a encryption and decryption of the uploaded and downloaded log files in the future.

As the main focus was not the web service, this lead to less testing of the web service. This means there are errors that are not handled correctly e.g. if a user tries to upload an a file with an invalid file extension, they will be redirected to an error 500 page telling the user that something went wrong with the system. Such errors should be handled more elegantly in case of further development of the web service.

In the future if TrueSec wants to make a web service with e.g. payment system with different package deals, then they would have to rework the register system or even the whole database itself.

9

References

What is Flask?, 2017-04-06, Retrieved from
http://pymbook.readthedocs.io/en/latest/flask.html

Jason Myers and Rick Copelannd, *Essential SQLAlchemy*, O'Reilly Media, California, 2016 p xiii

FlaskSQLAlchemy, 2017-04-06, Retrieved from
http://flask-sqlalchemy.pocoo.org/2.1

Flask-Migrate, 2017-04-06, Retrieved from
https://flask-migrate.readthedocs.io/en/latest/

Flask-WTForms, 2017-04-06, Retrieved from
https://wtforms.readthedocs.io/en/latest/crash_course.html

Flask-Login, 2017-04-06, Retrieved from
https://flask-login.readthedocs.io/en/latest/

Webassets, 2017-04-10, Retrieved from
https://webassets.readthedocs.io/en/latest/index.html#index

Steve, Suehring, *MySQL Bible*, Willey Publishing, Inc, New York, p. 11

Tim Ambler and **Nicholas Cloud**, *JavaScript: Frameworks for Modern Web Dev*, Spring Science+Business Media New York, New York, 2015, pp 1, 9

Jinja2, 2017-04-11, Retrieved from http://jinja.pocoo.org/

Bootstrap3, 2017-05-02, Retrieved from https://www.tutorialspoint.com/bootstrap/index.htm

Williams, Richards 2003, *UNIX Audit: Performing a Successful Unix Audit*, Computer Fraud & Security, vol. 2003, pp. 11-12

Muhammad Mushfiqur, Rahman, *Auditing Linux/Unix Server Operating Systems* , ISACA Journal Volume 4, 2015 pp 1

Mookhey, K.K. and Burghate, Nilesh, *Linux: Security, Audit and Control Features*, Information Systems Audit and Control Association, 2005, pp 33,56, 66

Dieter Gollmann, *Computer Security*, John Wiley & Sons, Inc, New York, NY, USA, 1999, pp 120, 122-123, 124

YAML, 2017-05-14, Retrieved from http://www.yaml.org/spec/1.2/spec.htmland,

Python Dictionary, 2017-05-14,
Retrieved from https://www.tutorialspoint.com/python/python_dictionary.htm

The Unix security audit and intrusion detection tool, 2017-05-03, Retrieved from http://www.nongnu.org/tiger/index.html#history.html

Tiger Unix security tool - Summary, 2017-05-03, Retrieved from
http://savannah.nongnu.org/projects/tiger

Steve Kemp, *Common Security check for a base installation - package reviewed*, 2002, Retrieved from https://lists.debian.org/debian-devel/2002/12/msg01566.html (2017-05-03)

Lynis, 2017-05-17, Retrieved from https://cisofy.com/lynis/

OpenSCAP, 2017-05-17, Retrieved from https://access.redhat.com/documentation/en-US/Red_Hat_Network _Satellite/5.5/html/User_Guide/chap-Red_Hat_Network_Satellite-User_Guide-OpenSCAP.html

Unix-Privesc-Check, 2017-05-17, Retrieved from https://github.com/pentestmonkey/unix-privesc-check

Linux Security Auditing Tool, 2017-05-17, Retrieved from http://usat.sourceforge.net/

Michael Haughland & Magnus Wecksten, *Linux Hoist Review: En undersökning avautomatiserade auditverktyg*, 2014, ss 7

Computer Oracle and Password System, 2017-05-04, Retrieved from
http://ftp.cerias.purdue.edu/pub/tools/unix/scanners/cops/cops.1.02.README

Animesh Patcha and Jung-Min Park, *An overview of anomaly detection techniques: Existing solutions and latest technological trends*, Blacksburg, United States, Bradley Department of Electrical and Computer Engineering, 2006, ss 2, 4

10

Appendixes

10.1 Appendix A

Tiger will do following checks:

- **check_accounts:** Checks the accounts provided in the system, looking for disabled accounts with cron, rhosts, .forward, and valid shells.
- **check_aliases:** Performs a check for mail aliases and improper configuration.
- **check_anonftp:** Determines if the anonymous FTP service is properly configured.
- **check_cron:** Validates the cron entries in the system
- **check_embedded:** Check if embedded path-names are configured properly
- **check_exports:** Analyses configuration files for NFS exported filesystems to see if access is properly restricted.
- **check_group:** Checks the UNIX groups available in the system, looking for conflicts and improper entries.
- check_inetd: compares against services definition, valid directory paths, non-existent binaries and active services.
- **check_known:** Search for known intrusion signs including backdoors and mail spools.
- **check_netrc:** Check the netrc files if it is insecurely configured
- **check_nisplus:** Search for wrong configuration in the NIS+ entries
- **check_passwd:** Looking for conflicts and improper entries
- **check_path:** Validates the binaries in user's path and path definitions used by scripts

- **check_perms:** Search for filepermissions and inconsistencies
- **check_printcap:** Check the configuration of the control file for the printer
- **check_rhosts:** Check the rhost files if the configurations are vulnerable for attacks
- **check_sendmail:** Check the configuration files
- **check_signatures:** Compare binary files signature to the one stored in the local database
- **check_system:** Calls a specific module
- **check_apache** Check the configuration file and reports which can lead to exposure and vulnerabilities in the system
- **check_devices:** Check device's permissions
- **check_exrc** Analyze the exrc files that are not in the home directory
- **check_finddeleted** Check if deleted files are still in used by the system
- **check_ftpusers:** Check the ftpusers file and see if there are users with administrative rights
- **check_issue:** Check the contain for appropiate content which is defined in the ISSUEFILE and ISSUENETFILE
- **check_logfiles:** Check log files
- **check_lilo:** Check the configuration files for lilo and grub boot loaders
- **check_listeningprocs:** Check the processes listening on TCP/UDP sockets in the system
- **check_passwdformat:** Check the password format
- **check_root:** Guarantee that the root is allowed to login to the local system
- **check_rootdir:** Checks permissions for the root file
- **check_rootkit:** Search for system which have been rootkited
- **check_single:** Check if single-user access is enable, if it will warn the user
- **check_release:** Check if the system is up-to-date
- **check_runprocs:** Check for running processes if they have permission to run

- **check_services:** Check the /etc/services file after services that should be configured
- **check_tcpd:** Test the existence of tcp wrappers and show which service is running in the wrapper
- **check_umask:** Check the umask configuration file
- **check_xinetd:** Check if the local user passwords are easy to guess

10.2 Appendix B

Following shell scripts are included in the Lynis security scan:

- test_accounting
- test_authentication
- test_banners
- test_boot_services
- test_containers
- test_crypto
- test_custom.template
- test_tests_databases
- test_file_integrity
- test_file_permissions
- test_filesystems
- test_firewalls
- test_tests_hardening
- test_homedirs
- test_insecure_services
- test_kernel
- test_kernel_hardening
- test_ldap

- test_logging
- test_mac_frameworks
- test_mail_messaging
- test_malware
- test_memory_processes
- test_nameservices
- test_networking
- test_php
- test_ports_packages
- test_printers_spools
- test_scheduling
- test_shells
- test_snmp
- test_solaris
- test_squid
- test_ssh
- test_storage
- test_storage_nfs
- test_system_integrity
- test_time
- test_tooling
- test_virtualization
- test_webservers

10.3 Appendix C

The following folders will be created inside the *results*-folder after a remote_job_linux_osx security scan:

- /results/cron_at_info
- /results/crontab_info
- /results/diskvolume_info
- /results/encrypted_disk_info
- /results/environment_info
- /results/firewall_info
- /results/groups_info
- /results/lastlog_info
- /results/modprobe_info
- /results/networkvolume_info
- /results/open_connections_info
- /results/passwdpolicy_info
- /results/processes_info
- /results/samba_info
- /results/sshd_info
- /results/startup_info
- /results/sudoers_info
- /results/suid_filers_info
- /results/system_info_info
- /results/users_info