Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Forensic Tool for Linux Desktop

Bachelor Thesis

2018
Roman Števaňák

Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Forensic Tool for Linux Desktop

Bachelor Thesis

Study programme:	Informatics
Study field:	2508 Informatics
Department:	Department of Computer Science
Supervisor:	RNDr. Jaroslav Janáček, PhD
Consultant:	Mgr. Lukáš Hlavička

Bratislava, 2018
Roman Števaňák

45150075

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Roman Števaňák
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Computer Science, Informatics
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title:	Forensic Tool for Linux Desktop
Annotation:	The thesis deals with analysis of forensic artifacts at Linux workstations and with creation of a tool for their detection, analysis and evaluation.
Aim:	- analyze available forensic artifacts at workstations with Linux distributions based on Debian and Red Hat - design and implement a tool for their detection, analysis and evaluation

Supervisor:	RNDr. Jaroslav Janáček, PhD.
Consultant:	Mgr. Lukáš Hlavička
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.
Assigned:	31.10.2017

Approved: 06.11.2017 doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

....

Student Supervisor

45150075

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Roman Števaňák
Študijný program:	informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor:	informatika
Typ záverečnej práce:	bakalárska
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Forensic Tool for Linux Desktop
Forenzný nástroj pre Linuxový desktop

Anotácia: Práca sa venuje analýze forenzných artefaktov na Linuxových pracovných staniciach a vytvoreniu nástroja na ich detekciu, analýzu a vyhodnotenie.

Cieľ: - analyzovať dostupné forenzné artefakty na pracovných staniciach s Linuxovou distribúciou založenou na distribúciách Debian a Red Hat
- navrhnúť a implementovať nástroj na ich detekciu, analýzu a vyhodnotenie

Vedúci:	RNDr. Jaroslav Janáček, PhD.
Konzultant:	Mgr. Lukáš Hlavička
Katedra:	FMFI.KI - Katedra informatiky
Vedúci katedry:	prof. RNDr. Martin Škoviera, PhD.
Dátum zadania:	31.10.2017

Dátum schválenia: 06.11.2017

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

študent

vedúci práce

Abstract

Nowadays, computer forensics does not help only with cases, where crime committed is on the computer, like cyber espionage or infrastructure intrusion. It can help with many cases, especially since significant part of communication comes through digital media.

There are numerous forensic tools, however, most them are proprierary, expensive and do not analyse Linux more than as a filesystem, even though it is being widely used as a desktop operating system.

In this thesis we analyse typical Linux system to identify forensic artifacts, then design and implement Fortool - Open source forensic tool for Linux, for their extraction and further analysis.

Keywords: Post-mortem forensics, Linux forensics, Forensic analysis

Abstrakt

V dnešnej dobe počítačová forenzná analýza neslúži len na vyšetrovanie trestných činov vykonaných počítačom ako napríklad vniknutie do počítačovej infraštruktúry, ale pri mnohých ďalších vďaka tomu, že veľká časť komunikácie prebieha v digitálnej podobe.

Existuje množstvo nástrojov, ktoré pomáhajú pri forenznej analýze, avšak napriek rozšírenému používaniu operačného systému Linux, mnohé z nich sa naň pozerajú len ako na súborový systém a samotné súbory neinterpretujú. Zároveň sú zväčša komerčné a drahé.

V tejto práci identifikujeme základné forenzné artefakty pre Linuxový sytém, navrhujeme a implementujeme Fortool - open sourcový nástroj na forenznú analýzu Linuxu na ich extrakciu a ďalšie skúmanie.

Kľúčové slová: Post-mortem analýza, Analýza Linuxu, Forenzná analýza

Contents

Introduction

Nowadays, computer forensics does not help only with cases, where crime is committed on the computer, like cyber espionage or infrastructure intrusion. It can help with many cases, especially since significant part of communication comes through digital media.

However, this is not a new field, and many forensic tools already exist with a rich history, such as FTK or Encase, that helped to apprehend BTK killer. However they usually do not focus on Linux as they do for Windows or MacOs, even though it is being widely used as a desktop operating system. Usually most tools stop at filesystem forensics and do not interpret content of files outside keyword or hash search and therefore most of Linux forensics has to be done by hand. Also most of the forensic tools are proprietary and very expensive. In this thesis we introduce Fortool - Open source forensic tool for Linux desktop, which can help analysts all over the world.

We focus on post-mortem analysis, where image used is mounted as read-only and in such a way, that tool has access into all files, including root-owned ones.

As a representative on which to look for artifacts and test the tool on, we use Debian-based Ubuntu 16.04.03 LTS, as it is at the time of writing latest long-term-support version of this distribution and one of the most popular ones, and RHEL-based CentOs 7 as it is also most recent version of this widely used distribution. These also bear strong similarity to other Linux distributions and most of the artifacts we examine are found in any other usual distribution.

In the first chapter we discuss what computer forensics is, and its challenges. Next, we take a closer look at what data on a typical Linux system, whether user-made or system-made can have value for forensic analysis and how and where is it stored. In the third chapter we go through our implementation of a tool to extract and analyse this data, how it is used and how it can be extended.

Chapter 1

Forensic analysis

Computer forensics involves the scientific examination, analysis, and/or evaluation of digital evidence in legal matters[28].

Digital trace is any data with informative value, that is stored or transmitted in a binary form. **Digital evidence** is digital trace that confirms or denies hypothesis and is admissible in court. **Forensic artifact**, as defined by SWGDE is information or data created as a result of the use of an electronic device that shows past activity[28].

There are two types of forensic analysis - live and post-mortem.

Former occurs when system is still active during analysis. In this scenario it is possible to acquire volatile data, such as RAM, running processes, internet connections and temporary files. If disc encryption is used, with this analysis, filesystem can be decrypted with cached key. On the other hand, this type of analysis requires more expertise and system is constantly modifying its data, which may hurt court admissibility. Analyst also should not trust any tools provided on the system.

Post-mortem analysis transpires only on drive image, and as read-only, which means data should stay intact.

Forensic process can be broken up into stages:

1. Data acquisition
2. Secure storage of data
3. Analysis of aforementioned data
4. Presenting conclusions

Data acquisition means extracting all important information from the system into some form of immutable storage, for example dumping RAM onto external sanitized drive or photographing a monitor.

With live forensics, standard procedure is to acquire data from most to least volatile. For example, we would like to extract RAM before copying hard drive. It is important to run only trustworthy software, best if it is on external drive, which analyst brought with them.

For post-mortem forensics extraction means copying hard drive bit per bit into an image. This can be done for example by linux utility `dd`. It is recommended to use hardware write blockers[32], which allow only reading from a drive on hardware level. This is to guard against automatic writing that can transpire on host computer, or unintentional one done by inexperienced analyst. It is important that image is bit per bit copy, because on most filesystems removing a file does not mean overwriting it with other data, and it usually can be recovered from unallocated space on drive. There should be hash sum made from both hard drive and its resulting image as a check for integrity. Booting a copy of an image on similar hardware can be insightful too.

Secure storage of data means storing all extracted information in a way that it can not be tampered with, and if so, it should be easily detected. In practice this means that after making image out of hard drive it is locked in a vault in a sealed container and analysis is always done on copy of the image. Before beginning it is also advisable to check hash sum of the image. Furthermore, everyone that comes to contact with it, should be listed in chain of custody form[32].

Analysis is helped by applying scientific method - first observations are made on gathered data. This can be done for example with keyword searches or reviewing system configuration.

Then, hypothesis is formed, that would explain the observations, and it is evaluated by making predictions based on hypothesis being true and validating them. It is also very important to try disproving said hypothesis by making predictions with assumption of it being wrong and testing those as well. If tests do not support hypothesis, it has to be revived[30].

Last step is to draw conclusions from proven hypotheses.

Presenting conclusions has to be done in an understandable way for usually not technical audience, and has to be demonstrably based on fact. It is also important to ensure availability, authenticity, integrity and in most cases also confidentiality of presented data.

Timeline is important to make clear chain of events, which can help with identification of more traces using deduction, or it can be used to present conclusions more clearly. Establishing it, however, can be complex task. If there are more systems involved, examiner has to know even small time offsets of each one and has to understand its format for each forensic artifact.

1.1 Time formats

Many applications store time differently, and to extract information from it, we need strong understanding of different time formats. In this section are listed most commonly used ones on a typical system and their take on some challenges such as leap seconds and years.

Leap year is defined in Gregorian calendar, is 366 days long and occurs in a year, that is divisible by 4. If it is divisible by 100 it is a leap year only if it is also divisible by 400.

Leap second comes from a need to synchronize length of solar based days, which gradually get longer with length of days defined as 86,400 SI seconds, which stays the same. At the time of writing, there has been 37 seconds added. Other way to deal with this problem is called second smear, which does not insert additional second,but rather over larger period of time makes several ones longer.

1.1.1 ISO 8601

This set of formats was standardized by International Organization for Standardization to reduce risk of misinterpretation when communicating across borders, because countries use local conventions. It is meant to represent date in Gregorian calendar and time in 24 hour timekeeping system.[27]

Any date preceding introduction of Gregorian calendar should be only used after agreement from both communication sides. This is because transition from Julian calendar included canceling accumulated time inaccuracies by going from 4th October 1582 to 15th October 1582.

This international standard defines format for one point in time, time span and recurring time interval.

Day formats are used to specify one day.

Expanded format, if agreed upon, can be used to represent years with larger number of digits than usual. In the agreement should be stated how many extra numbers will be used, and all smaller ones should be zero padded. Format also includes '+' or '-' as the first character to signify whether number of years is positive or negative.

We will represent date 14.3.2018 in all formats to show the difference.

General date is represented by year, month and a day. Complete representations can be 20180314 in basic format and 2018-03-14 in extended format. If there is no need for this degree of accuracy, other representations can be used. A specific month with format 2018-03, specific year as 2018 or specific century as 20.

Ordinal time consists of year and number of days in a year. It can be written in basic or extended format as 2018073 or 2018-073 respectively.

Week date is a year, number of weeks since its beginning and a number of weekday, where 1 is monday and 7 is sunday. It can be written as 2018W113, or 2018-W11-3. It is also accepted to omit day of the week for reduced accuracy.

Time of day formats are all based on 24 hour timekeeping. They contain hours in range from 0 to 24, minutes from 0 to 59 and seconds from 0 to 60. Midnight can be represented either as 24:00 of previous day or 00:00 of next day. Value of 60 in seconds is only used for positive leap seconds.

There should be time designator, 'T' used before time representation, where it is not clear it is time marking. Each format can be also written without colon ':' delimiters.

In each format we represent time of 15 hours, 26 minutes 37 seconds and 480 milliseconds. Accepted formats are complete, where it looks like 152637 or 15:26:37, with reduced accuracy, where only hour or hours with minutes are displayed. We can also use decimal fraction on seconds, such as 15:26:37,48, minutes as 15:26,625 or hours as 15,444. There has to be at least one numeral after the comma and there is no upper limit. Delimiter can be ',' or '.'.

Time zone can be signified directly after time representation. If there is none, time is considered to be local. Added 'Z' means it is already in UTC time. There can also be stated difference to UTC with '+' or '-' and precision in minutes or hours, as +01:00 or +01 for central European standard time. The colon ':' delimiting minutes and hours can be omitted.

Complete ISO 8601 timestamp combines any complete day, time and optional zone representation. Delimiter 'T' is used, unless agreed upon otherwise.

1.1.2 RFC 3339

This standard was proposed for internet communication and is a profile of ISO 8601 standard [33]. It only specifies time points representation, not timespan or recurring time. It contains these changes to ISO standard:

- Required complete representation of timestamp with four digit year and delimiters.
- Decimal fractions can be used only with seconds and delimiter can only be a full stop.
- Time zone has to be stated.

- Delimiter between time and date can also be a space, 'T' and 'Z' used can be lowercase, but still should be uppercase.
- Hour can only be from range 0 to 23, which means 24 can not be used.

1.1.3 RFC 3164

Standard, which describes BSD syslog protocol states, that timestamp field for each log should be in local time and in format of three letter abbreviation of english month, day in the month and colon delimited hours, minutes and seconds like Mar 14 15:26:37. If day of the month is one digit long, it has to be padded by space. Later in the standard it is stated, that if more detailed timestamps are desirable, they should be contained inside content of the log message.[34]

1.1.4 Epoch times

Unix time is a format representing seconds since what is called Unix epoch, 1970-01-01 00:00:00 in UTC [30]. If positive leap second occurs, the same number is repeated for two seconds. It is used mainly on Unix and POSIX systems. This epoch is also mostly used in Mozilla Firefox databases, however there it represents microseconds.

Windows times are several - Windows time, which contains number of milliseconds since system start and structure FILETIME represents it similarly to Unix time, but here the epoch is 1601-1-1 00:00:00 and each number is of 100 nanosecond periods. It can use local time as well as UTC, depending on which function returns it. Google chrome and chromium internal databases use this epoch too, but with 1 microsecond period.

UUID version 1 uses 60-bit timestamp in UTC or local time, with periods of 100 nanoseconds and epoch being reformation of Gregorian calendar, 15th October 1582.

1.2 Tools for Linux forensics

The Sleuth Kit [25] is selection of unix-based tools[31] to examine volume and filesystem. It supports among others NTFS, FAT, Ext4 and HFS. It can extract file data and metadata, however it does not interpret it.

It can also be used as a plug-in framework, where it provides reading of a file and what is called a blackboard for communication between modules. Its plug-ins allow it to for example calculate file entropy, generate HTML reports, look up hashes in databases or use RegRipper to analyse Windows registry hives. It is no longer under active development.

Autopsy [5] is open source GUI and platform for many forensic tools, including The Sleuth Kit. It has many capabilities - it can search for keywords, even through regular expressions, check file hashes against hash databases, analyse registry thanks to RegRipper or extract mails, EXIF information from pictures or web artifacts from Firefox, Chrome and Internet Explorer. It can also carve files using PhotoRec and check indicators of compromise with STIX. Android analyzer module allows it to extract text messages, contacts, call logs or gps data in google maps from some versions and vendors of android.

SMART for Linux [24] is commercial tool distributed by ASR Data. It can sort files by type, search through them for keywords or with regular expressions and supports looking up file hashes in hash databases.

OSforensics [19] is commercial tool made by PassMark software. It can check file hashes against a database, search for keywords in file contents, supports information extraction from web browsers and e-mails and showing it on timeline, and search for passwords. It works on Windows filesystems, as well as on Linux and Mac ones, however it does not provide some functionality on the latter, such as most recent activity, including browsing data and passwords.

1.3 Summary of artifacts

On a typical Linux system, there are many digital artifacts to be found. First, there is a need to map out how system behaves and how it is set up - this can be found in chapter 2.1.

User accounts described in 2.1.2 make for a good stating point, as they frequently contain personal information of the owner and also link together many actions across the system, which are done by it. Scheduled tasks in 2.1.1 can help us attribute certain actions that may have transpired seemingly without the cause, to the system or the user account that set them up, if it can be extracted. Reading metadata from files in 2.1.3 can help us link files to user accounts that may have created or had access to them and provide times of last access or modification to put in a timeline and can help with attribution. Installed software in 2.1.4 provides information about what the system is capable of. Logging, as described in 2.1.5 if set up thoroughly can give us complete look on what transpired in a system in a given time, from authentications to in some cases even software run. Configuration files of certain tools used for it can help us in finding and understanding log files.

Traces of individual user account activity are described in 2.2. Internet browsers can give us a lot of insight into activities that were done by user account, as can be seen in section 2.2.1. We can extract browsing history, downloaded files, stored cookies, currently or lastly open tabs and in some cases even what user filled into forms on some sites or even passwords. Also, as described in 2.2.2 we can also gain information about last used commands or usage of tools for encrypted communication such as openssh.

Chapter 2

Forensic artifacts

This chapter describes formats and locations of artifacts as described in 1.3. They can be split into user-created and system-created ones. Former are usually local for one user and usually placed in subfolders of their home folder, latter affect whole system.

2.1 System artifacts

2.1.1 Scheduled tasks

Task scheduling is widely used to automate jobs that need to be done regularly. On a typical Linux distribution there are two types of scheduling daemons: crond and anacron.

Crond

Crond can be usually configured in two ways - by vixie cron, that can be used by every user or by system-wide cron, which can only be set by root. More subtle difference between them is, that in system-wide crontab, there is a field to specify as which user job should be run, while in vixie cron all jobs are run as user who set it.

Crond is able to deal with changes in system time less than three hours, like for example daylight savings time, where if it is moved forward, all jobs that should be scheduled in the time that was jumped are executed at once, and if it is moved backward it prevents double execution[8]. If change is larger than three hours, no extra precautions are done.

System-wide crontab on a typical system, its configuration is in `/etc/crontab` and files contained in folder `/etc/cron.d/`. Its files are in following format:

```
SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
```

```
MAILTO=johndoe
# m h dom mon dow user  command
17 *    * * *   root    cd / && run-parts --report /etc/cron.hourly
25 6    * * *   root    test -x /usr/sbin/anacron ||
    ( cd / && run-parts --report /etc/cron.daily )
47 6    * * 7   root    test -x /usr/sbin/anacron ||
    ( cd / && run-parts --report /etc/cron.weekly )
52 6    1 * *   root    test -x /usr/sbin/anacron ||
    ( cd / && run-parts --report /etc/cron.monthly )
* * * * 13 5 root echo "It is friday 13th" >> /home/johndoe/day.txt
```

It is not recommended to edit `/etc/crontab`, but to add a file into `/etc/cron.d`. Configuration can contain commentaries, if line starts with #, lines setting variables, that look like NAME=VALUE or scheduling lines. First five space separated columns describe time, when job should be ran, next columns contain user, as which job should be run and command to be executed.

Time columns are formatted in sequence month, hour, day of month, month of year and day of week. For each field there are multiple ways to specify its value - * means that this condition passes for every possible number, lists like 1,2,3,4,5 mean that condition passes only for number in the list, which can be also written as range 1-5. One can also use steps that restrict variable it devides into steps that are divisible by the step, for example 3-11/3 means list 3,6,9 and it also can be used with asterisk, for example */2 are all even numbers, that can be used in given column. For columns month and day of week one can also use textual representations. If both day of month and day of week are used, job is executed only if both conditions are passed. Also, instead of five columns, one can use time specification nicknames, for example @reboot, or @annually, that stands for "0 0 1 1 *".[9]

When executing, crond uses shell defined for user it uses in `/etc/passwd`, unless variable SHELL is set, in which case it uses specified one. Local timezone for each crontab can be set by variable CRON_TZ.

Default /etc/crontab in Ubuntu 16.04 contains lines to check whether anacron is installed and if not, run `/etc/cron.hourly`, `/etc/cron.daily`, `/etc/cron.weekly` and `/etc/cron.monthly`, once an hour, day, week and month respectively.

Vixie cron This cron, also called user-level cron, can be configured using command `crontab`. Format of these is almost identical to system-wide ones, but they do not contain user column, as all jobs are run as owner of the crontab.

Rules on which users are able to use command crontab can be configured in files `cron.allow` and `cron.deny`. If file `cron.allow` exists, user must be listed in it in order to run it, otherwise if `cron.deny` exists user must not. Crontabs for each user are stored in `/var/spool/cron/crontabs`, with each file named after user who set it up.

anacron

Unlike previous two, this scheduling daemon does not assume computer is running continuously and checks whether jobs were executed within set up period since last execution, and if not, it executes them[3].

Its configuration is located in `/etc/anacrontab` and has following format:

```
SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
HOME=/root
# These replace cron's entries
1       5       cron.daily      run-parts --report /etc/cron.daily
7       10      cron.weekly     run-parts --report /etc/cron.weekly
@monthly        15      cron.monthly
    run-parts --report /etc/cron.monthly
```

Each line describes one job, sets variable or is a commentary. Its format consists of period in days with which job should be run, delay, which is how long anacron should wait from time it decides job should be run until it starts execution, unique name, which is also used as filename in `/var/spool/anacron/` folder for storing timestamp of its last run in format like 20180426 and lastly it contains command that should be executed.

Period can use nicknames such as @daily or @weekly which stand for 1 or 7 days, or @monthly, which executes once a month disregarding how many days it is. [4] Notable variables include RANDOM_DELAY, whose value is upper limit on how much minutes is randomly added to each job delay, START_HOURS_RANGE, which says in what time window is anacron allowed to start jobs, SHELL and PATH that are defined similarly to ones for crond.

2.1.2 User accounts

Information about what users are on the system are stored in `/etc/passwd`[30][20] in following format:

```
johndoe:x:1000:1000:John Doe,,,:/home/johndoe:/bin/bash
```

Each line represents one user account, and it contains, separated by colons, username, user id, group id of their primary group, their home directory, shell they use and optionally, comma separated additional information. By convention it contains contact and real name for person, that owns this account. Second field used to be filled with hashed password, but for security concerns it was moved into `/etc/shadow` and in their place 'x' is kept as a value, because each user has to be able to read `/etc/passwd`, and that would make extracting other users password hashes easy.

File `/etc/shadow` contains information about passwords for each user in format like this:

```
johndoe:$6$r2Ihh5P5$C[shortened]cwH1:17640:0:99999:7:::
root:!:17640:0:99999:7:::
nobody:*:17590:0:99999:7:::
```

Each line contains values separated by colon as follows: username, password hash, date of last change of password in days since 1.1.1970, minimum days since last password change to another one, upper limit of days that password can be valid, number of days warning before expiration is showed, number of days after password expiration when user is forced to change password after login, and date of account expiration[23].

Password hash is usually in format $id$salt$hash, where id tells us what hashing algorithm was used and salt is a random string, up to 16 bytes long, that is used to concatenate password to help mitigate dictionary attacks on hashes.There are several special values that signify special meaning: if it is empty, there is no password needed for logging into the account. If it is just '*', '!' or other value that could not be normally outputted hashing function, it means that this account is not accessible by authentication with password. When there is ! prepended to normally formated hash it probably means that account was locked.Value 0 in date of last password change means, that user will be forced to change password next login.

File `/etc/group` is analogous to `/etc/passwd`, but for user groups and has following format:

```
johngroup:x:1003:johndoe
```

For each group it contains its name, group id and comma separated list of members. There is also file `/etc/gshadow` which is similar to `/etc/shadow`, which looks like this:

```
johngroup:$6$dm81Pdb21w$4[shotened]n8.::johndoe
```

There for each line is name of the group, hash in same format as in `/etc/shadow`, list of administrators of the group and list of members. Administrators have the right to add or delete users and change password. Each administrator is also a member. Each user, that wants to access a group is prompted a password only if they are not a member.[11].

Each of `/etc/passwd`, `/etc/shadow`, `/etc/groups` and `/etc/gshadow` has a backup, which is its name with '-' appended.

In file `/etc/sudoers` we can find users sudo privileges. All this information can be important for establishing whether some user had access to specific files.

2.1.3 File forensics

Linux can run on many filesystems, usually it is ext4 or ext3. Here, filesystem is acyclic oriented graph, where each node is a file or directory, and each edge, which can go only from directory to file or another directory means, that it is located in a given directory. Every file and directory has assigned an i-node, which is where all its data are stored. Every file contains only its name and address to i-node.

There can also be hard and soft links, which are special types of file. Soft link contains path to file, which it links, hard link contains its name and i-node number of linked file. Once it is created, it is indistinguishable from the original file.

Every file and directory has owner, group and stored access rights for owner, group of the file and other users. There can be rights to read, write and execute set separately for each one.

File timestamps , also called modification, access and change times, are stored in i-node and updated every time given file is written into, read or its file rights or name are changed respectively. Furthermore, ext4 supports birth time, which is a timestamp that stays the same since creation of i-node.

File signatures or magic bytes are used to identify content type of a file. They are either in the beginning or at the end of file content. To recognize them, we can use utility `file`, which recognizes filetype, and in some cases can also output more metadata, depending on which type it is.

Forensic analysis of a filesystem on level this low is out of scope for this thesis, and thoroughly examined by numerous other papers and tools.

2.1.4 Installed software

There are numerous ways to get runnable software in a Linux system. Any file can get execute rights by `chmod a+x path_to_file`, and then be run.

Software can also be installed manually, usually from some compressed file, for example .tar.gz. First it needs to be decompressed, then script configure should be run to make sure of all dependencies, and in some cases convert `Makefile.in`, which is a template, into `Makefile`. Then is executed command `make` to build software and `make` `install` to copy it into their correct locations. Both of these commands act according to `Makefile`[16].

Another way is to install from package in format recognized by some package manager. Two most common are `.deb` and `.rpm` formats. These can be either downloaded from software manufacturer site or get into computer any other way. Managers can be used not only to install them, but to modify or remove them as well.

Most system also have high-level package manager, that handles communication with repositories - storages of packages on the internet, and can fetch new software user wants to install, resolve all its dependencies and also automatically update packages already installed.

Debian-based package managers

High-level package manager here is for example `apt`, which can have more high-level and user friendlier front-end interfaces such as `aptitude` or `synaptic`. It uses `dpkg` for package manipulation.

We can list installed packages either from `apt` - with command `apt list -installed` or from `dpkg` - `dpkg-query -l`.

We can read them directly from the file used by dpkg - `/var/lib/dpkg/availible` for all available packages and for even more information, for example whether package is marked for removal or if it is installed, we can look into `/var/lib/dpkg/status`. This file is also backed up daily in `/var/backups`, so we can also see how it was changing.

Furthermore, managers are by default logging into files `/var/log/dpkg.log` and `/var/log/apt/`.

RPM-based package managers

Here, package manipulation is done with `rpm` and high-level tools are for example `yum` and `dnf`.

List of installed packages from `rpm` can be queried by `rpm -qa`, where parameter q stands for query and a for all. Files, where all this data is saved are in folder `/var/lib/rpm` and are in format of Berkeley DB.

From `yum`, all installed packages can be queried by command `yum list installed`, where for each package is shown its name, version and which repository it was installed from.

Their files are saved in `/var/lib/yum/history` and `/var/lib/rpm` directories and they log into `/var/log/yum.log` and `/var/log/rpm` files[29].

2.1.5 Logging

Linux logging is a convenient way to save messages from software in a general way. It can be handled by several managers, most common are syslog, syslog-ng and rsyslog. There are minor differences between them, we will however focus on rsyslog, because both Ubuntu 16.04 and CentOs 7 by default use it.

On a typical system, `/var/log/` is the default logging folder.

rsyslog

Its configuration is usually saved in `/etc/rsyslog.conf`. Currently it supports three syntaxes: basic, also known as sysklogd, advanced known as rainerscript and legacy, which should not be used, because it is obsolete [7]. Basic syntax is similar to following example:

```
$IncludeConfig /etc/rsyslog.d/*.conf
auth,authpriv.*                 /var/log/auth.log
*.*;auth,authpriv.none          -/var/log/syslogot
```

Basic syntax can contain these types of lines - commentaries, directives and most importantly, rules by which logging is done. Also backslash '\' on end of the line concatenates it to the next one. Commentaries start with '#' and are ignored, as well as empty lines.

Directives start with '$' and set properties for logging daemon[22]. They can be used to load modules with $ModLoad, like omrelp, with which comes support for RELP protocol or omusrmsg, which can send logs to all logged in users. Directive $IncludeConfig can extend configuration by its parameter, for example by default there is extension by all files in `/etc/rsyslog.d/` that end with `.conf`. They are also used to define templates of logs and output channels with $template and $outchannel respectively.

Rule lines in basic syntax are comprised of filter and action, which are separated by spaces or tabs. [22] Filter is used to select for which logs action should be taken and is of format "facility.severity" where facility identifies subsystem log message came from and severity states how important it is. Both can be represented either as a keyword from predefined set [22] or numerical value corresponding to it. There are 8 levels of severity, lowest being 7 - debug and highest 0 - emerg or Emergency. By default, filter logs all messages with set level and all with lower value. Rsyslog also recognizes '!' and '=' before severity and then logs messages except set level or only set level. There can also be multiple facilities defined, when they are separated by ',' and '*' can be put in facility or severity field and means that system logs all possible values at that field. Each line can also have more filters if they are separated by ';'.

Action says what should be done with log message and can be of several types. Basic writing to a file is done by just setting an absolute path to the file. If it is prefixed with '-' it means that content of file will not be synced with each write, if it is globally enabled. Special handling is applied for terminals and consoles. Logs can also be written into named pipes, if its path is starts with '|'. They can also be sent through network by UDP or TCP, by specifying ip address with prepending '@' and '@@' respectively. Discarding messages can be done by keyword 'stop'. Action can also be a script, that is run with log message as a parameter. It starts with '^' and contains path to the script.

For each of these actions can be set a template by appending it with ';' and its name. There are several hard-coded templates like RSYSLOG_TraditionalFileFormat, which is usually default one, defined with directive $ActionFileDefaultTemplate, or it can also be a user defined one. All of this can give us a good image about where system logs are located and in what format.

Log rotation

Log rotation is used to conserve free space on drive and archive old logs. Most widely used tool is `logrotate`. Its configuration is located in `/etc/logrotate.conf` in this format:

```
weekly
su root syslog
rotate 4
include /etc/logrotate.d
/var/log/wtmp {
    missingok
    monthly
    create 0664 root utmp
    rotate 1
}
```

It consists of options specifications. Lines inside curly brackets are local to filename or filenames directly before them, otherwise lines are considered global. Latter options overwrite previous.[14]

Directive `include` takes a filepath as a parameter and includes the file, or in case path leads to directory, all files in its place. Directives `yearly`, `monthly`, `weekly`, `daily` and `hourly` say how often file should be rotated, while `rotate` says how many rotations are kept. `Su` is used to specify which user and group executes rotation, `Create` says which user and group are owners of old log files and optionally, what are its access rights. User in `su` has to be able to create files as specified by `create`. Custom scripts, that will be executed by sh shell on various points in rotation can be set by `prerotate`, `postrotate`, `firstaction`, `lastaction` and `preremove`. Last executed rotation for each file is recorded in `/var/lib/logrotate/status`.

Main log

With default settings, file `/var/log/syslog` for Ubuntu and `/var/log/messages` for CentOs contains main logs. According to default configuration of rsyslog, this file catches all generic non-auth logs. This file contains a wealth of information on everything from system state to messages of running programs, but it is elaborate filter out unimportant logs.

Authentication log

In Ubuntu `/var/log/auth.log`, in CentOs `/var/log/secure`, this log contains messages that are related to user authentication mechanisms.

Successful logins are also saved in `/var/log/wtmp`, and unsuccessful ones are in `/var/log/btmp`. These are not in plaintext format, and are read with commands `last` or `lastb`[12]. Optional parameters for extracting most of the information should include `-time-format iso` for time listing in ISO-8601 format, `-x` for output to include system shutdowns and runlevel changes, `-i` to show ip addresses along with usernames for non-local logins and `-w` for usernames to be displayed in full, otherwise they are shortened to 8 characters. We also need argument `-f` for specifying file to be read, otherwise they are read from default locations. Output contains time of login, username, what terminal was used and for each login in following format:

```
johndoe tty7 0.0.0.0 2018-04-25T16:20:53+0200-2018-04-25T16:20:55+0200
johndoe tty7 0.0.0.0 2018-04-25T16:20:49+0200-2018-04-25T16:20:53+0200
```

Audit logs

These are done by daemon `auditd` with configuration files in `/etc/audit/audit.conf` and rules for logging in `/etc/audit/audit.rules`. They can be also configured by command `auditctl`. Its logs are by default saved in `/var/log/audit/audit.log`. These can be searched through using command `ausearch` and `aureport`.

These logs can trace many things, depending on its rules. They can, among others, watch files and directories for access or modification and watch for system calls. Many logging rules however can significantly limit system performance.

2.2 User artifacts

2.2.1 Browser artifacts

In this subsection, we examine few of the most widely used web browsers[26][6] available for Linux - Mozilla Firefox and Google Chrome/Chromium. It is important to note, that statistics are based on http user agent, which can be easily spoofed, and statistics do not distinguish operating system.

Mozilla Firefox

Firefox is one of the most popular browsers. It is open-source and on some Linux distributions it is bundled as a default web browser. On standard installation, it uses folder `~/.mozilla/firefox/` as a storage[17]. In it, file `profile.ini` contains information about every user profile [18] in following format:

```
[General]
StartWithLastProfile=1

[Profile0]
Name=default
IsRelative=1
Path=yadwhz35.default
Default=1
```

Field StartWithLastProfile refers to setting in Firefox to start under last used profile[30], which is signified by variable Default. Path is a location to profile folder, and IsRelative modifies, whether it is relative to location of local Firefox folder or an absolute path.

Usually, profile folders are located in local folder, default one being named with random string concatenated with `.default`, in this example, `yadwhz35.default`
In each profile folder, there are several noteworthy files containing forensic artifacts.

places.sqlite is an sqlite database for storing history, bookmarks and downloads in one place. It stores all its data in following tables: [21]:

moz_annos contains additional information mozilla has to its entries in table moz_places. Most notably, it contains most information on downloaded files. Each entry contains its id, reference to entry in moz_places, type of its content, as a reference to moz_anno_attributes, the content and times of its addition and modification.

Each dowloaded file has two entries here. One contains path to the file and is of type `downloads/destinationFileURI`, the other one is of type `downloads/metaData`, which contains json dictionary of metadata about download.

moz_historyvisits each entry is a description of historical visit similar to following:

```
id|from_visit|place_id|visit_date|visit_type|session
476|0|419|1524614714758544|2|0
```

For each entry there is its id, from_visit, that is id in the same table of visits, from which user clicked through, place_id, that refers to moz_places and visit_date, that is time of the visit in UNIX epoch time in microseconds. If from_visit is equal to zero, it means user did not come from any site.

moz_places here Firefox keeps information about each place that it knows of, whether it was visited or added as a bookmark in similar entry:

```
id|url|title|rev_host|visit_count|hidden|typed|frecency|
419|http://fmph.uniba.sk/||ks.abinu.hpmf.|1|1|1|25|

last_visit_date|guid|foreign_count|url_hash|description|preview_image_url
1524614714758544|QvPh9iRzbHcj|0|125507778799192|
```

For each entry, is here unique id, universal resource location address of the place, its title, if it was provided, time of last visit and frecency[10], which is combination of words frequency and recency, and is used to assign single score for both of these criteria.

moz_bookmarks as the name suggests, it contains information on bookmarks user saved in browser. When creating one, there can be specified a keyword and a number of tags, by which can be bookmarks filtered. Each database entry, contains its unique id and type, which can be a number from 1 to 3, 1 for bookmark or a special folder such as Downloads or History, 2 for a folder and 3 for visual separator. It also contains field parent, which signifies id of folder under which this bookmark is saved. All tags have parent set on Tag folder. Field fk refers to id in table moz_places, where are url and all parameters such entry should have. This id is also referred from moz_keywords, where value of keyword for a bookmark is stored. Each entry also contains dates of creation and last modification. Each tag is also a folder, in which copies of all bookmarks that are tagged with it are saved.

formhistory.sqlite In database file `formhistory.sqlite` are saved fields and values used for autofill - information user input on a website, and can be automatically filled on subsequent visits in database similar format:

```
id|fieldname|value|timesUsed|firstUsed|lastUsed|guid
1|user|johndoe|1|1512244821926000|1512244821926000|P1lETCYVQKuv7otl
```

Each entry contains time it was created, as well as last used, both in Unix epoch time in microseconds, name of field, its value, and number of times it was used.

logins.json contains an array of saved passwords. It replaced `signons.sqlite`, which served the same purpose in older versions, and was in format of sqlite database. For each entry there are names of fields for username and password, times of creation, last use and last change, as well as encrypted username and password. To be more specific, their respective fields contain base64 coded ASN1 serialized object, which contains initialization vector and ciphertext encrypted with 3DES cipher in CBC mode. Key for their decryption is saved in `key4.db`.

key4.db is a sqlite database, which replaced `key3.db`. It stores all information needed to decrypt password to decipher entries in `logins.json`. It has two tables - metaData and nssPrivate. The first one is used to check the user chosen master password - when its contents uncipher into $'password - check\backslash x02\backslash x02'$ it is the right one, and can be used to decipher actual password from table nssPrivate.

cookies.sqlite stores web browser cookies. Only table `moz_cookies` contains entries, each representing one cookie:

```
id|baseDomain|originAttributes|name|value|host|path|expiry|
327|google.sk||NID|119=qR[shortened]qn7_ULRI|.google.sk|/|1528988150|

lastAccessed|creationTime|isSecure|isHttpOnly|inBrowserElement
1513176950865844|1512231910870960|0|1|0
```

Each entry contains its unique id, name and value of the cookie and other parameters, which can be set - host, which is url, from which cookie originated, domain and path, which default to host if not set, expiry, which is number of seconds since Unix epoch after which is cookie invalid and whether cookie has set parameters secure or HttpOnly, which mean that cookie should be sent only by secure connection e.g. HTTPS and not HTTP, or only sent by HTTP and HTTPS methods, not javascript respectively. Each cookie also has time of its creation and last access for internal purposes in Unix epoch time in microseconds.

sessionstore.js contains information about currently opened tabs, and can be used for tab restoration after closing. It is in json format and possibly compressed and named sessionstore.jsonlz4, which can be decompressed in python[15]. Its main structure is a dictionary. Under key 'cookies' it contains session cookies, enabling for session restoration. Under key 'session' it contains metadata about session such as time it started and when it was last updated. Key 'selectedWindow' only informs which in list of windows should be active. Under key 'windows' is a list of dictionaries, each of which contains information needed for restoration of concrete window, such as its size and position on screen, along with all opened tabs under 'tabs' and index of active tab under 'selected'. For each tab, under key 'entries' is list of dictionaries, each of which inform on one page in this tab history. It also has a time of last access.

Cached websites can be found in folder `~/.cache/mozilla`.

Google Chrome/Chromium

Google chrome and chromium are very similar, difference being, that chromium is open-source and google chrome, while based on chromium , also has number of proprietary features. That being said, locally stored data is almost identical.

Chromium stores local data in `~/.config/chromium` and google chrome in `~/.config/google-chrome`. There are folders for each user profile, usually one being named `Default`, which is folder for first profile, `Guest Profile` for guest user, and new profiles are usually named `Profile X` where X is a number, starting from 1. There are browser-wide folders, such as `Crash Reports`.

For each profile, there is a number of files containing forensically relevant data in its folder.

History is sql database, which contains browsing, search and download history. It stores information in tables:

urls similarly to moz_places table in Mozilla Firefox, it is a table, where are all through browser visited places, although it is not referenced from bookmarks. It has following scheme:

```
id|url|title|visit_count|typed_count|
1|http://fmph.uniba.sk/|Fakulta matematiky, fyziky a informatiky UK|1|1|

last_visit_time|hidden
13169140839261338|0
```

Place has its unique id, its address, title, if it was provided by http and time of last visit.

visits is table where all historical visits are recorded, similar to moz_historyvisits in this format:

```
id|url|visit_time|from_visit|transition|segment_id|visit_duration
1|1|13169140839261338|0|268435457|1|0
```

Each row represents one visit. Each has unique id and in field url reference to table urls, where we can find address of site visited. Column from_visit refers to id of another visit, from which user came to this site, and 0 if user came in some other way. It also has a time when it was loaded, which is formatted as a number of microseconds since 1st of January 1601 in UTC.

downloads tracks downloaded files. One row can look like this:

```
id|guid|
1|ce8ad348-f3c9-49e5-a098-4751cd29ebea|

current_path|
/home/johndoe/Downloads/SampleVideo_1280x720_10mb(1).mp4|

target_path|
/home/johndoe/Downloads/SampleVideo_1280x720_10mb(1).mp4|

start_time|received_bytes|total_bytes|state|
13169141641760714|10498677|10498677|1|

danger_type|interrupt_reason|hash|end_time|
0|0||13169141666257574|

opened|last_access_time|transient|referrer|site_url|
0|0|0||https://sample-videos.com/|

tab_url|
https://sample-videos.com/index.php#sample-mp4-video|

tab_referrer_url|http_method|by_ext_id|by_ext_name|
https://sample-videos.com/||||

etag|last_modified|
"34168d-a03275-5357ce5d96600"|Fri, 17 Jun 2016 17:43:52 GMT|

mime_type|original_mime_type
video/mp4|video/mp4
```

Each row represents one file, that was attempted to be downloaded. Most importantly it contains path to where it was downloaded, what size was the file, address of site, from which the download originated and times of start and end of downloading, in the same format as were visits.

Web Data contains autofill data - that is data input by user into fields on a website, that can be automatically filled by computer in following visits. It is a sqlite database. There are several tables, for browser supports more ways of autofilling data. There is table `autofill` used for basic substitution in following arrangement:

```
name|value|value_lower|date_created|date_last_used|count
login|johndoe|johndoe|1524667419|1524667419|1
```

Each entry simply contains name of the field and value with which it should be filled, value_lower, which is lowercase version of value, count of how many times this autofill was used and times of creation and last usage, unlike rest of databases, in UNIX epoch time. Tables starting with `autofill_profile` are meant for more complex filling in data, where all informations about user are saved, such as first, middle, last and full name in `autofill_profile_names`, phone number in `autofill_profile_phones` and email in `autofill_profile_emails` and tries to guess into which fields on internet page these entries fit. One profile is connected through identical guid in all tables. Browser also saves credit card credentials, which can be autofilled, with the exception of cvv number. Part of information is saved in table `masked_credit_card`, where it contains name of the holder, in what network the card is, last four digits of card number and expiration date. Table `server_card_metadata` then ties profile guid to id in `masked_credit_card` and also contains date of last use along with usage counter.

Login Data is table used to store passwords in plaintext. When browser is run, it usually detects whether system has kde wallet or gnome keyring present and if so, it stores passwords with them, otherwise in `Login Data`. This behavior can be controlled by using parameter `-password-storage=<kwallet|gnome|basic|detect>` when launching.

Database is in following format:

```
origin_url|action_url|username_element|
https://login.uniba.sk/|https://login.uniba.sk/cosign.cgi|login|

username_value|password_element|password_value|submit_element|
johndoe|password|johnpass||

signon_realm|preferred|date_created|blacklisted_by_user|scheme|[shortened]
https://login.uniba.sk/|1|13170284914861439|0|0|[shortened]
```

For each entry there is site, for which it is used, time of creation in a typical chrome form, names of fields to fill in and username with password in plaintext.

Cookies is a sql database, that contains all browser cookies in a scheme like this:

```
creation_utc|host_key|name|value|path|expires_utc|
13169141142726175|.uniba.sk|_gat||/|13169141202000000|

is_secure|is_httponly|last_access_utc|has_expires|
0|0|13169141142726175|1|

is_persistent|priority|encrypted_value|firstpartyonly
1|1|b'v11e\\x84\\xe5\\xe0\\xcfV(i\\x18\\x80\\x91 5\\xae\\xb8@'|0
```

For each cookie, there is its creation time, last access time and if provided, expiration time in typical chrome format, standard cookie parameters such as host and path, secure and http flags and name. Cookie value is unfortunately sometimes encrypted.

Bookmarks is a json file, which contains all data needed to construct saved bookmarks. It is a dictionary, where on first level are keys 'checksum', which contains check for integrity, 'version', which speaks for itself, and lastly 'roots'.

Under it is a dictionary, which consists of keys 'synced' where are synchronized bookmarks from other devices and in browser are called 'Mobile bookmarks', 'bookmark_bar', where are all bookmarks accessible from bookmark panel, and 'other'. Each bookmark is described by a dictionary, which usually contains date it was added and modified, again in typical chrome format, its display name and id, type, which can be 'url' or 'folder', depending on which it also contains key 'url', under which is address of bookmark, or 'children' where it is a list of similarly formatted bookmarks.

Saved last and current sessions can be read from files `Last Session`, `Last Tabs`, `Current Session`, `Current Tabs` and are in SNSS format. Website cache are stored in `~/.cache/chromium`.

2.2.2 Other user activity

In home folder of the user, we can find file `~/.bash_history` which contains list of last commands done by bash shell. Unfortunately, it does not say anything about time of their execution or their output. Similar file is `~/.sqlite_history` which stores database queries done through `sqlite3` utility or `~/.sh_history` which is for Korn shell[29].

`~/.bashrc` can give us some information on setup user uses, and also aliases for commands, which are by convention stored in `~/.bash_aliases`, but can be in `~/.bashrc` as well.

Folder `~/.ssh` is a default location for OpenSSH client on linux. It contains private and public keys of user, which may be encrypted. List of public keys, whose private keys are accepted as credentials is in file `authorized_keys`, and list of server public keys that this client knows and accepts in `known_hosts`. User-specific configuration is saved in `~/.ssh/config`, which overwrites default one in `/etc/ssh/ssh_config`.

Files, that were deleted by desktop environment Unity, KDE or XFCE, that are in trash, can be found in folder `~/.local/share/Trash`. There are two subfolders: `files`, where original files are stored and `info`, that for each file or folder contains its original filepath and date of its removal.

Chapter 3

Design and implementation

This chapter will describe structure we propose for implementation and usage of our implemented solution to extract and analyse forensic artifacts as described in chapter 2. Our implementation can be found on `https://github.com/rstevanak/fortool`.

3.1 Design

There should be two types of modules - parsers and analysers. Former take mounted image of a filesystem and output a file of artifacts in a more machine readable form and latter take that and try to extract information out of it.

3.1.1 Parsers

These are classes, that read one or more files which file path it got as argument, extract meaningful information, depending on its type and parse it into dictionary format with specified structure. All modules implement method `parse`, with argument `filename`, which returns python dictionary. These should work to implement extraction of artifacts that were defined in chapter 2.

Extraction using multiple parsers can be best specified by a configuration file. Module should also be given root to path of mounted filesystem to be examined, and run all specified parsers as if this was a root of a filesystem. All parsers should get root as input parameter too and if they parse another file in the filesystem, obligation to rewrite such file paths is with them. Next we propose configuration be of following format:

```
# parser_type   filename    place_in_resulting_artefact_tree
user_data   /etc    user_data
login_data  /var/log/wtmp   login_data.btmp
login_data  /var/log/btmp   login_data.wtmp
```

```
browsers.firefox ~/.mozilla/firefox  internet_browsers
```

Configuration can contain empty lines or commentary, if line starts with '#'. All other lines specify extraction by one parser module. Each line can be broken down into specification of parser to be used, filename that it should be given as an argument and path to where in a resulting dictionary should it be saved.

Columns are separated by any whitespace. Each column can also be written enclosed in single or double quotation marks if they contain whitespace. This is mainly for specifying filename, that contains spaces.

Filepath can also contain ~ which in linux usually refers to current user home folder. In this configuration it means, that parser is run for home folder of every user. These have to be extracted and under key 'user_data' in output dictionary, as is in `default_extraction.conf`. This is a feature, because some default filepaths, such as Mozilla Firefox or Google Chrome are defined relative to home folder of the user that installed it. However, when no such file is found, message is written into stderr for transparency, but execution is not stopped, because on a typical system there are a lot of user accounts, which do not belong to users, and these do not have for example Firefox installed.

Path in resulting dictionary is constructed as follows - each dot separated field is a dictionary key. First key is looked for in root of the resulting output dictionary, each next key in the value of previous one. If there is no such key, it is created, and as a value given another empty dictionary. For the last key, if it has no value, result from parser is placed there, if it does, in its place is created a list, which has two entries - first is the original value that was in that place and second is output from parser. This solves dilemma, what to do if more values are to be put in the same place in a way that avoids data loss. Analysers have to take into account that on some places, there can be lists instead of just one value.

3.1.2 Analysers

These classes take json format of parsed data to analyse it and produce output, that is human-readable or can be processed by other tools.

Timeline creation is important part of analysis, as described in chapter 1. Analyser for its creation should be easily configurable and flexible enough to use in other tools,like log2timeline[13]. What type of lines are outputted from input dictionary should be determined by configuration file in following format:

```
internet_browsers.*.profiles.*.history.*.time
    "{} - User visited {} under profile {}"
    internet_browsers.*.browser_meta.browser_type,
    .site,internet_browsers.*.profiles.^
login_data.wtmp.data.*.time
    "Successful login for username {} from terminal {} and ip {}"
    .username, .terminal, .ip
file_metadata.*.ctime
    "File {}  with permissions {} and uid{} gid{} was created"
    file_metadata.*, .permissions, .uid, .gid
```

Please bear in mind, that each line that starts with whitespace is in real file continuation of previous one. This line separation is only for demonstrative purposes. Each line is either a comment or represents one type of line to be outputted. There are three fields separated by whitespace.

First is path to timestamp, that this entry type will be sorted by. It consists of keys to dictionaries, separated by dots. It is resolved by getting the key from dictionary, which was a value of previous key, and first key is resolved from input dictionary root. Key can also be an asterisk '*', in which case there will be outputted line per each possible value for that key. This also works with lists.

Second is template of string to be outputted. This can contain '{}' to define where variables will be substituted and each can be also formatted in a python string formatting mini-language[1].

Third field is comma separated list of variables to be substituted into previously set places. Its format is similar to the first field, but asterisk '*' here will be substituted to be the same as the substitution in first column and '^' means key, not a value from the first column will be substituted. If key starts with dot '.', it will replace the last key of timestamp path. This is because usually there are many variables, which are contained in the same dictionary, and writing them all out would not be practical.

Successful bruteforce attack analysis detects subsequence of unsuccessful logins followed by a successful one for the same username, originating from the same ip address, within some time interval. How many bad logins, in what time frame can be influenced by parameters function analyse takes. Default ones are 5 bad logins within 600 seconds or 10 minutes. This automated analysis is done, because login data on system with several users is noisy and manual detection is laborious.

Algorithm works in a way, that achieves average case complexity of $O(n + m)$, where n is number of unsuccessful logins and m is number of successful ones, with a presumption, that input logs are sorted in chronological order. If any of input logs is not sorted, resulting average complexity is $O(n * log(n) + m * log(m))$

First, all successful and unsuccessful logs are divided into separate dictionaries, where key is a tuple of username and ip adress, and value is its timestamp. In average case should be putting a value into a dictionary $O(1)$, and this operation is done $n+m$ times, so average complexity of this stage is $O(n+m)$.

Then, sets of keys for successful and unsuccessful dictionaries are intersected. We iterate through keys of one, and for each we ask if it is as a key in the other. This has complexity of $O(n)$.

After that is for each user checked, whether their arrays of logins are sorted chronologically, which is done in $O(n)$ time, where n is length of searched field, and if it is not, it is sorted in $O(n * log(n))$ time. Worst case complexity of this stage is $O(n * log(n) + m * log(m))$, where n and m are counts of successful and unsuccessful logs.

Finally, all logs in a list are iterated through and queue of potential brute logins group is kept. Firstly, new log is added to the end of said queue, then all logs in it that are not in a timeframe within the new one are thrown out and then checked if there are at least preset number of bad logins to constitute brute force attack. If there are not enough we continue to another unsuccessful log, otherwise search in successful logins is launched, and if there is one within timeframe since the first log in queue, output line is created and if not we note what was the last searched index of successful logins that was not in the timeframe. When creating output line, we also add all unsuccessful logins leading up to the successful one for better readability. This stage has complexity $O(n+m)$, because we see each unsuccessful log exactly one time and each successful at most one time.

Hash search finds all files which hash value matches hash from the provided list. It first fills dictionary with all hashes to the files found by extraction, then goes through list of provided hash sums and looks for it in dictionary. Average complexity should be $O(n+m)$, where n is number of file hashes from extraction and m is how many hashes to look up were provided, as dictionary has $O(1)$ average complexity for getting and inserting an item.

3.2 Usage

This chapter describes usage of our implementation of forensic tool. It is assumed, that filesystem to be examined is mounted in read only mode and all files are readable.

3.2.1 Extraction

Extracting can be done either with only one parser, or multiple ones. Former is done with module `extract_file.py`. It has only one argument, filename, which is passed to chosen parser, and three parameters. First - `-p` or `--parser_type` is mandatory and will be asked for if not input. It specifies which parser should be used. Next, `-r` or `--root` says, what is a root of examined filesystem. It is not required with most parsers, but it is used for translation of some links in filesystem, such as when rsyslog configuration has directive include, and it is absolute path inside mounted filesystem, to get to the actual position of the file we need to prepend path to the root. Lastly, parameter `-o` or `--output` can set where output is stored.

Parsers that are in our implementation are split into packages and subpackages. For example parser for Firefox browser can be found in `parsers/browsers/firefox.py`, but when it is chosen in `-p` parameter, it should be referenced as `browsers.firefox`.

Extraction with multiple parsers can be done with `extract_root.py`. It has an argument to specify the root of examined filesystem and parameters `-o` or `--output` to set where resulting json should be written, if left blank it is printed into stdout, and `-c` or `--configuration` which is a path to configuration file to be used in format as described in subchapter 3.1.1. If left blank, it uses file `default_extraction.conf`.

Some parsers read other files, if they are according to its internal format extensions to original file, such as in `cron.d` or `logrotate.d`. Moreover, `rsyslog_config` parser not only reads its extensions, but also all log files that are mentioned inside it.

3.2.2 Analysis

Using analysers can be done only individually. Each should have its own interface, because usually these have more variation on how they should be run compared to parsers.

print.py only takes dictionary as an input and prints it using "pretty print"[2], what means that it uses newlines and tabulators to write each entry in neighboring list or dictionary directly under one another and keys of an item beside them, which makes it more readable to human eye.

timeline.py implements timeline creation as described in 3.1.2. It is run with one argument of filename to the extracted file and parameters `--output` or `-o` to specify where output should be written and `--config` or `-c`, which sets configuration file in corresponding format. If it is not used, default one named `timeline.conf` in the same directory as `timeline.py` will be used.

success_brute_login.py looks for successful brute force login with algorithm described in 3.1.2. It reads only data from `login_data`, which parses `/var/log/wtmp` and `/var/log/btmp`, and only if it is placed as in `default_extraction.conf`. Each series of bad or good logins is in consideration only if they are on same user and from same source ip.

It can be run with parameters `--threshold` or `-r` and `--timeframe` or `-f`, to define threshold of how many bad logins are considered noteworthy and in what timeframe respectively.

search_hash.py is a module, that checks all extracted file metadata for md5 hashes it gets. First and only argument takes path to file extracted by parsers and parameter `--hashes` or `-h` takes file where there is a one hash per line. On output it writes all found file metadata saved.

3.2.3 Example usage

We have a mounted filesystem we want to examine on `/media/forens`. This is how tool should be used:

```
python extract_root.py /media/forens -c extraction_config -o extracted
    This extracts everything, according to file extraction_config
    from filesystem mounted at /media/forens into file extracted
python analysers/timeline.py extracted -c timeline_config -o timeline
    Creates timeline according to timeline_config and extracted,
    outputs it into timeline
python analysers/success_brute_login.py extracted -t 400 -r 4
    In file extracted finds brute force logins with threshold of 4
    unsuccessful logins within 400 second timeframe
```

3.3 Extensibility

This program is written with extensibility in mind. It is supposed to be easy to look through and add new parsers or analysers to it.

Parsers only need method parse to be defined, which takes filename and filesystem root as inputs and outputs python dictionary with desired structure. Then, to be callable from `extract_file.py` and `extract_root.py`, it has to be placed into parsers folder, or one of its subfolders and its filename, without `.py` added into list ___all___ in file `__init__.py` in its directory.

If one wants to create new subfolder, one has to make `__init__.py` in it in format like this:

```
__all__ = ['log_rotation', 'rsyslog_conf', 'rsyslog_file']
from parsers.logs import *
```

There is item in field `__all__` for each module or subfolder in folder, and in import line there is path to new folder separated by dots. Also new folder name has to be added into its parent folder `__init__.py`.

This could be automated, however it is this way to control what files can be executed in a parsers directory.

Analysers can be written in any way, since they are run individually. If they are meant to be run from `analyse_file.py` it has to implement method `analyse`, which takes one argument, which is python dictionary and outputs list of strings to be printed out.

To work with parsers, analyser has to use format they output. Default extraction produces a dictionary formatted according to scheme in the appendix A. That schema is tree like representation of output dictionary. Each first word in a layer is key in a dictionary, and all keys in the same depth are in the same dictionary. Value of this key is in the next level depth. If after name comes `:array`, it means that value for the key in dictionary is not merely one instance of what is in next level, but an array of them. If name contains `:dict:[subkey]`, it means that value is not only one instance of whatever is in next level, but dictionary of them, and each is accessed by key described by [subkey]. If the name starts with $ it means that it is not a key in a dictionary but a value described by it. Every line should end with some $ starting element, but we omit it where it is obvious.

For example, if we come from root, it has key internet_browsers, under which is array of dictionaries containing keys browser_meta and profiles. If we choose latter, it is a dictionary, where keys are name of the profiles, and its values are next level dictionaries containing keys history, downloads etc. Under key history is then an array of dictionaries containing keys time, site or duration, under which are corresponding integers or strings.

Conclusion

In this thesis we introduced forensic tool for Linux desktop. First we created a comprehensive summary along with description of how and where are saved, of all forensic artifacts, whether system or user made, that can be found on a typical system. Then we created a tool, that can extract data on scheduled tasks, user accounts, metadata of all the files, installed software and logging configuration, as well as browsing data out of two major internet browsers. It can also create timeline out of all extracted data with complete timestamp, search through files with hashes and detect successful bruteforce attacks from logs used for logins. This all is done in configurable way, and tool is written to be expandable and transparent on how it works. In the table 1 is a comparison with other forensic tools for Linux.

Main contribution of this thesis is the tool, that can simplify work of many forensic analysts, with new features such as user enumeration or cron jobs parsing, that are not commonplace for Linux forensic tools. Another is summary of artifacts that can be usually found on most Linux distributions, along with their formats.

	File metadata	Browser artifacts	Log analysis	Timeline creation
Fortool	✓	✓	For logins	✓
OSForensic	✓	x	x	only for file metadata
TSK	✓	with plug-in	x	only for file metadata
Autopsy	✓	✓	x	✓
SMART	✓	x	x	x

Table 1: Comparison of forensic tools for Linux

Future work

In the future, we would like to extend this tool in many ways:

- Do file carving from unallocated space on drive using photorec
- Parse cache for internet browsers
- Make the tool more independent from operating system
- Make graphical interface to ease usage for people who do not like command line
- Implement searching through files based on keywords
- Implement log analysis

Bibliography

[1] 6.1. string — Common string operations — Python 3.4.8 documentation. https://docs.python.org/3.4/library/string.html#formatstrings. Accessed: 2018-04-26.

[2] 8.11. pprint — Data pretty printer — Python 3.6.5 documentation. https://docs.python.org/3/library/pprint.html. Accessed: 2018-04-26.

[3] anacron(8) - linux manual page. http://man7.org/linux/man-pages/man8/anacron.8.html. Accessed: 2018-04-26.

[4] anacrontab(5) - linux manual page. http://man7.org/linux/man-pages/man5/anacrontab.5.html. Accessed: 2018-04-26.

[5] Autopsy. https://www.sleuthkit.org/autopsy/.

[6] Browser market share. https://netmarketshare.com/browser-market-share.aspx. Accessed: 2018-04-25.

[7] Configuration formats - rsyslog 8.34.0 documentation. https://www.rsyslog.com/doc/v8-stable/configuration/conf_formats.html. Accessed: 2018-04-28.

[8] cron(8) - linux manual page. http://man7.org/linux/man-pages/man8/cron.8.html. Accessed: 2018-04-26.

[9] crontab(5) - linux manual page. http://man7.org/linux/man-pages/man5/crontab.5.html. Accessed: 2018-04-26.

[10] Frecency algorithm. https://developer.mozilla.org/en-US/docs/Mozilla/Tech/Places/Frecency_algorithm. Accessed: 2018-04-25.

[11] gshadow(5) - linux manual page. http://man7.org/linux/man-pages/man5/gshadow.5.html. Accessed: 2018-04-26.

[12] last(1) - linux manual page. http://man7.org/linux/man-pages/man1/last.1.html. Accessed: 2018-04-28.

[13] log2timeline. https://github.com/log2timeline/plaso/wiki.

[14] logrotate(8) - linux manual page. http://man7.org/linux/man-pages/man5/logrotate.conf.5.html. Accessed: 2018-04-28.

[15] lz4. https://pypi.org/project/lz4/. Accessed: 2018-04-25.

[16] make(1) - linux manual page. http://man7.org/linux/man-pages/man1/make.1.html. Accessed: 2018-04-28.

[17] Mozilla support for firefox profile. https://support.mozilla.org/en-US/kb/profiles-where-firefox-stores-user-data. Accessed: 2018-02-05.

[18] Mozillazine firefox profile folder. http://kb.mozillazine.org/Profile_folder_-_Firefox. Accessed: 2018-02-05.

[19] OSForensics. https://www.osforensics.com/osforensics.html.

[20] passwd(5) - linux manual page. http://man7.org/linux/man-pages/man5/passwd.5.html. Accessed: 2018-04-26.

[21] The places database. https://developer.mozilla.org/en-US/docs/Mozilla/Tech/Places/Database. Accessed: 2018-04-25.

[22] rsyslog.conf(5) - linux manual page. http://man7.org/linux/man-pages/man5/rsyslog.conf.5.html. Accessed: 2018-04-28.

[23] shadow(5) - linux manual page. http://man7.org/linux/man-pages/man5/shadow.5.html. Accessed: 2018-04-26.

[24] SMART for Linux |ASR Data. http://www.asrdata.com/forensic-software/smart-for-linux/.

[25] The Sleuth Kit (TSK). https://www.sleuthkit.org/sleuthkit/.

[26] User agent breakdowns. https://analytics.wikimedia.org/dashboards/browsers/#desktop-site-by-browser. Accessed: 2018-04-25.

[27] Data elements and interchange formats — information interchange — representation of dates and times. ISO 8601:2004, International Organization for Standardization, Geneva, Switzerland, 2004.

[28] SWGDE Digital and Multimedia Evidence Glossary. Technical report, Scientific Working Group on Digital Evidence, 6 2016.

[29] SWGDE Linux Tech Notes. Technical report, Scientific Working Group on Digital Evidence, 2 2016.

[30] Eoghan Casey. *Handbook of Digital Forensics and Investigation.* Academic Press, 2009.

[31] EC-Council. *Computer Forensics: Investigating File and Operating Systems, Wireless Networks, and Storage (CHFI), 2nd Edition (Computer Hacking Forensic Investigator).* Course Technology, 2016.

[32] Darren R. Hayes. *A Practical Guide to Computer Forensics Investigations.* Pearson IT Certification, 2014.

[33] G. Klyne and C. Newman. Date and time on the internet: Timestamps. RFC 3339, RFC Editor, July 2002.

[34] C. Lonvick. The bsd syslog protocol. RFC 3164, RFC Editor, August 2001.

Appendix A

This appendix contains description of structure created by default extraction of fortool. Its format is described in chapter 3.3.

```
internet_browsers:array  browser_meta  browser_type
                                       browser_version

                         profiles:dict:name  history:array  time
                                                            time_orig
                                                            site
                                                            duration
                                                            title

                                             downloads:array  filename
                                                              size
                                                              down_size
                                                              time_start
                                                              time_end
                                                              time
                                                              time_orig

                                             forms:array  time_last
                                                          time_last_orig
                                                          field
                                                          value
                                                          time_created
                                                          number_used

                                            passwords:array time_last
                                                            time_last_orig
                                                            site
                                                            time_created
                                                            time_created_orig
                                                            time_changed
                                                            time_changed_orig
                                                            encrypted_username
                                                            encrypted_password
                                                            username
                                                            password

                                             cookies:array  site
                                                            time_created
                                                            time_created_orig
                                                            name
                                                            value
                                                            encrypted_value

file_metadata:dict:path_to_file  atime
                                 mtime
                                 ctime
                                 crtime
                                 permissions
                                 uid
                                 gid
                                 filetype
                                 md5

logs  rsyslog  logfiles:dict:filename  lines:array  timestamp
                                                    hostname
                                                    tag
                                                    pid
                                                    message

               meta  directives:dict:name  $value
                     actions:dict:filter  $action
                     rainersctips:array  $script
      logrotate   logfile:dict:filename:array  $log_line
                  directives:dict:directive  $directive_value

user_data:dict:username home
                        shell
                        info
                        uid
                        main_gid
                        pass_hash
                        pass_age
                        pass_min_age
                        pass_max_age
                        pass_warn_before_expiry
                        pass_disable_after_expiry
                        pass_disabled
                        bash_history
                        sql_history

cron  system  jobs:array  cron_time
                          job
                          run_as_user
              daily_jobs:dict:filename  $job
              weekly_jobs:dict:filename  $job
              monthly_jobs:dict:filename  $job

      user:dict:username  jobs:array  time
                                      job

login_data  btmp  data:array  username
                              terminal
                              ip
                              time
                              time_orig
                              added_info
                  meta  begin_date
            wtmp  data:array  username
                              terminal
                              ip
                              time
                              time_orig
                              added_info
                  meta  begin_date
```

Appendix B

Appendix B contains implementation of Fortool as described in chapter 3. It is also available on `https://github.com/rstevanak/fortool`.