HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

COMPUTER SCIENCE

Malware Hunting

Bachelor Thesis

Departement of Computer Science
University of Applied Science Rapperswil

Spring Term 2017

Authors:	Nietlispach Oliver & Ehrbar Roman
Advisor:	Brunschwiler Cyrill
Project Partner:	-
Internal Co-Examiner:	Thomas Corbat
External Co-Examiner:	Benjamin Fehrensen

Abstract

Introduction

The analysis of potentially compromised workstations and servers has become daily routine for a security analyst. To help during the detection and analysis process, a triage toolkit is used which uses various methods to categorize data of a potentially comprised system. A good triage toolkit removes as much known data from the list, which leaves less work for the analyst. An approach to reduce the data set is using white- and blacklists of known software components. In a previous term project, a prototype called Maloney has been developed which sought to improve on existing solutions in matters of automation and recoverability. In this bachelor thesis the aim is to further analyze requirements and to extend Maloney.

Approach / Technologies

The bachelor thesis was separated into multiple week-long iterations for which the goals and results were individually defined. During these iterations analyses, approaches and solutions for individual requirements were formulated and implemented. Maloney is built on Java, Elasticsearch and The Sleuth Kit (TSK) and is an event-driven framework. An examination is broken down into smaller processes, called Jobs. These are run in sequence and generate events which then get passed on to further subscribed Jobs. The application uses Elasticsearch to speed up lookups of aggregated meta data. TSK extracts files and meta data from disk images. Additional features and technologies were added, such as MapDB for resilience and Jsign for the verification of signed software.

Results

Many new features have been added to Maloney during the bachelor thesis. Currently, the examination process itself supports hash and signature comparisons only. But further examination methods can be seamlessly added through the plug-in mechanism. These Jobs are now run in multiple threads. Tolerance to faults has been added through the inclusion of a recoverable, persistent storage for events. Even after an unexpected crash, the application can proceed with the examination. After all data has been extracted and examined, a report can be generated with categorization based on customizable rules. Alternatively, the data can be viewed in Elasticsearch or queried through a Command Line Interface (CLI).

Contents

1. Management Summary

Recently with public cases like WannaCry - a malicious software which demanded ransom to de-crypt users' data - computer security is coming more and more into focus of the public. To learn about these incidents and prevent any further intrusions, analyses are necessary. These analyses are conducted with forensic tool kits by specially trained forensic analysts.

A forensic tool kit is a software which helps security analysts determine whether or not a computer has malicious content on it. It was determined during a previous project that current forensic tool kits are lacking in terms of automation and reliability. For example Autopsy, a widespread open-source forensic tool kit, cannot in any way be automated, nor can the order of executed tasks be chosen. On the other hand Encase, a proprietary software, crashes frequently. And when it does, the whole process needs to be started anew. This is a huge problem when dealing with large amounts of data. With this gained insight, a new application prototype called Maloney was created which sought to improve on the faults of those other applications.

In this bachelor thesis, the prototype was further extended with functionality so it can support forensic analysts while working as autonomously possible. The application currently only has a small set of integrated examination methods. But new functionality can be added using plug-ins.

The stability of the application is paramount: On an crash, it can be restarted and will resume its work with only minimal loss of progress. With the addition of a filter to the application, uninteresting files - for example, any files which were created after an incident - can be left out of the examination. It is also possible to exclude potentially dangerous data, which could hinder the examination. Altogether, this enables better performance and resilience.

After all tasks have finished and the examination comes to an end, a report can be generated. The results in the report are categorized into known good, known bad and unknown files. The analyst is therefore able to gain a better insight into the data. If needed, new custom categories can be added to the report so that relevant data can be found even faster. For example, all files which were created after a date of an incident.

To conclude, Maloney enables an analyst to optimize his schedule and minimize the time effort needed for examining the data. Analysts need only to check the application periodically for the estimated time of arrival. After that, they can analyze, correlate and formulate their own conclusive report.

Date	Version	Change	Author
20.02.2017	0.0.1	Initial	Nietlispach Oliver
13.03.2017	0.0.2	Chapter Analysis added	Ehrbar Roman, Nietlispach Oliver
21.03.2017	0.1.0	Chapter Plug-in Architecture and early draft of Parallelism added	Ehrbar Roman, Nietlispach Oliver
03.04.2017	0.1.1	Chapter Parallelism finished	Nietlispach Oliver
24.04.2017	0.1.2	Chapter Identify Known Files finished	Ehrbar Roman
25.04.2017	0.1.3	Chapter Progress Tracker finished	Nietlispach Oliver
25.04.2017	0.2.0	Review II preparation	Ehrbar Roman, Nietlispach Oliver
02.05.2017	0.2.1	Chapter Fault Tolerance finished	Ehrbar Roman
09.05.2017	0.2.2	Chapter Case Management finished	Ehrbar Roman
08.06.2017	0.2.3	Chapter Reporting finished	Nietlispach Oliver
09.06.2017	0.2.4	Chapter File Exclusion finished	Nietlispach Oliver
09.06.2017	0.2.5	Chapter Software Signature Comparison finished	Ehrbar Roman
09.06.2017	0.2.6	Chapter Categorization finished	Nietlispach Oliver
09.06.2017	0.2.7	Chapter Simple Queries finished	Ehrbar Roman
15.06.2017	0.2.8	Chapter Summary finished	Nietlispach Oliver
16.06.2017	1.0.0	Revision and Publication	Ehrbar Roman, Nietlispach Oliver

Table 1.1.: Document change history

2. Introduction

2.1. Purpose and Scope

Maloney was created in a term project during the fall semester of 2016/2017 at the University of Applied Science Rapperswil (HSR) by Ehrbar Roman and Nietlispach Oliver. The tool was intended to be used for the process of detection & analysis of malware, as described in the *NIST Guide to Malware Incident Prevention and Handling for Desktops and Laptops* [SS13]. As stated in the term projects conclusion, there are missing functionalities before the software can be used in a production environment[EN16]. This document intends to explain the reasoning and mechanics behind the extensions to Maloney.

2.2. Audience

This document has been created for software developers and engineers who want to gain insight into the application called Maloney and the extensions done during the bachelor thesis project.

2.3. Document Structure

This document is separated into multiple sections. In the first section 3 the issues surrounding digital forensics and the process of Detection & Analysis is explained and requirements for a tool in that problem space are formulated. In the sections thereafter, issues are tackled one by one. Each section will contain the following:

- Description of the problem
- Possible solutions
- Decisions made
- Implementation details
- Discussion with advantages, disadvantages, and other noteworthy findings

3. Analysis

This section of the technical report concerns itself with the analysis of the problem space and current methods and tools to solve them. In the first part the field of digital forensics with a focus on the NIST Incident Response Life Cycle and its process of Detection & Analysis is outlined (section 3.1.1). In the second part of this chapter is a description of the capabilities needed for a forensic toolkit, such as extracting and viewing data from a media for further examination and analysis (section 3.2). These sections form the basis for the third section: Requirements for Maloney (3.3). Furthermore it is necessary to analyze whether or not the existing architecture of Maloney fulfills the requirements. The existing capabilities are therefore outlined (section 3.4).

3.1. Digital Forensics

Although the field of forensics has been going strong for a long time, digital forensics is a comparatively young field. There is no definitive framework for handling this kind of work yet, though many papers already have been created and published on this matter. Luckily they are all similar in their approach of the process itself. Because National Institute of Standards and Technology (NIST) is an established source for all matters computer security, its process model was chosen as a reference.

3.1.1. NIST Incident Response Lifecycle

There are different models for the processing of analyzing a system in order to identify intrusive software, or malware. Many of these models were created with a specific use case in mind, such as forensic police work or company internal workstation audits. Hence the slight differences of focus or content in the descriptions of the process. Though one often used guide for incident handling is *NIST Guide to Malware Incident Prevention and Handling for Desktops and Laptops*[SS13].

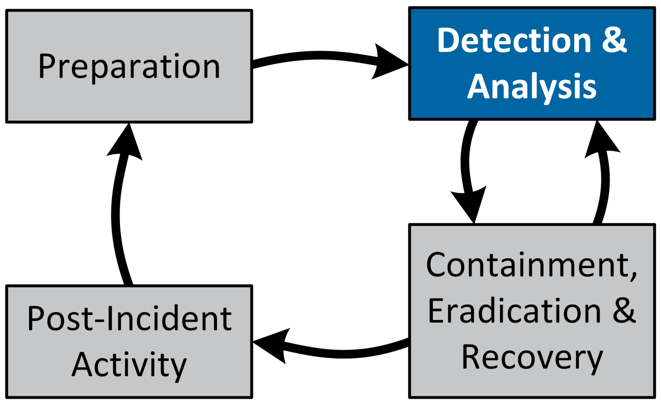


Figure 3.1.: NIST Incident Response Life Cycle[SS13]

The NIST Incident Response Life Cycle, as seen in figure 3.1, describes different phases in which malware incidents are handled. These can be divided into four general groups, which are paraphrased below[SS13]:

Preparation This phase is concerned with organization and infrastructure, including building know-how, coordination plans and tools to detect and handle incidents.

Detection & Analysis At this time, the prepared tools and organizations come into action. The intrusion has to be detected by either antivirus software, intrusion prevention systems or other means. The potentially infected systems are analyzed either actively, by executing it, or forensically. This forensic process is described in section 3.1.2. Oftentimes it is necessary to correlate different datasets (e.g. logs from 3rd parties such as networking with local system meta data) to get a definitive answer whether or not a certain component is a threat.

Containment, Eradication & Recovery During this phase, the infected systems are handled to hinder the spread and damage to the current and other hosts. Additionally, the aim is to completely remove any remaining infections. Additionally the systems and data get restored to a functional state.

Post-Incident Activity The lessons learned need to be documented for future incidents and revisions to structures, where needed, are made.

Although the *NIST Guide to Malware Incident Prevention and Handling for Desktops and Laptops* primarily lists networking tools to detect malware, the general structure of the process is still relevant in the case of this project. Especially the forensic process, which takes place inside the Detection & Analysis phase.

3.1.2. Forensic Process

The forensic process is part of the *Detection & Analysis* phase which is described in section 3.1.1. The special publication *NIST Guide to Integrating Forensic Techniques into Incident Response* states that "[...] forensics should be performed using the four-phase process [...]"[Ken+06]. These four phases are shown in figure 3.2.

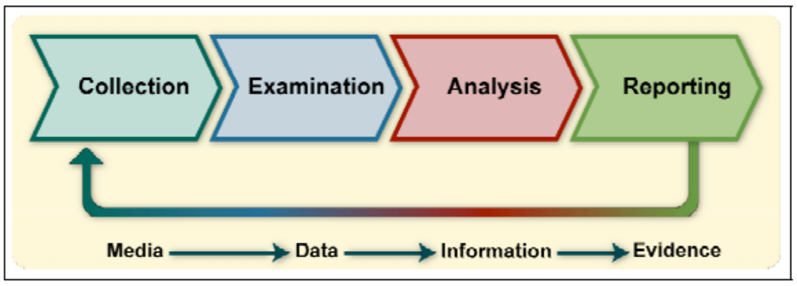


Figure 3.2.: Four phases of the forensic process according to NIST[Ken+06]

During the whole process it is important the integrity of the data is not harmed[Ken+06]. The whole process commences as follows:

Collection During this phase all related data is identified, collected, labeled and recorded[Ken+06]. This corresponds to the cloning of an image of a potentially infected system in a *post mortem* examination.

Examination This is the phase where forensic toolkits and techniques come into play. This phase can be automated. The collected data gets executed and extracted to get relevant information[Ken+06]. This is the place where Maloney and other tools will be used.

Analysis This phase is executed by an analyst, who tries to answer the question which started the process[Ken+06]. In case of this project, the question usually is whether or not a system has been infected with malware and what kind of malware it is.

Reporting In the final phase a report is generated with descriptions regarding the performed actions, the pending actions or recommendations to improvements to the organization or infrastructure to hinder incidents like this one in the future[Ken+06]. A forensic toolkit can assist in remembering the performed actions and in displaying the collected evidence in a useful manner.

3.1.3. Conclusion

For a toolkit to be useful in digital forensics it needs to be able to examine, execute and extract information from different data-sets and put them in a presentable format. During the analysis results from previous analyses should be available for comparison. The analyst then decides which information is relevant, and creates a report according to these findings. This results in various requirements which will be aggregated in section 3.3. Before that the various methods to examine and aggregate data needs to be analyzed. This will be the subject of the section 3.2.

3.2. Forensic Toolkit

As described in the forensic process in section 3.1.2, only raw data is collected during the collection phase. It is oftentimes necessary to further decompress or process those files before they can properly be analyzed. This can be automated or eased with forensic tools, which is the topic of section 3.2.2. But before these methods and tools can be described, the types of data which need to be examined and later analyzed are to be determined. This can be found in subsection 3.2.1.

3.2.1. Types of Data

In *NIST Guide to Integrating Forensic Techniques into Incident Response* is a set of various data which can be interesting to examine for an analyst who is looking for a malware infection on a system. To get a more complete picture of a system, it should not be gracefully shutdown. During a graceful shutdown swap-, hibernation- and other temporary files could be deleted [Ken+06].

- Configuration files (xml or plain files, Windows registry entries)
- Logs (OS- and application logs)
- Applications files (libraries, scripts, executables etc.)
- Application data files (documents, databases, binary encoded data etc.)
- Swap-, hibernation- and memory dump files
- Temporary files

3.2.2. Examination and Analyzing Methods and Tools

Depending on the type of data, various operations can help an analyst to perform faster. These operations should be included into a toolkit and are paraphrased from *NIST Guide to Integrating Forensic Techniques into Incident Response*[Ken+06].

Displaying Directory Structure The directory structure is a good place to get an overview of all the collected data and allows insight into the system, such as installed software or presence of certain kind of data.

Decompression Oftentimes data is saved in a compressed state or inside an archive. This kinds of data first need to be decompressed before they can be analyzed.

File Comparison The aim during this operation is to slim down the dataset, so that the analyst can already exclude well known data. One way to do this is hash comparison, in which the hash of any file found is calculated and compared with previously known good or known bad hashes. An example of such a set is the Reference Data Set (RDS) based on NIST National Software Reference Library (NSRL). It contains a list of known good software components.

String Searching or Pattern Matching Sometimes the analyst already has a suspicion of what the issue could be. A powerful search function or the ability to match certain patterns is useful in this case. As an example for a tool which performs pattern matching is YARA.

Accessing File Metadata Metadata contains useful information including creator, responsible organization, dates for the creation, changes and last access. Depending on the file, it contains other information which can be extracted and used to get a better picture of data in question.

File Viewer It might be necessary for an analyst to take a direct look into the file to correctly asses its content. When examining data it is important to keep in mind that the file ending of a file might not match with the contents of a file. File headers may provide a more exact description [Ken+06]. Manual examination is a very time consuming process and should be automated or supplemented with context for faster analysis whenever possible.

Included into the category of application data files in section 3.2.1 is data which sometimes is difficult to examine as they are in a special, perhaps proprietary, file format. Other tools are needed to analyze those files and to perform the above operations as stated in *NIST Guide to Integrating Forensic Techniques into Incident Response*. Additionally in case of an executable, it is oftentimes not viable to simply look at the binary code with a File Viewer. Examination of executables can be achieved with a sandbox tool like Cuckoo Sandbox in combination with YARA, to analyze the resulting patterns from execution in the sandbox environment.

With various tools in use, it is recommended to keep a log of the performed operations so that it can be reproduced[Ken+06].

3.2.3. Conclusion

There are various file formats, such as event logs, configuration files, or encoded application data to consider during examination and analysis. Additionally, the methods used to be able to closely examine the data can vary. Therefore a flexible, expandable toolkit is necessary. A list of requirements for such a toolkit can be found in the section 3.3.

3.3. Requirement Analysis

The forensic process, as described in 3.1.2, requires the integration of a vast variety of different approaches. The requirements can be divided into two groups, functional and non-functional. Whereas the functional requirements describe what the software should do, the non-functional requirements describe how it should behave. The latter have a big influence in the architecture of the software.

The requirements are designed to fit the scope of this project as defined in the initial assignment and the results of discussions between the project team and the supervisor.

3.3.1. User

The user of this software is an experienced computer user who is skilled in scripting, programming and forensics. He has a deep understanding in the different types of malware and how they can be identified on a computer system.

3.3.2. Functional Requirements

These requirements are prioritized into *MUST* and *CAN*, whereas *MUST* is the highest priority and has to be fulfilled by the end of this project. Some requirements were already fulfilled at the start of the project. These will be listed in subsection 3.4.2 Fulfilled Requirements.

Title	**FR-1 Examine Disk Images**
Priority	MUST
Description	The application accepts non-encrypted disk images (non-volatile data) of a computer as input and can perform the examination on this. It makes the containing files of the image available for further examination. All by Windows commonly used file systems need to be supported (NTFS, FAT, exFAT).

Table 3.1.: Definition of FR-1 Examine Disk Images

Title	**FR-2 Examine Directory Structures**
Priority	CAN
Description	The application accepts a root directory of a mounted file system as input and can perform the examination on this. It makes the containing files of the image available for further analysis.

Table 3.2.: Definition of FR-2 Examine Directory Structures

Title	**FR-3 Collect Metadata**
Priority	MUST
Description	The application collects metadata of every examined file for further analysis. The metadata should include, but not conclusive, following attributes: file modification, creation and access time, file path and name.

Table 3.3.: Definition of FR-3 Collect Metadata

Title	**FR-4 Automate Process**
Priority	MUST
Description	The application is able to perform a set list of tasks (gathering information, examining, generating a report) without the need of action from the user except an initial starting command. An analyst needs to specify the minimum required tasks to achieve his goal.

Table 3.4.: Definition of FR-4 Automate Process

Title	**FR-5 View Metadata**
Priority	MUST
Description	An analyst can access all collected data which was gathered during the examination in a human readable format. Using this data, he should be able to make conclusions about the incident.

Table 3.5.: Definition of FR-5 View Metadata

Title	**FR-6 Calculate File Hash**
Priority	MUST
Description	For every examined file, the hash (also known as message digest) should be calculated by the application. At least, the algorithms MD5 and SHA-1 should be supported.

Table 3.6.: Definition of FR-6 Calculate File Hash

Title	**FR-7 Identify Known Files**
Priority	MUST
Description	An analyst can provide a set of known files (Known Good, Known Bad or Custom) and the system tags every matching file with the name of the set and the matching category. Files are compared with the set based on their hashes, as described in FR-6 .

Table 3.7.: Definition of FR-7 Identify Known Files

Title	**FR-8 Manage Cases**
Priority	CAN
Description	An analyst may work on different cases on the same workstation. The examined data should not be mixed between different cases. The analyst needs the ability to specify an identifier for a case and can use this later to perform actions on the selected case.

Table 3.8.: Definition of FR-8 Manage Cases

Title	**FR-9 Compare to Golden Image**
Priority	CAN
Description	The software needs to compare two images and detect differences in added or removed directories and files, changed files and metadata. Therefore, the analyst provides a disk image (aka. golden image), which represents the baseline.

Table 3.9.: Definition of FR-9 Compare to Golden Image

Title	**FR-10 Examinate Windows Meta Data**
Priority	CAN
Description	When examining a disk image with a Windows installation, additional meta data should be extracted. Following data can be extracted: Windows version, patch-level, installed software, Windows Registry Files, Windows event logs, user accounts and groups, command history, recently accessed files and autostart applications.

Table 3.10.: Definition of FR-10 Examinate Windows Meta Data

Title	**FR-11 Configure Working Directory**
Priority	CAN
Description	An analyst can provide a path at the start of the application, which will be used as working directory. This directory contains the examined files and working copies generated during the examination and analysis phase.

Table 3.11.: Definition of FR-11 Configure Working Directory

Title	**FR-12 Exclude Files**
Priority	CAN
Description	There are multiple ways to hinder the analysis of a malware infection. Such malware, which exploits the toolkit and aims to harm the workstation (e.g. compression bombs), may be included in the disk image. Therefore, such potential dangerous files need to be recognized and further analysis prevented. Such files need to be marked as dangerous. An analyst can provide a list with names of files to exclude these from the examination.

Table 3.12.: Definition of FR-12 Exclude Files

Title	**FR-13 View Directory Structure**
Priority	CAN
Description	The analyst can view the directory structure of the examined file. This helps for example to identify the applications or how data was stored. The structure is enriched with file names, identifiers and modification, access and change times of the files.

Table 3.13.: Definition of FR-13 View Directory Structure

Title	**FR-14 Search for Strings in File Content**
Priority	CAN
Description	An analyst is able to search for strings based on the readable content of files. This search has to be independent of the file type, as long as it contains human readable text.

Table 3.14.: Definition of FR-14 Search for Strings in File Content

Title	**FR-15 Identify File Types**
Priority	CAN
Description	While examining files, the file type is determined based on their content. The type is stored as MIME media type as described in [RFC2046].

Table 3.15.: Definition of FR-15 Identify File Types

Title	**FR-16 Export Metadata**
Priority	CAN
Description	An analyst can export all metadata in a flat file format (e.g. CSV, hash sets) to include them in reports or for further analysis in a different application.

Table 3.16.: Definition of FR-16 Export Metadata

Title	**FR-17 Simple Queries**
Priority	CAN
Description	An analyst can provide a query to the application and recieves an output of all matching files. The query has to be a simple string. An analyst can also decide if the output should be displayed in the user interface or stored in a file.

Table 3.17.: Definition of FR-17 Simple Queries

Title	**FR-18 Validate Software Signature**
Priority	CAN
Description	An analyst wants to reduce the amount of unknown executables by using software signatures. Such signatures are provided by the manufacturer and are shipped with the software for validation.

Table 3.18.: Definition of FR-18 Validate Software Signature

3.3.3. Non-Functional Requirements

Following are the criteria specified which describes the operation of the system and not the specific behavior. These criteria are used to determine the basic architecture of the software.

Title	**NFR-1 Maintainability**
Sysnopsis	It is not possible to implement support for every file type, examination method and report type, because they may change over time. The application can be seamlessly extended to keep up with these changes and development is not dependent on a specific IDE.
Description	Changeability
Measureability	The software supports some kind of interface or extensible part where additional functionality can be added.

Table 3.19.: Definition of NFR-1 Maintainability

Title	**NFR-2 Efficiency**
Sysnopsis	During an incident, time is a critical resource. The application performs as fast as possible, and examines data only as far as specified by the user or needed to identify the threat.
Description	Time behavior
Measureability	Examination and analysis of data gets performed with all available system ressources and stops as soon as definitive answer to the threat of the data can be given (Known Good, Known Bad or Unknown).

Table 3.20.: Definition of NFR-2 Efficiency

Title	**NFR-3 Portability**
Sysnopsis	The application can be easily installed without the need for an IDE.
Description	Installability
Measureability	The installation is corresponds to the standards of the targeted platform on how to install software. The process is covered within a guide and can be performed without development tools.

Table 3.21.: Definition of NFR-3 Portability

Title	**NFR-4 Reliability**
Sysnopsis	The forensic process needs to deliver reliable results. Every execution with the same configuration needs to produce the same result. Even if the examination is interrupted.
Description	Recoverability
Measureability	The results of the analysis are persistent even after a crash of the application.

Table 3.22.: Definition of NFR-4 Reliability

Title	**NFR-5 Usability**
Sysnopsis	The user interface shows information about the progress and where the generated report and examined data is located.
Description	Understandability
Measureability	Progress, target locations of data and reports are shown in intervals of 3 seconds.

Table 3.23.: Definition of NFR-5 Usability

3.3.4. Conclusion

Some requirements were already defined in the term project “Malware Hunting” [EN16]. Back then, the focus was not on the process itself, but rather on the improvement of the basic framework and the integration of The Sleuth Kit (TSK). Due to a better understanding of the forensic process itself, more detailed requirements could be determined. They are suited for a basic framework or application which suits the needs of an analyst during the forensic process. Further decisions and development are based on the requirements noted above.

3.4. Maloney

Maloney is a new tool, and has not had a lot of development time. Still, it already fulfills some of the functional as well as non-functional requirements. But before those are discussed, the structure as of the beginning of this project is outlined.

3.4.1. Architecture

The architecture of Maloney is based heavily on the two ideas of extensibility and automation. The exact structure and designs of the application can be looked up in the technical report of “Malware Hunting”[EN16].

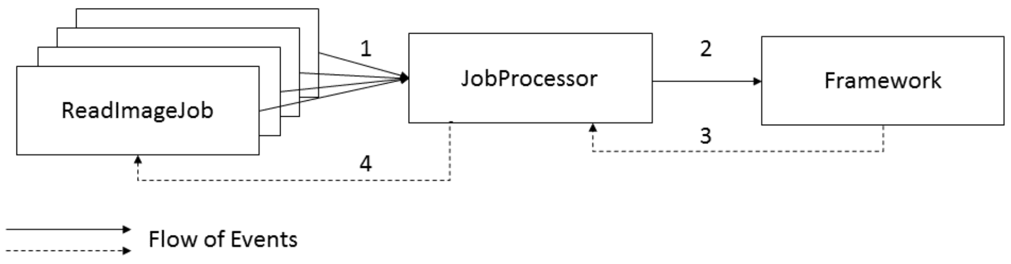


Figure 3.3.: Events get passed through the application

The whole application logic is structured into multiple encapsulated tasks, which are called `Jobs`. This can be seen in figure 3.3. These should be as small as possible and run once for each individual task. After completion, they create an `Event` which describes the result. These `Jobs` are executed by the `JobProcessor`. The `JobProcessor` forwards the generated events to the `Framework`. The `Framework` in turn decides what to do with these events[EN16]. This cycle and commences as follows:

1. An implementation of `Job` generates an event after completion
2. The instantiated implementation of `JobProcessor` notifies the `Framework` about those completions
3. The `Framework` decides which `Event` needs to be processed further by another `Job`
4. The instantiated implementation of `JobProcessor` starts new `Jobs` with the new `Event`

`Events` contains the name of the `Event`, information about the `Job` that created it and the unique identifier of the file concerned. `ReadImageJob` for example creates an `Event` for each file it extracted from a disk image. The instantiated implementation of `JobProcessor` then serves as a scheduler and (de-)multiplexer. Although at the end of the term project no multi-threading capabilities had been implemented. When the `Framework` receives the `Event`, it finds all interested `Jobs` for the new `Event`. These `Jobs` usually further examine these extracted result. An example would be queuing a `CalculateHashJob` into the `JobProcessor` after a new file has been extracted through the `ReadImageJob`.

3.4.2. Fulfilled Requirements

Some functionality has already been implemented into Maloney during the term project in the fall semester of 2016. The following requirements have already been fulfilled according to “Malware Hunting” in December 2016:

FR-1 Examine Disk Images TSK can extract files from an image to a desired location, which gets managed by Maloney’s `DataSource`

FR-3 Collect Metadata TSK reads meta data, which is then published into Maloney’s `MetaDataStore`

FR-4 Automate Process By building on an event based architecture, Maloney is able to follow predefined procedures which get passed to it by the Command Line Interface (CLI)

FR-5 View Metadata The meta data can be viewed with the help of Kibana on top of Elasticsearch

FR-6 Calculate File Hash MD5 and SHA1 hashes can be calculated with the *CalculateHashesJob*

NFR-3 Portability So far the application can be built and run without any IDE, only with the use of Gradle as the build tool

3.4.3. Conclusion

The existing components of Maloney were analyzed and described as an event-driven architecture which can be extended by creating new `Jobs`. The application thereby already fulfills some requirements as seen in subsection 3.4.2. With the problem space explored and the requirements defined and prioritized, the chapters 4 through 14 concern themselves with the solutions and implementations of the remaining requirements.

4. Plug-in Architecture

To achieve the goal of NFR-1 it is required to integrate some sort of interface to the framework. Through this interface new extraction methods should be added and made available for the forensic process. A common approach is to split the application in smaller chunks, which can be compiled independently and loaded as required at runtime. These small chunks, which extend the functionality, are often called plug-ins. This chapter describes existing approaches for plug-inss in Java and how the chosen approach is integrated into the architecture of Maloney.

4.1. Plug-in Frameworks

There are many existing frameworks which add support for plug-ins to an application. A brief research provided a list of different `plug-in` solutions for Java. Following are four well known frameworks outlined.

4.1.1. Java Class Loader

Due to the architecture of Java itself and the integrated reflection capabilities, a class to load other classes during runtime is already integrated and known as class loader. "Given the binary name of a class, a class loader should attempt to locate or generate data that constitutes a definition for the class. [. . .] Normally, the Java virtual machine loads classes from the local file system in a platform-dependent manner. For example, on UNIX systems, the virtual machine loads classes from the directory defined by the classpath environment variable." [17c] But to be exact, the class loader is not really a framework.

The foundation is laid by the class loader and could be used in the scenario to implement a plug-in architecture. The major drawback is that the application, which should be extended, needs to know the binary name of the class to load. It also has to know about the available methods and properties of the class. The latter can be addressed by using interfaces, which are implemented by the plug-ins. To use the class names, they have to be configured manually beforehand. New plug-ins are not automatically detected and can only be used after configuration through the user.

The ability to load classes at runtime and how the Java Virtual Machine (JVM) handles these introduces new problems. Using just the binary name, no version or location information are provided. Therefore, a different version as expected could be loaded, which may result in crashes or undefined behavior. A blog entry of Parlog describes this problem in detail, whereas he uses the term "JAR Hell" [Par15]. But as he concludes, this problem is not that common, especially when using a build tool like Gradle.

4.1.2. Java Service Provider Interfaces

This feature was made public with Java Version 1.6, but was used before that by the Java runtime. The following excerpt describes the goals of Service Provider Interface (SPI), which matches the requirement NFR-1 : "Developers, software vendors, and customers can add new functionality or application programming interfaces (APIs) by adding a new Java Archive (JAR) file onto the application class path or into an application-specific extension directory." [17e] This is achieved by using the SPI. "The set of public interfaces and abstract classes that a service defines" [17e] is added to the meta information of the Java Archive (JAR), which acts as the service provider.

Using this meta information, all implementations of a service by a service provider can be identified. Now, these implementations can be loaded, only with the knowledge of the interface. Behind the scenes, the class loader is used to load the concrete implementations. To make this work, the service provider needs to be located in the classpath, and the implementations require to have a public default constructor.

There are also some relevant limitations to the SPI: "You can use custom ClassLoader subclasses to change how classes are found, but ServiceLoader itself cannot be extended. Also, the current ServiceLoader class cannot tell your application when new providers are available at runtime. Additionally, you cannot add change-listeners to the loader to find out whether a new provider was placed into an application-specific extension directory." [17e]

4.1.3. OSGi

OSGi is the name of a corporation, the *OSGi Alliance*, and also a specification which "[...] facilitates the componentization of software modules and applications [...]" [17a].

The goal is to mitigate the problems initially introduced by the class loader and to improve the interoperability between components. OSGi brings many benefits by implementing the specification. According to OSGI one example of this is the solving of JAR hell: "OSGi technology solves JAR hell. JAR hell is the problem that library A works with library B;version=2, but library C can only work with B;version=3. In standard Java, you are out of luck. In the OSGi environment, all bundles are carefully versioned and only bundles that can collaborate are wired together in the same class space."[17b] There is also stated a support of dynamic detection of plug-ins: "Bundles can be installed, started, stopped, updated, and uninstalled without bringing down the whole system."[17b]

There are different implementations of the specifications. For example Apache Felix[1] or Eclipse Equinox[2], which is also the foundation of the Eclipse framework. Mainly, OSGi relies on meta information and an Application Programming Interface (API) provided by the frameworks. Many big applications rely on OSGi, as it is a de facto industry standard.

Both frameworks are quite complex and require a broad knowledge about OSGi. There are some examples specific to both frameworks. Even to understand the examples, which are more than a "Hello World", requires some time. No practical examples for a launcher, which starts all plug-ins from the command line, could be found. It was expected to pass some parameters or configuration from the CLI to the framework. This should not be mistaken with the shell of OSGi, which is used to control bundles.

4.1.4. Java Plugin Framework

This is another project based on Java with the goal to "[...] improve the modularity and extensibility of your Java systems and minimize support and maintenance costs." [JPF17] Based on the information provided by the project page [JPF17], the Java Plugin Framework (JPF) is inspired by Eclipse 2.0 and tries to mimic its plug-in framework. Therefore, it uses manifests to discover plug-ins at runtime.

JPF project seems to be discontinued. Based on the fact its current version was released for Java 1.5 in 2007, it is not guaranteed to be future proof. Even Eclipse has adapted its plug-in framework to support OSGi.

4.2. Solution Approach

Considering the plug-in frameworks and their approaches, as described in 4.1, their aptitude for this project has to be determined. All frameworks share the common goal of modularity and extensibility. They even behave similarly, by using embedded information from JARs.

[1]`http://felix.apache.org/`
[2]`http://www.eclipse.org/equinox/`

Regarding JPF, it is safe to say that OSGi is a better accepted solution by the industry. Implementing OSGi would exceed the project scope. This leaves SPI as the better choice as it is light-weight and all the requirements can be fulfilled. Also, it is possible to implement OSGi in the future without severe changes in the source code. With control over the source, runtime issues as described in 4.1.1, can be mitigated. The impact of such runtime issues are minimal in this project's scope. Also, by using classpath, it can be mitigated further.

> Facing the decision for a plug-in architecture, it was decided to implement the approach as described by SPI, and neglected using OSGi or JPF, to achieve improved modularity and extensibility, accepting the downside of possible runtime issues and the lack of support to manage plug-ins at runtime.

Gradle generates startup scripts for the installation packages, which defines the classpath environment variable for added dependencies. Therefore, as long as all plug-ins are built with the same dependency configuration like same packages and version as Maloney, version conflicts should not occur.

SPI introduce some terminology, which is used later again. For clarification of the terminology the following terms are as follows:

Service It is a well-known set of interfaces and (usually abstract) classes, mainly represented by a single type[17d]. The interface `Job` can be used as a service.

Service Provider The classes in a provider typically implement the interfaces and subclass the classes defined in the service itself. It provides additional functionality[17d]. An example of a service provider is the class `CalculateHashesJob`.

Service Provider Interface This is the API represented by the class `ServiceLoader` of the Java Runtime Environment (JRE). It is implemented in the application, which needs to be extensible through service providers implementing a specific service.[17d] The class `FrameworkController` would be an example of this.

The documentation *Class ServiceLoader* [17d] already contains a great example and some extensive information about specific implementation details.

To make it easier and deploy plug-inss with the installation package, a directory is required from which they are loaded. This cannot be done by just extending the classpath to this directory. A path to every JAR needs to be provided to work properly. This must be done before requesting all implementations of an interface or abstract class. Therefore the default class loader used by SPI needs to be extended. The class `URLClassLoader` already has a method to add new files at runtime, but this method is protected and cannot be accessed. With an extension, the visibility can be overridden, as shown in listing 4.1.

Listing 4.1: Custom class loader with public addUrl

```
public class CustomClassLoader extends URLClassLoader {
    public CustomClassLoader(URLClassLoader classLoader) {
        super(classLoader.getURLs());
    }

    // Make method public
    @Override
    public void addURL(URL url) {
        super.addURL(url);
    }
}
```

The directory for plug-ins is usually located beneath the bin-directory. Now, all JARs need to be added with `addURL` of `CustomClassLoader`. After enumerating and adding the files to `CustomClassLoader`, as shown in listing 4.2, this instance can be used with the SPI.

Listing 4.2: Resolving and adding plug-ins

```
URLClassLoader urlClassLoader = (URLClassLoader) ClassLoader.
   getSystemClassLoader();
CustomClassLoader myClassLoader = new CustomClassLoader(urlClassLoader);

try {
    // Extracts the plugins folder which is a sibling of the application
       folder (like libs)
    String path = FrameworkController.class.getProtectionDomain().
       getCodeSource().getLocation().getPath();
    File pluginFolder = new File(path).getParentFile().toPath().
       resolveSibling("plugins").toFile();
    File[] jars = pluginFolder.listFiles((dir, filename) -> filename.
       endsWith(".jar"));
    if(jars != null){
        for (File jar : jars) {
            myClassLoader.addURL(jar.toURI().toURL());
        }
    }
} catch (MalformedURLException e) {
    e.printStackTrace();
}
```

As mentioned in NFR-1 , the examination methods and reporting should be extensible. The examination methods are represented through `Jobs`. However there are no implementations for the creation of reports as of now. Therefore, this section focuses on the extension through concrete implementations of `Job`.

The SPI requires a class definition which describes the requested service. Additionally, a custom class loader can be provided to the `load` method, which is used to instantiate found classes[17d]. If `myClassLoader` is provided as second parameter, the service providers will be searched in classpath and in all JARss from the plug-in directory. The found classes will be instantiated and are available through an iterator. Listing 4.3 shows an example of how it can be used to load all defined implementations of the `Job` interface.

Listing 4.3: Loading of all available implementations of `Job`

```
Iterator<Job> iter = ServiceLoader.load(Job.class, myClassLoader).
   iterator();
while (iter.hasNext()) {
    Job job = iter.next();
    // do something with the job
}
```

However it is necessary for SPI to mark classes so that they can be detected. They need a special entry in the JARs meta information. This is usually done with a file named as the implemented interface or class. The file should contain a list of all classes to be published.[17d] In Maloney this file is located in "META-INF/services/ch.hsr.maloney.processing.Job" and contains a list of concrete implementations, as shown in listing 4.4.

Listing 4.4: Example service provider definition of `ch.hsr.maloney.processing.Job`

```
ch.hsr.maloney.processing.DiskImageJob
ch.hsr.maloney.processing.CalculateHashesJob
ch.hsr.maloney.processing.ImportRdsHashSetJob
ch.hsr.maloney.processing.TSKReadImageJob
```

4.3. Reference Implementation

A reference implementation for a plug-in was introduced with the task to identify known files, see 6.3. Therefore a new Gradle sub project was defined. The entire project structure of Maloney required an overhaul, because the framework was also the root project. The new structure can be seen in figure 4.1. All source files of the framework were moved into a new sub project and a new root project was introduced. The root references all sub projects. This brings three major improvements: First, references to other sub projects only by their name are possible. They have to be defined in the root project and are available for all other sub projects. Second, common tasks, for example a build, can be executed at once. Lastly, IDEs can detect dependencies correctly and provide additional features, like code completion. Before the refactoring, these features did not work all the time or tedious configuration was required to make it work.

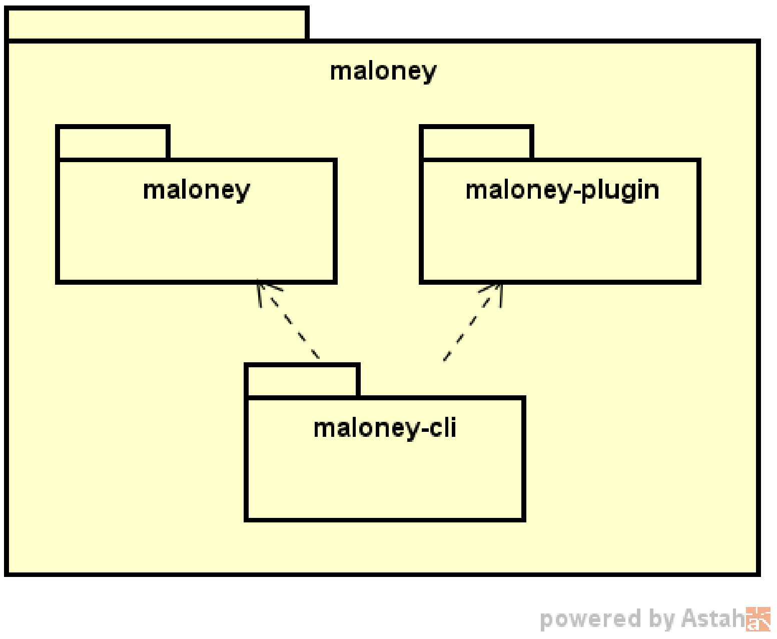


Figure 4.1.: The three sub projects maloney, maloney-cli and maloney-plugin in the root project maloney

4.4. Conclusion

By following the SPI approach, a simple and comprehensive solution can be achieved. It does not require an extensive knowledge of an API and the configuration to describe plug-ins is straight forward. Also, it has enough flexibility to be used in different parts of Maloney, such as reporting (see chapter 9). The same approach could also be used for reports, which can be provided and customized by the user.

Already existing implementations of `Job` will not be moved into their own sub projects, because this would just cost time and therefore has no priority. However these implementations can be detected and used the same way in the sub projects, due to the meta information in the base project. Newly created examination methods can be defined as sub projects.

The refactoring introduced with the reference implementation has led to a cleaner structure of the project over all and represents best practices. Further changes may be required, and should be tackled if there is enough time. For example removing the dependency to TSK from the framework core.

A specification like described by OSGi seems promising and should be considered in the future as complexity of plug-ins increase. This will be the case, as plug-ins depend on services provided by other plug-ins. Another reason would be if new functionality needs to be detected and managed at runtime. In this project, the complexity of a plug-in is expected to be low, as they only implement a few interfaces and have no dependencies on other plug-ins. Therefore, OSGi will not be implemented in the scope of this project.

5. Parallelism

To achieve the non-functional requirement of NFR-2 a mechanic to increase performance is necessary. This can be achieved through the use of better algorithms and efficient use of the present resources. An application with a single thread does often times not properly use the available resources and the time behavior is lacking. The most commonly used option is to introduce parallelism into the application by splitting up the logic into smaller tasks and execute them on separate processor cores, processors or even machines. Maloney's architecture enables the use of parallelism by separating the logic of the application into multiple smaller tasks, called `Jobs`. Additionally, the applications communicates through events. These could even be sent over the network. Three possible ways to achieve this will be outlined in this chapter. Afterwards, one of the options will be chosen and its implementation laid out. Lastly, the advantages and flaws are discussed.

5.1. Approaches for Parallelism

There are different approaches to achieve better performance. Generally, there are two options:

Scale-Up This is also known as vertical scaling. When giving a machine more resources, such as processing power and memory, these resources have to be used properly. As stated in *Java Concurrency In Practice*, "Threads are useful [...] for improving resource utilization and throughput."[Göe+06]. Therefore, splitting the work into multiple threads is one option.

Scale-Out Also known as horizontal scaling, this options spreads out the work over multiple machines or systems. This can be seen in distributed computing.

These two options are not mutually exclusive, both can be used to further increase the performance of a system.

5.1.1. Thread per Task

The first option is to create one thread per task as in figure 5.1. This enables running multiple tasks simultaneously on a system with more than one cores or processors.

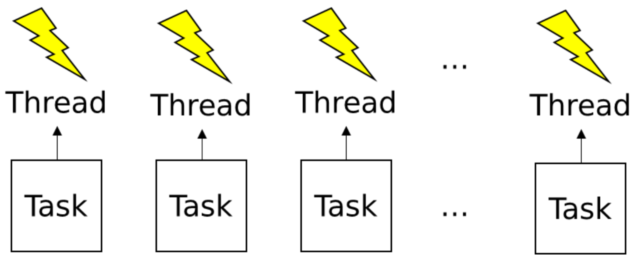


Figure 5.1.: Thread per Task

This takes load off of the main thread, but creates new problems, as stated in *Java Concurrency In Practice*[Göe+06]:

Thread Life Cycle Overhead The creation and teardown of threads is not free and consumes system resources. This varies according to the used platform, but nonetheless introduces latency and uses computation time.

Ressource Consumption When there are more threads then available processors or cores, they sit idle while still using memory.

Stability There is a fixed amount of available threads to every system which varies according to the platform and JVM parameters. Getting over the limit oftentimes results in an `OutOfMemoryError` which is difficult to recover from.

5.1.2. Thread Pool

This is the second option. In *Java Concurrency In Practice* it is stated that "[...] the task execution framework [...] simplifies management of task and thread lifecycles and provides a simple and flexible means for decoupling task submission from execution policy."[Göe+06] This simplification can be seen in figure 5.2. New tasks get queued up and as soon as a thread has finished its task, it takes a new one out of the queue. This way, threads get recycled and the amount of simultaneously running threads get limited.

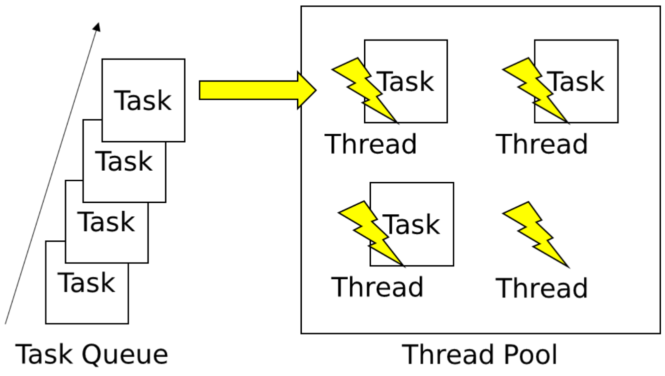


Figure 5.2.: Thread Pool

By taking over the management of threads, the thread pool fixes the issues stated in section 5.1.1. But it poses new issues as mentioned in *Java Concurrency In Practice*[Göe+06]:

Dependant Tasks When a task is dependent on "[...] timing, results or side effects of other tasks [...]"[Göe+06] it can create liveness issues, such as deadlocks.

Response-Time-Sensitive Tasks If a long running task gets submitted to a thread pool with few or only a single thread, responsiveness can become an issue.

Reused Threads Because thread pools reuse their threads, tasks which rely on thread-local variables may suddenly find unexpected values.

5.1.3. Distributed Computing

In *Distributed Systems: Concepts and Design* a distributed system is described as system "[...] which components located at networked computers communicate and coordinate their actions only by passing messages."[Cou+12] Splitting up the tasks not only into threads but onto different machines increases the potential performance. But by communicating over the network, latency, bandwidth and overhead to send messages need to be taken into account. Particularly transferring large amounts of data over the network is oftentimes not viable.

5.2. Decision

As described in the subsection 5.1.1, the creation of a thread per task has some shortcomings, as each thread creates overhead. Although it is unlikely in the case of Maloney that the amount of threads will create an `OutOfMemoryError` by itself, it surly will strain the system.

On the other hand, the issues that come with thread pools seem to be manageable:

- Tasks only get committed when their preconditions are met
- Any important management (i.e. user interface) can be separated from the thread pool
- Tasks can be implemented so that they do not rely on thread local variables

Distributed computing is interesting in concept, but poses more difficulties. The tasks have a chain in which they get performed: Perfectly modeling this process onto a distributed system gets increasingly difficult. Especially when new data gets extracted from an image or a file, it needs to be shared with the other workstations to fully utilize the distributed power. Sending this data over the network, when using images of around 200GB or bigger, does not seem viable.

The idea of distributed computing and usage of threads (per task and as pool) are not mutually exclusive. They can be implemented in the same system, although with much increased implementation time, as clients which get deployed on other systems need design and implementation time. The potential gain might not be worth the cost. Additionally, forensic work is oftentimes performed on a separate machine, isolated from the network, to minimize the risk of further spreading a malware infection.

Facing the decision for increased performance, it was decided to implement the usage of a thread pool, and neglect the approaches of thread per task and distributed computing, to achieve improved efficiency (NFR-2), accepting the downside of the potential problems that come with thread pools.

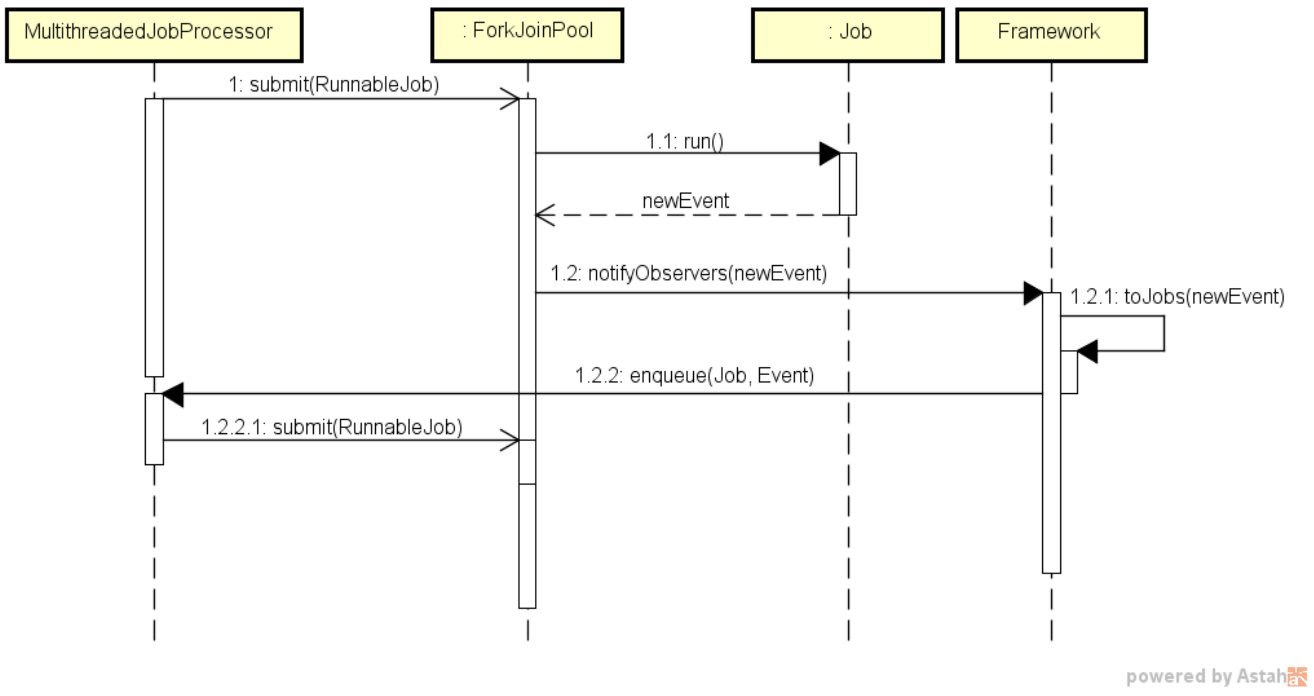
5.3. Implementation

With Java 1.7 Oracle introduced the `ForkJoinPool` which is an implementation of the `ExecutorService` interface. It executes all submitted tasks. These tasks are implementations of `runnables`, or for added syntactic sugar *lambda expressions*. It is not recommended to use the static `commonPool()`, because all new tasks, even ones from other parts of the application, automatically get submitted to it. Doing so could create unwanted side-effects when shutting said pool down, which is recommended for a graceful stop of the application. Instead, a custom pool should be used, which gets handled separately.

Inside the pool there is a Last In First Out queue for the submitted tasks. The pool can dynamically add, suspend and resume worker threads so that there are always some available. Normally the upper limit of worker threads is the number of processors available[Ora17]. This behavior can be overridden by creating a custom pool and specifying the level of parallelism in the constructor. Usually this is not necessary as this is best decided by whichever machine the application is run on.

The problem of deadlocks stated in section 5.1.2 can be avoided by only submitting tasks to the pool when they are ready to be run. This removes the possibility of idling tasks blocking all available threads and bringing the application to a grinding halt. It is important to keep the limited amount of threads in mind for the development of new `Jobs`.

Using a control thread which manages submissions to the thread pool and polling of results as well as tracking when the processing was finished turned out to be problematic: It resulted in deadlocks or unavoidable busy-waits. The usage of `Future` instances did not solve this problem either, as these still need to get polled by `isDone()` and then `get()` to get the result. The solution was to use a *Continuation Style*: To let the threads, in which the tasks were executed, notify the `Framework` of new events using the observer pattern. The `Framework` in turn enqueues the new tasks. This can be seen in the sequence diagram 5.3.



MultithreadedJobProcessor
: ForkJoinPool
: Job
Framework
1: submit(RunnableJob)
1.1: run()
newEvent
1.2: notifyObservers(newEvent)
1.2.1: toJobs(newEvent)
1.2.2: enqueue(Job, Event)
1.2.2.1: submit(RunnableJob)
powered by Astah

Figure 5.3.: Sequence diagram with pseudo code showing the sequence when queuing tasks (i.e. `Job`)

An option to implement this could be either to use `thenAccept` of `CompletableFuture` or to extend the anonymous implementations of `Runnable` in form of a *Lambda Expressions*. Because of easier readability the second option was chosen. A simplified extract of the source code of this can be seen in listing 5.1.

Listing 5.1: Submitting a new task to the pool via a lambda expression and notifying the `Framework` about changes through the *Observer Pattern*

```
    ForkJoinPool pool = new ForkJoinPool();

    runJob(Context context, Job job, Event event){
    pool.submit(()->{
        List<Event> result = job.run(context, event);
        setChanged();
        notifyObservers(result);
    });
}
```

Attention has to be paid to the type of a thread: When all non-daemon threads die, and the main thread is usually the only one, all daemon threads will be shut down. Threads inside a `ForkJoinPool` are defined as daemon by default. Therefore a mechanism to check whether all tasks have finished needs to be introduced. This was achieved by acquiring a semaphore at the start of a new task and releasing it again after it has finished. New tasks get queued up and acquire another semaphore, before the parent task has released its semaphores. When there are as many semaphores available as there were at the start, all jobs are confirmed as finished. To wait until all tasks have finished, the same number of semaphores are awaited as defined at the beginning. Therefore, the main thread uses the `waitForFinish` method to prevent exiting early. It replaces primitive counting methods and prevents busy-waits. To prevent deadlocks if multiple threads are using `waitForFinish`, all acquired semaphores are released immediately afterwards.

5.4. Conclusion

With the `MultithreadedJobProcessor`, a `JobProcessor` implementation which enables Maloney a better use of the available computational power. This increases the performance of the application and fulfills the non-functional requirement NFR-2 .

Using a thread pool introduces some restrictions: `Job` implementations must not wait inside of `run` on results produced by others, as it introduces possible deadlocks. To determine whether or not the `Job` can run, the corresponding `Event` should be used in conjecture with the method `canRun` to check if all conditions are met. Evaluation inside this method should be kept short. For accessing external resources or heavy operations, `run` should be used. Some deadlock scenarios may stay undiscovered on modern hardware and need to be verified on single core configurations.

For small tasks with low computing requirements or heavy disk requirements, the performance gain is only minimal. As of now, this is mostly the case. But as soon as there are longer running tasks or ones which require more computing power, the difference is more noticeable.

6. Identify Known Files

To address the requirement FR-7 , already known files should be marked. The simplest way to achieve this is by looking up the file hash in a set, which gets provided by the analyst. After the lookup of a file set, the results can be queried to gather a quick overview. This requirement has a precondition on FR-6 . This was implemented into Maloney during the term project and it includes a `Job` for indexing RDS archives.

This section focuses on the lookup of calculated hashes inside the indexed hash sets. It also shows how Elasticsearch is involved and where the limitations are. First, an approach is formulated. Afterwards already implemented parts will be revisited. Lastly the implementation gets described.

6.1. Approach

The best way to reduce the amount of files which are needed to be reviewed by an analyst is to compare them with already known files. All selected files get compared against files provided by the analyst and flagged, if they match. This could be done on bit level, but would require a lot of disk operations and storage for all known files. Hash algorithms can be applied on files to generate an unique fingerprint, which is already used for example in file integrity checks. Such checks rely on a set of hashes for each file to determine changes or defective files. Hash sets are also smaller when compared to the original files.

Hash sets can also be used to identify files. For every file the hash has to be calculated and compared against a set provided by the analyst. "Analysts should use validated hash sets, such as those created by NIST's National Software Reference Library (NSRL) project or personally created hash sets that have been validated [. . .]"[Ken+06] It is important to differentiate between hashing algorithms provided by this sets. Every hash set might provide its own subset of hash algorithms, in most cases only one. A lookup should be independent of the structure of hash sets and respect the algorithm.

6.2. Already Implemented Parts

Maloney already has some tasks implemented which are convenient when it comes to the task of identifying known files. The following implemented features will be examined: the tasks `CalculateHashesJob`, `ImportRdsHashSetJob`, and `ElasticHashStore`.

Hashes of examined files are calculated by the task `CalculateHashesJob`. These hashes are added as artifacts to an implementation of `MetadataStore`. In the case of this project, an implementation for Elasticsearch is provided and used as default. Artifacts get linked to file metadata using *nested datatype* [Ela17a]. These are stored and indexed separately and can also be queried independently.

An update script is in place to handle adding artifacts without re-adding already existing artifacts or loosing any. The said script uses parameters to create a new artifact and adds it to the existing ones. It is important to notice that script parameters need to be primitive types[EN16, section 2.8.5]. So, a deeper nesting of objects is not possible if this dynamic addition to the existing list should be kept as it is. Objects need to be serialized, for example as a string, to be used as a script parameter. This restriction is relevant at a later point, see 6.3.

The task `ImportRdsHashSetJob` indexes a provided RDS archive, which contains a set of hashes and information about operating system and product of origin. The implementation by `ElasticHashStore` already handles indexing and searches. Using elasticsearch to index and look up hashes later reduces the time for matching these against a file. And by using a separate Elasticsearch index, they can be managed and accessed independently from the analysis.

6.3. Implementation

Due to the fact that Elasticsearch handles relations different than other systems, the implementation needs to be adapted accordingly. This problem was observed and described in [EN16, section 2.8.3]. As a result, it was proposed to implement this using the *application-side join* approach. A new implementation of `Job` is used which is called `IdentifyKnownFilesJob`.

6.3.1. Querying Hash Sets

The new implementation is registered to receive events for new hashes (e.g. `MD5HashCalculated`, `SHA1HashCalculated`), which are produced by `CalculateHashesJob`. As soon as a new hash is calculated, it is compared to the set. For the lookup only simple search queries are used which provide all matches. One query uses a wildcard on the hash types, the other can be used to restrict the search on a specific type (e.g. MD5 or SHA-1). Although it is not possible to confuse the results of those two hash algorithms, they possess different levels of security, i.e. hash collisions in MD5 are more likely. The field declaration was introduced for additional clarity when the results are queried.

A *term query* is an exact match on a single field as seen in listing 6.1. To perform a search on multiple fields, a *multi-match query* is the weapon of choice as seen in listing 6.2. Every available search method has a corresponding factory provided by `QueryBuilders`.

Listing 6.1: Search statements for single field

```
SearchResponse searchResponse = client.prepareSearch(INDEX_NAME)
.setTypes(HASHRECORD_TYPE)
.setQuery(QueryBuilders.termQuery("hashes." + algorithm, hashValue)).
   get();
```

Listing 6.2: Search statements for wildcard search

```
SearchResponse searchResponse = client.prepareSearch(INDEX_NAME)
.setTypes(HASHRECORD_TYPE)
.setQuery(QueryBuilders.multiMatchQuery(hashValue, "hashes.*").type(
   MultiMatchQueryBuilder.Type.BEST_FIELDS)).get();
```

An exact match is forced by the mapping *keyword datatype* inside Elasticsearch. If no mapping is provided, Elasticsearch will generate it by itself. Normally text fields are *datatype text*, which gets analyzed. Analyzed means that it will be divided into single words: Punctuation and spaces are separators between words. This would be disadvantageous in the case of hashes, where an exact match matters. Therefore it is essential to know of what type the data is to write an appropriate query.

6.3.2. Potential Near Real-Time Issues

While elaborating this new feature, potential issues have arisen regarding *near real-time* feature of Elasticsearch. It manifests itself in generated artifacts which are not visible to the subsequent tasks. Elasticsearch's index is refreshed every second (per default), so only data which was indexed before the last refresh can be queried. This problem was already discussed in [EN16, section 2.8.6] regarding unit testing.

This potential near real-time issues mainly applies to queries. Accessing stored data by their ID will always force a refresh and return the entire document. A query can be configured to wait for a refresh.

This may result in a longer query time and is not recommended. For more information and how to change the behavior, see Elasticsearch Documentation[1]. Developers need to keep this in mind while using Elasticsearch as the data store.

It is no problem in the case of hash lookups against a separate index. Indexing of hash sets has to be done prior to a hash lookup, to guarantee that all entries are used. While examining an image, the hash index should not be changed. But Maloney will not prohibit it.

6.3.3. Storing Results and Data De-Normalization

Queries deliver multiple hits, or matches. Adding these hits as artifacts to a file is exactly what *application-side join* stands for. Adding an entire hit as an artifact is only possible as serialized object in a string. Searching for a single field inside this object introduces more problems. The field `artifacts.value` is interpreted as text, therefore a relation between single fields and values is problematic. For example, if all files should be found which are identified as Known Good, it would match all fields and not only on the type field. To solve this, the hit has to be de-normalized further. The types of artifacts were extended by using a dollar sign as separator between the type and field name (e.g. `ch.hsr.maloney.storage.hash.HashRecor`
This is done for the fields `type` and `sourceName`. If the hit is still saved as whole, it can be later used by reports or other tasks.

Because such a lookup is executed for every raised hash event, it may lead to duplicate artifacts while using hash sets which contain multiple algorithms. This is the case in RDS sets, but other sets often contains a single type. Because Elasticsearch has no way to remove duplicates, they need to be identified by the job and skipped. Hash sets have to be indexed before a lookup happens, otherwise there will be no hits nor artifacts.

A new event (`KnownFileFound`) is produced after a successful match. If there is no match in any of the supplied hash-sets, the file should be marked anyway. Therefore, another artifact is produced containing the tested hash algorithm and that the result was negative. This case also produces a different event, namely `KnownFileNotFound`.

6.3.4. Plug-in

The task `IdentifyKnownFilesJob` is the first implementation of `Job` in a separate JAR. It uses the newly introduced plug-in architecture as described in chapter 4 Plug-in Architecture, and acts as a reference implementation.

How to add custom functionality and develop a plug-in is described in chapter C Developer Handbook.

6.4. Conclusion

The task `IdentifyKnownFilesJob` is able to compare examined files with hash sets, which is part of requirement FR-7 . Files can be queried for the presence of generated artifacts and the type of the matching hash set. The other part of the requirement is depending on tasks to import such hash sets. As of now support for the most common set, RDS, is already implemented. Therefore the requirement is fulfilled.

To support hash sets in a different format additional tasks can be implemented. They can directly benefit from this implementation of a generic lookup.

An entry of a hash set can contain additional information about the operating system and product. This information could be used to identify installed applications or detect software which should not be present. Because different products sometimes use the same libraries or other files, this might lead to

[1]https://www.elastic.co/guide/en/elasticsearch/reference/5.0/docs-refresh.html

false positives. Conclusions based on this data must be taken with caution. It may be possible that it generates false positives by marking files as known, which should not be present on a given system.

A detection of duplicate hash entries is not intended. An analyst should be able to use different hash sets which may contain duplicates. Therefore, they are not removed in the lookup process.

There was an issue found concerning the extraction with TSK. An exception was thrown which resulted in extraction of zero-byte files. More information to this sporadic error can be found in subsection 15.2.1 Invalid FS_Info Object. It is important to note that although the extraction is faulty, zero-byte file hashes are handled the same way as other hashes and do not lead to any further issues.

7. Progress Tracker

Currently, the entire log output is displayed by the CLI. This is not really useful for a general overview of the progress. To achieve non-functional requirement NFR-5 a meaningful way to store and display progress inside the application is needed.

7.1. Analysis

It is difficult to determine how long the application will take to fully analyze the selected image or files. But some data can be helpful to figure out just how much of the image already has been analyzed, and to which degree.

The overall processing speed should also be tracked so that an estimated time of arrival (ETA) can be calculated.

As stated in NFR-5 Usability, this information has to be updated steadily and displayed to keep the user up to date on the current progress of the application.

7.2. Approaches

To get meaningful answers to the question of how far the processing is, various approaches can be taken. It has to be decided which metrics are relevant, the most sensible point to gather this information and how to convey this to the user.

7.2.1. Metrics

Maloney handles all the internal communication through instances of `Event`. These are sent on completion of a single task and contain information on the result. When tracking these, an insight into the current progress can be gained when put into relation with previous analyses. In addition, if the number of found files on an image and registered `Jobs` in the application are put into context with the number of observed events, tracking of the progress becomes viable. However, predictions are still problematic, because it cannot be foreseen how often a `Job` will be executed and how long it will take.

7.2.2. Location for Tracking

There are multiple locations where tracking of progress with the afore-mentioned metrics can happen:

Task-based Tracking of progress on the lowest level and smallest steps inside the application. On this level it is possible to deliver exact time behavior statistics of a task. Tracking on this level alone is disadvantageous, because it is impossible to know how many tasks are left. In addition each implementation of `Job` (i.e. each task) would have to introduce redundant code for operations which every task does. For example every task has a start and an end event which needs to get tracked.

Scheduler-based When using the scheduler for reporting of progress, some redundancy for reporting can be taken out from tasks. But because the scheduler only executes tasks and does not know all registered tasks, it cannot predict which task is the last one. In addition, the exact behavior of tasks is not known either.

Task-chain-based On the highest level, predictions can be made concerning the time-behavior and probability of execution of registered tasks (i.e. `Job`). This way the overall time can be estimated, but the effective result can vary greatly from the first estimation because the chain is not always executed to the last task, as a definitive answer to the character of th data may be given earlier.

These options are not mutually exclusive, they even complement each other in some aspects. For example task-specific events (e.g. new file found) may be tracked on a task basis, but overall task progress can be tracked on data acquired by the scheduler, thereby creating less redundant code in implementations of tasks.

7.2.3. Output

It is not enough to simply store the progress inside the application. It has to be displayed to the user somehow too. This can be achieved through different representations:

Graphical When the application is started a window is opened where the progress can be seen. The Graphical User Interface (GUI) displays the collected metrics and can use progress bars or cake diagrams to convey the information to the user.

Textual The collected metrics are printed on the CLI in a simple textual representation. This might happen in real-time or periodically to reduce required updates.

Designing and implementing a GUI consumes a lot of time, whereas simply printing out the numbers into the console is a more simple matter. Additionally a user interface consumes more system resources.

The collection of only the most basic metrics suffices, as a rough estimate should be enough to create an average processing speed of tasks.

> Facing the decision for a progress tracker, it was decided to collect basic metrics and print them with an estimated time of arrival in the console, and neglected using a GUI, to achieve non-functional requirement NFR-5 usability, accepting the downside of less appealing visuals.

7.3. Implementation

The interface as planned during the previous project was redone with a more simplistic but extensible design. An `enum` was created for standardized counters, which can be extended with custom counter names. This information can be passed to and processed by the implementation of `ProgressTracker` through the method `processInfo`.

The implementation created is called `SimpleProgressTracker` and it stores the information in a `Map`. The keys are either the custom name or the predefined `enum` values. The values of the map are counters, that get increased on every call. It was decided to refrain from using sophisticated estimations for the remaining time with regard to the task-chain. Instead the average processing speed of tasks over the last 30 seconds and a estimation of how long it will take to finish the remaining unfinished tasks gets calculated.

The `Framework` handles creation of progress information for new and processed tasks, to save redundancy in `Job` implementations. On the other hand any task-specific event, such as unpacking or identifying of known good or bad files gets handled by the respective `Job`.

For future extensions - and a possible GUI - the `ProgressTracker` gets injected through the `FrameworkControll` The `FrameworkController` in turn creates an instance of `ScheduledExecutorService` with a single thread to schedule the periodic output of the progress (see listing 7.1). This executor service is more reliable than `Timer`, because it will not stop executing the `Runnable` if the thread has crashed.

Listing 7.1: Thread which periodically prints the stored information in `ProgressTracker`

```
private ScheduledExecutorService scheduledThreadPoolExecutor =
    Executors.newSingleThreadScheduledExecutor();
    // ...
    scheduledThreadPoolExecutor.scheduleAtFixedRate(() -> {
        // ... Fill message ...
        System.out.println(message);
    }, START_TIME, CONSOLE_UPDATE_FREQUENCY_IN_SECONDS, TimeUnit.SECONDS)
       ;
```

For the calculation of the average speed the class `ETACalculator` in the `util` package has been introduced. It calculates the average processing speed over a specified amount of nodes, which is specified in the `RELEVANT_CYCLES` field. Currently the last ten measurements are used for the average (see listing 7.2). A `LinkedList` is used for the storage of these measurements, because it has the fastest insertion and deletion methods with *O(1)*.

Listing 7.2: Calculating the average task-processing speed

```
public double getAverageSpeed(){
    if(measurementList.size() < 2){
        return 0.0;
    }

    double speedSum = 0;
    int lastFinished = 0;
    long lastTime = 0;
    int avgElements = 0;

    for (Measurement c : measurementList) {
        if(lastFinished != 0 || lastTime != 0){
            speedSum += ((c.getFinished() - lastFinished) /
                        (c.getTime() - lastTime * 1.0));
            avgElements++;
        }
        lastFinished = c.getFinished();
        lastTime = c.getTime();
    }
    return speedSum / avgElements;
}
```

After the average processing speed is calculated in `getAverageSpeed()`, an ETA can be calculated with the help of the remaining unprocessed tasks.

This measurement is very naive, because it assumes that every task has the same complexity and uses the same time to process. Additionally one of the first `Job` implementations which gets executed is the `TSKReadImageJob`, which takes a long time. But after it has finished and other `Job` implementations get executed, this method provides a better estimation.

7.4. Conclusion

The implementation of `ProgressTracker` called `SimpleProgressTracker` holds different counters and fulfills the need as stated in NFR-5 . This way the analyst gets a feedback on the current progress and can compare this value with others gained through experience. The `ETACalculator` then provides an estimate of how much longer it will take by dividing the remaining tasks through the average speed over the last 10 measurements.

Because `TSKReadImageJob` is currently one of the first and longest running `Job` implementations which gets executed, the first estimations are inaccurate. After that `Job` has finished, and more measurements have been taken it gets more accurate.

8. Fault Tolerance

An analyst must be able to rely on his tools and their results. These tools must produce the same results if the same configuration applies, and they must behave predictable. This is also the main concern of non-functional requirement NFR-4 : The application has to be fault tolerant. Because the analysis in chapter 3 does not state requirements concerning availability and performance, the main focus lies on the results produced while used in the forensic process described in section 3.1.2.

This chapter starts with the analysis of things that can go wrong and which parts of Maloney are affected. Then a strategy has to be developed how the risks can be mitigated, followed with concrete implementation details. Lastly, the result is discussed.

Even tough the exclusion of files could be part of this chapter, it is separately discussed in chapter 13 File Exclusion.

8.1. Analysis

First it must be known how faults manifest themselves. Improving fault tolerance is a continuous process over the lifetime of an application, because not all causes might be detected at the beginning. Now, only faults concerning the defined requirements of section 3.3 can be addressed.

Mainly, there are four different aspects, which need to be addressed in this project:

Cancellation During Examination An analyst might want to cancel an examination, due to various reasons. It may also be possible that the examination is interrupted unintentionally, for example by a shutdown of the workstation. No progress should be lost, which was generated up until this moment.

Crash of a Job A `Job` can also crash during the examination. There are countless sources of faults, which might result in unexpected behavior. Therefore a single `Job` should not be able to influence the execution of others. It should also not be able to prevent other planned tasks to be executed.

Framework Issues The framework Maloney relies on different systems and resources, for example on a database or disk space. It should detect problems, which might prevent it from working properly.

Attacks There might be attacks against a forensic analysis tool, as implied by [Ken+06]. These aim to prevent the analysis or to make it more difficult. For example, there are compression bombs or malicious content, which reveals itself while examinating an infected system.

In contrast to the first three aspects, the last one concerning attacks is difficult to mitigate. There are countless forms or manifestations, and they can influence the analysis over side channels. It requires monitoring over the entire workstation only to detect manipulations at all. A way to mitigate effects introduced by attacks seem hard, if not impossible.

> Facing the detection and mitigation of attacks, it was decided not to implement a detection mechanism for attacks, to keep the focus on the examination of files, accepting the downside of possible malware infections or canceled examinations.

8.2. Approach

Because Maloney does not require further user intervention after it is started, the pattern *Minimize Human Intervention* [Han07, p. 57] is already in place. Therefore, this behavior should not change in case of failures. For example, there are no user requests asking for "are you sure?".

The overall approach is to *Maximize Human Participation* [Han07, p. 60]: The user has the overview and total control. Because the application can be configured in different ways and use additional parts, which may be developed by third parties, the faults have to be handled different in each case. To allow the user to make the right decisions, he requires information. This can be provided as a log file or an output in the user interface. log4j was already selected to handle the logging[EN16]. It can be extended with additional appenders, for example to notify errors in the user interface. By using the log levels (i.e. `trace, debug, info, warn, error, fatal`) and clearly formulated messages, the user should be able to detect problems quickly. The verbosity of the log can be selected either through a different configuration for log4j or the command line parameter `-v`.

Following with the ability to cancel and restart an examination, planned `Job` execution need to be tracked and stored permanently. This is required in combination with the configuration to recreate the application state. In addition the state of a planned tasks has to be tracked, either if it is running or if it has finished. A `Job` should not be executed twice, if it is not required. It still might produce duplicate results, for example if the execution has been canceled before the state transition was recorded. In the case of an intended cancellation, all running tasks should be completed before the application exits. If a set of unfinished tasks were found, the user can decide if he wants to process these tasks again or start anew. The framework also requires a new event, which is generated as the processing of a stored set begins, to replace the default start event. Otherwise, everything would be started anew. This approach correlates to the *Restart* [Han07, p. 151] pattern, with the tasks and states as *Checkpoints* [Han07, p. 166].

All data is persisted to the working directory or the databases. Therefore, these systems also need to be able to handle spontaneous operating system restarts. Both Elasticsearch and the local system drive are suited for this cases.

The propagation of errors because of crashes from a `Job` needs to be prevented. Therefore each `Job` represents a single *Unit of Mitigation* [Han07, p. 37]. The framework needs to detect an unexpected error (mainly unchecked exceptions) and handle them. If this happens, generated events need to be considered as faulty and therefore discarded. Accordingly, the file has to be marked that it could not be examined correctly. The analyst can detect invalid results based on that marking. Crashed `Jobs` should also be tracked and made available for a later restart, in case the fault was of transient nature. Because the results may still be important, it can be compared with the *Rollback* [Han07, p. 154] pattern.

A `Job` also needs the ability to tell the framework that it wants to cancel and be executed at a different time. This might be the case if not enough disk space is available. In this case the examination will not yield the correct results and needs to be prevented.

8.3. Implementation

The entire application state can be recreated with the configuration and the stored events. The configuration can be provided in a configuration file, which Maloney already supports. The events are produced during runtime and need to be stored persistently. The library MapDB provides collections as a lightweight solution to store all events. Every produced event is stored in the list, and removed after all jobs have finished related to this event. MapDB also features transaction support. This is required to keep the store fault tolerant [Kot17, Chapter Performance and Durability]. For the same task, a database or a message queue could also be used.

> Facing the decision for the persistent storage, it was decided to use MapDB and neglected to use a database or message queue, to keep the application simple, reduce latency and minimize dependencies to external systems, accepting the downside to manage more low level functions.

The class `EventStore` was introduced to keep track of job executions and corresponding events. This way, only a minimal interference of the framework was achieved and the usage of MapDB is transparent. During an examination thousands of events are produced, MapDB needs to be configured accordingly.

As shown in listing 8.1, transactions are enabled as well as memory mapping. All lists are configured to use the same database file of MapDB. Further mentions of a database file are related to this file. To reduce maintenance operations, for example introduced if the database file needs to be expanded, the increment size of new allocations was increased. This is also the size of a write ahead log used for transactions, which is not mentioned in the documentation.

Listing 8.1: Configuration of MapDB

```
db = DBMaker.fileDB(file)
    .fileChannelEnable()
    .transactionEnable()
    .allocateStartSize(128 * 1024 * 1024) // 128MB
    .allocateIncrement(64 * 1024 * 1024)  // 64MB
    .closeOnJvmShutdown()
    .make();
```

Every transaction needs to be committed, which merges the write ahead log into the database file. This process blocks the execution for some time. A commit after every operation results in many disk operations and slows down the entire process. Therefore, multiple additions and removals of events are combined into a single commit, which is executed in an interval. As an improvement the commit is only planned if changes are pending. MapDB also requires transactions in order to provide fault tolerance. Because events are only removed after all corresponding jobs have finished their execution and added new events, it is impossible that the entire chain of events gets lost. It may be possible that some new events are lost, but they can be reproduced by starting with the previous one. Committing multiple changes at once improves the performance while maintaining just a little gap of work which has to be redone.

The `FrameworkController` checks for existing events at startup and enqueues them automatically for execution. Only a message indicates that it is processing stored events instead of a complete run. In this case, the default startup event is replaced by a new one with the name "restart". Tasks could use this event as an indicator that they have to perform additional checks and cleanups. To prevent a restart and dismiss all stored events, the `FrameworkController` provides the method `clearEvents`. This can also be achieved with the command line switch `-clear-events`. No events will be discarded automatically, if the user does not explicitly say so.

Every `Job` is a separate *Unit of Mitigation*. The framework has no clue how to handle a fault introduced by a `Job`, but it has to prevent spreading of a fault. Therefore, every job is executed in its own try-catch block, which even handles unchecked exceptions like division by zero. If it still happens, the job execution will not be removed like other successfully finished ones. Also, the event remains stored on the disk for later processing. A `Job` has now also the ability to throw a `JobCancelledException`, which will behave like an unchecked exception. It can be used if the execution should be canceled prematurely. This option should be used rarely, because after a restart, the event will be executed again for every interested job.

Finally, a shutdown hook is added to the Java runtime by the `FrameworkController`.It provides a cleanup procedure to close open files, flush changes to disk and close other handles. Such a hook will either be executed if the JVM is interrupted, for example through a Ctrl-C, or if `System.exit` is called. The procedure also stops gracefully the processing of events and waits for a short amount of time. Already running tasks should have enough time to finish before the application exits.

8.4. Conclusion

With the aforementioned implementation, the fault tolerance could be improved drastically. Faults introduced through programming failures in an implementation of `Job` will not lead to a crash of the entire framework.

All generated events are recorded and can be executed at a later time, for example as soon as more storage is available. The user still has the ability to decide whether he wants to restart the process or to start anew. But there is also a downside to this permanent storage: speed. Every produced event has to be stored and retrieved from disk. This is an expensive operation. It slows down the entire processing of events. Still, the time all these storage operations take is a fraction of the entire examination. The possible time saving in case of faults is worth this investment, assuming an examination takes an entire day.

The persisted database file also needs to be handled by the framework and is not entirely human readable. It contains lots of information used by MapDB and can be viewed with a hex editor. This is a result of serializing the events as Javascript Object Notation (JSON). This file can not be edited manually, because it uses checksums to verify its integrity. Thus, it can be deleted and a new one will be created, if the database is not readable anymore. There are still some options to recover such a file, but the current configuration should be enough.

This implementation has a big impact on the framework and required many little changes, for example in the `JobProcessor` described in chapter 5. It is still possible to improve fault tolerance further, but the biggest issues are now mitigated. The current implementation is a trade-off between speed, complexity and time expense for development.

The ability to restart an examination is dependent on the case management features implemented in chapter 10. While testing an issue with MapDB was discovered related to memory mapping. Chapter 10 also contains the description and the solution of the issue.

9. Reporting

After the examination has been completed through the execution of various tasks, the found information needs to be presented in a readable format to the analyst. This step is described as the report in chapter 3.1.2 Forensic Process.

Located in this chapter are the analysis of the issue and motivation, the approach, and finally specifics on the implementation of the reporting mechanism in Maloney.

9.1. Analysis

After the examination and analysis is complete a report has to be generated as stated in functional requirement FR-16 Export Metadata.

When generating a report the focus should be directed to the data, that was identified as malicious or was not identified at all. Special attention should be paid to scripts, macros, libraries and anything executable. The reason for this is that those categories pose an increased security risk through a higher chance of containing malicious code.

The report can have various formats, such as a PDF-, HTML- or a CSV-File. The priority is on extracting the data from the application into a portable, independent, human readable format.

9.2. Approaches

9.2.1. Phases

A report needs to be generated at the very end of the whole process to encompass all gained information. Determining when the application is finished is complicated though. The order of tasks, which get executed, can vary every time and the end of a chain of tasks is difficult to determine when there are various conditions inside tasks. Therefore a mechanism has to be introduced, which ensures that all tasks have finished, before the report gets generated. The most reliable way to achieve this is by introducing phases to the application, which get executed sequentially. There are two different approaches for this:

Phases Inside the Application One option would be to introduce phases into the application and let the tasks decide in which phase they want to be executed. The report generating task can then specify that it should be executed in the last phase. This will ensure that it is executed after all examination tasks.

Phases Through Multiple Executions Another way would be to execute the tool multiple times but with different parameters. By choosing which tasks get executed, phases are introduced. This requires that the application does not randomly choose a temp folder where to extract and analyze the data. A proper case management is required.

The former introducing phases as a state inside the application, and lays the decision to the developer when a task should be executed. In the latter approach phases are represented through the various executions of the tool and the decision when to generate a report is put into the hands of the analyst.

Because Maloney is designed for power users, who most certainly desire to tweak an application to suit their specific needs, the decision when to generate a report should be put into the hands of the user:

Facing the decision for generating a report, it was decided to create phases through multiple executions of the application, and neglected using phases inside the application, to achieve guaranteed execution of the report at the end of all examination tasks, accepting the downside of user responsibility in starting the process.

9.2.2. Export format

There is also the issue of the format of the exported data, and how to the results are visualized. There is no requirement for the visualization and format of the collected data. However the purpose of visualizing data into bar or pie charts is to gain some information out of them. For this to work properly it is necessary that the result from the examination get categorized. This is handled in chapter 12 Categorization.

At the start of the report, an overview should be created. The overview should show all categories and many results lay within these categories. Additionally the overview should display meta information about the process itself. Meta information include the starting parameters, the time and date of the examination, the executed tasks, and the used datasets for comparison.

9.3. Implementation

There was no mechanism to retrieve all stored information in the `MetadataStore`. The first step was to implement an iterator to enable traversing over all stored data. To enable iterating, the data has to be retrieved first. It is common to work with disk images with size of 250 GB or more. To retrieve large amounts of data in Elasticsearch the scroll search is recommended [Ela17b].

The scroll search creates a snapshot of the current index and returns 100 data-sets. These consist of file meta data and their associated artifacts. They will get cached by the client together with a scroll-id. When sending another request with the received scroll-id, the next set of data gets sent in response. When reaching the end of the cached results, new results will be retrieved as seen in listing 9.1.

The only disadvantage is that a scroll uses more memory than other searches, and is thus required by Elasticsearch to have a timeout. After this timeout runs out, the results of the scroll will be lost. To accommodate even slower report jobs, the timeout has been set to 60 seconds.

Listing 9.1: Method to recieve next set of hits from Elasticsearch

```
private boolean continueScrolling() {
    this.scrollResp = client
        .prepareSearchScroll(this.scrollResp.getScrollId())
        .setScroll(new TimeValue(EXPIRY_TIME_IN_MS))
        .execute()
        .actionGet();
    if(this.scrollResp.getHits().getHits().length == 0){
        // Zero hits mark the end of the scroll
        return false;
    } else {
        extractResults();
        return true;
    }
}
```

The report itself gets generated through an implementation of the `Job`. This enables usage of the plug-in architecture for reports.

With all requirements fulfilled on the side of the framework, a simple `Job` called `ReportJob` was implemented which extracts all stored information in the `MetadataStore` and prints it into a CSV-file.

9.4. Conclusion

To accomplish functional requirement FR-16 Export Metadata, the necessary functionality framework-wise was implemented to enable generating of reports. A `Job` implementation to print all gathered data in form of meta data and artifacts into a CSV-file was added.

Because the data does not have meaningful categorization, no further implementations were added until such a thing exists in the application. When this is in place, HTML or PDF reports encompassing a more visual reporting.

10. Case Management

It cannot be expected that only one examination is executed, which then can be closed and all results become obsolete. In practice an analyst switches from one incident to another. Either the priority of an incident changes, newly gained insights require a revision or incidents need to be compared.

This chapter addresses the two requirements FR-8 and FR-11 . They are closely related because one comprises of managing cases and the other of handling working directories.

The analysis will state what belongs to a case and how it is related to an incident. Based on this information the approach describes how the requirements can be achieved. Thereafter follow some details about the implementation. Finally a conclusion is outlined.

10.1. Analysis

An incident can contain just one or multiple cases. A case therefore contains everything required for the examination by Maloney. The forensic process as described in section 3.1.2 can be applied multiple times to a single case. As a consequence Maloney should be applied multiple times to the same case. Results of different cases can then be used to form an overall conclusion of the incident.

A case consists of the following parts:

- Temporary files produced by a `Job`
- Examined or extracted files, for example of a disk image
- Generated reports
- Log files
- Last applied configuration
- Produced, but not processed events
- Examined metadata

The hash sets are not part of a single case. They are used globally over all cases, so that they can be managed in a single location. A hash identifies files exactly, therefore it carries a meaning which transcends over all or multiple cases. For example, if a file has malicious content, it should always be identified as such regardless of the originating disk image or system.

If a specific case needs to be selected, there has to be an identifier. Most of the time identifiers consist of alpha-numeric characters with some punctuation. *CVE-2014-0160* is an example used to identify the vulnerability also known as the Heartbleed Bug[1] or *JENKINS-43495* references an issue of installation dependencies of Jenkins[2]. A user will provide such an identifier as described by the FR-8 . Whenever the same identifier is provided, the operations are executed on the same case. An analyst will also have to find all files and metadata related to one case. Therefore, the identifier should be used to group the data for a single case, which also prevents mixing data with other cases.

In addition to specifying an identifier, users may want to specify where all data is stored on the disk. This enables selecting of a portable hard drive or a RAM drive as the storage medium for the case.

[1] `http://heartbleed.com/`
[2] `https://issues.jenkins-ci.org/browse/JENKINS-43495`

10.2. Approach

To group all data for a case, the identifier needs to be used both on the file system as well as for the metadata storage. This introduces some restrictions for such an identifier, because not all characters are valid or reserved for internal use. For example, asterisks or slashes have a special meaning on file systems, dots or underscores are on the other hand reserved by Elasticsearch, which is used to store metadata. Usable without issues are alpha-numberic characters and dashes. For easier handling a new identifier should be generated, if none is provided. The generated identifier must not exist on the system and must be unique.

A directory should be provided, which points to the location where all cases will be stored on disk. This directory will be further referenced as working directory. If none is provided, the temporary folder provided by the operating system is used. Every case is a subdirectory of this working directory.

10.3. Implementation

The implementation is straight forward. Every class with case related files requires to know where to store them. Therefore the `FrameworkController` provides the path and the case identifier to every interested class. These classes are:

- `EventStore`
- `LocalDataSource`
- `MetadataStore`

Additionally log4j is also reconfigured to output a file in the working directory of the current case. The configuration during runtime requires lots of factories. There is a manual on how to extend Log4j 2 configuration[3], but it is not up to date with the used version in this project. Because the configuration is done in the constructor of `FrameworkController`, it might be lost if the configuration is reloaded. This happens for example, if a configuration file was provided and changed after the programmatic configuration.

Further restrictions are introduced through Elasticsearch. While no documentation about naming of could be found, Elasticsearch will not accept every name as an index. It was determined by trial and error that lowercase characters as well as numbers and dashes are supported. In addition an index name cannot start with a dash, underscore or plus sign. The index names used by Maloney consist of the lowercase case identifier and a prefix "maloney-". This prefix also minimizes the risk of hitting a keyword of Elasticsearch, for example `_search`. Collisions are still possible, if multiple working directories are used with the same case identifier.

Depending on the file system other issues may arise: An example is NTFS which ignores the casing of characters. This may result in unexpected behavior.

> Facing the decision of a case identifier, it was decided to only use lowercase characters a-z, numbers and dashes, to support different file systems and Elasticsearch, accepting the downside of fewer possible combinations and additional validation.

The case identifier is validated using a simple regular expression. If the identifier is determined invalid, a new one will be automatically generated. A counter is used prefixed with the current date. This counter is incremented if the generated id already exists in the working directory. For example, "20170503-2" or "20170508-1" could be possible generated case identifiers.

After initializing the working directories and case names, the configuration is backed up in the case directory. This enables analysts to retrace their steps. Applying these configuration files in chronological order will recreate the current state of the examination.

[3]`https://logging.apache.org/log4j/2.x/manual/customconfig.html`

While testing the new implemented features, a problem of MapDB arose. The database file could not be opened with enabled memory mapping while using a working directory on a non-system drive. In this case, it was a shared folder of the virtual machine. Listing 10.1 shows a part of the corresponding stack trace. As it seems Java sometimes has issues of memory mapping files using `RandomAccessFile`. Luckily MapDB already has an alternate solution using `FileChannel`. By replacing the call to `fileMmapEnableIfSuppor` through `fileChannelEnable` in the MapDB configuration (listing 8.1), the issue could be mitigated.

Listing 10.1: Stack trace of MapDB with memory mapped files

```
Caused by: java.io.IOException: Invalid argument
at sun.nio.ch.FileChannelImpl.map0(Native Method)
at sun.nio.ch.FileChannelImpl.map(FileChannelImpl.java:926)
at org.mapdb.volume.MappedFileVol.<init>(MappedFileVol.java:104)
```

10.4. Conclusion

By introducing the case management as described in this chapter, a key feature of the framework was done. It not only allows to restart a previous failed examination, but also enables the further examination on top of an existing one. Also, the analyst is able to choose a different working directory. The examination could be performed on a faster disk for example. Hereby, the requirements FR-8 and FR-11 are met.

The libraries log4j and MapDB could be customized quite well, but required a lot of time to do so. The issue with MapDB could have been prevented by using `FileChannel` from the beginning. Because `fileMmapEnableIfSupported` was mentioned in the example configuration for high performance, no further research was done on memory mapping to save time.

11. Software Signature Comparison

Code signing is the process of digitally signing executables and scripts, to ensure their integrity and provide information about the author. It verifies that the software was not altered or corrupted since it was signed. As described by the requirement FR-18 , this information can be used to reduce the amount of files, which need to be reviewed by analysts.

This chapter will provide information about the motivation for such a feature and Authenticode as the implementation by Microsoft. It relies heavily on the solution implemented in “Forensik Triage Kit” [TV16], which was developed for Autopsy.

11.1. Analysis

The beginning of Authenticode dates back to mid 1996, when it was introduced to improve the security of Internet Explorer 3.0. Since then, it was improved over the time and broadly used by Microsoft itself, e.g. to sign drivers, and other software developers. The trust is built on root certificates, which are distributed with the operating system, similar to other digital signature processes using the X.509 standard. For more information, see *Authenticode* [Mic] and *Windows Authenticode Portable Executable Signature Format* [Mic08].

Authenticode is intended to work with Portable Executables (PEs), which include .exe, .dll, and .sys files. The signature may be embedded into the signed file or as a stand alone signature catalog (.cat files). Such a catalog contains signatures for multiple files, which accounts for the biggest share on Windows systems. Just a few PEs are signed directly. Only the parts of an PE containing code is signed: Some header fields are excluded from the hash algorithm used. [Mic08]

There are different tools for code signing for Microsoft Windows which are part of the Windows SDK (e.g. signtool or chktrust). There are open source alternatives such as osslsigncode or Jsign. Although most of these tools concentrate on the code signing process, but not the verification. The verification is part of Windows and can be accessed through its API. The Authenticode plugin for Autopsy uses a modified version of Jsign, which is also able to extract certificates or verify the signature. While it can handle code signing, it is not able to verify the chain of trust.

One major restriction exists: "The signature itself does not convey any information about the intent or quality of the software." [Mic08]

11.2. Approach

The two variants, embedded or stand alone, need to be handled differently. First of all, the signatures need to be extracted from those files. After that the hash of every PE needs to be calculated and compared against these extracted signatures. Signatures are tied to a specific case and should not be shared with other cases. Earlier calculated hashes from requirement FR-6 can not be used, because they also hash the signature information, which is excluded by the Authenticode algorithm.

The proposed default hashing algorithm for hashing is SHA-1. MD5 is only used for legacy implementations and should not be used to sign new content. [Mic08] A peek into different signature catalogs of driver packages revealed that other algorithms are used, for example SHA-256.

Attached to a PE is a certificate. To validate the signature of the PE with the certifacte the following steps need to be performed:

Validate Signatures Signed with a code signing certificate and co-signed by a Time Stamp Authority (TSA).

Ensure Certificate Usage The certificate used must allow the usage for code signing.

Validate Chain of Trust All certificates in the chain of trust for the signature are valid.

To validate the certificate chain, it should be possible to verify it without the root certificates provided by the examined system. Some certificates may have been replaced or added to cover up modifications.

Because the solution should be platform neutral, the Windows API can not be used. Therefore, the modified Jsign will be the replacement.

The results from these examinations will also be added as artifacts, so they can be analyzed later. The priority is to link files with their signature.

11.3. Implementation

The implementation consists of multiple tasks. These tasks are represented as implementations of *Job*. Following is a brief overview over their responsibilities:

- AuthenticodePEJob
  - Detect PEs
  - Calculate hash of relevant content
  - Extract certificates of PEs
  - Verify embedded certificates
- AuthenticodeCatalogJob
  - Detect signature catalogs
  - Extract signatures of signature catalogs
  - Extract certificates of signature catalogs
  - Verify embedded certificates
- AuthenticodeSignatureLookupJob
  - Match signatures with PEs

In `AuthenticodePEJob` the magic value (first four bytes) is used to determine if it is a PE or not. If the content is `0x4D5A` (ASCII `MZ`), it should be processed further. To calculate the hashes with SHA-1 and SHA-256, the class `net.jsign.PEFile` provides the algorithms required to only hash the parts of the file relevant to Authenticode. `net.jsign.PEVerifier` is used to verify the signature and to extract the certificates.

`AuthenticodeCatalogJob` uses an optimistic approach: The file extension is used to determine if it is a signature catalog. The class `net.jsign.CatalogFile` is used to extract the signatures and certificates. All signatures are stored for a later lookup, because it is possible that not all hashes of the PEs are calculated. The current implementation also uses Elasticsearch for this purpose, similar to the implementation in chapter 6 Identify Known Files. Signatures can be stored in the same index as the meta data by using a different type identifier. In this case "code-signature" is used.

By using Elasticsearch, the signatures can be persisted and compared in a separate execution using `AuthenticodeSignatureLookupJob`. This `Job` uses all files in the meta data store and compares their hashes with the extracted signatures of signature catalogs. Therefore this job has to be executed after a full run of both previously described `Job` implementations, `AuthenticodePEJob` and `AuthenticodeCatalogJob`.

A proper verification of the certificates is not implemented due to the complexity of this topic and the available time. A placeholder is available and a TODO placed where the verification needs to be implemented. Until this feature is fully implemented, every certificate obtains the state `UNKNOWN`. There are two variants which were discussed for this purpose: The Bouncy Castle library or the OpenSSL toolkit. Other libraries which verify the certificate chain could not be found.

Before validation of the extracted certificates with OpenSSL can take place, they need to be organized. First, all intermediate certificates have to be extracted in a separate file, beginning with the one closest to the root certificate. Then, the certificate used for the signatures can be validated using this separate file, as shown in listing 11.1.

Listing 11.1: Example of certificate validation using OpenSSL in the command line

```
openssl verify -CAfile intermediate.pem signerCert.pem
```

A validation using Bouncy Castle was attempted, but was discarded after difficulties with the identification of root certificates. The approach which relies on the detection of self signed certificates was not working as expected. The method used is shown in the listing 11.2, which always failed with a `SignatureException`. It may be possible that the root certificates were not included in the test files. Due to the milestone *Code Freeze*, no further time was invested.

Listing 11.2: Method to detect self signed certificates

```
public static boolean isSelfSigned(X509Certificate cert) throws
   CertificateException, NoSuchAlgorithmException,
   NoSuchProviderException {
    try {
        // Try to verify certificate signature with its own public key
        PublicKey key = cert.getPublicKey();
        cert.verify(key);
        return true;
    } catch (SignatureException sigEx) {
        // Invalid signature --> not self-signed
        return false;
    } catch (InvalidKeyException keyEx) {
        // Invalid key --> not self-signed
        return false;
    }
}
```

The used Jsign version is integrated as git submodule and referenced as local Maven dependency. It can be built and installed in the repository using the Gradle task `installJsign`.

11.4. Conclusion

The implementation introduced in this chapter fulfills the requirement FR-18 only partially. Comments in the source code will assist on implementing the missing validation of the certificates used for the signature.

The topic of software signature verification in its entirety is quite complex. The found documentation is not up to date and covers only a fraction of the entire scope of Authenticode. Therefore, more time and research is required to implement this feature correctly.

To allow the results already to be categorized, as explained in chapter 12 Categorization, two implementations of `Category` exist: `AuthenticodeCategoryKnownBad` and `AuthenticodeCategoryKnownGood`. These extend the default categories and therefore help with the identification of interesting files.

12. Categorization

When creating a report, as seen in chapter 9, simply printing out all found data is usually not helpful, because there is immense amounts of data. Structured results are needed for the analyst to gain insight into it.

This chapter describes why categorization is necessary, how it should be approached, and the way it is implemented in Maloney.

12.1. Analysis

When printing out the results of the examination, there is just plain data. To create a useful report as specified by functional requirement FR-16 Export Metadata, the analyst has to perform queries, correlate the data and categorize the results.

Doing this manually when the resulting dataset can be large as a quarter million entries, the analysis becomes very cumbersome. As described in functional requirement FR-4 Automation, it should be driven forward as much as possible.

Therefore, a forensic toolkit should be able to perform some basic categorization on its own, so that the work of the analyst gets easier.

12.2. Approach

12.2.1. Qualifiers

By default there are three different categories which can be applied to every examination: Known Good, Known Bad, and Unknown.

Known Good This data has been identified as non-harmful and can be safely ignored

Known Bad Data tagged with this category is harmful and has to be reported to the client

Unknown The examination could not determine the character of the data directly, the analyst needs to take a further look

Files can belong to a category by matching with its qualifiers. Because possible tasks and their results are extensible, so should the qualifiers which determine when a category applies. Therefore, it should be possible to enrich existing default categories with additional qualifiers so that they match the created examination results. These qualifiers should be definable for developers of plug-ins, because the knowledge of how to interpret examination results lies with them. A task which compares hashes with a database of well known files for example adds the result `KNOWN GOOD` to the data-set. It then adds this information as qualifier for the categorization so it can be identified by the framework, and later by the analyst, too.

12.2.2. Custom Categories

When the analyst can create custom categories, he can further reduce the amount of time he needs to determine the character of a file. So not only should it be possible to enrich the pre-existing categories with new qualifiers, but also to create custom categories which can then be used for easier identification.

With the addition of custom categories it should become possible for a file to have multiple matching categories as opposed to only one. For example an analyst can create a category for all files which were created before a specific date, but never accessed or changed again. Those files can then be identified faster with this category.

12.2.3. Reproducibility

When working with multiple categories, it is important to know which qualifiers were used for them. They need to be visible for the analyst, just like any other configuration. This is needed because the steps taken during the analysis are important for reproducibility.

12.3. Implementation

12.3.1. Categories and Qualifiers

To model categories and their qualifiers, categories have been separated into the `Category` interface as the root element and `RuleComponent` as the qualifiers (see figure 12.1). This way qualifiers, i.e. `RuleComponent`, can be added seamlessly to the `Category`. With the two different composites `AndRuleComposite` and `OrRuleComposite` the relation of qualifiers can be selected:

AndRuleComposite Every rule of the composite has to be true for a match

OrRuleComposite Any rule of the composite has to be true for a match

It is important to note that an `RuleComposite` or any of its inheritors with no added `RuleComponents` will always be a negative match. This is because there are no `RuleComponent` which were successfully matched, and therefore the qualifier is not fulfilled.

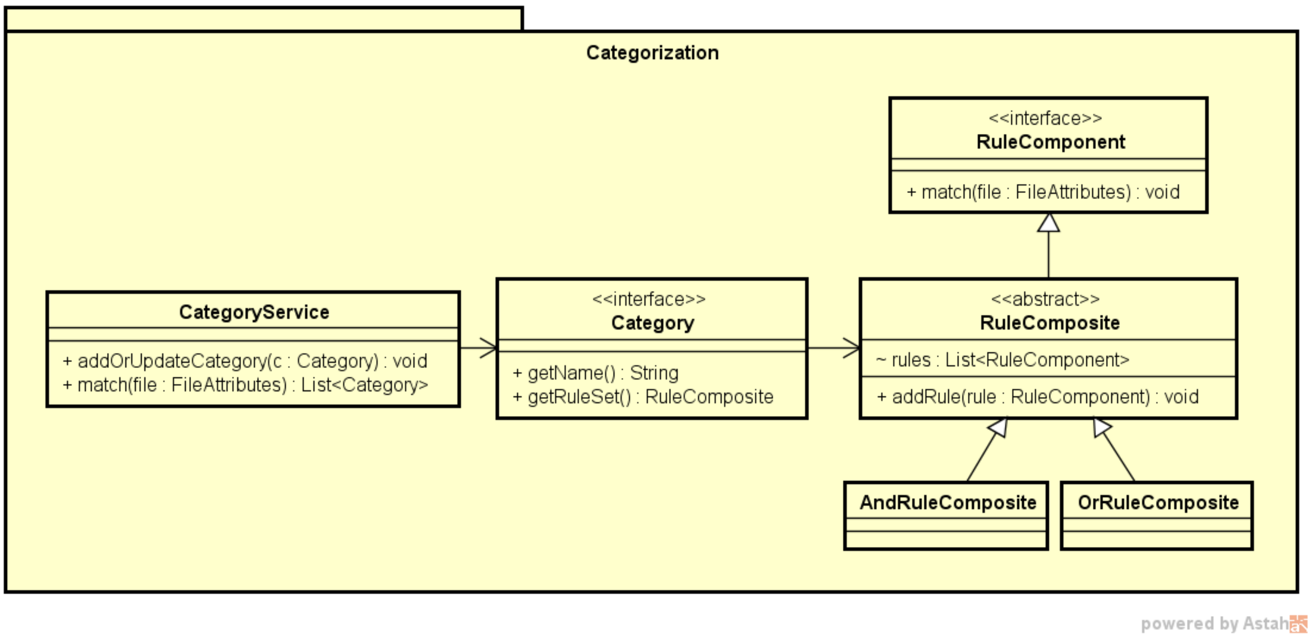

Categorization
<<interface>>
RuleComponent
+ match(file : FileAttributes) : void
CategoryService
+ addOrUpdateCategory(c : Category) : void
+ match(file : FileAttributes) : List<Category>
<<interface>>
Category
+ getName() : String
+ getRuleSet() : RuleComposite
<<abstract>>
RuleComposite
~ rules : List<RuleComponent>
+ addRule(rule : RuleComponent) : void
AndRuleComposite
OrRuleComposite
powered by Astah

Figure 12.1.: Simplified UML-Diagram for the composite structure of `RuleComponent` and inheritors in context of `Category`

The `CategoryService` has been added to the `Context` class. This way it can be accessed by any `Job` which requires it, especially all `Job` implementations which create some form of output such as reports. The existing `ReportJob` has been modified to add the found categories to its output.

12.3.2. Adding Qualifiers to Categories

The qualifiers for categories can be added to the service by calling `addOrUpdateCategory`. If the name of supplied `category` matches any of the existing categories, its qualifiers will be added to the existing one.

Furthermore, extension of qualifiers and categories has been implemented by loading them via the `CustomClassLoader`. All implementations of `Category` in the plug-in folder get loaded and added to the service. This works the same way as described in chapter 4 Plug-in Architecture. This also enables the reproducibility and insight for the analyst, as all qualifiers of a category can be seen in these separate classes.

12.4. Conclusion

With the addition of categorization, the analyst can determine which data may need further analysis or give a thorough report of the state of the analyzed system. With the `CategoryService`, all examination tasks can add their definition of categories such as *Known Good*, *Known Bad* or customized categories to the framework. Those definitions get loaded like `Job` implementations by the `CustomClassLoader` as described in chapter 4 Plug-in Architecture.

13. File Exclusion

According to functional requirement FR-12 Exclude Filess this feature is necessary. Not only because of the possibilities of attacks on the analysts workstation, but because not all files are necessarily interesting. If the analyst decides that for example a certain file ending or files created before a specific date need not be investigated, he should be able to do so.

In this chapter a analysis for file exclusion, as well as the possible approaches and its implementation in Maloney are described.

13.1. Analysis

When it comes to analyzing potentially infected systems, there is a small chance that the system in question is containing malicious code that can spread to the analyzing workstation. Other potential threats such as compression bombs exists too.

Beside the threat of an attack there also might be files which are of no concern for the analyst and therefore can be safely ignored. For example it might be helpful to ignore .txt files, or files which were created before the date of an incident. Reducing the amount of files which need to be analyzed by the analyst also reduces the total time required for the whole process.

13.2. Approach

13.2.1. Position of Filtering

There are multiple options as to when the filtering out unwanted files should happen: Either all the time on any event, or only when a new file is added.

On Any event Anytime a new event is created, check for the criteria. If any of them match, discard the event. This uses the most processing time but can intercept events independently of their source.

On File Add When a new file is added for examination, check for the criteria. If any of them match, do not create a new event and thereby stop the events propagation. The issue with this method is that when a file gets added to the system trough a different event than specified in this filter, it cannot be intercepted.

> Facing the decision for the position of the criteria check, it was decided to check on file add, and neglected checking on any event, to achieve better time-behavior, accepting the downside of missing events or files with a different sources then those specified.

13.2.2. Criteria

The criteria for filtering out files can be broken down into rules for every description or previous examination result. Because of this criteria the design described in chapter 12 Categorization can be applied to this problem too.

These criterias should be available for usage on a single case or over multiple cases. They should get loaded dynamically on every execution of the application until it is specified otherwise.

13.3. Implementation

With the decision to filter events on file add a new `Job` implementation has been added. It is called `ExclusionJob`. It uses the categorization feature described in 12 Categorization.

A new instance of `Category` is created through the factory in chapter 14 Simple Queries. By using this factory regular expressions can be passed to the new `Job` as job configuration. The job then checks those criteria on every `newFile` event. If it does not match, it creates an `addedFile` event.

A refactoring to `SimpleQuery` was necessary to accommodate `OrRuleComposites`, so that multiple rules can be added.

In addition, all examination jobs now listen to the event `addedFile` instead of `newFile`. In other words it intercepts events for new files and maps them to addedFile event as long as the criteria for exclusion are not a match.

13.4. Conclusion

With the new `ExclusionJob` any unwanted file can be excluded from further examination. It is possible to apply any filter through passing regular expressions as the job configuration. The job uses functionality from 12 Categorization and 14 Simple Queries. This fulfills functional requirement FR-12 Exclude Files.

14. Simple Queries

The user experience of Maloney should be as independent as possible of the used technologies. Therefore the CLI should offer a way to query all stored meta data. The requirement FR-17 Simple Queries requests the ability to use a simple string, which then can be interpreted and translated for the specific technology.

This chapter outlines the need for this feature and analyze the scope. Following is the approach on how to achieve the goal and details about the implementation. The implementation uses functionality introduced in chapter 12 Categorization and 9 Reporting. Finally, the result is discussed.

14.1. Analysis

The goal of an analyst is to dig into the data and find the interesting parts. There is data generated for every examined object, including file meta data, artifacts and other reports. This leads to a massive overflow of information for the analyst. A method to search in this data is necessary to increase the efficiency for the user.

Maloney supports different technologies to store the meta data and artifacts. Each technology has their limits and special requirements. For example, Elasticsearch uses its own domain specific language, which requires a deep understanding of how the data is stored. Therefore the query used by the search should be independent of the technology and offer a format known to an analyst. It should also remove the need to know, how the data is stored and the special treatment which is required to handle the data. Maloney should act as an intermediate layer and abstract the access for this different technologies. Consequently is a query tool required which handles the conversion.

A query should also be customizable and easily editable by an analyst. Predefined queries will not provide the same amount of flexibility. Changing the query needs to be quick, so that the resulting data set can be reduced further. An analyst does not want to spend time with a complicated interface. The queries should also be changeable and reusable.

An output is expected either on the user interface itself or in a file. The latter is required to use different tools, editors or search functions to get a quick overview over the data. It is similar to the feature discussed in chapter 9 Reporting.

To improve the overview it would help if the displayed properties can be selected.

14.2. Approach

For simplicity the query is built up on simple key-value pairs. This format is simple to write and does not require a deep understanding of the physical storage structure. There are multiple approaches how the query can be parsed into a format, that can be interpreted efficiently. The research revealed that either regular expressions or Antlr would be well suited:

Regular Expression They can be used to detect simple patterns of a search query and divide the identifier from the actual value to search.

Antlr Antlr is a tool to generate powerful parsers for many different tasks. For example, it can also be used to define domain specific languages, compilers or to improve the usability of search engines. Antlr uses regular expressions to define a grammar, which then can be used to generate parsers, visitors, and so on. [ANTLR] A generated parser could be extended to write a category or even to translate the query directly for the different back ends.

An example for the power of Antlr is it allows to build a compiler for Java, using an entire book as the input. With this extensibility, the learning curve is steep and requires a deep understanding of the tool itself. An experiment revealed, that it could be integrated into the build process using plug-ins for Gradle. The classes in the source code generated by Antlr can be extended to suit the current task. Extending the grammar allows to extend the abilities of queries. For example, natural language processing would be possible instead of a strict format.

> Facing the decision of the method to parse the query, it was decided to use regular expressions and neglected to use Antlr, to keep the parsing simple and the build process slim, accepting the limited depth of the queries and increasing complexity with later extension.

Only the data provided by `MetadataStore` will be available for queries. The `DataSource` can be processed with different file system tools. It reduces the complexity of the query mechanism and also allows an analyst to use other tools, like `grep`, `tr` or `awk`.

The CLI provides different streams for in- and outputs. These streams can be redirected into a file or other tools using pipes (depending on the operating system or shell). Therefore, the standard output stream should be used to display the data to the user. If the query is a string, everything can be executed inside the CLI or from a script.

Implementing the search inside Maloney instead of the implementation of the `MetadataStore` reduces the need to re-implement the feature for every new technology. It allows the reuse of existing components and therefore reduces the complexity.

14.3. Implementation

The feature is implemented in the class `SimpleQuery`, which is directly invoked by the `FrameworkController`, if required. Therefore, additional command line switches were introduced: `-q` or `-query` to supply a query and `-filter` to supply an additional filter. It is neither implemented nor acts like a `Job`. It does not require events or generate artifacts. When executing a query, it should not execute any other tasks. This includes scheduling unfinished events or running any other `Job` implementations.

A regular expression is used to parse the query. The pattern is shown in listing 14.1. Both groups can be referenced by their names `property` and `expression`. It is a simple list of key-value pairs, where the `expression` is enclosed by double quotes. No specific separator is defined for the separation of pairs. These pairs are then used to dynamically generate a `Category`. `Category` is further described in chapter 12 Categorization. Also the filter uses a similar pattern, but just for the `property`.

Listing 14.1: Used regular expression pattern to split the query into relevant parts

```
((?<property>[a-zA-Z]+)="(?<expression>[^"]+)")+
```

A new `RuleComponent` is added fto the custom `Category` or every property. The `RuleComponent` matches against the string of the selected property using the `expression` from the pair. Therefore, it also supports regular expressions in `expression`. Attention has to be paid that no double quotes are used in `expression`, otherwise it will be misinterpreted. Examples for valid queries are shown in listings 14.2 and 14.3. Every `RuleComponent` is aggregated using an `AndRuleComposite`, so every one must match.

When the query cannot be interpreted with the pattern, the search uses the query string and checks if it is contained in the properties "fileId" or "fileName" as a fall back.

Listing 14.2: Example query string matching all files in a directory called "windows" with the file name beginning with "reg"

```
fileName="reg.?" filePath="(?i).*windows.*"
```

Listing 14.3: Example query string files with a given hash

```
artifactType="(?i)MD5" artifactValue="86fb269d190d2c85f6e0468ceca42a20"
```

There are multiple values supported as properties, which map to the logical format of `FileAttribute`. In table 14.1 are the possible values. These are case insensitive. The filter can be controlled by providing a list of these values. Their order will also be used to format the output. If none is provided, every property is printed. The property `artifacts` contains also `artifactType`, `artifactOriginator` and `artifactValue`. They cannot be selected independently in a filter.

Property	Query	Filter
fileId	✓	✓
fileName	✓	✓
filePath	✓	✓
dateAccessed	✓	✓
dateChanged	✓	✓
dateCreated	✓	✓
artifacts		✓
artifactType	✓	
artifactOriginator	✓	
artifactValue	✓	

Table 14.1.: Recognized values for property in a query or filter

The value part of the pair can use any regular expression pattern. The online documentation of Java about Patterns[1] is a useful resource when writing one. If a date should be queried, it will be formatted using `DateTimeFormatter.ISO_DATE_TIME` as default. An example of a date would be "2011-12-03T10:15:30+01:00[Europe/Paris]".

The implementation shares some similarities with the `ReportJob` introduced in chapter 9 Reporting. It gets every `FileAttribute` stored in the `MetadataStore`. Using a custom implementation of `Category`, every `FileAttribute` is matched before it will be printed to the standard output.

A tabulator is used as default delimiter of the values. This is relevant if the output should be processed in a different application. The order of the properties in the filter also determines the order of the output.

14.4. Conclusion

The functionality provided by the class `SimpleQuery` fulfills the requirement FR-17 Simple Queries. It facilitates to query all the stored information to find the desired information.

With the implementation of `Category` in chapter 12, some code could be reused and no new matching algorithm had to be developed. This reduced the time required for the implementation. Some technologies usable as `MetadataStore` already provides querying capabilities, which are yet unused. Translating the queries for each implemented `MetadataStore` could increase the processing speed and reduce the amount of transferred data. During the development and testing phase no performance issue were observed.

Dates and times are another difficulty to handle. It was not examined which timezone offset they include or if they are always extracted in the Coordinated Universal Time (UCT). Therefore, it could be difficult to pinpoint an issue to the correct hour they occurred. It requires further time to streamline the timezone compensation over the whole application.

The parser is also used by the feature implemented in 13 File Exclusion and therefore supports the same query syntax.

[1] https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

15. Summary

In this chapter, the achieved work is examined again: The requirements are reviewed, still open and unhandled issues explained and possible future developments formulated in an outlook.

15.1. Requirements

In chapter 3 Analysis the requirements for the project were formulated. After the project is finished, these requirements need to be revisited. It needs to be evaluated which ones were fulfilled, and which ones were not so that future developers have a reference where to start.

15.1.1. Fulfilled Requirements

At the start of the project some requirements were already fulfilled. But many new features were added which extended the capability of Maloney. The following requirements are considered fulfilled:

FR-1 Examine Disk Images This requirement was already fulfilled at the start of this project.

FR-3 Collect Metadata This requirement was already fulfilled at the start of this project.

FR-4 Automate Process This requirement was already fulfilled at the start of this project.

FR-5 View Metadata This requirement was already fulfilled at the start of this project. An additional technologically agnostic version was added through 14 Simple Queries.

FR-6 Calculate File Hash This requirement was already fulfilled at the start of this project.

FR-7 Identify Known Files Comparison mechanisms have been implemented as in chapter 6 Identify Known Files and chapter 11 Software Signature Comparison.

FR-8 Manage Cases A case management with case identifier, working directories and separation of global and case-based data has been introduced in chapter 10 Case Management.

FR-11 Configure Working Directory This requirement was fulfilled and described in chapter 10 Case Management.

FR-12 Exclude Files The functionality to exclude dangerous or uninteresting files has been added and described in chapter 13 File Exclusion.

FR-16 Export Metadata The functionality to generate a report has been added and described in chapter 9 Reporting.

FR-17 Simple Queries In addition to the ability to querry data over Kibana, a technology agnostic querrying functionality has been added and described in chapter 14 Simple Queries.

The non-functional requirements are an invariant to the whole application and were regarded as this during the whole development.

15.1.2. Unfulfilled Requirements

It was clear from the start that not all formulated requirements could be fulfilled by the end of the project. The following requirements were not implemented due to time constraints and the subsequent re-prioritization of requirements.

FR-9 Compare to Golden Image Golden Image comparisons are not as common as hash or signature comparisons.

FR-10 Examinate Windows Metadata

FR-13 View Directory Structure The file structure can also be seen directly on the image and is not necessarily needed in Maloney.

FR-14 Search for Strings in File Content

FR-15 Identify File Types

15.2. Open Issues

Although most problems could be solved, there were some sporadically appearing errors or errors with no traceable source. These have to be kept in mind for future development, as they are key in making the application even more stable.

15.2.1. Invalid FS_Info Object

During development, some sporadic exceptions with the message "Invalid FS_INFO object" have been observed, which resulted in empty files and therefore zero-byte file hashes. This behavior was misinterpreted as problems concerning *near real-time* as described in chapter 6 Identify Known Files. This misinterpretation was mostly because the exceptions were only thrown when no debugger was attached. When a debugger was attached, a match was found. The root cause is not originating in this implementation, but has to be resolved in the future.

This error originates in TSK and could not be successfully reproduced, but appeared sporadically. Usually a complete reboot of testing machine fixed the issue.

15.2.2. Memory Access Violation

Occasionally during an examination the JVM crashed with a memory access violation. This only sporadically appeared and is linked to TSK and its required libraries.

Again, a restart of the application and subsequently the machine itself solves the issue.

15.2.3. Out Of Memory Error

During the analysis of a large image, it is possible that Elasticsearch in conjunction with Maloney provokes an out of memory error. After the JVM is terminated, the application can be restarted and the examination continued.

It is possible to prevent this error entirely by assigning more memory to the JVM. An experimental run with the JVM parameters `-Xms 12g -Xmx 12g`, which assign 12 GB of memory to the application, lead to a successful execution.

The memory footprint of the application may be reduced by refactoring. The following issues have been identified:

Double Curly Brace Initialization It is possible to create memory leaks when using double curly brace initialization in conjecture with the `CustomClassLoader`. "They contain a reference to their enclosing instance, and that is really a killer." [Ede14] By eliminating them, this possibility can be removed.

event Pull Instead of Push The `MultiThreadedJobProcessor` could be refactored so that it pulls events from the framework. Queued events could then be offloaded from the memory onto the disk. The mechanism to do so was already implemented in chapter 8 Fault Tolerance with MapDB.

15.2.4. Extracted Elements Count in Working Directory

During the examination all found files on the image get extracted to the specified working directory. A discrepancy can be found when checking the amount of files in the meta data store, i.e. Elasticsearch: There is one more file then there is on the file system.

This is due to the registering of the image file itself in the application, but it not being copied to the working directory.

15.3. Outlook

Although the project is finished, there are still multiple possibilities how Maloney can be extended. For starters, there is the list of unfulfilled requirements in subsection 15.1.2 Unfulfilled Requirements. Apart from these, there are many other possible extensions the application could profit from. This includes the addition of more examination methods or the extension of existing ones. Some possible improvements can be found in the conclusions of their respective chapters.

Extension of Signature Comparison Add functionality to validate the full certificate chain (see chapter 11 Software Signature Comparison).

Configuratble Framework Components Some framework components are not dynamically configurable; E.g. the name of the Elasticsearch instance is statically defined in the implementation of `MetadataStore` and cannot be changed dynamically.

Post Case Clean Up When a case is finished, the data needs to be deleted on both the local storage and the meta data storage; This could be automated.

Command Line Tool Execution Template An option to drastically increase the possibilities for examination would be to include some form of generic command line examination tool template. E.g. YARA can detect patterns in generated log files after an execution inside Cuckoo Sandbox sandbox. It could also detect patterns in log files from the execution of other commands.

Index Plain Text Content All plain text content could be made searchable, so that all log files or other documents can be searched on the image. This gives the analyst the ability to correlate data better.

Distributed Computing The possibility to share the computational effort between multiple machines. This is discussed in chapter 5 Parallelism as well.

Remote Analysis The analysis of a live system could be integrated. The required extraction mechanism for remote files could be a `Job` implementation.

Windows OS Support Currently the only operating system Maloney can be run on is Linux. The reason is that the used TSK version cannot be built on Windows. A future version of TSK may fix this and Windows support can be re-added.

16. Attachments

A. User Documentation

This chapter contains all the necessary steps to create, install and examine a disk image with Maloney. Currently, only Linux is supported as the operating system to develop and to perform an examination. For information about the windows support, see “Malware Hunting” [EN16].

A.1. Setting up the development environment

The setup includes all steps required to prepare the development environment for Maloney. After executing all steps, Maloney with all dependencies can be built.

Gradle is used as the build tool, which provides a lot of functionality. This guide only covers the most important use cases and therefore not all possible ways on how to use Gradle are described. For more information and some advanced details see *Gradle User Guide* [DM15].

This guide is tested with *Kali Linux 2017.1 Light 64 bit*, but other distributions may also work. The commands need to be changed accordingly to the used distribution.

A.1.1. Requirements

- Installed Linux operating system
- Internet access
- Optional: IDE for Java, for example IntelliJ IDEA Community or eclipse

A.1.2. Install Prerequisites

All commands are entered into a terminal with normal user privileges. The command *sudo* is used to elevate the permissions, where required.

1. Make sure the machine is up to date. Enter the following commands:

   ```
   $ sudo apt update && sudo apt upgrade
   ```

2. There are some dependencies to external software packages and libraries: automake, autoconf, libtool, gcc, Java Development Kit (JDK), git, ant, maven, libewf, afflib and zlib. This dependencies can be installed through the package management of the distribution[1]. Execute the following commands to install them:

   ```
   $ sudo apt install automake autoconf libtool build-essential openjdk
       -8-jdk git ant maven libewf-dev libafflib-dev zlib1g-dev
   ```

3. Optional: Some Unit Tests will require an Elasticsearch instance. This integration tests are excluded from the gradle test task, but sometimes useful while debugging. Install Elasticsearch and Kibana as described in A.3. Alternatively, both can be executed directly from the command line without installing. For more information, see section "Installation Steps" on Download Elasticsearch[2].

[1]On some distributions, alternative package-source *universe* needs to be activated

[2]https://www.elastic.co/downloads/elasticsearch

4. Clone into the git repository, currently hosted on Github[3]

   ```
   $ mkdir ~/projects && cd ~/projects
   $ git clone https://github.com/rehrbar/maloney
   ```

5. Init TSK submodule and build it. This step needs to be repeated, if a hard reset of the working directory is performed or the reference was updated. More information about building is described in *~/projects/maloney/sleuthkit/INSTALL.txt* Enter the following commands:

   ```
   $ cd ~/projects/maloney/
   $ git submodule init sleuthkit
   $ git submodule update sleuthkit
   $ cd sleuthkit
   $ ./bootstrap
   $ ./configure
   $ make
   ```

6. Init the Jsign submodule and install it into the local Maven repository. This step needs to be repeated, if a hard reset of the working directory is performed or the reference was updated. Enter the following commands:

   ```
   $ cd ~/projects/maloney/
   $ git submodule init jsign
   $ git submodule update jsign
   $ ./gradlew installJsign
   ```

If the build of TSK and Jsign succeed, everything is configured correctly and all dependencies are available.

The unit tests of Jsign require a broadband Internet connection to work properly. Using the cellular network might result in failing tests.

A.1.3. Build Maloney

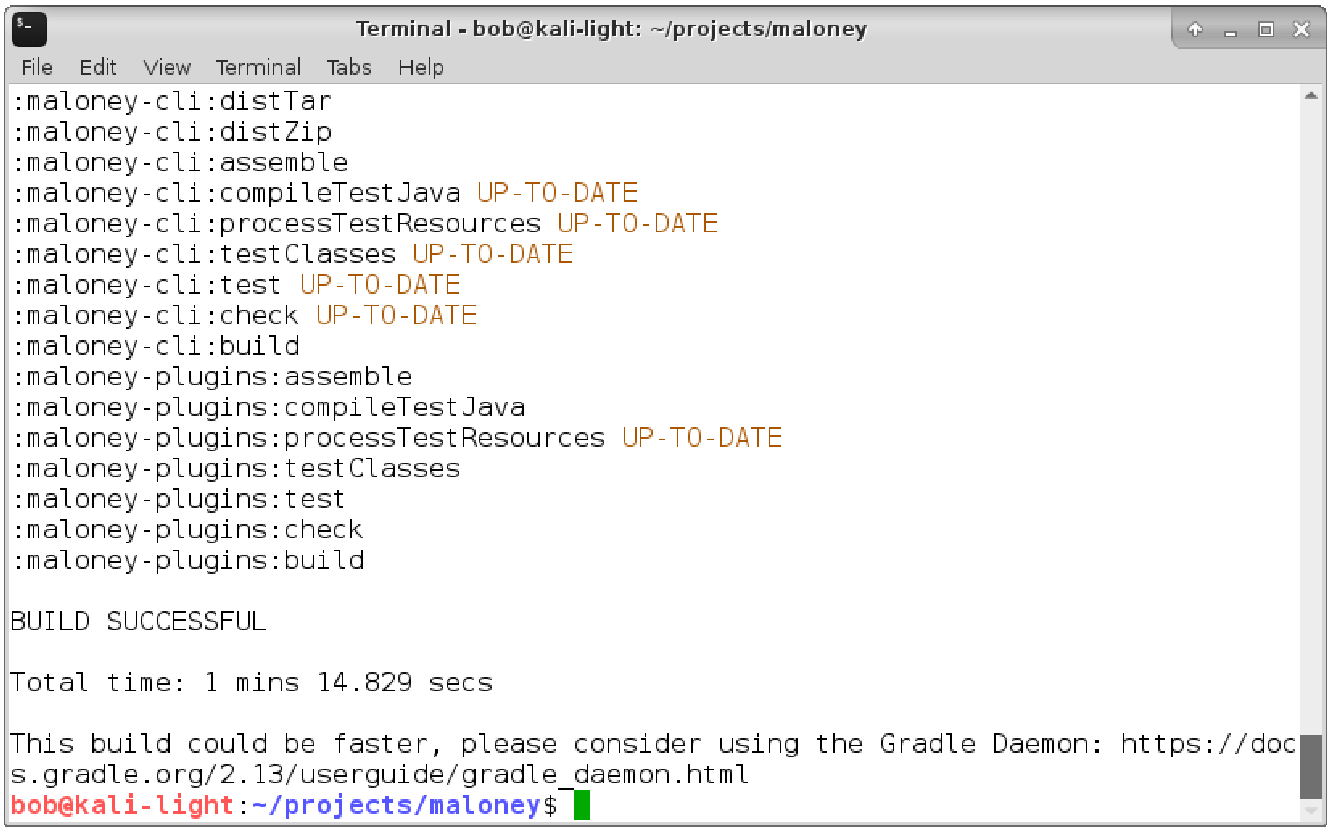
The project itself is structured with three Gradle sub-projects: *maloney*, *maloney-plugins* and *maloney-cli*. The latter is the command line interface, which can be distributed as an application. The projects *maloney-cli* and *maloney-plugins* are referencing to *maloney*, which will automatically be built if required. Similarly, *maloney* has a reference to TSK bindings. Additionally, *maloney-cli* refrences *maloney-plugins* as runtime dependency, which allows debugging of the plug-ins. As a side effect, the library of *maloney-plugins* is included automatically in the libraries directory of the distribution package.

The root project contains the Gradle wrapper, which will download the required gradle version automatically. It also includes some shared configuration for all sub-projects.

To perform a build of *maloney-cli* and its dependencies, run the following commands:

```
$ cd ~/projects/maloney/maloney-cli
$ ./gradlew build
```

[3] https://github.com/rehrbar/maloney


Terminal - bob@kali-light: ~/projects/maloney
File Edit View Terminal Tabs Help

```
:maloney-cli:distTar
:maloney-cli:distZip
:maloney-cli:assemble
:maloney-cli:compileTestJava UP-TO-DATE
:maloney-cli:processTestResources UP-TO-DATE
:maloney-cli:testClasses UP-TO-DATE
:maloney-cli:test UP-TO-DATE
:maloney-cli:check UP-TO-DATE
:maloney-cli:build
:maloney-plugins:assemble
:maloney-plugins:compileTestJava
:maloney-plugins:processTestResources UP-TO-DATE
:maloney-plugins:testClasses
:maloney-plugins:test
:maloney-plugins:check
:maloney-plugins:build

BUILD SUCCESSFUL

Total time: 1 mins 14.829 secs

This build could be faster, please consider using the Gradle Daemon: https://doc
s.gradle.org/2.13/userguide/gradle_daemon.html
bob@kali-light:~/projects/maloney$
```

Figure A.1.: Output of a successful build

The output is located in the folder *build/libs*, which contains mainly the Java archives. This files cannot be executed without the dependencies in the Java classpath. The task *installDist* will create the required directory structure and startup scripts, including all dependencies. The output of *installDist* is located in *build/install*.

A.1.4. Run Unit Tests

Some integration tests are excluded to run in the default configuration. Mainly tests which have external dependencies, such as disk images or to Elasticsearch.

To run all configured tests, enter the following commands:

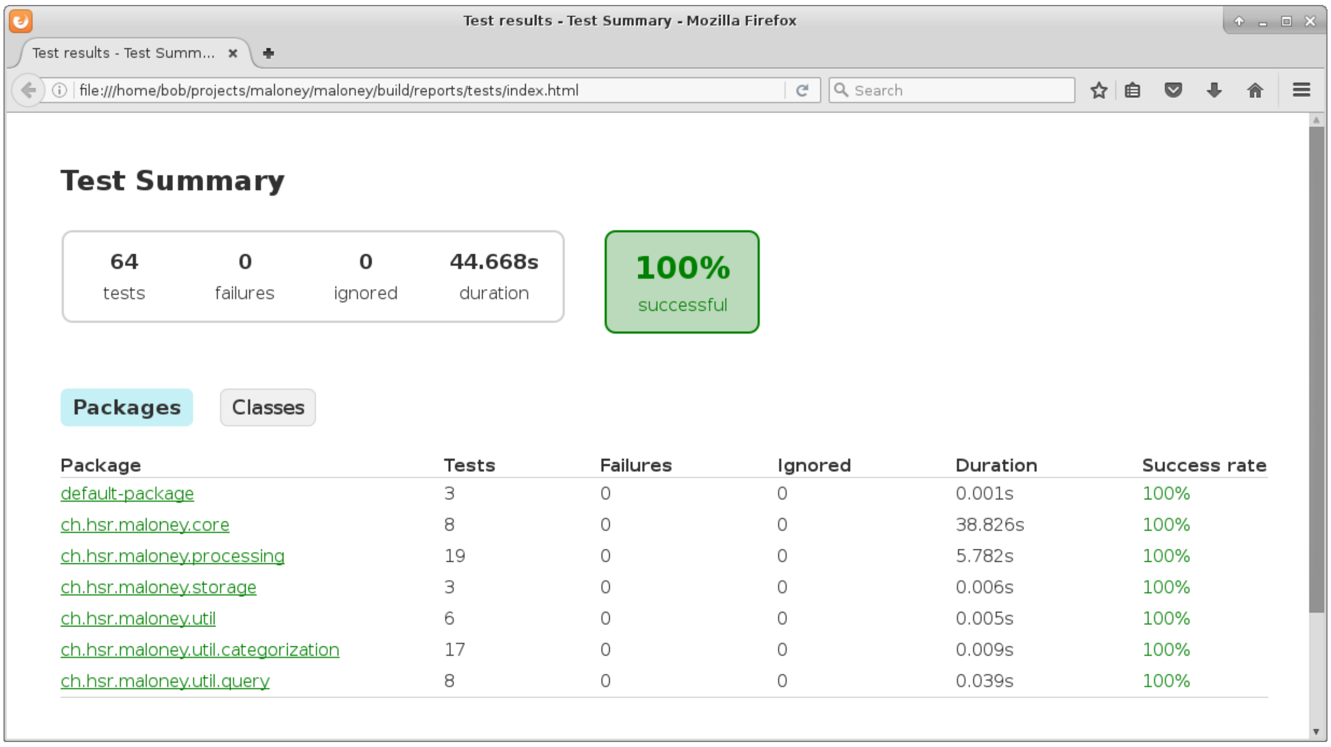
```
$ cd ~/projects/maloney
$ ./gradlew test
```

If not all tests are required, a single test class can be executed. Only test classes which are not excluded in *build.gradle* are available. See the following command as an example:

```
$ ./gradlew test -Dtest.single=ch/hsr/maloney/storage/LocalDataSourceTest
   *
```

An IDE can also run single tests. In this case, the integrated test runner is used. Sometimes it is necessary to configure JVM-Options and environment variables before running a test. This is dependent on the used IDE and will not be covered in this documentation.

All detailed test results are located in *build/reports/tests* of every sub-project containing tests. The test results can be viewed in the browser by opening the *index.html* file, shown in figure A.2. This report allows to drill down into every executed test.


Test results - Test Summary - Mozilla Firefox
Test results - Test Summ...
file:///home/bob/projects/maloney/maloney/build/reports/tests/index.html
Search

Test Summary

64	0	0	44.668s	100%
tests	failures	ignored	duration	successful

Packages Classes

Package	Tests	Failures	Ignored	Duration	Success rate
default-package	3	0	0	0.001s	100%
ch.hsr.maloney.core	8	0	0	38.826s	100%
ch.hsr.maloney.processing	19	0	0	5.782s	100%
ch.hsr.maloney.storage	3	0	0	0.006s	100%
ch.hsr.maloney.util	6	0	0	0.005s	100%
ch.hsr.maloney.util.categorization	17	0	0	0.009s	100%
ch.hsr.maloney.util.query	8	0	0	0.039s	100%

Figure A.2.: Generated test summary in firefox

A.1.5. Integrated Development Environment

A modern Java IDE can import the Gradle projects. In some cases, Gradle can be used to generate the necessary project files itself. How to use and import the project files into an IDE are not part of this guide.

By using Gradle, any text editor could be used. There is no need to use a specific IDE.

A.2. Creating a distribution package

This guide describes the necessary steps to create a distribution package, which can be installed on other machines.

A.2.1. Requirements

- Already set up development environment as described in section A.1 Setting up the development environment
- Internet access
- Gradle needs to build the project successfully

A.2.2. Building a Distribution Package

The project *maloney-cli* is configured with the *distribution* plug-in of Gradle. It provides some useful tasks to create a single archive, which contains all dependencies and start scripts.

Executing the *build* task will also build the distribution package. To build the distribution package manually, enter the following commands:

```
$ cd ~/projects/maloney/maloney-cli
$ ./gradlew assembleDist
```

All generated archives (distribution packages) are located in *build/distributions*. Both archives, Zip and Tar, contain the same files. The only difference is that the Tar archive can contain file permissions, e.g. the run permission, and is therefore suited for Linux. The startup scripts can be run on Linux and Windows (*bat file extension*). For additional information about the *distribution plugin*, see chapter 35. The Distribution Plugin in *Gradle User Guide* [DM15].

A.2.3. Add Additional Content to the Distribution

The distribution plugin also has the ability to inlcude additional content: "Static files to be added to the distribution can be simply added to src/dist. More advanced customization can be done by configuring the CopySpec exposed by the main distribution." [DM15, chapter 50. The Application Plugin] Files in *maloney-cli/src/dist* will be placed in the root of the distribution package. Folder structures are also supported.

The project *maloney-cli* is configured to include a specific dependency to a TSK library, which is not automatically recognized. Therefore, this library is added with the following extension to *build.gradle*:

```
distributions {
    main {
        contents {
            // adding missing library which is required by libtsk_jni
            from("../sleuthkit/tsk/.libs/") {
                include "libtsk.so.13"
                into "lib"
            }
        }
    }
}
```

A.3. Installing Maloney

This guide describes how to install the distribution package, as described in section A.2, on a new machine.

This guide is tested with *Kali Linux 2017.1 Light 64 bit*, but other distributions may also work. The commands need to be changed accordingly to the used distribution.

A.3.1. Requirements

- Installed Linux operating system
- Distribution package of Maloney
- Internet access
- Free hard disk storage of approximately 120% of the image to be analyzed
- At least 4 GB of memory

A.3.2. Installation

All commands are entered into a terminal with normal user privileges. The command *sudo* is used to elevate the permissions, where required.

1. Make sure the machine is up to date. Enter the following commands:

   ```
   $ sudo apt update && sudo apt upgrade
   ```

2. There are some dependencies to external software packages and libraries: JRE, ewf-tools and afflib-tools. This dependencies can be installed through the package management of the distribution[4]. The tool *curl* is used for testing. Execute the following commands to install them:

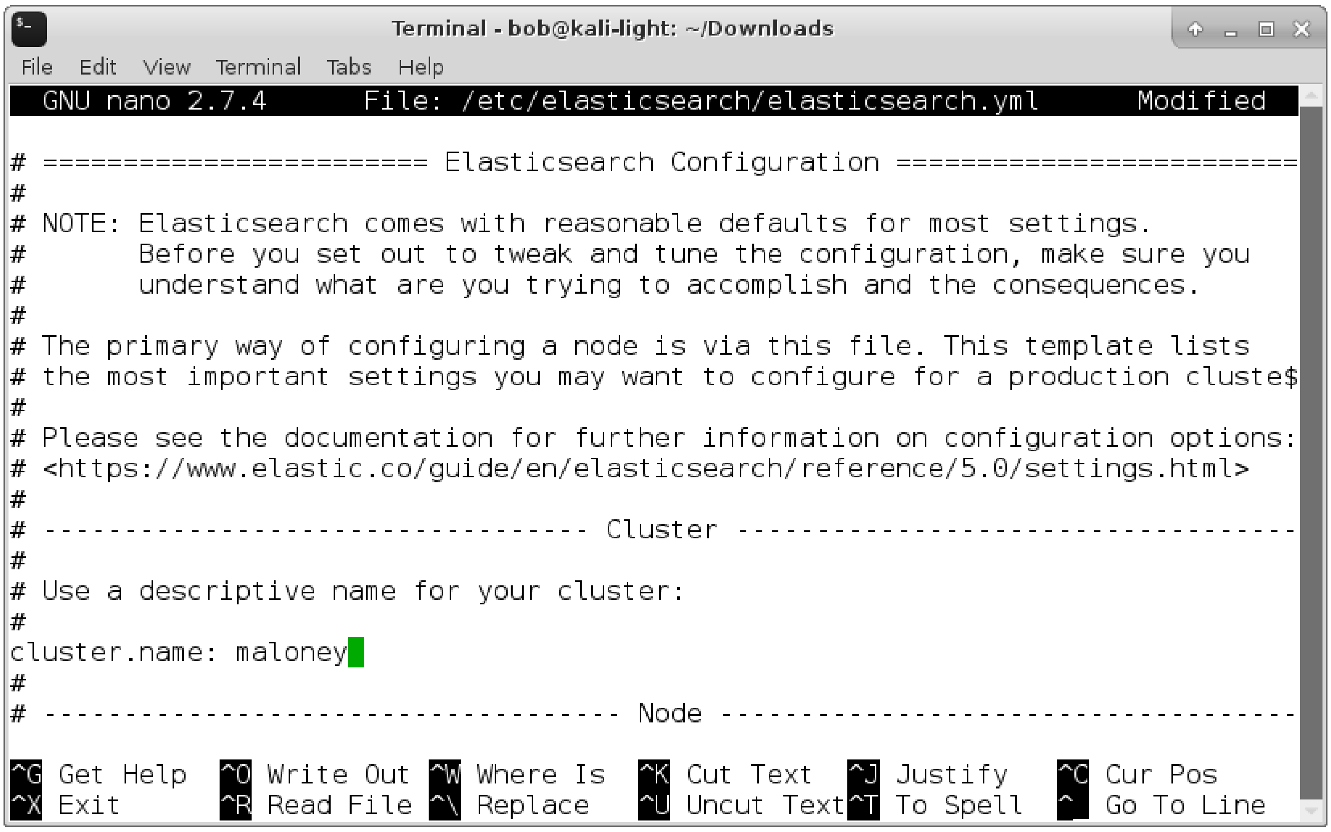
   ```
   $ sudo apt install curl openjdk-8-jre ewf-tools afflib-tools
   ```

3. Download and install Elasticsearch by using the Debian installation package. If the used distribution is not based on Debian, use an alternative package from Elasticsearch Download[5]. Maloney is tested with version 5.0.2, newer bugfix-releases might also work.

   a) Enter the following commands to download and install Elasticsearch:

      ```
      $ cd ~/Downloads/
      $ wget https://artifacts.elastic.co/downloads/elasticsearch/
          elasticsearch-5.0.2.deb
      $ sudo dpkg -i elasticsearch-5.0.2.deb
      ```

   b) Change the clustername of Elasticsearch to "maloney", like in figure A.3, so it will not join a default cluster automatically. Open */etc/elasticsearch/elasticsearch.yml* with an editor and change the property. Save the file afterwards.

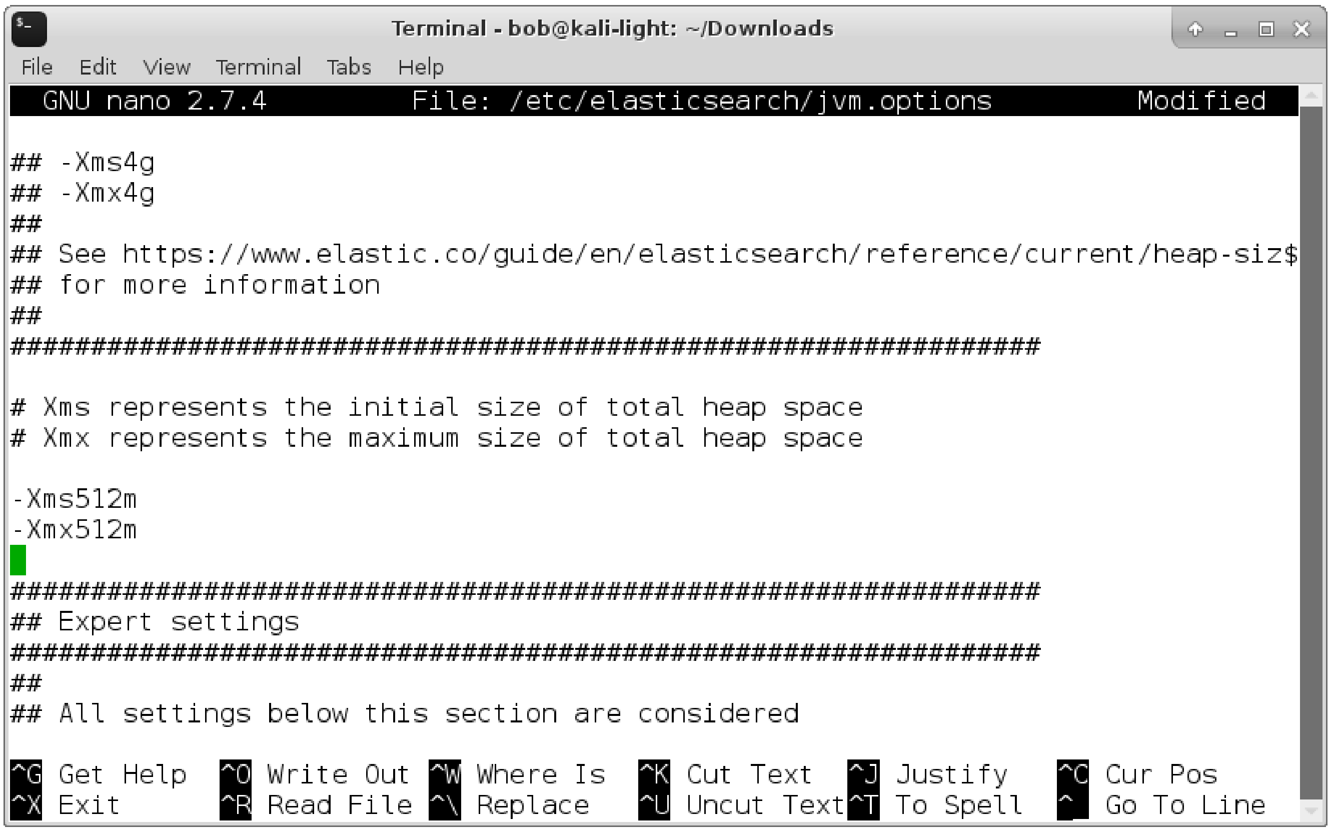

Terminal - bob@kali-light: ~/Downloads
File Edit View Terminal Tabs Help

```
  GNU nano 2.7.4      File: /etc/elasticsearch/elasticsearch.yml      Modified

# ======================== Elasticsearch Configuration =========================
#
# NOTE: Elasticsearch comes with reasonable defaults for most settings.
#       Before you set out to tweak and tune the configuration, make sure you
#       understand what are you trying to accomplish and the consequences.
#
# The primary way of configuring a node is via this file. This template lists
# the most important settings you may want to configure for a production cluste$
#
# Please see the documentation for further information on configuration options:
# <https://www.elastic.co/guide/en/elasticsearch/reference/5.0/settings.html>
#
# ---------------------------------- Cluster -----------------------------------
#
# Use a descriptive name for your cluster:
#
cluster.name: maloney
#
# ------------------------------------ Node ------------------------------------

^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell   ^_ Go To Line
```

Figure A.3.: Changed cluster name in *elasticsearch.yml*

[4] On some distributions, alternative package-source *universe* needs to be activated
[5] `https://www.elastic.co/downloads/elasticsearch`

c) Optional: Tweak ram usage of elasticsearch, especially if your machine has less than 4 GB. "The standard recommendation is to give 50% of the available memory to Elasticsearch heap, while leaving the other 50% free. It won't go unused; Lucene will happily gobble up whatever is left over." [Ela16] The default is 2 GB. To change it, open */etc/elasticsearch/jvm.options* with an editor and change both lines starting with "-Xms" and "-Xmx". See figure A.4 as a reference. Safe the file afterwards.


Terminal - bob@kali-light: ~/Downloads
File Edit View Terminal Tabs Help

```
  GNU nano 2.7.4          File: /etc/elasticsearch/jvm.options          Modified

## -Xms4g
## -Xmx4g
##
## See https://www.elastic.co/guide/en/elasticsearch/reference/current/heap-siz$
## for more information
##
################################################################

# Xms represents the initial size of total heap space
# Xmx represents the maximum size of total heap space

-Xms512m
-Xmx512m

################################################################
## Expert settings
################################################################
##
## All settings below this section are considered

^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell   ^_ Go To Line
```

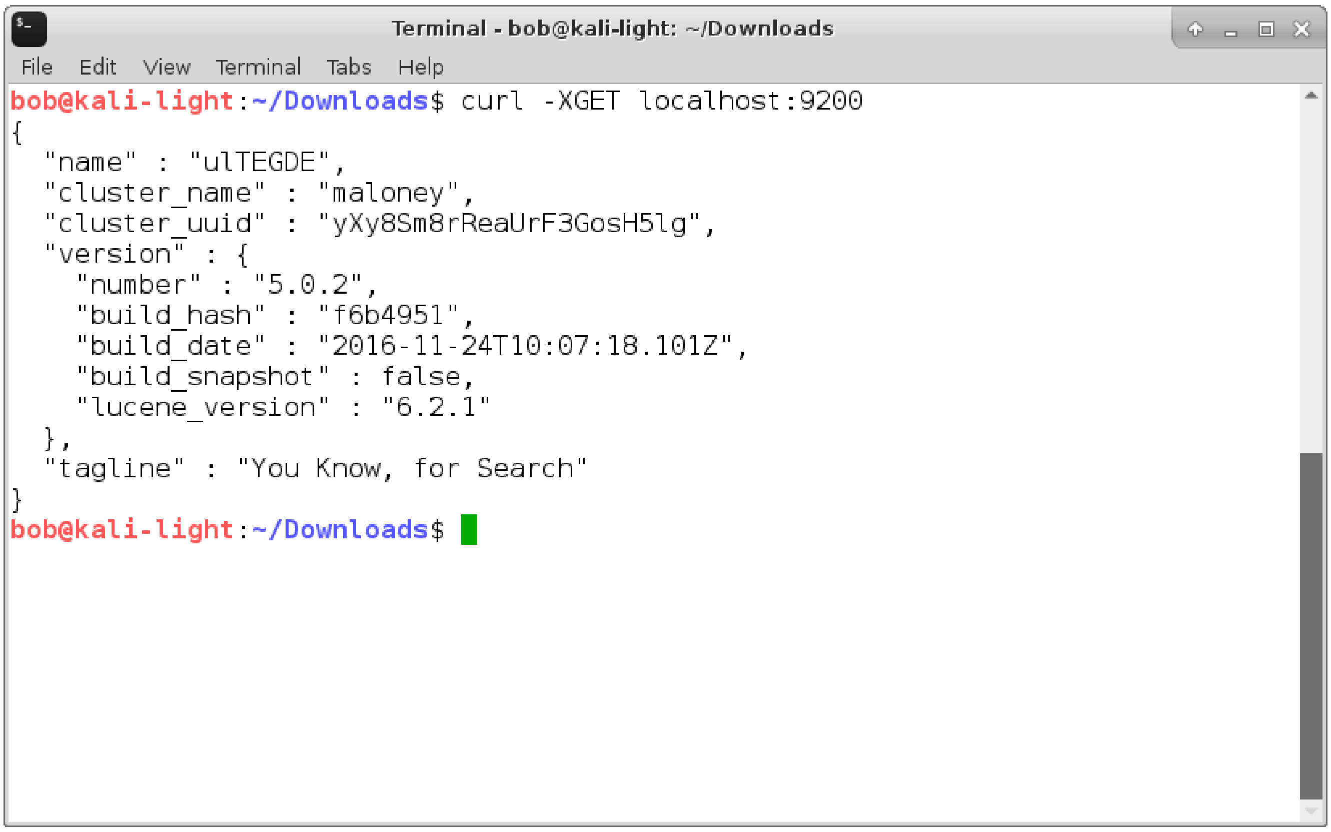
Figure A.4.: Changed heap settings in *jvm.options*

d) Make sure Elasticsearch is configured correctly and it will start. The startup may take some time. Execute the following command:

```
$ sudo systemctl start elasticsearch
```

e) When Elasticsearch has started, it can be tested with an easy HTTP-GET-Request. If the result looks like figure A.5, it should be ok. Use the following command:

```
$ curl -XGET localhost:9200
```


Terminal - bob@kali-light: ~/Downloads
File Edit View Terminal Tabs Help

```
bob@kali-light:~/Downloads$ curl -XGET localhost:9200
{
  "name" : "ulTEGDE",
  "cluster_name" : "maloney",
  "cluster_uuid" : "yXy8Sm8rReaUrF3GosH5lg",
  "version" : {
    "number" : "5.0.2",
    "build_hash" : "f6b4951",
    "build_date" : "2016-11-24T10:07:18.101Z",
    "build_snapshot" : false,
    "lucene_version" : "6.2.1"
  },
  "tagline" : "You Know, for Search"
}
bob@kali-light:~/Downloads$
```

Figure A.5.: Querying Elasticsearch instance with curl

f) Optional: Enable automatic startup of Elasticsearch service with the following command:

```
$ sudo systemctl enable elasticsearch
```

4. Extract the distribution package of Maloney. The location needs to be accessible for the user who will execute the analysis. For the purpose of this documentation, Maloney will be installed to */opt/maloney-cli-2.0-SNAPSHOT*. This location is further referenced as the **installdir**. Enter the following command and replace */path/to/maloney-cli-2.0-SNAPSHOT.tar* with the location of the distribution package:

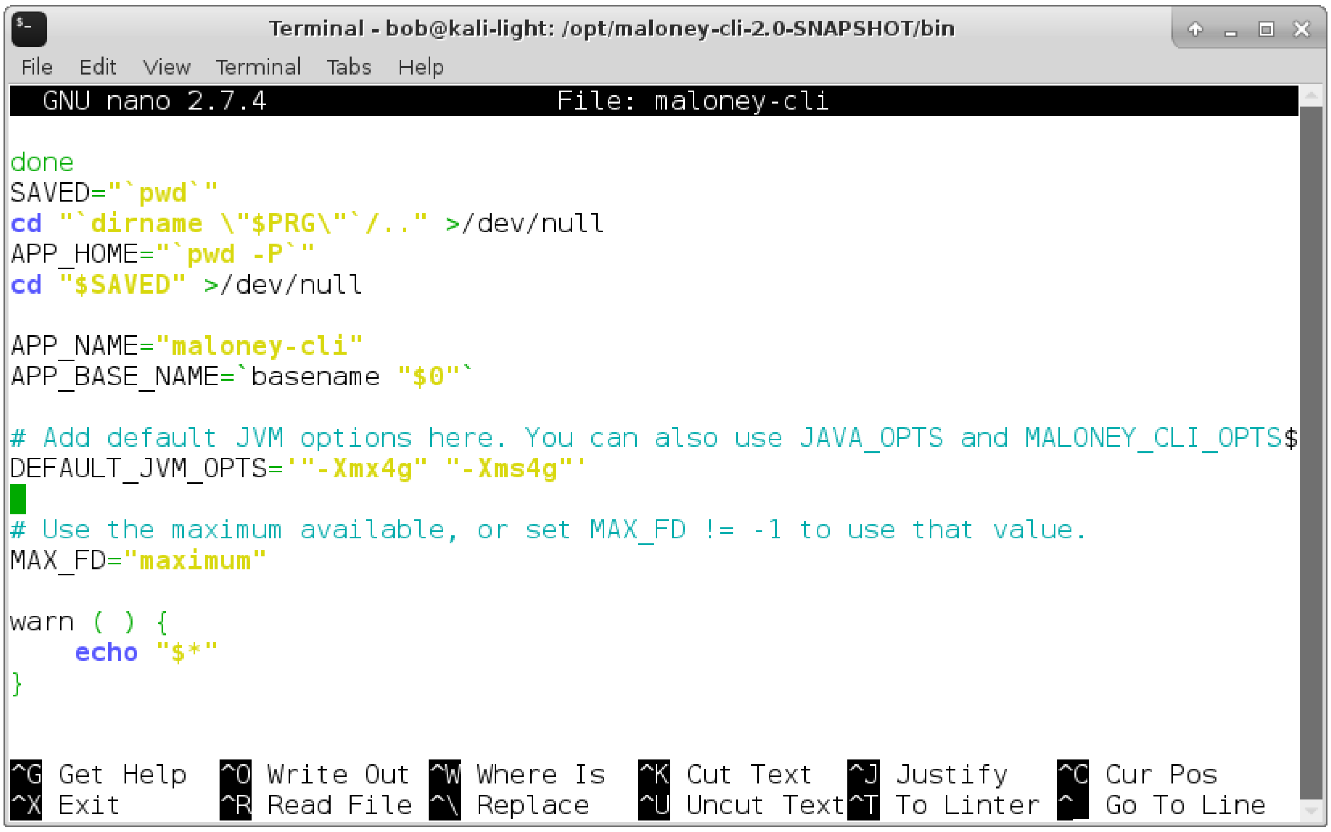
```
$ cd /opt/ && sudo tar -xf /path/to/maloney-cli-2.0-SNAPSHOT.tar
```

5. Optional: Add a link for Maloney to the binaries folder, so it's available from every location. Enter the following command and replace *installdir*:

```
$ sudo ln -s installdir/bin/maloney-cli /usr/local/bin/maloney-cli
```

6. Optional: Increase the available memory for Maloney if an image over 100 GB should be analyzed or if out of memory errors occur. Edit the start script *installdir/bin/maloney-cli* with an editor and change *-Xmx* and *-Xms* in property *DEFAULT_JVM_OPTS*. Save the file afterwards. Following is an example which allows the application to use up to 12 GB:

```
DEFAULT_JVM_OPTS='"-Xmx12g" "-Xms12g"'
```



```
  GNU nano 2.7.4                File: maloney-cli

done
SAVED="`pwd`"
cd "`dirname \"$PRG\"`/.." >/dev/null
APP_HOME="`pwd -P`"
cd "$SAVED" >/dev/null

APP_NAME="maloney-cli"
APP_BASE_NAME=`basename "$0"`

# Add default JVM options here. You can also use JAVA_OPTS and MALONEY_CLI_OPTS$
DEFAULT_JVM_OPTS='"-Xmx4g" "-Xms4g"'

# Use the maximum available, or set MAX_FD != -1 to use that value.
MAX_FD="maximum"

warn ( ) {
    echo "$*"
}


^G Get Help    ^O Write Out   ^W Where Is    ^K Cut Text    ^J Justify     ^C Cur Pos
^X Exit        ^R Read File   ^\ Replace     ^U Uncut Text  ^T To Linter   ^_ Go To Line
```

Terminal - bob@kali-light: /opt/maloney-cli-2.0-SNAPSHOT/bin
File Edit View Terminal Tabs Help

Figure A.6.: Increase available memory for Maloney

7. Optional: Install Kibana for further analysis of the gathered data. Kibana should be the same Version as installed Elasticsearch, in this case 5.0.2. Again, if the distribution is not based on Debian, use an alternative package from Kibana Downloads[6].
   a) Enter the following commands to download and install Kibana:
      ```
      $ cd ~/Downloads/
      $ wget https://artifacts.elastic.co/downloads/kibana/kibana
        -5.0.2-amd64.deb
      $ sudo dpkg -i kibana-5.0.2-amd64.deb
      ```
   b) Optional: Enable automatic startup of Kibana service with the following command:
      ```
      $ sudo systemctl enable kibana
      ```
8. Optional: Remove the downloaded files.

A.4. Using Maloney

Currently, Maloney provides a set of functionality, which is shipped with the framework. This guide should help to provide a quick overview of what can be accomplished with the current version.

This guide is tested with *Kali Linux 2017.1 Light 64 bit*, but other distributions may also work. The commands need to be changed accordingly to the used distribution.

A.4.1. Examine Disk Image

This is the main task of Maloney. It will extract all files from the image, generate hashes for every file and detects PEs.

[6]https://www.elastic.co/downloads/kibana

1. Make sure local Elasticsearch instance is running. For Linux, enter the following command:

   ```
   $ sudo systemctl start elasticsearch
   ```

2. Mount the storage device which contains the disk image to analyze.
3. Create a new configuration file for Maloney, which defines various parameters of the application. Configurations can be reused for different cases.

   a) Use *maloney-cli* to generate a new configuration template. The name and location is irrelevant, but for later reference */path/to/config.json* is used. Enter the following command:

      ```
      $ maloney-cli -sc /path/to/config.json
      ```

   b) Open */path/to/config.json* in a text editor.

   c) Add the job configuration for *DiskImageJob* with the path to the disk image. As example, */path/to/diskimage.dd* is used as image which should be analyzed. The *JobConfigurationMap* should look as followed:

      ```
      "jobConfigurationMap": {
          "DiskImageJob":"/media/sf_shared/diskimage.dd"
      },
      ```

   d) Optional: Define the working directory. If nothing is provided, it defaults to a temporary directory of the operating system. On Linux, it uses */tmp/maloney/*. Keep in mind that this location may be cleaned up after a reboot or automatically if not used. The line should look as following:

      ```
      "workingDirectory": "/some/directory/",
      ```

   e) Save the modified configuration.

4. Replace */path/to/config.json* with the previously created configuration. As case identifier, *incident42* is used. The identifier can be changed, but it can only contain lowercase characters and numbers[7]. Enter the following command:

   ```
   $ maloney-cli -c /path/to/config.json -id incident42
   ```

5. Optional: View the results with Kibana.

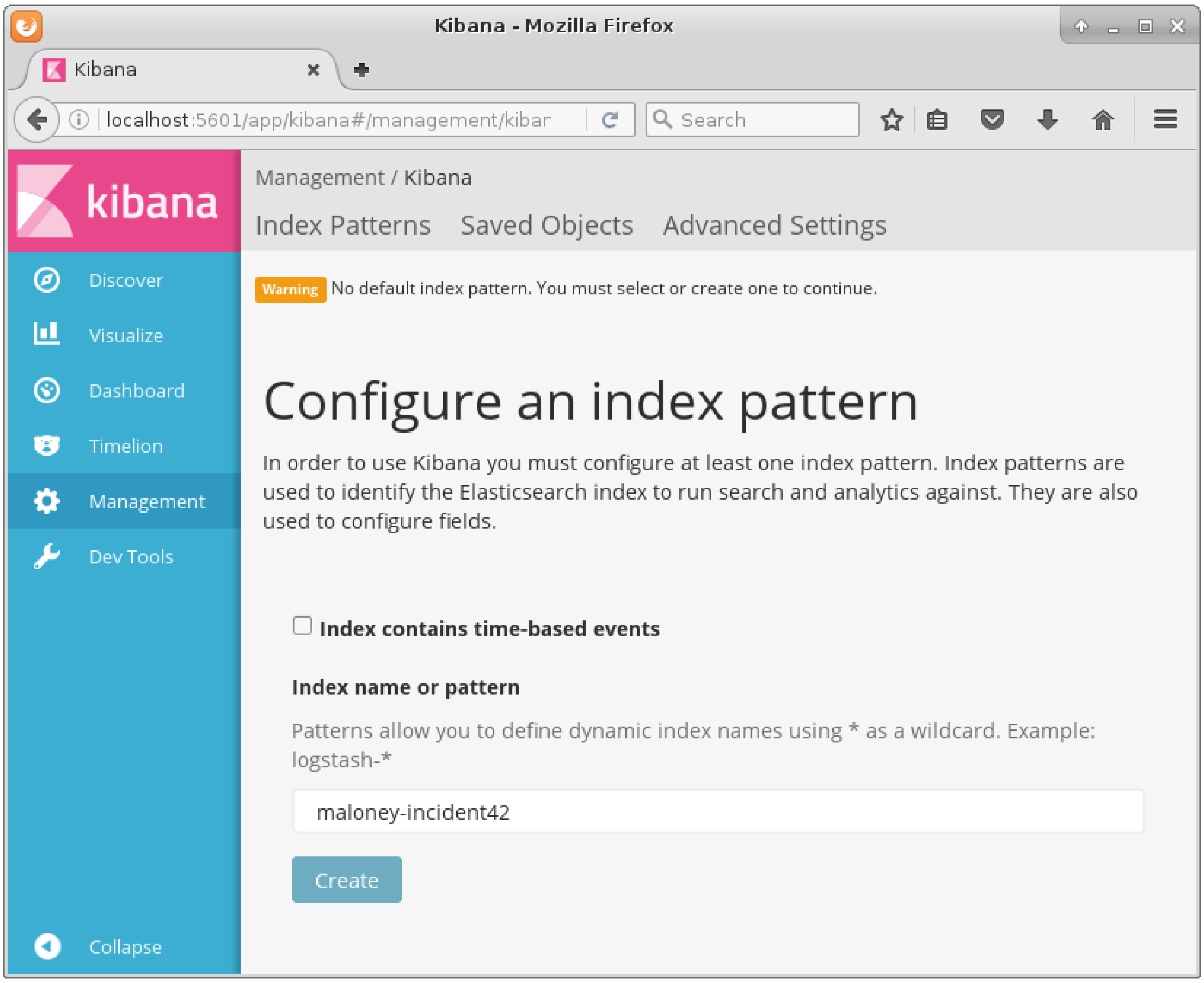
   a) Make sure a local instance of Kibana is running. For Linux, enter the following command:

      ```
      $ sudo systemctl start kibana
      ```

   b) Open your web browser and navigate to `http://localhost:5601/`.

   c) Configure the index pattern for Maloney. If no index pattern is available, the form is automatically displayed to create a new one. Otherwise, go to *Management >Index Patterns >Add New*. Uncheck all checkboxes and enter "maloney-incident42" into the field *Index name or pattern*, so it looks like figure A.7, and click *Create*. The index name consists of the case identifier prefixed with "maloney-".

[7]See chapter 10 Case Management for more information about case identifiers.



Warning No default index pattern. You must select or create one to continue.

Configure an index pattern

In order to use Kibana you must configure at least one index pattern. Index patterns are used to identify the Elasticsearch index to run search and analytics against. They are also used to configure fields.

☐ Index contains time-based events

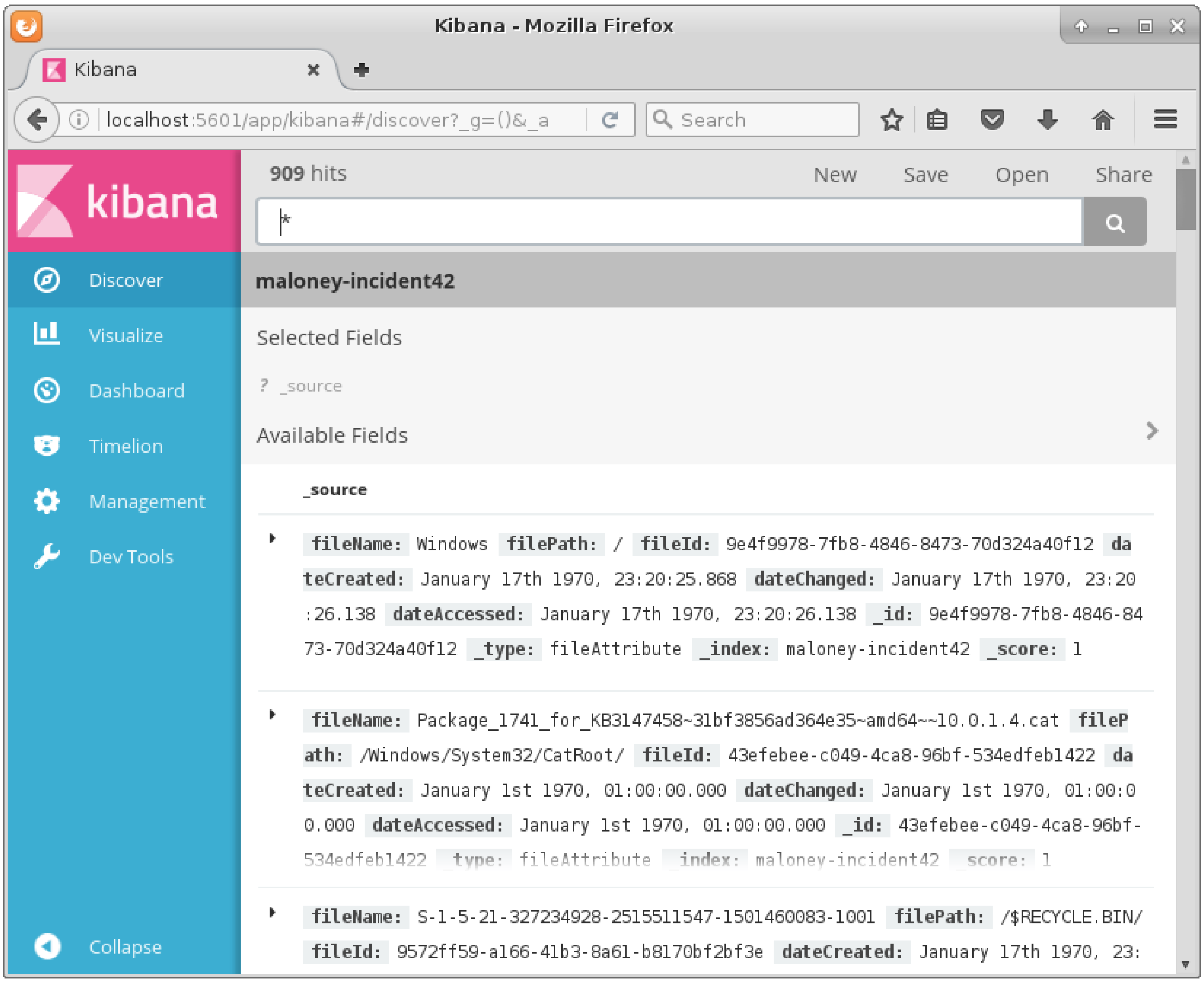
Index name or pattern

Patterns allow you to define dynamic index names using * as a wildcard. Example: logstash-*

maloney-incident42

Figure A.7.: Configure an index pattern in Kibana

d) View all results inside index *maloney-incident42*. Go to *Discover* and make sure the index is selected. Any query can be executed against the indexed data. The window looks similar to figure A.8.



```
fileName: Windows filePath: / fileId: 9e4f9978-7fb8-4846-8473-70d324a40f12 dateCreated: January 17th 1970, 23:20:25.868 dateChanged: January 17th 1970, 23:20:26.138 dateAccessed: January 17th 1970, 23:20:26.138 _id: 9e4f9978-7fb8-4846-8473-70d324a40f12 _type: fileAttribute _index: maloney-incident42 _score: 1

fileName: Package_1741_for_KB3147458~31bf3856ad364e35~amd64~~10.0.1.4.cat filePath: /Windows/System32/CatRoot/ fileId: 43efebee-c049-4ca8-96bf-534edfeb1422 dateCreated: January 1st 1970, 01:00:00.000 dateChanged: January 1st 1970, 01:00:00.000 dateAccessed: January 1st 1970, 01:00:00.000 _id: 43efebee-c049-4ca8-96bf-534edfeb1422 _type: fileAttribute _index: maloney-incident42 _score: 1

fileName: S-1-5-21-327234928-2515511547-1501460083-1001 filePath: /$RECYCLE.BIN/ fileId: 9572ff59-a166-41b3-8a61-b8170bf2bf3e dateCreated: January 17th 1970, 23:
```

Figure A.8.: Discover the indexed file attributes and artifacts in Kibana

A.4.2. Access Extracted Files

After the examination of an image, as explained in section A.4.1, the files can be examined manually. All files of a case are located in a sub folder of the working directory, provided in the configuration. It defaults to the temporary directory of the operating system. For the name of the sub folder, the case identifier is used. If no identifier was provided, it defaults to the current date including an incrementing number. For more details, see chapter 10 Case Management. Using the previous example, the data is located in */some/directory/incident42*.

For every case, a set of different folders and files are created. The table A.1 contains a brief overview. Even if the configuration does not examine an image, this structure is still generated.

The analyst can apply any desired tool to the examined files. Information provided by Maloney can be used to identify the most interesting files to begin with.

File-/Folder-name	Description
files/	Contains all extracted files from the image. Also including additional files, which were generated during the examination. The filename correlates to the file id in the `MetadataStore`.
jobs/	Contains several working directories, which are used by jobs. They always begin with the job identifier followed by an generated id. Even jobs with the same name get different folders.
config_*	Backup of used configuration. Every execution creates a configuration file including all provided parameters.
events.db	Persistent storage of the events.
events.db.wal*	Write ahead logs for the persistent storage.
maloney.log	Log file of Maloney. Search for errors and warnings in this file.

Table A.1.: Recognized values for property in a query or filter

A.4.3. Clear Case Results

An examination takes up a lot of space. Among other things, since all files are extracted. After all evidence and reports are secured, the case can be removed to free up disk space. The following steps describe how to accomplish this.

1. Identify the location of your case on the disk. Section A.4.2 describes where to find it. It is */some/directory/incident42* for this guide, with *incident42* as the case identifier.
2. Delete the entire case directory with the following command:

   ```
   rm -rf /some/directory/incident42
   ```

3. Delete the generated elastic search indexes. It can be accomplished using the developer tools in Kibana or with the command *curl*. Enter the following command, if *curl* is available:

   ```
   curl -XDELETE localhost:9200/maloney-incident42
   ```

If there is an acknowledgement, everything related to the case is gone.

A.4.4. Import RDS Hash Set

Maloney can index the NIST NSRL RDS. Every calculated hash for a file is looked up in this index during an examination. Following steps describe how to accomplish this.

1. Download the RDS unique set from NSRL Downloads[8]. This may take some time, because the file has a size of about 1.7 GB.
2. Make sure the local Elasticsearch instance is running. For Linux, enter the following command:

   ```
   $ sudo systemctl start elasticsearch
   ```

3. Create a new configuration file for Maloney, which defines various parameters of the application. Configurations can be reused for different cases.
   a) Use *maloney-cli* to generate a new configuration template. The name and location is irrelevant, but for later reference */path/to/config.json* is used. Enter the following command:

      ```
      $ maloney-cli -sc /path/to/config.json
      ```

   b) Open */path/to/config.json* in a text editor.

[8]`http://www.nsrl.nist.gov/Downloads.htm`

c) Add the job configuration for *ImportRdsHashSetJob* with the path to the disk image. As example, */path/to/rds_254_u.zip* references to the earlier downloaded file. The *jobConfigurationMap* should look as followed:

```
"jobConfigurationMap": {
    "ImportRdsHashSetJob":"/path/to/rds_254_u.zip"
},
```

d) Save the modified configuration.

4. Enter the following command and replace */path/to/config.json* to start the analysis with the configuration file:

```
$ maloney-cli -c /path/to/config.json
```

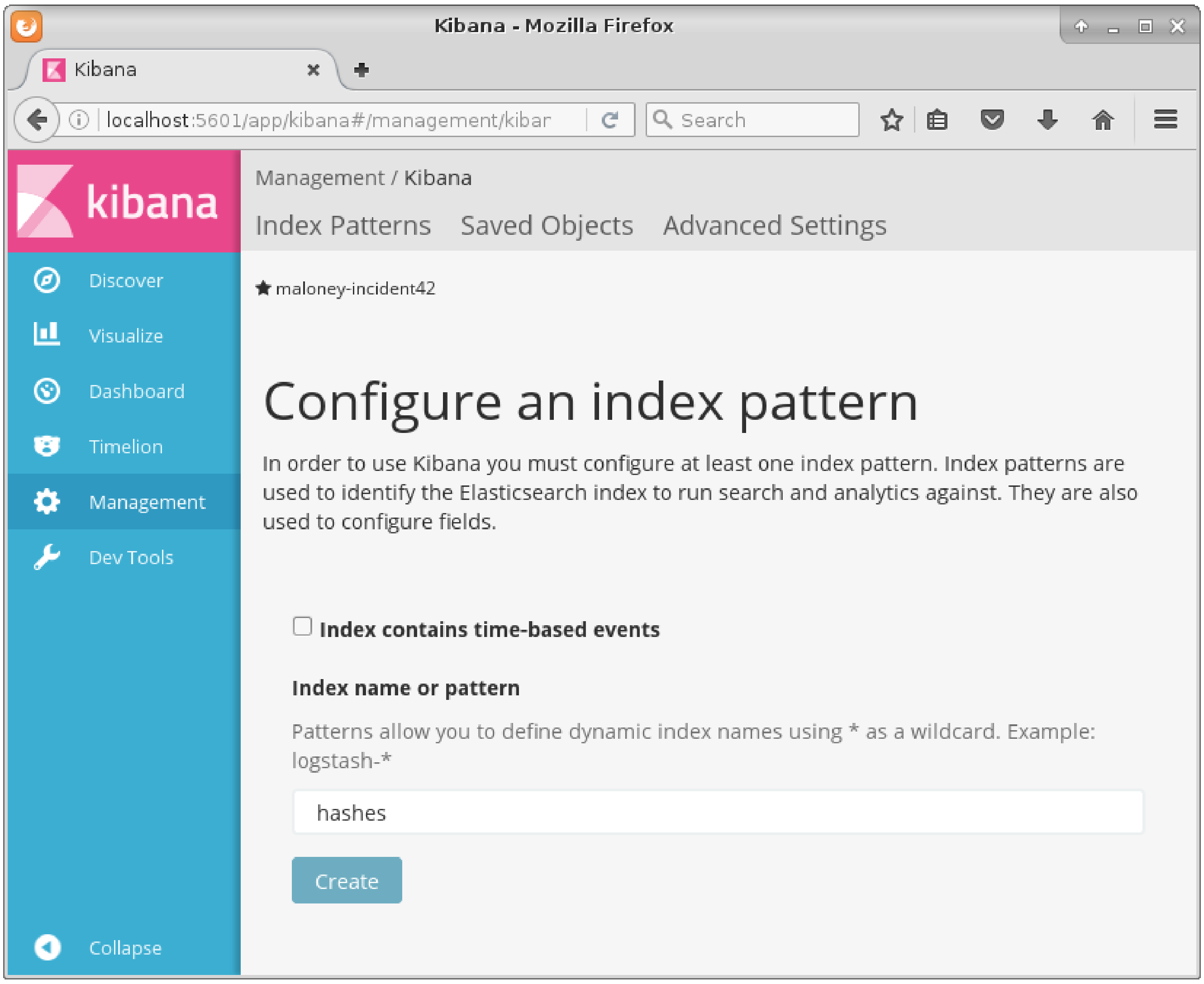
5. Optional: View the results with Kibana.

a) Make sure a local instance of Kibana is running. For Linux, enter the following command:

```
$ sudo systemctl start kibana
```

b) Open your web browser and navigate to `http://localhost:5601/`.

c) Configure the index pattern for maloney. If no index pattern is available, the form is automatically displayed to create a new one. Otherwise, go to *Management >Index Patterns >Add New*. Uncheck all checkboxes and enter "hashes" into the field *Index name or pattern*, so it looks like figure A.9, and click *Create*.



Configure an index pattern

In order to use Kibana you must configure at least one index pattern. Index patterns are used to identify the Elasticsearch index to run search and analytics against. They are also used to configure fields.

☐ Index contains time-based events

Index name or pattern

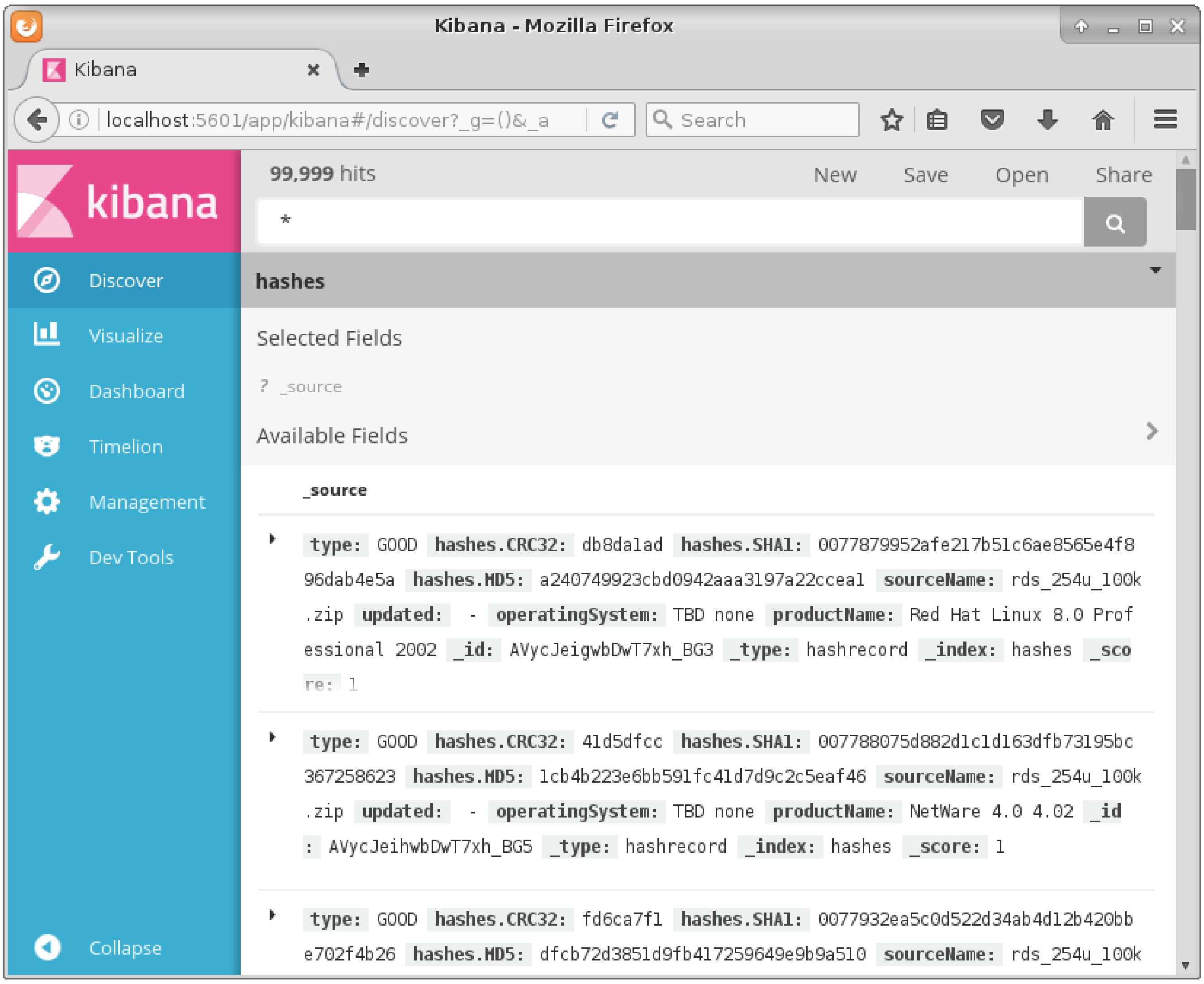
Patterns allow you to define dynamic index names using * as a wildcard. Example: logstash-*

hashes

Create

Figure A.9.: Configure an index pattern in Kibana

d) View all results inside index *hashes*. Go to *Discover* and make sure the index is selected. Any query can be executed against the indexed data. The window looks similar to figure A.10.



```
type: GOOD hashes.CRC32: db8da1ad hashes.SHA1: 0077879952afe217b51c6ae8565e4f8
96dab4e5a hashes.MD5: a240749923cbd0942aaa3197a22ccea1 sourceName: rds_254u_100k
.zip updated: - operatingSystem: TBD none productName: Red Hat Linux 8.0 Prof
essional 2002 _id: AVycJeigwbDwT7xh_BG3 _type: hashrecord _index: hashes _sco
re: 1

type: GOOD hashes.CRC32: 41d5dfcc hashes.SHA1: 007788075d882d1c1d163dfb73195bc
367258623 hashes.MD5: 1cb4b223e6bb591fc41d7d9c2c5eaf46 sourceName: rds_254u_100k
.zip updated: - operatingSystem: TBD none productName: NetWare 4.0 4.02 _id
: AVycJeihwbDwT7xh_BG5 _type: hashrecord _index: hashes _score: 1

type: GOOD hashes.CRC32: fd6ca7f1 hashes.SHA1: 0077932ea5c0d522d34ab4d12b420bb
e702f4b26 hashes.MD5: dfcb72d3851d9fb417259649e9b9a510 sourceName: rds_254u_100k
```

Figure A.10.: Discover the indexed hashes in Kibana

As shown, the instructions are quite similar to section A.4.1 Examine Disk Image. Importing the RDS is handled the same way as an examination and therefore a case identifier is required. If none is provided, it will be generated. The created case can be cleaned up following the instructions provided in section A.4.3 Clear Case Results.

A.4.5. Clear Stored Hashes

All hashes are stored in one index of Elasticsearch. All examinations use the same hashes for look ups. If a new one is required due to different reasons, it has to be cleared first. If *curl* is available, use the following command:

```
$ curl -XDELETE localhost:9200/hashes
```

If only the Kibana developer tools are available. Enter the following command into the console window and execute it:

```
DELETE hashes
```

A.4.6. Generate a Report

Every execution of Maloney generates a new report using all available categories. The report is usually located in the working directory of *ReportJob*. It consists of a comma separated list and is named *report.csv*. This file will be overwritten after every execution which also includes a *startup* event.

A.4.7. Query Indexed Data

The examined meta data can be queried using the CLI. To identify the correct case, a configuration file and the case identifier is required. Following steps will show how to query the data.

1. Optional: Create a new configuration file. A previously used configuration file can be used without creating a new one too. A query will not execute anything else.
   a) Use *maloney-cli* to generate a new configuration template. The name and location is irrelevant, but for later reference */path/to/config.json* is used. Enter the following command:
      ```
      $ maloney-cli -sc /path/to/config.json
      ```
   b) Open */path/to/config.json* in a text editor.
   c) Define the working directory. If nothing is provided, it defaults to a temporary directory of the operating system. On Linux, it uses */tmp/maloney/*. Keep in mind that this location may be cleaned up after a reboot or automatically if not used. The line should look as follows:
      ```
      "workingDirectory": "/some/directory/",
      ```
   d) Save the modified configuration.
2. A query requries a query term, which represents a list of key-value pairs. The filter is optional and influences the displayed attributes. As configuration file */path/to/config.json* is used together with the case identifier *incident42*. To perform a query, enter a command similar to the following:
   ```
   $ maloney-cli -c /path/to/config.json -id incident42 -q 'filename=".*
     exe" --filter 'fileid filename'
   ```

The output can either be viewed on the CLI or piped into a file. The output is always tab delimited. Stored files can be analyzed and reviewed with any text editor. For more information see chapter 14 Simple Queries.

A.4.8. Create Disk Image

The easiest way to create a disk image is by using a Linux live CD. Most live CDs include the tool *dd*, which can create raw disk images from a hard drive. To get a list of all drives, the following commands can be used:

```
$ df -h
$ sudo fdisk -l
```

Enter the following command to create an image, replace the `X` with the drive letter identified before and replace */path/to/image.dd* with a location large enough to store the entire image. The resulting image is uncompressed and uses the same size as the source.

```
$ sudo dd if=/dev/sdX of=/path/to/image.dd
```

A.5. Security Aspects

Unlike other databases or similar systems, Elasticsearch has no integrated security system. Everyone which has access to the instance can do absolutely everything: View, change and delete indexes, reconfigure the instance, and so on. Keep this in mind if a confidential disk image should be analyzed. Use a firewall to prevent access to Elasticsearch and, if used, Kibana. The configuration allows to bind both to a specific network interface. This guide will not describe, how to do this, because it is too specific to the environment.

There is also a commercial plugin for Elasticsearch and Kibana to add authentication and encryption. For more information, see Elasticsearch Security[9].

It is also important to note that all extracted files are unprotected: There are no additional permissions settings or encryption whatsoever. Any software which runs with the same permissions as the user or higher can read these files.

[9] `https://www.elastic.co/products/x-pack/security`

B. Configuration Handbook

In this chapter located is a list of all `Job` implementations provided by the current version of Maloney, their descriptions and configuration parameters. The list is alphabetically sorted. For a better overview of the logical order of execution, see figure B.1. The figure contains only the used events. To describe the implementations, following schema is used:

Name The Name of the `Job` implementation.

Canonical Name The name which can be used in an import statement.

Description A brief explanation of what it does.

Requires Event Which `Event` the implementation requires so it can execute.

Produces Event Which `Event` the implementation will generate on a successful execution, if any.

Job Configuration What information this implementation expects in the job configuration string, if any.

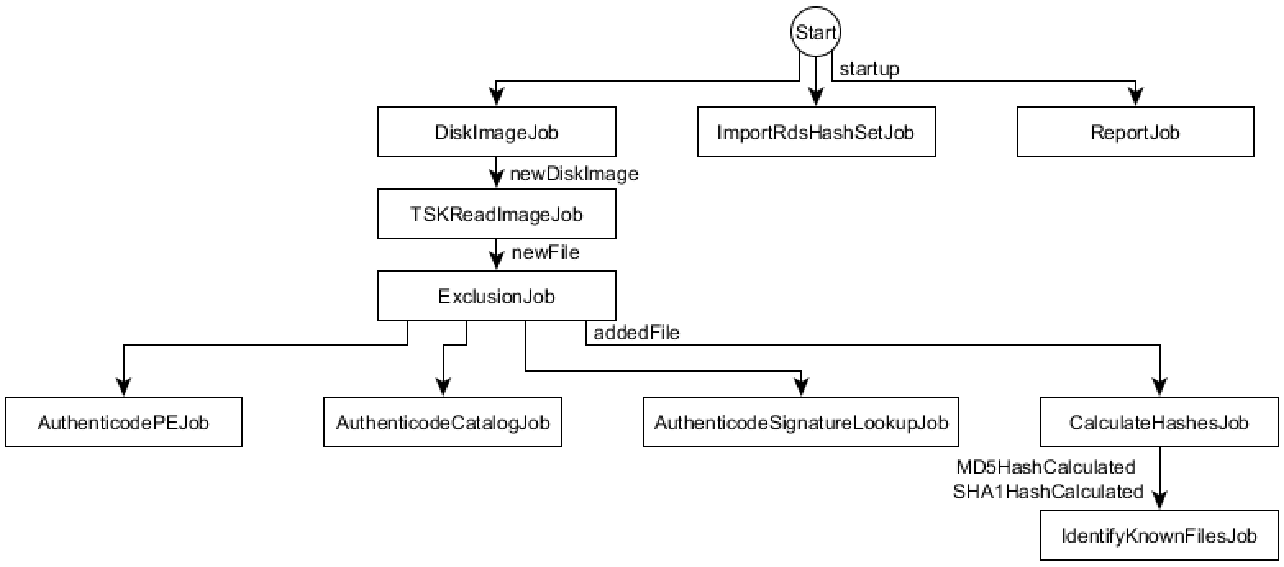


Figure B.1.: Chains of `Job` implementations and `Event` in between

B.1. AuthenticodeCatalogJob

Name	AuthenticodeCatalogJob
Canonical Name	ch.hsr.maloney.maloney_plugins.authenticode.AuthenticodeCatalogJob
Description	Extracts Authenticode signature information of catalog files.
Requires Event	`addedFile`
Produces Event	-
Job Configuration	-

Table B.1.: Description of `AuthenticodeCatalogJob`

B.2. AuthenticodePEJob

Name	AuthenticodePEJob
Canonical Name	ch.hsr.maloney.maloney_plugins.authenticode.AuthenticodePEJob
Description	Extracts and validates Authenticode signature information of a portable executable if a signature exists.
Requires Event	`addedFile`
Produces Event	-
Job Configuration	-

Table B.2.: Description of `AuthenticodePEJob`

B.3. AuthenticodeSignatureLookupJob

Name	AuthenticodeSignatureLookupJob
Canonical Name	ch.hsr.maloney.maloney_plugins.authenticode.AuthenticodeSignatureLookupJob
Description	Compares signature extracted through `AuthenticodeCatalogJob` with the ones from `AuthenticodePEJob`.
Requires Event	`addedFile`
Produces Event	-
Job Configuration	anything but `null`, will not run otherwise.

Table B.3.: Description of `AuthenticodeSignatureLookupJob`

B.4. CalculateHashesJob

Name	CalculateHashesJob
Canonical Name	ch.hsr.maloney.processing.CalculateHashesJob
Description	Calculates MD5 and SHA1 hash and stores them as `Artifact`.
Requires Event	`addedFile`
Produces Event	`MD5HashCalculated` and `SHA1HashCalculated`
Job Configuration	-

Table B.4.: Description of `CalculateHashesJob`

B.5. DiskImageJob

Name	DiskImageJob
Canonical Name	ch.hsr.maloney.processing.DiskImageJob
Description	Adds a new disk image to the application to be examined.
Requires Event	`startup`
Produces Event	`newDiskImage`
Job Configuration	Path to image file, will not run otherwise.

Table B.5.: Description of `DiskImageJob`

B.6. ExclusionJob

Name	ExclusionJob
Canonical Name	ch.hsr.maloney.processing.ExclusionJob
Description	Filters out files as defined by the job configuration, only creates `addedFile` event if filter does not match.
Requires Event	`newFile`
Produces Event	`addedFile`
Job Configuration	Key-value pairs, which use a field name and a regular expression in quotes. A pair should be structured like the following example: `fileName=".*txt"`. The case of the field name (i.e. `fileName`) does not matter.

Table B.6.: Description of `ExclusionJob`

Possible field names for the filter are as follows:

- `FileId`
- `FileName`
- `FilePath`
- `DateAccessed`
- `DateChanged`
- `DateCreated`
- `ArtifactType`
- `ArtifactOriginator`
- `ArtifactValue`

B.7. IdentifyKnownFilesJob

Name	IdentifyKnownFilesJob
Canonical Name	ch.hsr.maloney.maloney_plugins.IdentifyKnownFilesJob
Description	Compares artifacts from `CalculateHashesJob` with stored hashes from `ImportRdsHashSetJob` and adds artifacts to the file if a match is found.
Requires Event	`MD5HashCalculated` or `SHA1HashCalculated`
Produces Event	-
Job Configuration	-

Table B.7.: Description of `IdentifyKnownFilesJob`

B.8. ImportRdsHashSetJob

Name	ImportRdsHashSetJob
Canonical Name	ch.hsr.maloney.processing.ImportRdsHashSetJob
Description	Import a zipped National Institute of Standards and Technology (NIST) National Software Reference Library (NSRL) Reference Data Set (RDS) available at http://www.nsrl.nist.gov/Downloads.htm.
Requires Event	`startup`
Produces Event	-
Job Configuration	Path to NIST NSRL RDS zip-file, will not run otherwise.

Table B.8.: Description of `ImportRdsHashSetJob`

B.9. ReportJob

Name	ReportJob
Canonical Name	ch.hsr.maloney.processing.ReportJob
Description	Generates a report based on the stored meta data and artifacts. Extends every entry with category tags where they match.
Requires Event	`startup`
Produces Event	-
Job Configuration	-

Table B.9.: Description of `ReportJob`

B.10. TSKReadImageJob

Name	TSKReadImageJob
Canonical Name	ch.hsr.maloney.processing.TSKReadImageJob
Description	Extracts all files, directories and unallocated spaces including their meta data. Adds the extracted data to the `DataSource`.
Requires Event	`newDiskImage`
Produces Event	`newFile`, `newDirectory` and `newUnallocatedSpace`
Job Configuration	-

Table B.10.: Description of `TSKReadImageJob`

C. Developer Handbook

This handbook is concerned with the necessary steps and the limitations of developing new `Job` for Maloney. This includes implementations for more examination methods or reporting. In addition, it explains how to create custom categories.

C.1. Creating a Job

For a `Job` implementation to be loaded into Maloney, two things are necessary. First is the `Job` implementation itself. Secondly the jar needs a service provider definition. This service provider definition needs to be located at `META-INF/services/ch.hsr.maloney.processing.Job`. Inside this file the new `Job` implementations need to be added with their canonical names. For an example, see listing C.1.

Listing C.1: Service provider definition with new `Job` implementations

```
new-project.maloney-plugin.NewExaminationJob
new-project.maloney-plugin.NewReportJob
```

After the class has been created and the service provider definition added, the implementation can begin. The resulting JAR can be added to the plug-in folder of Maloney and it will be loaded on application start.

C.1.1. Job Logic

Create a new class which implements the `Job` interface. Implement the methods as requested by the interface, which also contains a documentation. Generally, a `Job` implementation gets executed in the following order:

1. `shouldRun`
2. `canRun`
3. `run`

Both methods `shouldRun` and `canRun` should be short and non-blocking and shall not change any data. Only the method `run` should do that.

If anything fails during the `run` method, a `JobCancelledException` should be thrown. E.g. when there is not enough disk space available for the `Job` to successfully complete.

C.1.2. Position in Event Chain

Before any execution takes place, the `Framework` checks whether all registered `Job` implementations can be run as is. For this, it checks the methods `getRequiredEvents` and `getProducedEvents`. The application only starts when a proper chain with `Events` can be built. For a list of all standard `Events` check chapter B Configuration Handbook. Generally, `Event` names should be short and human readable.

C.1.3. Job Configuration

For the application to start, a configuration file has to be supplied. If there is a configuration inside the `jobConfigurationMap` in form of `"jobName":"Configuration As String"`, it gets passed to the `Job` implementation after its instantiation with the method `setJobConfig`. The provided configuration is valid for the entire lifetime of the instance and will not change.

C.1.4. Limitations

It is important to note that some limits apply when creating new `Job` implementations.

First off is the need to synchronize the access on local variables. It is possible that multiple threads run the same method of the same instance. This becomes a problem if local variables or class fields are used. Therefore it is recommended to get all necessary data through the `Context` and only use the scope of the method. If this is not possible, proper synchronization of those variables is necessary.

The framework uses the default constructor to create an instance. Other constructors are ignored. If a configuration through parameters is required, the method `setJobConfig` should be used.

C.2. Create a Category

Similar to creating a new `Job`, creating a custom `Category` involves the implementation of the `Category` interface and adding a service provider definition. This time the definition is located in `META-INF/services/ch.hsr.maloney.util.categorization.Category`.

C.2.1. Category Rules

First, it needs to be decided whether all rules need to apply for the rule to match or any of them. An `AndRuleComposite` is for the former and `OrRuleComposite` for the latter (see chapter 12 Categorization).

After that, more composites or rules can be added to the base composite. Rules are single parameter methods an can be replaced with lambdas. For an example, see listing C.2.

Listing C.2: Example of a Category with one rule

```
public class EvilExeCategory implements Category {
    @Override
    public String getName() {
        return "EvilExeCategory";
        // Alternativley, if the default category should be extended:
        // return DefaultCategory.KNOWN_BAD.getName();
    }

    @Override
    public RuleComposite getRules() {
        RuleComposite ruleComposite = new OrRuleComposite();
        ruleComposite.addRule(fileAttributes -> fileAttributes.getName().
           equals("evil.exe"));
        return ruleComposite;
    }
}
```

C.3. Further Reading

For more information on how the plug-in mechanism works, check chapter 4 Plug-in Architecture.

List of Figures

List of Tables

Glossary

API Application Programming Interface. 15, 16, 18, 42, 43

Authenticode Code signing implementation by Microsoft. 42–44, 82

Autopsy GUI-based solution to analyze disk images; using TSK; `http://www.sleuthkit.org/`. 42

Bouncy Castle Cryptographic library for Java and C#. 44

class loader Loads Java classes during runtime. 14–16

classpath Environment variable which is used by Java to look up class files. 14–17

CLI Command Line Interface. 13, 15, 29, 30, 50, 51, 72

CSV Comma-separated values. 10

Cuckoo Sandbox Open-source sandbox tool for examination of malware. 7, 55

Data Transfer Object A data transfer object is an object that carries data between processes. 82

distribution package Software artifact, which can be used to distribute an application on clients. 60

DTO Data Transfer Object. 82, *Glossary:* Data Transfer Object

Elasticsearch Distributed, RESTful search and analytics engine. 13, 25–27, 33, 37, 40, 43, 50, 54, 55, 57, 59, 62–66, 69, 71, 73, 80

ETA Estimated Time of Arrival. 31

event Signifies a change inside the application, in Maloney used for communication between tasks. 12, 27, 48, 54, 80

git Distributed version control system. 58

Gradle Build management tool for different languages based on Ant and Maven. 13, 14, 16, 18, 44, 51, 57, 58, 60

GUI Graphical User Interface. 30

IDE Integrated Development Environment. 11, 13, 18, 57, 59, 60

JAR Java Archive. 14–17, 27, 78

JDK Java Development Kit. 57

Job Application logic of Maloney separated into multiple encapsulated tasks. These are called `Jobs`. 12, 16–18, 20, 22, 24–26, 29–34, 37–39, 43, 46, 49, 51, 55, 78

JPF Java Plugin Framework. 15, 16

JRE Java Runtime Environment. 16, 62

Jsign Java implementation of Authenticode. 42–44, 58

JSON Javascript Object Notation. 35

JVM Java Virtual Machine. 14, 21, 34, 54

Kibana Tools to visualize and navigate data stored in Elasticsearch. 13, 53, 57, 65–71, 73, 80

Last In First Out Description of behavior in a queue: The last queued item is the first to get out of it if the queue gets polled. 22

log4j Common logging library to extend the capabilities of an application. 33, 40, 41

Maloney Name of the developed application during this term project. 1, 3–5, 12–14, 16–18, 20, 21, 24, 25, 27, 29, 32, 33, 36, 39, 40, 45, 48, 50, 51, 53–55, 57, 61, 62, 64–66, 68, 69, 72, 74, 78, 80, 82

MapDB Java library which provides Collections supporting off-heap or on-disk storage. 33–35, 41, 54

Maven Build management tool primary for Java applications. 44, 58

MD5 Hashing Algorithm. 8, 26, 42

MIME Multipurpose Internet Mail Extensions. 10, 83, *Glossary:* Multipurpose Internet Mail Extensions

Multipurpose Internet Mail Extensions Internet standard that extends the format of emails to add extra functionality. 83

NIST National Institute of Standards and Technology. 4–6, 69, 77, 80

NSRL National Software Reference Library. 6, 69, 77

NTFS File system used by Microsoft Windows. 40

OpenSSL Toolkit for TLS and SSL including a general-purpose cryptographic library. 44

OSGi Open Services Gateway initiative. 15, 16, 18, 19

PE Portable Executable. 42, 43, 65, 83, *Glossary:* Portable Executable

plug-in Software component that adds a specific feature to an existing computer program. 14–19, 27, 37, 58, 78

Portable Executable Executable application in a binary format. 42, 83

RAM drive A block of random access memory which is treated as a disk drive. 39

RDS Reference Data Set. 6, 25, 27, 69, 71, 77, 83, *Glossary:* Reference Data Set

Reference Data Set A colletion of signatures of software from various sources, which are known and traceable applications. This software is collected in the National Software Reference Library `http://www.nsrl.nist.gov/`. 6, 77, 83

service provider Implementation which provides a specific service. 14, 15

SHA-1 Hashing Algorithm. 8, 26, 42, 43

SHA-256 Hashing Algorithm. 42, 43

SPI Service Provider Interface. 14–18

The Sleuth Kit Forensic toolkit to recover and analyze files `http://www.sleuthkit.org/`. 12, 83

Time Stamp Authority Public available authority to provide a validated time stamp. 43, 83

TSA Time Stamp Authority. 43, 83, *Glossary:* Time Stamp Authority

TSK The Sleuth Kit. 12, 13, 18, 28, 54, 55, 58, 61, 82, 83, *Glossary:* The Sleuth Kit

UCT Coordinated Universal Time. 52

X.509 Standard which defines the format of public key certificates commonly used in cryptography. 42

YARA Multi-platform tool for pattern matching in context of malware research and detection. 6, 7, 55

Bibliography

[17a] *About Us*. OSGi Alliance. Mar. 13, 2017. URL: https://www.osgi.org/about-us/.

[17b] *Benefits of using OSGi*. OSGi Alliance. Mar. 13, 2017. URL: https://www.osgi.org/developer/benefits-of-using-osgi/.

[17c] *Class ClassLoader*. Java Platform SE 8. Oracle. Mar. 13, 2017. URL: http://docs.oracle.com/javase/8/docs/api/java/lang/ClassLoader.html.

[17d] *Class ServiceLoader*. Java Platform SE 8. Oracle. Mar. 13, 2017. URL: http://docs.oracle.com/javase/8/docs/api/java/util/ServiceLoader.html.

[17e] *Creating Extensible Applications*. The Java^{TM} Tutorials. Oracle. Mar. 13, 2017. URL: https://docs.oracle.com/javase/tutorial/ext/basics/spi.html.

[ANTLR] *Antlr Github Repository and Documentation*. URL: https://github.com/antlr/antlr4 (visited on 06/08/2017).

[Cou+12] George Coulouris et al. *Distributed Systems: Concepts and Design*. 2012. ISBN: 978-0-13-214301-1.

[DM15] Hans Dockter and Adam Murdoch. *Gradle User Guide*. 2015. URL: https://docs.gradle.org/2.13/userguide/userguide.html (visited on 06/12/2017).

[Ede14] Lukas Eder. *Don't be "Clever": The Double Curly Braces Anti Pattern*. Dec. 8, 2014. URL: https://blog.jooq.org/2014/12/08/dont-be-clever-the-double-curly-braces-anti-pattern/ (visited on 06/15/2017).

[Ela16] Elasticsearch BV. *Elasticsearch: The Definitive Guide. Heap: Sizing and Swapping*. Dec. 16, 2016. URL: https://www.elastic.co/guide/en/elasticsearch/guide/current/heap-sizing.html.

[Ela17a] Elasticsearch BV. *Nested datatype*. Elasticsearch Reference 5.0. Mar. 28, 2017. URL: https://www.elastic.co/guide/en/elasticsearch/reference/5.0/nested.html.

[Ela17b] Elasticsearch BV. *Scroll*. Sept. 5, 2017. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/search-request-scroll.html.

[EN16] Roman Ehrbar and Oliver Nietlispach. "Malware Hunting". Term Project. University of Applied Sciences Rapperswil, 2016.

[Göe+06] Brian Göetz et al. *Java Concurrency In Practice*. Mar. 1, 2006. ISBN: 978-0-321-34960-6.

[Han07] Robert S. Hanmer. *Patterns for Fault Tolerant Software*. 2007. ISBN: 978-0-470-31979-6.

[JPF17] JPF Team. *Java Plugin Framework*. Mar. 14, 2017. URL: http://jpf.sourceforge.net/.

[Ken+06] Karen Kent et al. *NIST Guide to Integrating Forensic Techniques into Incident Response*. Aug. 1, 2006. URL: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800 86.pdf.

[Kot17] Jan Kotek. *MapDB Documentation*. 2017. URL: https://jankotek.gitbooks.io/mapdb/content/.

[Mic] Microsoft. *Authenticode*. URL: https://msdn.microsoft.com/en-us/library/ms537359(v=vs.85).aspx.

[Mic08] Microsoft Corporation. *Windows Authenticode Portable Executable Signature Format*. Mar. 21, 2008. URL: http://www.microsoft.com/whdc/winlogo/drvsign/Authenticode_PE.mspx.

[Ora17] Oracle. *Java API Documentation: ForkJoinPool*. Mar. 14, 2017. URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html.

[Par15] Nicolai Parlog. *JAR Hell*. Oct. 19, 2015. URL: http://blog.codefx.org/java/jar-hell/.

[RFC2046] Ned Freed and Nathaniel S. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. RFC 2046. RFC Editor, July 1995. URL: https://tools.ietf.org/html/rfc2046.

[SS13] Murugiah Souppaya and Karen Scarfone. *NIST Guide to Malware Incident Prevention and Handling for Desktops and Laptops*. June 1, 2013. URL: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-83r1.pdf.

[TV16] Luca Tännler and Mathias Vetsch. “Forensik Triage Kit”. Bachelor Thesis. University of Applied Sciences Rapperswil, 2016.