Alternative scalable HIDS with investigation capability

Hosed based intrusion detection system without hashing

Bachelor thesis

In this thesis, I show how a host-based intrusion detection system can be built for scalability. More specifically, I show how the sleuth kit can be used to create an intrusion detection system that does not rely on hashing, but on filesystem attributes. This way, it gains speed, which enables us to take the risk not to calculate hashes.

Degree programme:	BSc in Computer Science
Specialisation:	IT Security
Author:	Julian Stampfli
Tutor:	Dr. Bruce Nikkel
Expert:	Armin Blum
Date:	June 9, 2019

Abstract

Many tools exist to help a company to protect itself against cyber attacks. One type of such a tool is called an intrusion detection system (IDS). Those are created to detect attacks that somehow got through other measures and infected one or many hosts in the network. One type of IDS is called network-based intrusion detection system (NIDS). They operate on a network level and analyze the incoming and outgoing traffic for anomalies. Those anomalies usually signal an intrusion. When they find such an intrusion, they usually generate an alert for a system administrator or security professional to analyze.

There is also another type of IDS called host-based intrusion detection system. They operate directly on the host and try to find attacks there. They are more effective at finding intrusions that are dormant and don't do anything for some time. They mostly operate on the file system and sometimes go beyond that. HIDS have been quite effective at finding intrusions on file basis in the past by creating cryptographic hashes of hashes and comparing them to previous executions. However, as file sizes have been growing, they began to struggle to execute within a short time frame. Calculating a hash is seen as the only reliable way to find changes to the file system, and with more data, it is taking increasingly long to calculate them. The situation has grown out of proportions because the time to scan now takes so long, that the intrusion detection system can't reliably find intrusions within a useful timespan.

In my thesis, I try to show a different solution to this problem. I created a host-based intrusion detection system that works at a file basis but does not calculate any hash. Instead, it finds intrusions by evaluating the file system attributes like modification time and permissions. This approach is risk-based because it is less reliable, but by increasing the speed, the host can be scanned multiple times more often than if hashes get calculated. I am using an open source forensic investigation tool called the sleuth kit (TSK). It offers much functionality for file system analysis and works on most operating systems. With this tool, I can extract the file system attributes reliably and fast, without touching the files themselves.

There is another advantage that I hope to give with my system. Forensic investigators usually struggle to reliably create a timeline of what happened on a file system after an intrusion. This timeline is essential because it can lead them to find out what exactly happened and often can make future intrusions harder. Here I want to help as well. Other than the other HIDS, my system stores all the executions. This way, an investigator can look at this history and sees how the attack started. One nice side-effect of that is that my system is very flexible. After changing something on the host, the system automatically adjusts.

With my tool, system administrators and forensic investigators have another option to tackle intrusion detection. Taking the risk-based approach can lead to many fast detections that otherwise would not be detected in time. Additionally, investigators have more data at their disposal to protect from future attacks.

Acknowledgements

Firstly, I want to tank my tutor Bruce Nikkel and for his outstanding guidance and ongoing support during this project.

Next I want to thank Amir Blum for supervising this work as my expert and for giving valuable feedback at the start of the project.

Further I want to thank the opensource community behind The Sleuth Kit (TSK) and pytsk3 without whom this project would not have been possible. Especially Joachim Metz who is very active in the pytsk3 community and helped me find the solution to one of my issues.

Contents

1. Introduction

For modern businesses it is imperative that their computer systems are secured. This way they can focuss on their busines without worrying about the computer network. To support the securing of their computer system, there are many tools, which can be grouped into different types of families. Those families have multiple different purposes, some are responsible to keep any malicious activity outside of the network while others try to find out whenever something manages to come through. Those type of system are named intrusion detection system (IDS). An IDS generally works by tracking and evaluating some form of activity from hosts and networks and then tries to find anomalies. This is usually done by comparing the current activity to some sort of configuration which usually contains a form of a whitelist and blacklist. Found anomalies are then alerted or logged such that investigators and system administrators can evaluate the anomaly and maybe take action.

There are two flavours of IDS, one, which deals with data from the network, named network based intrusion detection system (NIDS), and another which evaluates data from the host system, named host based intrusion detection system (HIDS). NIDS are used to detect unusual behavior of the network like communication from hosts which usually don't communicate and suspiciously large amounts of data. HIDS, on the other hand, are used to detect anomalies on a host. This process works by detecting changes in the way processes are used or when the file system is changed. Both types of IDS have different advantages, and they should be used in conjunction for best results. [6]

This thesis is about writing of a HIDS that operates on the file system. As already mentioned a HIDS operates on the host machine and detects anomalies by comparing resources available on the host. For detecting changes to the file system a HIDS usually generates cryptographic hash functions and compares them to previous executions. If the cryptographic hash function changes, then the file has been altered. If this alteration is detected as a anomaly, it is alerted. This approach has one weakness. The calculation of a cryptographic hash function takes time. Specifically, to calculate the hash the HIDS needs to read the whole content of the relevant files. Because those files are stored on some storage medium, it can actually take some additional time to read the file from the file system and then calculate the hash [12, 16]. This is from a historical perspective not relevant as it was efficient enough that the entire file system could be hashed in a small amount of time. However, storage media grew, and with it, the amount of data on a server [17]. With that, the calculation of cryptographic hash functions needs more time. So much indeed, that traditional HIDS can't scan big systems within a valuable amount of time anymore.

There is another way though. Instead of reading the entire file and calcualating the hashes for each, the file system already contains some information that could be used to find intrusions. The file system contains a lot of metadata. Things like modification date, permissions and file size. Accessing this type of information is cheap and it can be done without touching any type of content within the file. This data can be used to find anomalies, while not as reliable as hashing, the speed gain can justify sacrificing a little bit of reliability. This risk has to be considered, because finding an intrusion fast can be the difference between an attempted intrusion and an exfiltration of all the busines relevant data. [8]

Another advantage that this HIDS has is the built in support for forensic investigations. Sometimes it is not sufficient to find intrusions, because every IDS can miss some intrusions which usually are investigated. For those investigations to be successful, more data is better because it makes it easier to reconstruct the attack. This way, they can evaluate what has happened and assess the risk. Most HIDS solutions don't offfer much help in this regard. The solution developed within this thesis, however, has this idea of a forensic investigation built in.

1.1. Drawbacks

This implementation doesn't calculate any cryptographic hash functions. This results in some blind spots. The file system metadata can be changed after an infection and without calculating a cryptographic hash function those changes will not be found. [29] This risk needs to be weighed against the gained speed and the benefit of finding intrusions fast. This implementation does not claim to be the ultimate tool which finds all intrusions; rather, it is one piece of the puzzle. Further information about attack scenarios and mitigations are available in section 3.4.3 and section 3.4.4.

2. Technical Background

In this chapter, I provide some technical introduction to relevant topics. Further information is available in the linked resources.

2.1. Definitions

2.1.1. Filesystem

storage media takes many forms. There are the traditional Hard Disk Drive (HDD) and newer Solid State Drive (SSD). Both are built based on different assumptions and using different technologies. There exist more storage media like Compact Disc (CD), Digital Versatile Disc (DVD), Universal Serial Bus (USB) flash drives and more. storage media operates on blocks of data. One such block is typicaly 512 bytes or 4 Kilobyte (KB). [17]

An operating system typically needs to store files of data. This is where file systems come in. They create a layer of abstraction for the storage media. A file system uses the blocks of the storage media and stores the files in a data structure called inode [8]. The file system also keeps track of metadata like creation time, accessed time, permissions, etc. Which metadata is stored depends on the file system. [17]

2.1.2. Cryptographic Hashing Function

A hash function is, in general, a function which takes an input of unspecified size and generates an output of a fixed size. Hash functions are used widely in programming and databases to access certain data within a data structure easily. By design, hash functions have collisions, meaning that multiple inputs generate the same output. This statement must be true if you consider the unlimited input size and limited output size. If there are more inputs than outputs, there must be at least one value that is assigned to multiple inputs. For data storage and other use cases this is not a huge problem because collisions can be handled, and if the hash function has good distribution, collisions are unlikely. Additionally, in many systems, a hashing function with weak collision resistance is deliberately chosen because it is faster to execute. [7, 12]

In a cryptographic context, this tradeoff cannot be taken. Two big factors play into why not. Firstly in cryptography hashes are often used as an assurance that the content of some data has not changed. If collisions are easy to find, the data can be altered in ways that result in the same hash, meaning that the hash no longer fulfills the use case. Additionally, potential gain can be big. If a collision can be provoked, even if challenging, data can be changed. This data could be a legal document or a bank transaction, neither of which we want to change. For those reasons, a cryptographic hash function needs to be highly collision resistant. The drawback of collision resistant algorithms over easier hashing functions are operations needed for the calculation, while current hashing functions are performant and secure, they still take some time for much data. [20]

One cryptographic hash function is called Secure Hashing Algorithm - 256 (SHA-256). SHA-256 is a standard published by the National Institute of Standards and Technology (NIST). It creates a 256-bit output and has not yet been successfully attacked. [15] This hashing algorithm is used to make sure that the configuration of this implementation has not changed between multiple runs.

2.1.3. HIDS

A HIDS works by detecting changes on the local host. It does that by looking at files, processes, configuration, logs, or other indicators. In this thesis, I focus on files. It is important to note that the other sources are a valuable source of information. Any of those sources might have signs of intrusions. Especially running processes and the configuration can hold relevant data. However, this implementation only covers file-based information due to the lack of resources.

In a HIDS that operates on the file system, all the valuable information comes from unexpected changes to the file system. Most hosts do not have any changes on the file system except for software updates. Additionally, the file system usually contains too much information to cover in a handwritten configuration. Thus, the easiest approach is to compare the current state of the file system to previous ones. If it has changed significantly or in an unexpected way, something foul might be happening. The negative side effect of this approach is the false positives that come from legitimately changing the host. Those legitimate changes could be new versions of the webpage that is running, newly updated configuration or updated ssh keypairs. However, those changes come scheduled, and the alerts can then be quickly checked and acknowledged.

A good HIDS should be able to handle such valid changes without much change. Additionally, it should be able to find intrusions reliably and in a timely fashion. Maybe it is too late if the HIDS finds an intrusion a week after it infected the system. To detect changes on the files reliably popular HIDS calculate a hash of the files and compares that hash to previous runs. If a cryptographic hash function is used, each file always generates a unique hash which can not be forged. The main drawback of using good cryptographic hash functions is that they take a longer time. Some implementations thus use weaker implementations of cryptographic hash function or even non-cryptographic hash function. The obvious drawback of this approach is that a collision can be generated and such a file can be altered or replaced without the HIDS noticing.

In the following sections, I present two of the most used HIDS, Tripwire, and Aide. They built the main competition of the integration developed in this thesis.

Tripwire

In 1992 the first HIDS named tripwire was created and publicly released as a free tool. In 1997 the creator of tripwire then created the company Tripwire Inc. and bought the naming rights for tripwire. The free version was monetized, and they released new versions of tripwire. [11, 1] Now they mostly market to enterprise and industrial customer and have more products than only one for file integrity. [23]

As it is a commercial product, it is hard to see how they work in detail since most of the information is only available to paying customer.

AIDE

Aide is an opensource alternative to tripwire. It was created after tripwire went commercial. [14, 4]

When aide is first executed, it generates a reference database. This database contains all the information for each file that is within a path of interest. Depending on the config it contains more or less information, including cryptographic hash functions. Each subsequent execution then compares the files found to this initially created database. The database needs to be updated manually if the system has a scheduled change. It generates a log of all the changes and distributes that per email or similar depending on the configuration. Both the configuration and the database are usually stored on the file system that is scanned. [3, 21]

Aide can compare multiple cryptographic hash functions and non-cryptographic hash functions and some metadata. It runs on many modern Unix system [3] and uses GPG keys to sign their releases. It has a strong community behind it, but only offers a comand line interface (CLI) and has no fancy user interface and is written in C. [4]

2.1.4. NIDS

Compared to a HIDS, a NIDS can be used to detect intrusions over multiple hosts. NIDS are used to analyze anomalies on the network. An anomaly might be an unexpected session from a server which usually doesn't communicate with the internet or unusually large or frequent traffic. Certain intrusions can be tracked by analyzing the content of the packages. Some intrusions are easier to detect on the network. Especially such ones that extract much data or are very aggressive at spreading within a network. The main advantage here is that multiple sources can be combined on the network level. Another advantage this approach has is that current malware almost always contains some network communication [5, 28].

However, a NIDS doesn't only have advantages. Certain kinds of attacks can only be detected on the network with much difficulty. For a corrupted file upload cannot be detected by only checking the network. For best results, both types of IDS can be used in conjunction to detect attacks.

In this thesis NIDS are not in the center. The implementation doesn't look at network traffic or mimic other NIDS functionality. However, as mentioned above, I highly encourage everyone to use an appropriate NIDS to improve the chance to find an intrusion.

2.1.5. The Sleuth Kit

TSK is an opensource toolkit used to investigate disk images. It is based on the coroner's toolkit [22] and contains multiple command line interfaces and an Application Programming Interface (API) for various purposes. [19, 2] It is mainly written in C, runs on Linux, OsX, Windows and more and can be used to analyze many different file systems. It is heavily used in forensic investigations to find deleted or corrupted files and for other information gatherings on a disk image.

fls

One of those command line tools is fls. It can be used to access directories, files, and the attributes of each. With it, the directories can be displayed recursively, and for each file, the attributes can be printed to the console. [9] This tool plays a central part in this thesis as it is used to recover all the files and attributes to detect intrusions. On how exactly it is used, please refer to section. 3.2.1

2.1.6. Python

Python was the programming language of choice for this product. Python offers many advantages over other languages. Some of those are:

- Platform independent

- Good community support in the forensic community
- Active library for TSK
- Small overhead
- Easy for prototyping
- Easy to read

This decision was not made very lightly. Other programming languages were considered. For more information on those refer to section B.1.

pytsk3

Pytsk3 is the library mentioned above that creates bindings for python to the API of TSK. As TSK this library is opensource and is still active. It is hosted on github and offers most of the functionality of the TSK API. Pytsk3 is used extensively in the scanner part of the implementation. For further information refer to section 3.2.1.

2.2. Proposed solution

The main principle behind a HIDS is the detection of changed files. As already discussed in section 2.1.3, most tools rely on the calculation of hashes. This is generally a good approach since changes can be found very reliably; however, as already mentioned, it can drastically hinder the performance of the HIDS. Sadly the actual performance lost cannot be clearly stated, as it heavily depends on what hardware is in use. However, considering the computational overhead of calculating a cryptographic hash function it is clear that the time it takes grows with bigger file systems. As storage media has grown from a Megabyte (MB) to Terrabyte (TB) so has the requirement to store more data. Creating hashes over such big systems is not viable as it can take a long time to create a hash of big amounts of data. [12, 10, 16]

2.2.1. Time issues

In this thesis, I propose a different approach. Ignore the hashing and the file content and focus on the file system metadata instead. This approach should be sufficient to find intrusions and thanks to not creating hashes, the HIDS can execute more often. If a conventional HIDS might take several days to complete a run on a file system with multiple TB of data, the proposed approach would take some minutes to hours. It could then be run multiple times a day and find new and changed files within a short amount of time. It is possible that the HIDS misses some changes if the attacker can change all the relevant metadata before the system rechecks the same file, but this is a risk that has to be taken to gain the opportunity to scan large file system. It is also possible to scan highly critical sections of a system with a traditional HIDS and the rest with the proposed solution. This way, one has both advantages. The whole system can be checked in a timely fashion with the proposed solution, and for certain smaller parts of the system, a general HIDS can detect changes by using strong cryptographic hash functions.

This solution uses TSK via pytsk3 to extract the file system metadata. The main advantage that this gives is the interoperability with different file systems. Additionally, by directly accessing the attributes from the file system, no metadata is changed. The files themselves are never touched. Furthermore, it can also be used to get the files and attributes of an image that has been extracted or of a virtual machine.

2.2.2. Investigation capabilities

Another improvement upon existing HIDS proposed is that investigation capabilities need to be built in from the start. Traditional HIDS are good at finding intrusions and alerting them. However, they don't offer support for the investigation of the incident. They don't have information beyond what was configured. This lack of information makes sense is dangerous cryptographic hash function. If something was missed in the configuration, an investigator can't use the output of those systems to gain further knowledge. They only have the alerts and nothing further.

This solution stores all the available metadata for each scanned file for each run. This way, an investigator can use this output to gain more information about how the attacker proceeded. He can look at the changes of permissions, modification of files, even if the alerts might have been ignored. It is also possible to generate a timeline out of this data to form an extended view on what happened when on the file system.

2.2.3. Flexibility

Aide works by comparing runs to one first execution. This is practical as it detects one intrusion multiple times. However, it generates many messages, and users are then less likely to take them seriously. Additionally to that, after a legitimate change to the system, aide always generates alerts until the initial run is reset. This can lead to undetected intrusions a short time after an upgrade, which might also be the most critical window for an intrusion because the update might have created a security risk. Thus, it is possible that an intruder can gain access shortly after an update which is alerted by aide, but ignored because of the false positives.

The proposed system does not have this issue, at least not as strongly. As all the data gets collected for each run, it makes sense to compare each run to the previous. This has the benefit of legitimate changes being adopted into the accepted one run after it has been finished. This results in overall fewer messages, which increases the importance of each. If it is required that the system should always compare against one specific run, this could also clearly be done. The system would need to be configured to not check against the latest, but a specific, predefined run. Only the configuration would then need to be changed to update this predefined run.

2.3. Scope

In this project I create a HIDS which uses TSK to detect changes. It covers the three main changes discussed in section 2.2. This service uses a SQL database to store the executed runs. It includes user documentation and this thesis documentation.

Out of scope is the creation of the timeline mentioned in section 2.2.2. Also, out of scope are extensive alerting functionality and extensive example configurations for commonly used operating systems and tools. Furthermore, any big data analysis of the runs, while very interesting, are also out of scope.

3. Results

In this chapter I will talk more about the implementation details of the HIDS. I will also look into the security of this implementation and what measures can be implemented to improve it.

3.1. System Architecture

As seen in figure 3.1 the system itself contains of two components, the scanner and the investigator component. It is split this way to gain flexibility, specifically, that both components can be executed independantly. This has the advantage that the scanner and the investigator don't need to be run on the same system. Also it means that previous results of the investigator can be recreated by running the investigator on the previously captured data. Additionally, if someone is only interested in the results of the scanner, the investigator doesn't need to be run.

Besides the scanner and the investigator which were produced in this thesis, there are four other components which build the environment of the HIDS. There is the file system which contains the data in which we are interested. As already mentioned, this could be a mounted device that is directly accessible through the operating system. It could also be run on the disk images of virtual machines or on previously created disk images. To run it on a previously created disk image might be interesting to see a history if an image was taken at multiple different times. Also it could be used on backup images, if backups are made in such a way that they create a disk image.

Then there is the forensic component. This component is the combination of TSK and pytsk3. The main purpose of this component is the abstraction of the file system into python classes which can then be used within the Scanner. This abstraction is important to be compatible to multiple different file systems. TSK offers a lot of different functionality, in this thesis I am only using the utility that is also used for the fls command described in section 2.1.5.

The scanner and investigator are explained in detail in section 3.2 and 3.2.2, the database in section 3.1.2 which leaves the alerting component. This component will be part of the section 3.2.2. One component that is not on the diagram is the configuration. This component is special because it influences every other component and at the same time is relatively simple. The configuration is covered in section 3.1.1.

3.1.1. Configuration

The configuration file is provided in YAML. YAML is a human friendly data serialization language used in many programming languages. Its main advantage over other languages for configuration files is the readability and the ease to extend already existing configuration files. There were more reasons why YAML is used documented in section B.2.

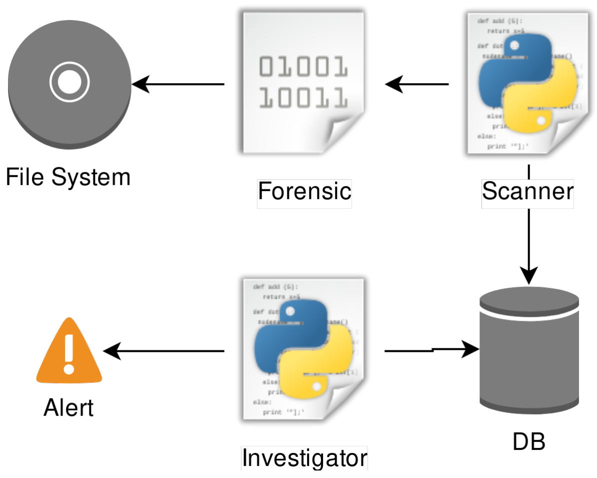

Figure 3.1.: System Architecture

Database Configuration

The configuration consists of three parts. The first part is the database configuration. Options are to either use the system with a sqlite database file, which is the most simple solution, or to use it with a postgres database server. For the sqlite configuration only the filename is required as seen in listing 3.1. For a postgres server there needs to be more configuration. The host and port need to be provided, as well as the database, the user and a password as evident in listing 3.2. This configuration defines where the data will be sent to and read from in the scanner and investigator parts respectively. As both parts need access to the database this part of the configuration is in a seperate part. It is possible to extend the system to use more DataBase Management System (DBMS), it is not part of the scope of this thesis though.

Listing 3.1: SQLite Configuration

```
sqlite:
        filename: fids_db3.db
```

Listing 3.2: Postgres Configuration

```
sqlite:
        filename: fids_db3.db
```

Scanner Configuration

The second part of the configuration is the part that defines the scanner. An example config can be seen in 3.3 It consists of one main key which is named scan. If this config entry is missing, the scanner part of the HIDS will not be executed by default. Then it contains the following subkeys:

- **image_path** Path to the file system that is used for the scan.
- **scan_paths** List of paths to scan. Paths will be scanned recursively. "/" will thus scan all the paths.

- **ignore_paths** Paths that should not be scanned. Can be a subdirectory of any path in scan paths. The recursion will stop at this path and not continue downwards. Practical if certain directories are not interesting for intrusion detection.

Listing 3.3: Scanner Configuration

```
scan:
        image_path: /dev/nvme0n1p5
        scan_paths:
                [
                        "/",
                        "/nonExisting",
                ]
        ignore_paths:
                [
                        "/temp/"
                ]
```

Investigator Configuration

The investigator configuration is simmilar to the scanner configuration in the way that it contains a top level node called investigator. If this is missing the investigator part will not start. As can be seen in listing 3.4 it is actually a lot more complex than the scanner config.

- **same_config** This configuration specifies whether a changed config should result in an alert. Defaults to True.
- **validation_run** This can define a run that is used for validation. If empty the second last one will automatically be used.
- **Rules** Rules are ways to create templates on what should be used to compare the files. Rules are explained more through below.
- **Investigations** Investigations define which paths and which files should be checked for intrusions. Investigations are explained more through below.

Listing 3.4: Investigator Configuration

```
investigator:
        same_config: True
        validation_run:
        rules:
                - name: php
                  rules:
                        - m
                        - i
                        - l
                        - n
                        - a
                  equal:
```

```
                            - meta_creation_time
                            - meta_size
                    - name: logs
                      greater:
                            - meta_size
                      equal:
                            - meta_creation_time
        investigations:
                    - paths:
                            - '/etc'
                      fileregexwhitelist:
                            - '*.php'
                      rules:
                            - php
                    - fileregexwhitelist: '/*'
                      fileregexblacklist: '*evilfile*'
                      rules:
                            - logs
```

Rules

Rules represent templates that are later used in the investigations to find anomalies. Rules can be based upon other rules to extend them. Recursive rules are allowed. They simply extend each other. An example of the rule configuration can be seen in listing 3.4, the fields are explained below.

- **paths** For investigations different paths can be defined. This changes the behaviour in such a way that only the specified path will be investigated. Usefull if certain paths need special observations.
- **rules** The rules which are used within this investigation is based on are defined here. They are referenced by name.
- **fileregexblacklist** All files which are found may not include this regex. If their path with filename results in a match with this regex it will be alerted.
- **fileregexwhitelist** The whitelist has a different behaviour than the blacklist. It will work simmilarly to the paths config. Only files will be analyzed that match this regex.

Additionally to the configurable rules there are some predefined rules. Those were heavily influenced by the aide configuration. The ruleswhich are preconfigured can be seen in listing 3.5

Listing 3.5: Investigator Configuration

```
self.rules.append(RuleConfig(name='p', equal=['meta_mode']))
self.rules.append(RuleConfig(name='ftype', equal=['meta_conten']))
self.rules.append(RuleConfig(name='i', equal=['meta_addr']))
self.rules.append(RuleConfig(name='l', equal=['meta_link']))
self.rules.append(RuleConfig(name='n', equal=['meta_nlink']))
self.rules.append(RuleConfig(name='u', equal=['']))
self.rules.append(RuleConfig(name='g', equal=['meta_gid']))
self.rules.append(RuleConfig(name='s', equal=['meta_size']))
self.rules.append(RuleConfig(name='b', equal=['']))
self.rules.append(RuleConfig(
        name='m',
        equal=['meta_modification_time',
                'meta_modification_time_nano']))
self.rules.append(RuleConfig(
        name='a',
        equal=['meta_access_time',
                'meta_access_time_nano']))
self.rules.append(RuleConfig(
        name='c',
        equal=['meta_changed_time',
                'meta_changed_time_nano']))
self.rules.append(RuleConfig(name='S', greater=['meta_size']))
self.rules.append(RuleConfig(name='empty'))
```

Investigations

The investigation configuration defines the objects which are scanned for intrusion, as well as how they are scanned. They contain rules and some configuration on what should be scanned. An example of the investigation configuration can be seen in listing 3.4, the fields are explained below.

- **name** Each rule has a name. If two rules with the same name are defined the one defined below will be used but without guarantee. The name is used to extend rules.
- **rules** The rules which are used within this rule are defined here. They are referenced by name.
- **greater** Those are the properties which are allowed to grow between the runs. Greater also includes equal.
- **equal** Those are the properties that are suposed to stay equal during all runs.

3.1.2. Database

The database is another component that is shared between the scanner and the investigator. As already mentioned, multiple DBMS can be used in conjunction with this HIDS. In this section I will not focus on the DBMS used. The database consists of five relations. Firstly, I will explain some reoccuring types, how they are stored and what they represent after that, the relations and then quickly how the different DBMS can be used.

Some ideas are not specific to a relation. Most relations contain timestamps and Universaly Unique IDentifier (UUID). How they are stored can be seen below:

Timestamps To save space, Timestamps are stored as an integer value. This value represents the UNIX timestamp.

UUID IDentifier (ID) are stored as UUID in their hexadecimal representation.

Enum TSK defines many enums. Most of those enums have an int representation. To save space, this representation is stored in the database. The enums can be viewed in the TSK API reference. [24]

FIDS_RUN

The run table is relatively simple. It contains a ID so that each run can be identified. This ID will be used in the other relations as well to create a link to the run. Besides that it contains a SHA-256 hash of the configuration that it was started with. It then contains a start and and end time of this particular run. The table definition is shown in listing 3.6

Listing 3.6: Fids Run Table Definition

```sql
CREATE TABLE FIDS_RUN(
        id varchar(32),
        config_hash varchar(64),
        start_time int,
        finish_time int,
        PRIMARY KEY(id)
);
```

FIDS_ERROR

With each run there is the possibility of errors. Those errors are stored in this table. It is rather simple as well. Additionally to the run ID it contains an ID as well. Next to that it contains a description and a location of where it occured. The table definition is shown in listing 3.7

Listing 3.7: Fids Error Table Definition

```sql
CREATE TABLE FIDS_ERROR(
        run_id varchar(32),
        id varchar(32),
        description text,
        location varchar(255),
        PRIMARY KEY(run_id, id)
);
```

FIDS_FILE

This table contains most of the information. It again has ID additional to the ID of the run. Then it has the path in which the file was located and all the meta information about the file. Most attributes start with meta, while the rest start with name. This is a reference to TSK which has multiple structs for meta and name information about each file. I kept the naming of TSK, thus more information about each attribute can be found on the API reference of TSK. There are also two indeices. One on the inode address and one on the combination of path and file. Those are made so that the investigator can be executed faster. [27] The table definition is shown in listing 3.8.

Listing 3.8: Fids File Table Definition

```sql
CREATE TABLE FIDS_FILE(
        run_id varchar(32),
        id varchar(32),
        path text,
        meta_addr int,
        meta_access_time int,
        meta_access_time_nano int,
        meta_attr_state int,
        meta_content_len int,
        meta_content_ptr int,
        meta_creation_time int,
        meta_changed_time int,
        meta_creation_time_nano int,
        meta_changed_time_nano int,
        meta_flags int,
        meta_gid int,
        meta_link int,
        meta_mode int,
        meta_modification_time int,
        meta_modification_time_nano int,
        meta_nlink int,
        meta_seq int,
        meta_size int,
        meta_tag int,
        meta_type varchar(255),
        meta_uid int,
        name_flags int,
        name_meta_addr int,
        name_meta_seq int,
        name_name int,
        name_size int,
        name_par_addr int,
        name_par_seq int,
        name_short_name int,
        name_short_name_size int,
        name_tag int,
        name_type varchar(255),
        PRIMARY KEY (run_id, id)
);
CREATE INDEX inode ON FIDS_FILE(meta_addr);
CREATE INDEX fullpath ON FIDS_FILE(path, name_name);
```

FIDS_FILE_ATTRIBUTE

In TSK each file can contain multiple attributes. Those attributes are stored in this table. It contains the previous ID and another one for each attribute as each file can have multiple attributes. The attributes contain flags, a name and a type. The flags and the type enums called 'TSK_FS_ATTR_FLAG_ENUM' and ‘TSK_FS_ATTR_TYPE_ENUM’. More information available in the TSK API reference. [25, 24] The table definition is shown in listing 3.9

Listing 3.9: Fids File Attribute Table Definition

```sql
CREATE TABLE FIDS_FILE_ATTRIBUTE(
        run_id varchar(32),
        file_id varchar(32),
        id varchar(32),
        flags int,
        tsk_id int,
        name varchar(255),
        name_size int,
        at_type varchar(255),
        PRIMARY KEY (run_id, file_id, id)
);
```

FIDS_FILE_ATTRIBUTE_RUN

Attributes can contain multiple data runs. Those data runs are represented in this relation. It contains all the ID from the attributes and an additional for the run. It then has a block address and a lenght. More information can again be found in the TSK API reference. [26] The table definition is shown in listing 3.10

Listing 3.10: Fids File Attribute Run Table Definition

```sql
CREATE TABLE FIDS_FILE_ATTRIBUTE_RUN(
        run_id varchar(32),
        file_id varchar(32),
        attribute_id varchar(32),
        id varchar(32),
        block_addr int,
        length int,
        PRIMARY KEY(run_id, file_id, attribute_id, id)
);
```

3.1.3. Shared

There are other shared components. Mainly the model of the system. It contains of multiple classes for the types. There is one class for each relation. They can parse TSK objects and they can parse database rows. This way both the scanner and the investigator can work with the same classes. Except for parsing they don't have any functionality.

3.2. Application Flow

The main application flow consists of first reading the config file. If no config file path is passed the default path is used. It then starts the scanner if a scan configuration is found. After the scanner is completed it starts the investigator if the detection config is found. It works if any of the two configs are found. If none is found then the system will simply do nothing.

3.2.1. Scanner

The scanner component is the first specific component. It is executed when the system is run and a scan config is available. It is responsible for getting all the information from the file system for the configured paths. It has multiple stages.

1. Initialization
2. Scan
3. Error Logging
4. Storing
5. Error Logging
6. Finalizing

In the initialization phase the scanner creates a run object. This object is already saved to the database. This way user know just by looking at the database if there are any runs still running. This also creates a first opportunity to look for inconsistencies. If there are a lot of started runs, something suspicious might be going on. It also creates the hash of the configuration and already saves it to the database.

The scan phase has more steps to it. First it creates the pytsk3 object called 'img_info'. This is done by passing the path to the disk image to pytsk3. Then the actually important 'fs_info' object is created by passing the img info object. This way we have access to the file system. The scan actually starts by calling 'open_directory_rec' for each scan path. This function keeps track of all diretories it already traversed into as to avoid circular loops and unnecessary steps. Then it checks if the path is in the ignored paths. It then iterates over all objects in the directory.

For all entries it checks if it is a valid entry with the required attributes to continue. Afterwards it checks if it is a directory. For all the directories the function first checks if the directory has already been visited and if not it calls itself with the new directory as the parameter. If the entry is a file, it is parsed into a python object and then stored into a local list of found files. Any errors that occur are saved by creating an error object that is stored into a list of errors.

In the error logging phase all errors that have occured in the scan process are stored in the database. This is done by iterating over all the errors and saving them one by one. The database is then commited and even if the files can't be saved for some reason, the errors will be persisted.

After the first error logging phase is the file saving phase. Here the files are saved again by iterating over them and saving one by one. Should any additional error occur while saving the files, they are again added to a list of errors.

There is a second error logging phase. In this phase the errors that occured while saving the files get stored into the database. The functionality is the same as for the first error logging phase.

In the finalizing phase the run object gets updated and the endtime added. It is then also updated on the database. At this point the scanner is finished. It has written all the files from the file system to the database. Also all errors that occured during this process are logged. The run object on the database will have a valid start and end timestamp.

3.2.2. Investigator

The investigator component is responsible for finding intrusions. As discussed in section 3.1.1 it does so with predefined rules and investigations. To best explain how the investigator works it is best to split it into different steps as with the scanner.

1. Parsing configuration
2. Fetching runs
3. Fetching files
4. Parsing top level information
5. Iterate through the files
6. Create alert

First the investigator needs to parse the configuration. This is more work than the other configuration parts of the system, because of the alerts. One investigation path has none or many alerts, which themselves can have multiple alerts again. To break this structure into a less complex structure the first step is to flatten it out. Simply, by eliminating the nesting of alerts and creating one list of alerts this can be done. Then to make it even simpler, the configurations in each alert are taken out so that the investigation has one list for equal and one for greater. Those lists are then pruned by removing duplicates and by making sure that each database column only appears at most once.

Afterwards, the run ids need to be fetched. By default all the runs are fetched and the most recent is compared to the second most recent. This can be changed if the validation run configuration is set. Then this will be taken as comparison for the most recent run.

After the runs are aquired, the investigator gets the files. This is done mostly with a simple sql query as shown in listing 3.11. Which joins the file table with itself on the inode address or path and name. This way the system can make good use of the indices that were created on this table. After the files get fetched, they are parsed and put into a list of tuples with the matching files. This process takes some time for a long list of files.

Listing 3.11: SQL Querry for files

```sql
SELECT * FROM FIDS_FILE first
        LEFT JOIN FIDS_FILE second
                on (first.meta_addr=second.meta_addr or
                        first.path=second.path and
                        first.name_name=second.name_name)
        WHERE first.run_id = ? and second.run_id=?
                or second.run_id is null;
```

After the files have been parsed the actual investigator starts executing. The first thing that gets checked is the configuration hash. This can be deactivated, but it increases the security by making sure that the configuration file has not changed since the last run.

Then each file gets checked for each rule. This is done by using the previously generated greater and equals list. Each attribute in either list is checked for the appropriate usecase. Should something not be correct an alert is created. Those alerts can then in the end be used to generate a report.

Finally, the alerts are used to generate a report. Currently this report is only written to stdout. This means that no real alerts are generated. Those can be generated though by using the output of this system as an input for another tool.

3.3. Examples

In this section I will give some general usecases and the configuration that would be used. This is to make it easier to understand the HIDS and how it can be used. I will mainly use the apache webserver as an example. There will be some assumptions on the directories and contents for both scenarios. Additionally I assume that the file system is accessible through '/dev/sda1'

For additional help there is also a user documentation in apendix section D

3.3.1. Webpage hosted in Apache

For those examples I will take the assumption that the apache webserver is installed and has a htdocs directory that is located in '/var/www/'. I assume that there is a webpage running with some php files. It also has html, css, javascript and image files in a static folder. Then it has a log directory where the webpage itself stores all the logs and an upload directory where the user can upload all kind of files. There are 4 types of files that are watched, some more and some less closely. Example Configurations will be given for each type. For the scanner config the config in listing 3.12 can be used.

Listing 3.12: Scanner Configuration

```

sqlite:
    filename: apache.db

scan:
    image_path: /dev/sda1
    scan_paths:
        [
            "/var/www/htdocs/",
        ]
```

PHP files

The webserver has some php files in different directories. Those are used for varios purposes, but they should never change, except for an update to the webpage. All of them lie in some subdirectory of the htdocs directory. Thus they all are covered in the scan from the configuration in listing 3.12.

The tricky part is, that we have multiple different php files in different directories. The configuration should cover that even if the exact location and folder structure changes a bit and new folder are added. As the scanner automatically scans recursively, the exact folder structure does not really matter. The files will be found when they are in the htdocs directory. As the files should not change but might be accessed at any time, watching everything should be what we want. Especially important are the modified, changed and created times. Also important are size, path, name and inode.

This much for the rule, the investigation is then relatively easy. There is not a special path that has to be watched since the scanner was configured only to scan htdocs. By using the same database, the investigator will only validate this data as well. Important though is the whitelist. There is an upload directory in which we expect changes. Thus a file name regex of '.php$' will only validate php files.

An example investigator configuration is listed in listing 3.13. For this configuration I did not use the preconfigured rules.

Listing 3.13: Scanner Configuration

```
sqlite:
    filename: apache.db

investigator:
    same_config: True
    rules:
        - name: equal_but_accessible
        equal:
            - meta_creation_time
            - meta_creation_time_nano
            - meta_modification_time
            - meta_modification_time_nano
            - meta_changed_time
            - meta_changed_time_nano
            - meta_size
            - path
            - meta_addr
            - name_name
        greater:
            - meta_access_time
        investigations:
            - fileregexwhitelist: '.php$'
              rules:
                - equal_but_accessible
```

Upload directory

As already mentioned, there is an upload directory. This is more tricky, since there files might get upladed and deleted all the time. What we do care about is that there are no php files uploaded. This way an intruder could create a reverse shell in the browser. That would be very bad and we definitely want to detect that with every scan even. It would be cought with the rule from listing 3.13 because this scanns for all php files. However, as an exercise and even just to ensure it is actually caught there is another rule for the upload directory.

For the rule we definitely need to add the special rules 'new_files_ok' and 'deleted_files_ok' since both is actually ok. Then generally greater access and modification times are ok. For the investigation, a file regex blacklist will be added. Additionally only the upload directory needs to be searched. With that php files can easily be excluded. The example configuration is shown in listing 3.14

Listing 3.14: Scanner Configuration

```
    sqlite:
        filename: apache.db

```

```
investigator:
    same_config: True
    rules:
        - name: upload
          greater:
            - meta_modification_time
            - meta_changed_time
            - meta_access_time
        investigations:
            - fileregexblacklist: '.php$'
              paths: [
                "/var/www/htdocs/upload/"
              ]
              rules:
                - upload
                - new_files_ok
                - deleted_files_ok
```

HTML, CSS, Javascript and images

The webpage also has some static context, namely HTML, CSS, Javascript and image files. Those are in a directory called 'static'. Those files are like the php files, they should not change unless someone creates an update to the page. This is rare. In this static folder only those files are acceptible, in the configuration this results in a blacklist for those file types which is negated. This way files that don't equal these file types are alerted. Additionally the previously used rule named 'equal_but_accessible' should do the trick to find modified files. An example configuration is shown in listing 3.15

Listing 3.15: Scanner Configuration

```
sqlite:
    filename: apache.db

investigator:
    same_config: True
    rules:
        - name: equal_but_accessible
          equal:
            - meta_creation_time
            - meta_creation_time_nano
            - meta_modification_time
            - meta_modification_time_nano
            - meta_changed_time
            - meta_changed_time_nano
            - meta_size
            - path
            - meta_addr
            - name_name
          greater:
            - meta_access_time
        investigations:
            - fileregexblacklist: '.(js|css|png|ts|jp(e)?g|htm(l)?)$'
              blacklist_inverted: true
              paths: [
                "/var/www/htdocs/static/"
              ]
              rules:
                - equal_but_accessible
```

Log directory

The last directory that is not yet checked is the log directory. Log files usually grow in size until they reach a certain limit. Then they stay static and a new file is created, which then again grows. To create a configuration for that it is important to check the size for growing. Additionally, all the timestamps should also only grow. The file ending could also be checked, if wished this can be copied from the other configurations. One important detail is the 'file_rename_ok' rule as the rolled over log files might be changed. Also the 'new_files_ok' rule needs to be added. The example configuration is available in listing 3.16

Listing 3.16: Scanner Configuration

```
sqlite:
    filename: apache.db

investigator:
    same_config: True
    rules:
        - name: logs
          equal:
            - meta_creation_time
            - path
            - meta_addr
          greater:
            - meta_creation_time_nano
            - meta_modification_time
            - meta_changed_time
            - meta_size
            - meta_access_time
    investigations:
        - paths: [
            "/var/www/htdocs/logs/"
          ]
          rules:
            - logs
            - file_rename_ok
            - new_files_ok
```

Complete configuration

The partial configurations can then be combined into one bigger configuration. Some rules can be reused, which makes everything a bit easier. The complete configuration is shown in listing 3.17.

Listing 3.17: Scanner Configuration

```
sqlite:
    filename: apache.db

scan:
    image_path: /dev/sda1
    scan_paths:
    [
        "/var/www/htdocs/",
    ]

investigator:
    same_config: True
    rules:
        - name: upload
          greater:
            - meta_modification_time
            - meta_changed_time
            - meta_access_time
        - name: equal_but_accessible
          equal:
            - meta_creation_time
            - meta_creation_time_nano
            - meta_modification_time
            - meta_modification_time_nano
            - meta_changed_time
            - meta_changed_time_nano
            - meta_size
            - path
            - meta_addr
            - name_name
          greater:
            - meta_access_time
        - name: logs
          equal:
            - meta_creation_time
            - path
            - meta_addr
          greater:
            - meta_creation_time_nano
            - meta_modification_time
            - meta_changed_time
            - meta_size
            - meta_access_time
        investigations:
            - fileregexwhitelist: '.php$'
              rules:
                - equal_but_accessible
            - fileregexblacklist: '.php$'
              paths: [
                "/var/www/htdocs/upload/"
              ]
              rules:
                - upload
                - new_files_ok
                - deleted_files_ok
            - fileregexblacklist: '.(js|css|png|ts|jp(e)?g|htm(l)?)$'
              blacklist_inverted: true
              paths: [
                "/var/www/htdocs/static/"
              ]
              rules:
                - equal_but_accessible
            - paths: [
                "/var/www/htdocs/logs/"
              ]
              rules:
                - logs
                - file_rename_ok
                - new_files_ok
```

3.3.2. SSH private keypairs

Another example in which the HIDS can be used is the tracking of private and public keypairs. Usually they are read when a webserver is started. Afterwards it is cached in the memory of the webserver and the actual files should not be read again. This makes an updated access time a possible intrusion. For this scenario I assume that the keypair is located in the directory '/var/www/.ssh/'. An example configuration is shown in listing 3.18.

Listing 3.18: Scanner Configuration

```
    sqlite:
        filename: ssh.db

    scan:
        image_path: /dev/sda1
        scan_paths:
        [
            "/var/www/.ssh/",
        ]

    investigator:
        same_config: True
        rules:
            - name: equal_not_accessible
              equal:
                  - meta_creation_time
                  - meta_creation_time_nano
                  - meta_modification_time
                  - meta_modification_time_nano
                  - meta_changed_time
                  - meta_changed_time_nano
                  - meta_access_time
                  - meta_access_time_nano
                  - meta_size
                  - path
                  - meta_addr
                  - name_name
        investigations:
            - rules:
                  - equal_not_accessible
```

3.4. Security

A HIDS is a system that tries to improve the security of a host by detecting intrusions. Thus, it is extremely important to evaluate the security of such a system. This is the goal of this section. First, I will show some limitations of the produced system. It has some downsides and it is important to look at them. After that I will loook at some risk associated with running the HIDS. Those risks might be associated with all competitors. Then I will give some attack scenarios. Those will include some scenarios where the proposed HIDS is better suited than previous HIDS, but also some that specifically attack the limitations and risks. Lastly, I will dive into how to circumvent those attack scenarios. Some might be defeated if we change some small things, some will need more work.

3.4.1. Limitations

There are multiple limitations that I will list below. I will reference those in the attack scenarios and mittigations. I will also list how easily they can be exploited and how easy it is to detect those limitations.

Changing Attributes

As already mentioned multiple times, the proposed system doesn't use cryptographic hash functions for detection of intrusions. This is a big issue as it means that changed files might not be found. Attributes can be changed which could lead this implementation to miss an intrusion. [13, 29]

It is rather easy to change the file system attributes. However, it is not trivial to change all of them. Some will leave traces and for some attributes root access is needed. Additionally the attacker would first need remember the times which would be correct such that he can correctly reset the times. This can be quite challenging. However, it is likely that the attacker will not change all attributes or that he sets them incorrectly, such that the system can still discover the changes. Some changes to the attributes will leave other traces, such as kernel level logs. Those can be evaluated to find this kind of intrusion.

Non file based intrusion

Another limitation are intrusions that are not file based, which are getting more popular even as advanced persistant threat (APT). Many of the biggest threats for web applications according to Open Web Application Security Project (OWASP) are not creating any files on the host. [18] There are also other attacks which don't need to use files if they can hijack an already started process. Those attacks are not detectable because the system only works on file basis.

Those kind of intrusions are getting more common and as already said, can't be discovered by only checking the file system. However, they leave other traces. They often leave logs in the application because for most of the OWASP top ten, many trial and error is required. For APT it is more difficult to find them. They mostly leave traces as well in form of weird network behaviour. Because to achieve persistance they need to redownload the malware after a host has been rebooted.

Intrusion in non watched location

When the scanner is started it is fed with which paths to scan and which to ignore. If an intrusion injects a file in a place that is not watched, it will not be found. This limitation comes from a faulty configuration.

For non watched locations it is as if the host would not run a HIDS. Maybe the intrusion will at some point write or read something in a watched location, but otherwise the threat can go unnoticed for a long time. To avoid such intrusions it is recommended that at least the scanner part of the HIDS on the whole system. Additionally it is recommended to run the investigator on the whole system as well, but maybe with a lower frequency.

Preexisting intrusions

The system can only detect intrusion by unexpected changes. If it is started on an already running host, it is possible that this host is already infected. This infection can not be seen by the system, as long as the files from the intrusion stay consistent.

Preexisting intrusions are hard to detect if they don't alter their files. Luckily malware behaves like a normal software project. At some point it will probably change its executable. It might also read or write other files that seem suspicious. Additionally, if the HIDS is deployed with new machines additionally to the already running machines, those are going to detect the intrusion as soon as the already infected hosts try to infect the newly deployed ones.

Shut down HIDS

The system relies on the running of the scanner. On a host this would probably be done by creating a scheduled task. If this can be disabled, then the scanner will no longer run and no further intrusions will be detected. Additionally the investigator could also be stopped. This would lead to the same issue.

Should the scanner be shut down then the investigator will no longer find any intrusions. This can be detected if the database is checked on new entries from another source. This is easiest done when an external database is used or when the sqlite file is copied from the host to an external machine at some points. Also, if logs are configured to be written after each execution, it can be suspicious if those entries are missing or are showing to be evaluating the same runs over and over again.

This limitation is not highly likely, because the intruder first needs to get administrative access to the host to be able to shut down the HIDS. This is not always easy to achieve and once the attacker has administrative access he can do whatever he want on the system.

3.4.2. Risks

Aside from the limitations there are some risks with running this system. As with the limitations, I will give some comment to each risk and talk about how to evaluate them.

Running unknown code

As for any program, it is always a risk associated to execute code that was downloaded from the internet. It is possible that it was modified or that it is exploitable. However, this is an opensource program. This makes it easier to evaluate the code. The same would need to be done for all the dependencies, more specifically for pytsk3 and TSK. This is possible but requires a lot of work. However, the opensource community is generally well trusted, which might be enough. To protect from alterations to the tool that was downloaded, hashes can be validated. Additionally, the system behaves very straightforward. It will scan the file system and then write to a database. It will then read from the database and create alerts. This functionality can be monitored. Should the system do something else it might either have a bug or is being abused.

File access

Any HIDS that operates on file basis needs access to the files. Depending on the data that is stored on the host this might be a cause for concern. The system has full access on any of those files. This can not be circumvented. What can be done is using only the scanner on this system. Let it write either to a centrally managed database or a sqlite file. Then let the investigator run on the data from another host. This way the system can be monitored to make sure it doesn't open any other communication.

Root access

The system needs access to the disk image which is mounted. To gain access to that it needs to run with administration privileges. With those permissions the system could theoreically do anything on the host. The best way would be to limit the amount of time that it is executing with those privileges. To do that the scanner and the investigator can be executed in different processes where only the scanner has administrative access. The investigator only needs to read the database and send alerts, for neither it needs administrative privileges.

3.4.3. Attack Scenarios

In this section I describe several attack scenarios. I will only explain what the intruder does and how he does it. I might give some description of the host and which processes are running. I will also list which limitations or which risks the attacker uses. In section 3.4.4 I explain how those attacks can be avoided or detected and what is needed for the detection. Finally in the discussion I give a short summary on how my system stands against other systems like Aide.

Classic Intrusion with persistance

The classic way to gain a foothold on a host is to write a file on the file system. This file is then executed remotely and the host is compromised. How exactly the attacker was able to upload a file is not relevant for this scenario. The attacker will not do anything special with the file and leave it where it lands. He does not change any attributes or uses any other tactic to hide.

This attack actually exploits none of the risks and limitations as this is the basic kind of attack that the system is made to find.

Intrusion with persistance, attacker changes all metadata

The attacker is able to change a file or create a file on the host and then resets the attribes in such a way that the system can't find anything wrong.

Risks and Limitations:

- Changing Attributes
- Intrusion in non watched location

Intrusion without persistance to use as intermediate host

The attacker is able to exploit a running process. He uses the host to gain access to other machines in the network and does not read or write anything from or to the host.

Risks and Limitations:

- Non file based intrusion

Intrusion without persistance to exfiltrate data

As in scenario named 'Intrusion without persistance to use as intermediate host' the attacker gains access through an exploit of an already running process. However, in this scenario he does read files of interest, for instance private keys for encryption.

Risks and Limitations:

- Partially: Non file based intrusion

Intrusion without persistance but as apt

This scenario is simmilar as the previous two. Here the attacker is again able to exploit a process that is already running. He then goes on to use this process to do things that are uncharacteristic for this process. He does neither read nor write files from the host. His goal is to stay hidden for as long as possible. Additionally he wants to reinfect the host whenever the process is restarted.

Risks and Limitations:

- Non file based intrusion

Attacker exploits HIDS to gain administrative privileges

Differently from before, this time the attacker somehow can exploit the HIDS to gain administrative privileges. The most likely way is to give it a different configuration or change the code that is already running on the system. It needs to be either because the scanner subsystem does not have any additional entrypoints.

Risks and Limitations:

- Non file based intrusion
- Root access

- Running unknown code

Attacker changes the code maliciously

The attacker is able to change the code that the user downloads. He injects his own malicious code in the version that was downloaded. It is then executed ant the attacker is able to take over the host.

Risks and Limitations:

- Preexisting intrusions
- Root access
- Running unknown code

3.4.4. Attack Mittigations

To avoid the attacks described in Attack Scenarios there are multiple things one can do. First I give some information how the specific scenarios can be detected if possible. Then there are some general recommendations on what can be done to increase the security of the system or of the host.

Classic Intrusion with persistance

This attack scenario is actually the easiest. It will be detected if the system is configured correctly. Compared to other HIDS, this kind of attack will be found faster, because the system can be run more often.

Intrusion with persistance, attacker changes all metadata

This kind of attack can only be found, if the attacker takes longer than the system has to scan. It is possible that the HIDS discovers it while the attacker has not yet changed all the attributes. Additionally, it is possible that the attacker did not change all the relevant attributes. In both cases the system will find the intrusion and alert it. Should the attacker be fast enough the system will not find the changed file. Then it depends on how well the attacker hides. If he does not do anything on the file system, it is possible that this kind of intrusion will go unnoticed for a long time by the HIDS.

Here a NIDS or a different HIDS comes into play. The host can be configured in such a way that there is a different HIDS running that only runs on a part of the file system that has more exposure. A NIDS can also detect this kind of intrusion, because the attacker will need to communicate to or from the compromised host at some point.

Intrusion without persistance to use as intermediate host

This type of attack can not be detected by any HIDS that only works on file basis. Here again a NIDS can help. Most likely the attacker wants to gain access to other hosts in the network. When there is a NIDS installed, it should be able to detect this traffic and alert them. Another way would be to deploy an additional HIDS that does not work on file basis, but checks network information. This way it would find the intrusion and can alert it.

Intrusion without persistance to exfiltrate data

This type of attack is again very visible for a NIDS. Large amounts of data will be transfered in a way, that might not be usual. This can result in alerts from a NIDS. Here the proposed system might be able to alert the attack as well. As the attacker reads the data, it changes the modified timestamp. If the attacker does not reset this timestamp, the system will be able to alert the change.

Intrusion without persistance but as apt

This type of intrusion is again not possible to detect by a HIDS that works on file basis, except the attacker reads files. Here again a NIDS is needed which will be able to detect it, because after the infected process is restarted, another host will reinfect it. This is the kind of traffic a NIDS should be able to detect.

Attacker exploits HIDS to gain administrative privileges

For the system, this attack is probably not detectable, because the attacker changes the system itself. The best way to protect against that is to closely monitor what it does. The scanner part should only be allowed to read data from the disk image and write it to the specified data. To protect further, the investigator should be executed seperately without admin privileges. This way the attack surface is smaller. The only thing that can be used to generate an exploit would be the scanner config. Another way to notice the change might be the changed configuration which might already have been saved to the database. This way one might notice that something is wrong.

Attacker changes the code maliciously

This attack is not detectable from the system again, because firstly, the system itself was changed and behaves abnoramally. And secondly, because when the system was executed, the intrusion already happened. To protect from this form of attack, the system administrator should check the hash of the HIDS. Additionally, he should verify the Pretty Good Privacy (PGP) key. This way an administrator can make sure that the package which is installed has not been altered. Further, the source code can be checked and compiled by the system administrator himself. This would take more work but the possibility for malicious inclusions are fewer.

3.4.5. General defense

Generally it makes sense to use a NIDS in conjunction with any HIDS. Some intrusions are easier to detect on the host level and others on a network level. Both make sense so both should be used. Additionally, it makes sense to use an additional HIDS which is specialized in detecting intrusions on process or network activity level. This way the host is monitored fully and intrusions can be even better detected.

For the system specifically, it makes sense to run the scanner and the investigator seperately. To gain the most security, the scanner should run on the system and write the output to a DBMS which is outside the host with a user that can only append to most tables and additionally update the endtime column of the fids run relation. The investigator should then be started on a seperate host that only validates intrusions. This way the attack surface is as small as possible.

4. Discussion

The designed and implemented HIDS works differently than previous HIDS. Many will argue that because it does not generate cryptographic hash functions it is insecure. I argue against that, because finding intrusions always takes risks. If a conventional HIDS finds more intrusions but takes a long time to scan the whole system, it is not useful. Finding an intrusion days or even weeks after it happened might not be interesting. The attacker then has a long time to either hide, move on or extract all the information he is interested in. Finding intrusions in a timely fashion is very important and this is where my implementation shines. Additionally, even if it didn't find the intrusion, the database can be very helpful for forensic investigators to find out what the attacker did. This might help finding similar attacks in the future by adapting the configuration or by chaning other components. The argument this system makes is not even to be the one and only. In its current form this does not make sense. I hevily advise people to use a NIDS or a HIDS with other focusses. I also advise people to use a HIDS on file basis which uses cryptographic hash functions for highly critical parts of the system. However, if only a hash based HIDS is used, then many attacks will be found to late, or not at all. Another benefit that this implementation has is that it runs on any system. TSK runs on Linux, OsX and Windows, so does python. By using those components, this system should work on any of those systems as well.

During my work I realized the system and I tested it with modified data. However, the system has not yet been used in a productive environment and has not yet detected an intrusion. This means that it can not be said without doubt if it would work. The main reason why it has not been tested is that the time ran out. Besides that there were ethical and judical conserns of creating a host that is easily exploitable just so that it can be attacked. This would lead to criminals gaining access to a host to do their work which is not in my interest. Even if my system would find their intrusions, it is still possible that they can abuse the host for some time. Additionally, it is possible that they would use an attack which my system can not detect. This would mean that they have access to the host for a prolonged time. The system would needs to be tested in a live environment where attacks naturally happen. Sadly, I did not have access to such a system.

5. Conclusion

In this thesis the goal was to create a HIDS which finds intrusions in big file system fast and helps with forensic investigation. Such a system was built. I am using TSK to only check file system metadata to find intrusions. This way the system is extremely fast compared to systems that use cryptographic hash functions. My system thus takes a risk based approach to finding intrusions. It will not find them as reliably as other systems, but it will find them faster.

5.1. Future Work

It would make sense to extend this HIDS to extend past the file system. It would be great if the scanner can also scan the processes and network connections. Not only would this data be important to finding intrusions by the system, but it could also give investigators much more information. Information which they are currently mostly missing.

The system should also be extensively tested. For this it needs to run for a prolonged time in a productive system. The output and the found intrusions would then need to be compared to other system to gain important information about how fast and how many intrusions are found using this system.

One other path that could be improved on would be the investigator. Currently it operates only on a configuration which someone must write. It would be great to autonomously analyse the scanner output and find anomalies on them. This could be done if a lot of data has already been collected on many different types of hosts. The types could then be grouped and an algorithm could detect similar patterns to find simmilar intrusions.

Generally, the field of finding intrusions has a lot of opportunities for research. This system can be one part of an extensive system that checks for intrusions. Especially since disk sizes and data usage is still growing it is important to have such a system that can find intrusions fast.

It would also be interesting to add the output of the scanner to a super timeline. It would help seeing what happened on the host at any point in time.

Declaration of primary authorship

I hereby confirm that I have written this thesis independently and without using other sources and resources than those specified in the bibliography. All text passages which were not written by me are marked as quotations and provided with the exact indication of its origin.

Place, Date:	Biel, June 9, 2019
Last Name, First Name:	Julian Stampfli
Signature:	....

Acronyms

API Application Programming Interface. 5, 6, 13, 14, 16

APT advanced persistant threat. 27

CD Compact Disc. 3

CLI comand line interface. 5

DBMS DataBase Management System. 10, 13, 32

DVD Digital Versatile Disc. 3

HDD Hard Disk Drive. 3

HIDS host based intrusion detection system. 1, 4–10, 13, 20, 25, 27–33, 35, 49, 51

ID IDentifier. 13–16

IDS intrusion detection system. 1, 5

KB Kilobyte. 3

MB Megabyte. 6

NIDS network based intrusion detection system. 1, 5, 31–33

NIST National Institute of Standards and Technology. 4

OWASP Open Web Application Security Project. 27

PGP Pretty Good Privacy. 32, *Glossary:* PGP

SHA-256 Secure Hashing Algorithm - 256. 4, 14

SSD Solid State Drive. 3

TB Terrabyte. 6, 7

TSK The Sleuth Kit. iii, 5–9, 13–16, 29, 33, 35, 40

USB Universal Serial Bus. 3

UUID Universaly Unique IDentifier. 13

Glossary

anomaly An unexpected change in the file system. For a change to be unexpected it needs to be covered by the configuration changes are unexpected only if the configuration says so. Some examples that might cause anomalies, changed rights, new files, deleted files.. 1, 5

collision Multiple different inputs with the same hash value.. 3

cryptographic hash function A deterministic one way function that fullfills collision resistance and other cryptographic properties. Implementations are the SHA-2 and SHA-3 families.. 1, 2, 4–7, 27, 33, 35, 40

file system A file system is used to create a layer of abstraction between the hardware of the storage medium and the operating system. There are multiple file system types which are in use with different capabilities. Additionally to the files the file systems keep track of meta data to each file. This meta data includes some timestamps (created, last accessed, etc) and more.. 1–7, 9, 10, 17, 18, 27, 29, 31, 35, 40

github A platform for opensource projects. It is free to use and hosts the source code for many projects.. 6

hexadecimal A number system with 16 digits. It uses the numbers 0-9 and the Letters A-F.. 13

intrusion An unauthorized access to a system or to data.. 1

malware Malware is any program that is designed to harm a computer system. The term includes well known terms like Virus, Worm, Adware, Keylogger, Trojan, etc. Malware usually tries to hide it's traces to achieve longer infection periods of time.. 5

metadata Data that gives information on other data. In this thesis it is mostly used to describe attributes of the file system that describes files. Exmaples are creation date and permissions.. 1–3, 5, 7, 35

non-cryptographic hash function A deterministic one way function that does not the hard to achieve properties that make a cryptographic hash function. Often used when speed is more important than collision resistance.. 4, 5

opensource Software where both the source and the software is freely accessible and changable.. iii, 5, 6, 29, 40

PGP Standard for encryption and signatures defined in RFC4880. 32

postgres An opensource database server. It is rather lightweight and heavily used.. 10

pytsk3 Python bindings for TSK. iii, 7, 9, 17, 29

regex Regular Expressions are a standard way to find certain patterns in a string.. 12

SQL A domain specific language used for querying a relational database.. 8

sqlite A very lightweight relational database implemnentation. It does not have a dedicated server but instead writes the database to a file which is written to the local host.. 10

storage media media that is used to store data in a computer system.. 3, 6

UNIX timestamp The number of seconds since 00:00:00 on the 1st of January 1970. 13

YAML YAML Ain't Markup Language. 9

Bibliography

[1] . https://www.tripwire.com/company/. [Accessed; 2019-06-10].

[2] *About*. https://www.sleuthkit.org/about.php. [Accessed; 2019-06-10].

[3] *AIDE*. https://aide.github.io/. [Accessed; 2019-06-10].

[4] *aide source code*. https://github.com/aide/aide. [Accessed; 2019-06-10].

[5] Ulrich Bayer et al. "A View on Current Malware Behaviors". In: *Proceedings of the 2Nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*. LEET'09. Boston, MA: USENIX Association, 2009. URL: http://static.usenix.org/events/leet09/tech/full_papers/bayer/bayer.pdf.

[6] *CITE NEEDED PLEASE LOOK FOR SOMETHING*. https://startpage.com/. [Accessed; 2019-06-10].

[7] *Cryptographic and Non-Cryptographic Hash Functions*. https://dadario.com.br/cryptographic-and-non-cryptographic-hash-functions/. [Accessed; 2019-06-10].

[8] *Detailed Understanding Of Linux Inodes With Example*. https://linoxide.com/linux-command/linux-inode/. [Accessed; 2019-06-10].

[9] *FLS*. http://www.sleuthkit.org/sleuthkit/man/fls.html. [Accessed; 2019-06-10].

[10] *How to Calculate a Hash of a file that is 1 Terabyte and over?* https://stackoverflow.com/questions/22724744/how-to-calculate-a-hash-of-a-file-that-is-1-terabyte-and-over. [Accessed; 2019-06-10].

[11] Gene H. Kim and Eugene H. Spafford. "The Design and Implementation of Tripwire: A File System Integrity Checker". In: *Proceedings of the 2Nd ACM Conference on Computer and Communications Security*. CCS '94. Fairfax, Virginia, USA: ACM, 1994, pp. 18–29. ISBN: 0-89791-732-4. DOI: 10.1145/191177.191183. URL: http://doi.acm.org/10.1145/191177.191183.

[12] Lyudmil Latinov. *MD5, SHA-1, SHA-256 and SHA-512 speed performance*. https://automationrhapsody com/md5-sha-1-sha-256-sha-512-speed-performance/. [Accessed; 2019-06-10].

[13] *Linux – Fake File Access, Modify and Change TimeStamps*. https://www.shellhacks.com/fake-file-access-modify-change-timestamps-linux/. [Accessed; 2019-06-10].

[14] Arthur Messenger. *AIDE to the Rescue – An Open Source Security Tool*. http://www.drdobbs.com/aide-to-the-rescue-an-open-source-sec/199101554. [Accessed; 2019-06-10].

[15] National Institute of Standards and Technology (NIST). *FIPS PUB 180-4: Secure Hash Standard (SHS)*. Aug. 2015. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[16] Wim Nevelsteen and Bart Preneel. "Software Performance of Universal Hash Functions". In: *Advances in Cryptology — EUROCRYPT '99*. Ed. by Jacques Stern. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 24–41. ISBN: 978-3-540-48910-8.

[17] Bruce Nikkel. *Practical Forensic Imaging*. No Starch Press, 2016. ISBN: 1593277938.

[18] *Open Web Application Security Project*. https://www.owasp.org/images/7/72/OWASP_Top_10-2017_(en).pdf.pdf. [Accessed; 2019-06-10].

[19] *Overview*. https://www.sleuthkit.org/sleuthkit/. [Accessed; 2019-06-10].

[20] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. 1st. Springer Publishing Company, Incorporated, 2009. ISBN: 3642041000, 9783642041006.

[21] *The AIDE manual*. https://aide.github.io/doc/. [Accessed; 2019-06-10].

[22] *The Coroner's Toolkit (TCT)*. http://www.porcupine.org/forensics/tct.html. [Accessed; 2019-06-10].

[23] *Tripwire*. https://www.tripwire.com/. [Accessed; 2019-06-10].

[24] *tsk_fs.h File Reference*. http://www.sleuthkit.org/sleuthkit/docs/api-docs/4.2/tsk__fs_8h.html. [Accessed; 2019-06-10].

[25] *TSK_FS_ATTR Struct Reference*. http://www.sleuthkit.org/sleuthkit/docs/api-docs/4.2/structTSK__FS__ATTR.html. [Accessed; 2019-06-10].

[26] *TSK_FS_ATTR_RUN Struct Reference*. http://www.sleuthkit.org/sleuthkit/docs/api-docs/4.2/structTSK__FS__ATTR__RUN.html. [Accessed; 2019-06-10].

[27] *TSK_FS_FILE Struct Reference*. http://www.sleuthkit.org/sleuthkit/docs/api-docs/4.2/structTSK__FS__FILE.html. [Accessed; 2019-06-10].

[28] G. Vigna and R. A. Kemmerer. "NetSTAT: a network-based intrusion detection approach". In: *Proceedings 14th Annual Computer Security Applications Conference (Cat. No.98EX217)*. Dec. 1998, pp. 25–34. DOI: 10.1109/CSAC.1998.738566.

[29] Sander van Vugt. *Working with Linux file system attributes*. https://searchdatacenter.techtarget.com/tip/Working-with-Linux-file-system-attributes. [Accessed; 2019-06-10].

List of Figures

List of Tables

Listings

APPENDICES

A. Project Management

A.1. Goal

Before the start of the project the following main goal was defined:

Building of an HIDS that detects unauthorized or unusual behaviour on the file system. Compared to traditional HIDS file system integrity checking, it should scale with a lot of data and have the possibility to be used for investigation (retain historic data) built in from the start.

A.1.1. Sub goals

From this primary goal, the following sub goals were defined.

Scanning

The system is capable of scanning the file system for certain properties. The search is done by leveraging the sleuthkit tools. Thus the system is capable of interpreting the results from sleuthkit. It will further analyze them and decide on what to do with the results. Especially importance is given to the finding of differences.

Recording

The system records all findings. Including new, changed and deleted files in comparison to an earlier point in time. This recording enables the use of investigation as the evolution of the data can be viewed at any time. This data can also be used for machine learning algorithms to detect anomalies that are out of the scope of this thesis.

Evaluation

The system is capable of evaluating the results by applying predefined rules. Those rules can be adjusted by configuring the system.

It is thinkable that the system analyzes the recordings and makes decisions based on the historical behavior of the specific host and behavior from different similar hosts. This approach is not part of this thesis as it requires much historical data that is not present at the time of this thesis.

Alerting

The system is capable of being run continuously. This capability enables it to find anomalies automatically. The system can report those anomalies by creating alerts. It allows configuration of these alerts.

Scaling

The run of the system on a big file system completes in an appropriate amount of time. This speed allows the finding of anomalies that appeared recently. Additionally, it allows the storing of more states of the system which results in a her probability of capturing short-lived anomalies for future investigations.

A.2. Workpackages

From those goals the workpackages in table A.1 were defined. For a better overview they are assigned to categories. The categories are Architecture, Implementation, Validation and Administrative. Architecture is about defining how the system will look like and how it should work. Implementation is the effective implementation work for getting the system to run, this includes configuration and coding. Validation is about testing of the system. Administrative is everything that deals with project management and other workpackages that don't directly influence the system but need to be done.

The ID is a combination of the first letter of the category and a unique index. Administrative is shortened to D due to the conflict with Architecture.

The priority is a value of high, medium and low.

The workpackages are chrononically ordered. Meaning they should be worked on in approximately the order that they are given.

A.3. Planning

For the planning of this project the following milestones were created. Each coveres multiple workpackages. The mapping can be seen in table A.2. The milestones can also be seen in figure A.1. There they are displayed with an assumed and actual finish date.

Table A.1.: Workpackages

ID	Short description	Prio	Category
D00	Setting up LaTeX-document	h	Admin.
D01	Define workpackages and set deadlines	h	Admin.
A00	Research other HIDS and the sleuthkit tools	h	Archit.
A01	Decide on a Programming Language	h	Archit.
I00	Setup the developer environment	h	Impl.
I01	Add the ability to scan the whole system using sleuthkit	h	Impl.
A02	Decide which database connectors should be used	h	Archit.
I02	Add one database connector	h	Impl.
I03	Implement a recording functionality	h	Impl.
A03	Decide how the rules should be defined	h	Archit.
I04	Add template rules and ability to parse them	h	Impl.
I05	Add functionality to parse output according to rules	h	Impl.
V00	Verify that the system runs on a big file system	h	Valid.
I06	Add functionality of repeated scans	m	Impl.
A04	Define which alerting methods make sense	m	Archit.
I07	Add alerting functionality using one method	m	Impl.
V01	Verify the functionality of the software by changing the system	h	Valid.
V02	Verify the functionality of the software by running it on an infected system	m	Valid.
V03	Verify the alerting of the software by running it on an infectable system	m	Valid.
I08	Add multiple database connectors to different systems	m	Impl.
I09	Add multiple alerting methods	m	Impl.
A05	Define how to protect system and configuration from tampering	l	Archit.
I10	Implement software hardening	l	Impl.
D02	Finish user documentation	m	Admin.
D03	Finish project documentation	h	Admin.
D04	Create project presentation	h	Admin.
D05	Create project poster	m	Admin.
D06	Create project video	l	Admin.

Table A.2.: Milestones

ID	Short description	Workpackages
00	Setup	D00, D01, A00, A01, I00
01	Initial functionality	A02, I01, I02, I03
02	Rules	A03, I04, I05, V00
03	Alerting	A04, I06, I07
04	Exhaustive testing	V01, V02, V03
05	Usability	I08, I09, D02
06	Software Hardening	A05, I10
07	Presentation	D02, D03, D04, D05, D06

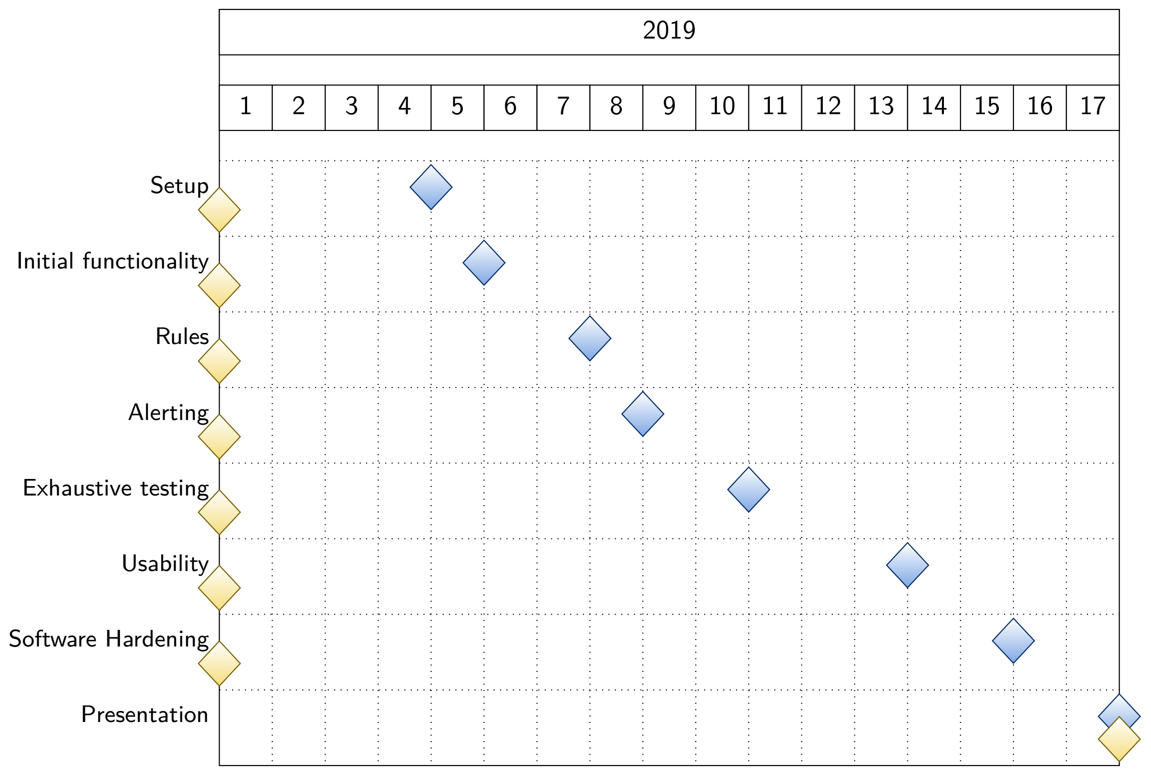

Figure A.1.: Milestones

B. Implementation Decisions

B.1. Programming Language

B.2. Language for Configuration File

C. Journal

D. User Documentation

E. Content of CD-ROM

Content of the enclosed CD-ROM, directory tree, etc.