RUHR
UNIVERSITÄT
BOCHUM

RUB

RUHR-UNIVERSITÄT BOCHUM

Mitigating Anti-Forensics: A Schema-based Approach

Phil Knüfer

Bachelor's Thesis – January 5, 2014.
Chair for System Security.

1st Supervisor: Prof. Dr. Thorsten Holz
2nd Supervisor: Dipl. Inf. Johannes Hoffmann
Advisor: Dipl. Wirt.-Inf. Martin Wundram (DigiTrace GmbH),
M.A. Alexander Sigel (DigiTrace GmbH)

hgi SYSSEC

Abstract

The goal of this thesis is to find an improved way to deal with the ever-growing anti-forensic risk. The situation today is that most testing is conducted unstructured and insufficiently organised. We present a new schema-based approach that tries to counter this behaviour. Therefore, we first design our own schema and give ideas on how test cases can look like. We then implement examplary test cases and describe this step in detail to give ideas on how to build own ones. At last, we evaluate a cross-section of forensic tools with the test cases and and find out that our implementation work is well-suited to find flaws in today's forensic software products. We conclude that there is still much work to be done to enhance security against the anti-forensic threat.

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Date	author

Contents

1 Introduction

1.1 Motivation

The field of Cyber Forensics is part of the digital world since the early days of personal computers, but techniques that were once mostly a manual search of data have been a subject to changes in the course of time [Cha09]. Nowadays digital forensic investigators have to handle a massive volume of digital data and the amount is constantly growing. To facilitate the examination process software programs have been created that automate a lot of tedious tasks, such as extraction of all archive files, user-defined keyword search and creation of human-friendly reports about USB devices that have recently been connected to the investigated system or software that is installed on it.

As forensic reports are often used in court and thus have an important influence on the decision between freedom and jail, it is extremely important that investigators work thoroughly and that their results are always correct, or that errors are at least detected whenever they occur. Because of that it has become common practice in all forensic sciences to verify the correctness of the tools in use [ILA02]. The ISO standard 17025 ([ISO05]) describes an abstract approach to testing, and some laboratories decide to show their compliance to it by getting certified.

Following the idea of testing, various people have tried to also verify the correctness of the software tools used for *Digital* Forensic tasks [Bru11]. With the Computer Forensic Tool Testing Project (CFTT) the US-American National Institute of Standards and Technology (NIST) tried to formulate a standardised methodology of testing such software[NIS13a]). However, one cannot evaluate their results as test cases are not published directly on the CFTT website and the documentation on test assertions and test plans remains rather abstract to be applicable to a broad range of tools. Carrier ([Car10]) presents image files of hard disk partitions containing specifically crafted files that are meant to complement the work of the CFTT. Unfortunately the vast majority of testing of forensic software performed to date follows a functional driven approach. This is a technique typically used in software development which analyses if the intended functions of a software work as they should. However functional testing does not cover problems that may arise when software is (mis)used in a way the developer did not think of. First unstructured tests ([WFM13], [NPSB07]) have already shown that there is a huge potential in finding and exploiting software bugs by applying other testing techniques.

Another problem is the lack of structured test environments. Given the fact that everybody tests on its own one can easily imagine how incomplete the results must inevitably be. Different investigators make different experiences in their everyday work and when testing their equipment they will most certainly focus on scenarios that they are familiar with.

This thesis tries to develop a systematic testing approach that can serve as a guideline during forensic tool testing. By organising the information that flows into and out of software as general as possible we want to help testers think outside of their typical scope. Additionally, test cases are derived from the laid out schema to give examples on the methods one could use to go beyond functional testing. However, it is extremely important to always keep in mind the limitations of this thesis specified in Section 1.3.

1.2 Related Work

Literature on Anti-Forensics

The field of finding attacks against digital forensic software, often called anti-forensics, is relatively new. Therefore, one does not find as many papers as in other disciplines when consulting relevant databases. For example, when searching for the term "anti forensic" we found 760 papers on *Google Scholar*, from which over 90% are written after the year 2005. One of the earliest sources covering anti-forensics is [Rog05]. Rogers there defines the term and presents some basic but still mostly passive ideas (defensive measures without *attacking* anything). It is very noteable that although there has not been much research in this field until 2005, Rogers already has a few slides that cover the idea of directly targeting the forensic software.

Two years later Newsham et al. have published a paper that shows the rapid development in the research field [NPSB07]. They leave the path of functional testing and use fuzzing as well as "targeted fault injection" to find bugs caused by programming errors and corner cases that the developer did not think of. A major drawback of this work is that the testing is performed rather unstructured and no explanation is given why a certain test-case has been created or why others have been left out.

Wundram, Freiling and Moch [WFM13] present attacks that build upon the work done by Newsham et al. They revisit well known but still relevant vulnerabilities and expand them by showing the first code injection attack against a commercial forensic tool. The idea to target the HTML reporting function is quite interesting and useful but again the paper lacks an overall structure of the outlined vulnerabilities.

Literature on Taxonomies of Software Vulnerabilities

While there are not that much papers covering anti-forensic topics the situation is completely different for work that defines taxonomies of software vulnerabilities. One of the most important articles in this field has already been published in 1994 by Landwehr et al. [LBMC94], and it is by far not the first one. Many other taxonomies are based on this paper and we found references to it numerous times during our literature research. However, due to the early date the picture of security flaws is somewhat outdated. Additionaly, Landwehr et al. write in their "Limitations" section that "the development of this taxonomy focused largely, though not exclusively, on flaws in operating systems". This is why their categories, *genesis, time of introduction, and location*, are not really useful as a base for this thesis. It is improbable that a forensic tool contains intentional flaws as well as it is uninteresting from the user's perspective whether a flaw has been introduced during the creation of the application or during maintenance, e.g., by an update.

For a better overview on different taxonomies Igure and Williams [IW08] have created an extensive roundup of numerous security based taxonomies published between 1974 and 2006. They split the paper in both attack and vulnerability based taxonomies. As no actual taxonomy is given on its own their work is not helpful for this thesis in a direct way, instead it shows that none of the mentioned work does cover forensic software and the related risks as a central aspect. However, one should keep in mind that Igure and Williams may have missed a taxonomy, or that a new one has been produced recently.

One of these unmentioned taxonomies, though a very mentionable one, is the paper "Seven pernicious kingdoms" by Tsipenyuk, Chess and McGraw [TCM05]. The *seven kingdoms*, a term the authors adopted from biology, are groupings for different types of software flaws. These kingdoms build the origin of the schema presented in this thesis. They are also the core of the Common Weakness Enumeration (CWE) project managed by the MITRE corporation. A deeper insight in this project is given in Section 2.5. At this point it should only be noted that MITRE is the organisation behind the well known Common Vulnerability Enumeration identifier system that has a knowledgeable reputation among computer (security) professionals. After working with the taxonomy of Tsipenyuk et al. in more detail we found some *shortcomings* for our special use case, such as the strong focus on source code. This aspect is, together with others, discussed in Section 3.1.

To conclude the review of related work we would like to note that, although there is not much scientific work in the area of anti-forensics, not every paper or presentation found during the research phase can possibly be mentioned in this section. On the taxonomy side the variety is even more vast. The presented work only shows the most important influences on this thesis from both research fields.

1.3 Contribution and Limitations

In this thesis we present a schema that can serve as a structured testing guideline. We therefore build upon the taxonomy by Tsipenyuk et al. and enhance it to better suit our needs. We design a tree structure that divides the abstract term *Forensic Tool* into different areas that can be of interest during testing. Due to the limited time given when working on a Bachelor's thesis we focus on software tools only and exclude the fact that hardware can be attacked as well. Interested readers will find a very brief passage on hardware attacks in the conclusion at the end of this thesis.

From the previous chapter we learned that there are many taxonomies dealing with security flaws, even if they use different terms for that fact. Still, all of them treat user input only as one of many aspects, but it clearly is the main part of forensic work, where data of other people has to be analysed. This is why we designed our schema tree to reflect various different types of data in its leaf nodes.

In the second part of the thesis we implement multiple test cases that are derived from these leaf nodes. Again we have to set limits to our work because of the sheer amount of forensic tasks, storage systems in use and different file formats one could use to encode data. We focus on the most common and best researched forensic analysis method, the *Post Mortem Analysis*. There, complete hard disk copies or loose files gathered from a computer are looked at on a system different from the investigated one (hence the term post mortem in contrast to a live analysis). Because everybody who expects to get into a forensic investigation one day can prepare such files prior to that, we believe that they constitute a major risk for forensic software and should be tested most. Certainly, every investigator testing her software in use should also think of the other leaf nodes in the schema and never forget that all testing is in vain if the forensic workstation itself is not secure from hacking attempts happening *live*, that is, during the investigation of the evidence. The test cases are evaluated by analysing them with common digital forensic software tools. We will depict their usefulness depending on the results. Not least because of the high cost often associated with highly specialised software, the evaluation is limited to some well-known commercial software products, as well as to some open source (or at least free) software.

Additionally to this thesis we will propose the results to standardisation institutions such as NIST and promote them within the forensic community, possibly by publishing them on the inofficial CFTT mailing list [Anoa].

1.4 Organisation of This Thesis

The structure of this thesis follows mostly the order used in the previous section. First of all we provide some background on the topic and explain basic concepts in Chapter 2. We then outline our schema and explain its development process, as well as derive test cases from it in Chapter 3. In Chapter 4 we describe the precise implementation of them, thus giving ideas on how one could create own test sets. The evaluation of the test cases follows directly in Chapter 5. Finally we conclude the results and present an outlook into the future in Chapter 6.

2 Background

This chapter provides insight into the most important ideas needed to understand the concept of digital forensics and a possibly related anti-forensic risk. Furthermore, some important standardisation and classification institutions are presented.

2.1 Digital Forensics

A common question asked by people that are not familiar with computing and digital data is about what computer forensic scientists exactly do. They know the term *forensics*, but associate it with forensic doctors examining mortal remains or with police officers gathering fingerprints at a crime scene. In general forensics comprises everything court related, as the word originates from the Latin word *forum* – market-place, which was where in ancient Rome people would gather for judicial reasons [Oxf13].

Digital forensics is the subfield of gathering evidence linked to everything related to IT systems. The term *system* in that case includes devices (personal computers, servers, mobile phones, ...) as well as storage media (external hard drives, USB sticks, CDs ...). Most of the time the data residing on these artifacts is of forensic interest, but it is also important to investigate the hardware itself.

When finding an item of interest, a digital forensic investigator must first make sure that it remains unchanged. Depending on the situation it must be decided whether to do a *live analysis* of the system or to only analyse switched-off devices, an approach called *post mortem analysis*. The data on it must then be acquired safely, so that it is modified as little as possible and all changes are documented. In the post mortem case this is mostly achieved by using dedicated devices that physically prevent write access to a device, so called *writeblockers*. After making a working copy of the evidence data it has to be analysed. Depending on the type of forensic investigation this step can include different tasks. If for example the computer of a person accused to possess or distribute child pornography is investigated, the analysis step would include searching for image or video files, but also analysing the visited websites and possible chat protocols or email conversations. Finally, a report has to be written which is understandable for people not being technical experts, e.g., judges. This step has to be conducted with particular care, because not all forensic investigators are directly employed by law enforcement authorities and might therefore not be exactly familiar with the formal requirements.

2.2 Digital Forensic Software

Investigators are nowadays challenged by a surprisingly simple problem. Digital storage amounts are constantly growing, a fact that we encounter almost every day, when reading about new data centers or just by comparing a typical computer setup from the daily newspaper advertising supplement, with one we saw one or two years ago. When analysing such a system under forensic viewpoints there is today *much* more data to look at than in the early days of computing. To keep up with the developing technology, hard- and software tools have been created to support digital forensic tasks.

There is many specialised software, some of it is downloadable free of charge while other programs are advertised as complete packages and cost lots of money. A certainly not representable list of tools with more than 75 entries can be found on the wikipedia [Wik13].

Software can help forensic experts during all steps of an investigation. The bit-level cloning or logical copying of data, whichever approach is chosen, would not even be possible without the help of software reading the data. Other software helps reading out proprietary data formats or automates part of the analysis step. All big commercial software tools are for example able to automatically traverse through all directories of a hard disk image and search for files containing special terms that the investigator defined beforehand. Therefore, most of the tools can automatically extract archive files found to search their contents as well. It is even possible to generate parts of the final report by using software. Some tools can create nicely formatted lists of all removable devices that were once connected to a computer, or of all the software that is installed on it.

As a conclusion we can retain that there is a multitude of software assistants for the daily routine in a forensic laboratory.

2.3 Anti-Forensics

Anti-Forensics, sometimes also named counter forensics, is in general a term describing every approach that makes the life of a forensic investigator harder. A formal definition of the term, and the one we use in this thesis has been formulated by Dr. Marcus K. Rogers in 2005 [Rog05]. The choice fell on it because it is the oldest one that we found.

> Attempts to negatively effect the existence, amount and/or quality of evidence from a crime scene, or make the analysis and examination of evidence difficult or impossible to conduct.

The relatively recent date shows that the topic has long been outside of the focus of the scientific community. However, methods known for a much longer time are since then called anti-forensics, for example cryptography that is heavily researched for at least 40 years now (standardisation of DES in 1977). Rogers, as well as others ([Har06], [Gar07]) categorise some *classic* anti-forensic threats which all describe the passive methods mentioned in Section 1.2. All classifications know some kind of *data hiding*, where evidence is stored in unusual spaces, encrypted, or packed into other unsuspicious files by use of steganography techniques. A second method is called destroyal or wiping of data. In that case files are not simply deleted, but overwritten with a series of either random or zero bits. This technique can be more stealthy than the hiding approach, but obviously wiped data is lost and can not be accessed at a later time. The third technique known in classic anti-forensics belongs into the area that Rogers calls "trail obfuscation". Around the real evidence, multiple instances of fake evidence are created to consternate the investigator. A practical example of obfuscation and annoyance is a collection of script files uploaded to github in 2011 [Int11] by a user called *int0x80*. One of them is for example a shell script creating lots of files using random data and encrypting it with an also random key. Afterwards the files are deleted *insecurely*. If an investigator would search for deleted files after this script has been run he would recover thousands of encrypted archives and had to choose whether to try bruteforcing them or leave them out, risking to skip valuable evidence.

More interesting in the field of anti-forensics is the *modern* threat of targeted attacks against the forensic software. This is what lastly made anti-forensics popular as a research field ([NPSB07] [WFM13]). While the classic techniques pose no threat to the reliability of the evidence that could be found (one could *only* miss real traces), modern anti-forensics targets the investigation process as a whole. Data is made unreliable, as the security and integrity of the software that acquired it is undermined. By exploiting flaws found in forensic software an attacker can basically reach one of the following goals:

1. Crashing the forensic software (Denial of Service)

2. Causing unspecific misbehaviour of the forensic software, such as leaving out a certain subfolder

3. Gaining control over the software, thereby defining its output, with the possibility to damage the original acquired data or to add false evidence of either incriminating, exculpating or none-credible and thereby legally attackable kind.

The first two aspects are already quite serious. Often an investigation has to be finished in a short time frame and software that constantly crashes costs much of the precious time. Knowing all weaknesses in the software a forensic expert could use multiple tools to be sure everything has been analysed completely, but that again would cost time for setup and comparison of the results.

The last aspect however is a disaster. If software could be a target of such attacks its reliability in court is very doubtful, as stated by Ridder [Rid07]. It could already be sufficient for the defense lawyer of an accused person to raise enough questions and thus making the judge decide to not consider the findings of the forensic analysis in its sentence.

2.4 Standardisation of Digital Forensic Tool Testing

Standardisation is an approach that is more typical for forensic works in the USA than it is in Germany, a fact explained in the following. One of its main uses is to provide help in courtrooms, both to forensic witnesses who can support their findings by pointing out that the tools they used have been tested by well-known organisations, as well as to judges who often are no experts in the technical field and thus could not easily assess if the forensic witness used reliable equipment. Ridder [Rid07] explains different criteria, known as the so-called *Daubert standard* after a famous trial, that may be used by U.S. courts to evaluate the reliability of evidence, among them being the factors "whether the theories and techniques employed by the scientific expert have been tested", "whether they have been subjected to peer review and publication" and "whether the theories and techniques employed by the expert enjoy widespread acceptance." All of these might be supported by standardisation approaches.

Although such concepts do not automatically apply to other countries standardisation might as well be helpful when reasoning in German trials. Typically, *§ 261* Strafprozessordnung (StPO) applies which is known as "Freie richterliche Beweiswürdigung" [HR10] –free consideration of evidence by the judge. In real life scenarios this means that an expert witness has to make plausible that her report is correct by explaining the abilities of the tools used, most ideally also by pointing out their shortcomings and reasoning why these do not pose a threat to the concrete findings. One can easily imagine that a tool being standardised by well-known organisations such as NIST can be a supportive argument.

2.4.1 Computer Forensic Tool Testing Project (CFTT)

NIST is known for publishing standards in different areas of interest. Recently it has been in the news along with the discussions around the Snowden leaks and possible backdoors inside a standardised cipher [Zet13].

One of the projects running at NIST is the Computer Forensic Tool Testing Project, which goal it is to raise the reliability of computer forensic tools, both soft- and hardware, by "development of general tool specifications, test procedures, test criteria, and test hardware" [NIS13a].

In a typical approach, NIST [NIS01] decides on a feature to test and than chooses tools that support this feature. Requirements are formulated and test assertions derived from that requirements are developed. Then test cases are produced that reflect the requirements. Finally NIST publishes the results alongside with a documentation of the test assertions and the created test cases.

When reviewing the test case descriptions we found test case *DA-25* describing the reaction of a tool to corrupt image files [NIS05]. It would have been possible to define all test cases created in this thesis as specialisations of this one, as it is very abstract and anti-forensically prepared evidence always deviates from the normative behaviour, making it corrupt. However, we feel that the requirements defined by the CFTT are not strict enough for our needs. The only required behaviour of a tool is that it "executes in execution environment XE", but XE is completely unspecified and only an abbreviation for any execution environment one can think of. Plain-talking, this does not mean more than that the tool starts. All other assertions that are to be tested remain optional. In this thesis, more requirements are made obligatory. Details on this topic are given in Chapter 5, the test cases are listed in Appendix F.

The abstract definition of testing is a general problem all documents published by the CFTT have in common. It remains unclear why a certain tool is chosen to be tested, which test assertions were made up for what reasons and why test cases were created while others might have been left out. Additionally, there are categories that seem as if they would have been forgotten somewhere in the testing process. The NIST website for example lists a category "Forensic String Search" where only a single document, a public draft for comments on the tool requirements specification, has been published since 2008. When browsing the website it remains unclear whether or not the category is still subject to active research.

In spite of all drawbacks the program might have, the CFTT is the only major approach to testing forensic software, and instead of talking it down, support would be in the interest of everyone in the forensic community.

2.4.2 Computer Forensic Reference Data Sets

Computer Forensic Reference Data Sets (CFReDS) is another project residing at NIST [NIS13b]. It is mainly a website listing about a dozen data sets that are meant to be test cases for forensic tools. The website states that some of the test cases are provided by the CFTT project but it remains unclear which of them these are and from which of the various CFTT tests they stem. The website also mentions briefly how own test cases can be created and how they should be documented. The information thereon does not go into detail, but the advice to completely document everything that a user needs to fully reproduce the results helped in improving the test case documentations that we created (see Appendix F). Hyperlinks to

other test images are provided, for example to those of Brian Carrier [Car10] whose goal it has been to "fill the gap between extensive tests from NIST and no public tests".

CFReDS seems to be a much smaller project in comparison to CFTT, but it could be a place to publish community-based test cases dealing with anti-forensics, such as those that are developed in this thesis. Not least, having the test cases published by a project associated with an organisation as big as NIST will certainly enhance the reception of them.

2.5 An Examplary Schema: MITRE's CWE

The non-profit organisation MITRE is the founder of the CVE system. It is well known for providing a numbering scheme of vulnerabilities detected in real life software. With the CWE identifiers MITRE tries to achieve a similar organisation schema for general weaknesses as *cross site scripting* or *buffer overflows* in contrast to actual vulnerabilities. The work of MITRE is based on several sources, among others the paper mentioned in Section 1.2 and the Preliminary List of Vulnerability Examples for Researchers (PLOVER)[Chr06].

The CWE list is not only a flat chronological numbering of weaknesses but more a hierarchical classification of them. On the website [Cor13] one can both access online, as well as download different views of the complete list. All of them are available in graphical form as tree-like structures and as lists that can be read from top to bottom. The website links various information to a CWE entry when browsing it, for example Common Vulnerability Enumeration (CVE) identifiers that are associated with this type of weakness, an explanation of the possible impacts and sometimes examples of bad code.

Unfortunately the complete CWE list is quite complex, enumerating 940 different weaknesses that vary from very general ones as CWE-693 ("Protection Mechanism Failure") to very specialised entries as CWE-82 ("Improper Neutralization of Script in Attributes of IMG Tags in a Web Page"). Due to to the complex nature and the general purpose approach of the CWE system we decided to base our schema not directly on it but on its sources, however taking over the idea to organise the schema as a tree structure.

3 Schema and Deduction of test cases

Taking into account the basic concepts of digital forensics and its related software, explained in Sections 2.1 and 2.2, we now set up our schema to give a structure to forensic software testing. As explained in Section 1.2, the schema is based on the ideas of Tsipenyuk et al. While working with their ideas we found however that there are drawbacks when applying them to forensic software. This is why we first have a more detailed look on the basics of their taxonomy in Section 3.1 before building our own schema in Section 3.2. Finally, Section 3.3 deduces the test cases implemented in Chapter 4 and gives ideas on how to build further tests according to the schema.

3.1 Review of *Seven Pernicious Kingdoms*

In Section 1.2, a first look on the paper in which Tsipenyuk et al. originally described their taxonomy (in the following referred to as the *kingdom taxonomy*) is given. There, it is already pointed out that some aspects do not work out that well in conjunction with our needs.

First of all the *kingdom taxonomy* is based on the assumption that one has access to the source code of an application. This is perfectly fine for the target audience Tsipenyuk et al. had in mind, namely the developers of the software themselves. Although developers of forensic software should of course do extensive testing before releasing it, we want to target the average forensic investigator that most of the time is just a user. Among the commonly used forensic software tools, most have commercial license models and do not provide access on a source code level. This is why we need an approach that treats software as a black box component, as shown in Figure 3.1.

To understand the other shortcomings of the *kingdom taxonomy*, a list of all kingdoms together with a brief description is given in Table 3.1. A core concept of forensic software is that it heavily relies on input provided by untrusted users. Input validation should therefore not only be one of many equal aspects. In general, all vulnerabilities in software are only accessible if some input is given to trigger them. The *API abuse* kingdom is well-suited to be included in our schema. It describes a special type of input and provides a good naming scheme to separate it from other input classes. *Security features* though are more an abstract concept than a concrete

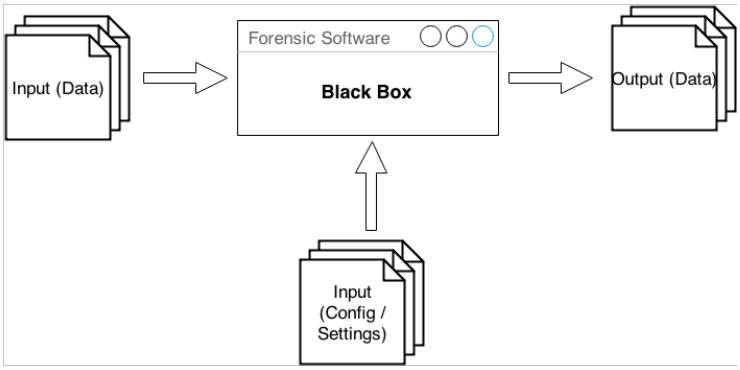

Figure 3.1: The forensic software, depicted as a blackbox.

Table 3.1: The kingdoms from the taxonomy of Tsipenyuk et al.

Name of Kingdom	Description
Input Validation and Representation	Mistakenly trusting input, causing *buffer overflows*, *cross site scripting*, *SQL injections* and other vulnerabilities
API Abuse	Possibility to misuse an API and thereby causing security issues
Security Features	Errors related to authentication, access control, cryptography and other software functions *directly* related to security
Time and State	Vulnerabilites related to temporal aspects. Examples include *race conditions* and *deadlocks*.
Errors	Wrong error handling or error messages that give out too much internal information.
Code Quality	Vulnerabilites that are based on poor quality of the source code.
Encapsulation	Failure to separate certain parts of an application from each others. This could mean not correctly sandboxing an application or not separating different user accounts enough.
Environment	Vulnerabilites that are not caused by the source code but introduced through other means, e.g., by automatic compiler optimisation

software feature. They are not directly mentioned in our schema but have a branch inside the tree where they belong to, together with the *time and state* vulnerabilities. Section 3.2 has more details about that. For the *errors* kingdom, a similar idea as for input validation holds. Every error needs a trigger and the exploitation of error handling will furthermore lead to another vulnerability, such as the possibility to overflow a buffer. It is somewhat redundant to have a dedicated kingdom for this type of weaknesses but it can make sense when reviewing programs on a source code level. As we do not want to do that, we keep this kingdom out of our schema. In contrast, *code quality* is a kingdom we do not see any use for, not even in the form Tsipenyuk et al. use it. Obviously every weakness is somewhat based on poor code quality, with the exception of errors related to the last kingdom, *environment*. Anyway, due to the fact that we target the software as a black box it is irrelevant how a vulnerability was introduced. This is something developers of software have to think about, not forensic investigators. Finally, *encapsulation* is again more a concept that gets violated by a specific vulnerability. Often one does not know which parts of software should be encapsulated from each other by just looking at them, but when a flaw in the handling of specific user input is found an attack can be crafted to violate the concept. Certainly, the exact implementation details of attacks are too fine-grained to be covered by our schema.

At first sight it might now seem that the *kingdom taxonomy* is not suited at all for our needs, having just a single kingdom that is directly adoptable. The next section will however show that most of them are found in the schema tree we develop, although they might not be directly visible. Their basic ideas are still valid and well-researched concepts we can build upon.

3.2 Construction of a New Schema

As already said, the goal of the schema presented in this thesis is to guide **users** of forensic **software**. It categorises different types of **input** to help investigators structure their search for possible vulnerabilities. Before coming to the input part the root level of the schema is created. Figure 3.2 shows two general possibilities of analysing a forensic software tool, which we define as schema base, or *root node*. Although we want to focus on a black box approach it is important to keep in mind that source code analysis is in theory also possible and can at least be applied to open source forensic software. The node is in some way a renamed version of the *code quality* kingdom, but not entirely as quality aspects also play a part in the the black box branch of the schema.

Looking at source code analysis one can basically do two different things. A good start can be to automatically analyse the source code with tools like RATS –the **R**ough **A**uditing **T**ool for **S**ecurity [Che09], which is chosen as an example because of its open source availability. Automatic analysis tools are quite strong in finding

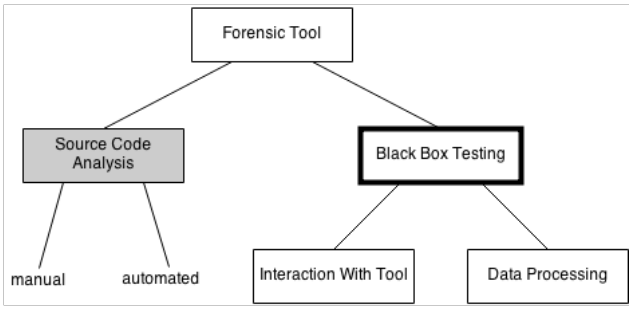

Figure 3.2: The root level of the schema tree.

memory errors through *use after free*, use of uninitialised variables, use of deprecated or unsafe functions and similar vulnerabilites. They are however still quite error-prone and automatic analysis should always be accompanied by a manual code review [OWA08]. For a deeper insight into this topic the reader is advised to have a look into the OWASP Code Review Guide cited above, which will likely be available in an updated version as of January 2014.

By only reviewing source code, one has no possibility to detect vulnerabilities belonging to the *environment* kingdom. These errors only get introduced at compilation time and are found during a dynamic analysis of the software. As one has often to deal with commercial software we call that branch *black box testing*, but it is of course also possible to perform dynamic analysis with knowledge of the underlying source code. The further subdivision of the schema, depicted in Figure 3.3, is not affected by this difference. Dynamic analysis can, as well as source code analysis, be done both automatically and manually. Therefore, from this level on the schema focusses more on different input types the software can have. During the implementation in Chapter 4, we also present an automated approach called fuzzing.

The two main types of relevant data inputs into forensic software are data generated by interaction with the tool on the one hand and data put in to be processed by the software on the other hand. The latter is the more interesting type because it is under the control of potentially malicious subjects, so we will start with a description of interactive data before turning to the data processing. In general interactive usage can both be intentionally malicious or indadvertently damaging. It is something only the investigator should have access to. Software errors found in this area are likely to belong either to the *time and state* or to the *security features* kingdom. This is because when interacting with the software it is very important that it correctly manages all kinds of access-right related functions, such as secure login and different user accounts. It is advised that forensic software is only run in trusted laboratory environments. However, software used during live forensic analyses must inevitably run in dangerous environments. Therefore, the interactive part might not be as hard

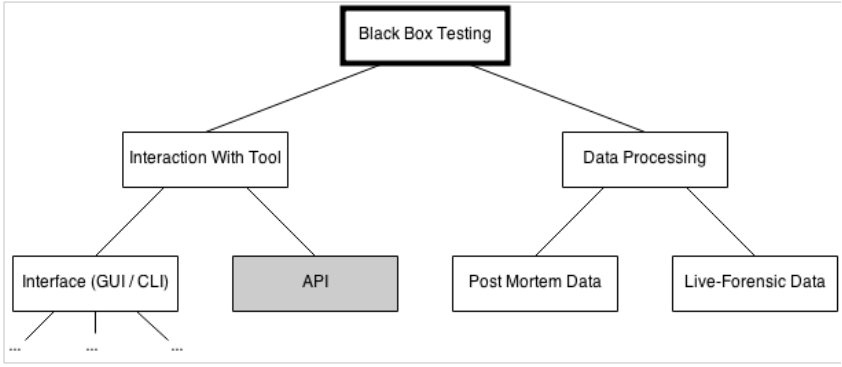

Figure 3.3: The middle level of the schema tree.

to exploit by a malicious attacker as thought, and if the attack is succesful the result can be disastrous.

One can interact with software in different ways, either by use of an API provided by the software developer or by using an interface, both graphical or textual. The API node is identical to the *API abuse* kingdom explained earlier. The node is marked to make clear that it is not a real *leaf* node. It can be subdivided further, for example to differentiate different API types, but this is outside of the scope of our schema, also because public APIs are rather uncommon in forensic software.

On the other hand there is the usage of an interface. For reasons of clarity the leaf nodes are only shown in the full schema view in Appendix C. Interaction through an interface comprises all settings that can be made, as well as all text fields, buttons, menu entries and combinations of them. The kind of the interface is most of the time irrelevant, and one must keep in mind that while textual command line interfaces accept by nature more different inputs of any kind than graphical user interfaces, these in turn contain other dangers such as the accessibility via a web browser. This part of interaction is considered in the full schema, which further divides the interface into a *remote* branch. Some software has a client-server architecture to be accessible from multiple computers in a network. Sometimes computationally intensive tasks are also designed to be run on a dedicated server machine, primarily those that are related to big data volumes. The configuration of the server is often done through a web interface, as most developers have knowledge of web programming and see it as an easy and fast way to provide graphical feedback to the user. This is in fact a dangerous idea as the *OWASP Top Ten list of Web Application Errors 2013* [OWA13] lists eight out of ten errors with a prevalence of at least *common*, the highest one being *very widespread*.

That harmless looking web application errors can lead to a complete compromise of

the whole investigation environment was shown by *DigiTrace GmbH*. There, a test of a tool believed to be a hardware writeblocker was conducted and it was found out that the device was indeed an embedded Linux system. Even worse, it had a vulnerable web configuration interface running under the super user account. After finding a command injection the investigators were able to write to storage devices containing evidence data [Wun13].

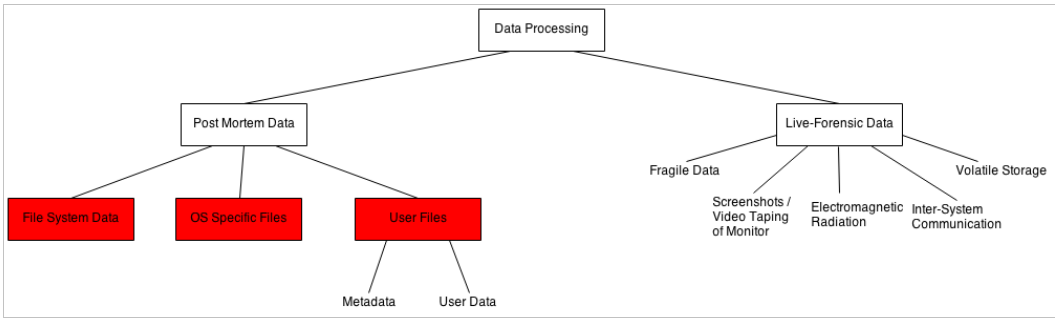

Figure 3.4: The relevant leaf nodes of the schema tree.

The right branch, *data processing*, is further refined in Figure 3.4. As explained in Section 2.1 a forensic investigation is either a live analysis, or the later analysis of loose files and disk images collected from a system. Live-Forensic data is acquired if one is interested in evidence that would change after the system has been switched off. This is for example the case if reasonable hope exists to observe the actions of an attacker that still actively uses a system. Such evidene includes data residing in *volatile storage* areas as Random Access Memory (RAM) and even processor registers. It also includes data that is indeed saved on non-volatile storage but would not easily be accessible. This *fragile data* mostly resides inside of cryptographically secure disk images (TrueCrypt, Bitlocker, ...) that are mounted in a running system. Note that RAM-disks are typically considered fragile data over volatile storage, as they are mounted as normal hard drives. In live analysis one is also interested in data that is never stored on a computer. This can be any kind of *inter-system communication*, such as network traffic (GSM, WiFi, Ethernet, ...) or communication with attached devices (over USB, SATA, Firewire, ...). *Electromagnetic radiation* of devices or the *information shown on the monitor* also belong to those never-stored materials. With the rise of new techologies, leaf nodes in this part of the schema will probably be subject to change and should be rechecked regularly.

The biggest risk of a live-forensic analysis is that the investigator must invevitably work on the "opponent's" system. One can in that case never be perfectly sure that the data is authentic. Hendrik Adam at *DigiTrace GmbH* has developed a proof-of-concept rootkit, called anti forensic rootkit (afr), to raise the awareness for that problem. The rootkit *lives* on the computer of a suspect and hijacks cer-

tain processes if an analysis is detected. Currently, for example output of false hash values as well as randomly overwriting the RAM to crash the system are supported [WA13].

A big and well researched part of forensic analyses is however post mortem data. Most forensic software that we have access to for evaluation is targeted at it, therefore we weight it the most important branch. In Figure 3.4, three nodes are highlighted. These are considered the most important nodes in the schema and in fact, Chapters 4 and 5 are based only on them. As a starting point, the next section will derive the test cases implemented.

3.3 Deduction of Test Cases

The basic problem to date is that testing of forensic software is most of the time done by only a few people in an unstructured way. As a fully automated approach would be at least complex, if not impossible, due to the high amount of tools available that all have many different functions, a semi-automated solution is presented in the following. Like for source code analysis, automation can never replace manual search. It is instead an extension to it and can accelerate the testing for basic vulnerabilities.

At first we will have a look at disk images and loose files, because they can easily be prepared not only by tool testers, but also by potentially malicious suspects.

File System Data is strictly speaking all part of data on a storage device that is responsible for making the device read- and writeable by computer systems on a very low level. In practice many different file system types are known, the most important ones being *FAT 32* for general purposes, *NTFS* for Windows systems, *HFS+* for Mac computers, and *EXT4* which is the standard file system of Linux. In our test cases we invalidate the structure of every file system, possibly causing problems for the software tools analysing them. One level below the file system one finds the so-called partition tables. These are data structures, either the older Master Boot Record (MBR) or the newer GUID Partition Table (GPT), that tell the operating system of a computer where different partitions on a file system can be found and what size they are. Because of partition tables a storage device can have more than one partition and therefore also different file systems. The test cases we develop also contain multiple examples for both MBR and GPT partition tables that are corrupted in one or another way.

The second node belonging to post mortem analysis in the schema is called *OS specific files* with the abbreviation *OS* referring to the operating system of a computer. We call all files OS specific that are not directly created by the user of a computer, but instead represent data managed by the operating system itself. Of course, this data

is different for every OS, therefore we build a test set containing different test cases for Windows, Mac OS X and Linux. For all operating systems, we have a look on the database where the file search index is stored. This is an index containing information about the folder structure and sometimes even about contents of single files, which is why it has a very high value for forensic investigators. In the case of Windows, the desktop search stores its index in the proprietary *ESE* database format. Other proprietary formats we manipulate are the *Windows registry*, where meta information about all types of system activity is stored, *event log files* containing logging information of different kind and the files managing *Jump Lists*, a feature introduced in Windows 7 and explained in more detail in Section 4.3.1.

On Mac OS X the search function is called Spotlight, and the *Spotlight Database* is of interest for us as well. Instead of Windows, where the registry is stored in only a few single files, Mac OS X manages similar settings in many small files of the type *plist* which we manipulate, too. A bit less interesting are Mac logfiles, because they have a plain text format. This is not less dangerous by definition, but it is neither encouraging to write (attackable) software for analyses if a file format can also be read manually. We manipulate logfiles nevertheless to show the dangers of parsing seemingly harmless file types.

With Linux being an open source system, even more logfiles and settings have a simple plain text format. This makes it on the one hand easy to administrate for experienced users, but hard to manipulate maliciously. Injection attacks are here still possible, but the harder to detect crashing attacks by manipulating tiny parts of complex file structures can not be applied. For example purposes we again manipulate the search database, this time belonging to the command line program *locate*, and the *.bash_history* file, where every command typed is stored by default.

The third schema leaf node, *User Files*, is also the broadest one. Thousands of applications exist to make the life of computer users easier and many of them have proprietary data formats. The situation for forensic software could not be harder, however software developers try to make their programs understand as many file types as possible, thereby introducing many additional attack vectors. Even programs targeted at a single file format often have security vulnerabilities, as has more then one time been demonstrated by the *Acrobat Reader* of Adobe Inc. [Ado13]. It is therefore very likely that errors in forensic software do exist. To find them we create a multitude of different files used by applications typically present on a computer. Among them are image file formats, music and video data, office files and different email formats. Some of them are manipulated by hand, others are automatically fuzzed.

Due to the massive amount not everything can be targeted by our example test cases. When creating own tests one can on the one hand expand the examples given in this thesis by choosing other file formats that the forensic application of choice supports. A good example are SQLite databases, which are heavily used by

modern web browsers, in Windows 8 and by smartphone apps. These three software types will probably make up a large amount of future evidence, and provide a good starting point for own test cases, as a testing of SQLite databases had to be left out due to time restrictions. When concentrating on a special software one can most certainly create manipulated files that are more complex than the ones presented, simply because every format takes some time to understand before the real dangerous exploits come to mind.

On the other hand it is possible to do more research on the leaf nodes not represented in the test case implementations. Using broken config files could crash a software program, while making contradicting settings in different places could lead to security holes. When interacting with software, it is possible that clicking through menu entries in a very special order that the developer did not think of, unwanted states in the program flow can be reached. It is also very interesting to research the effect of corrupt memory dumps or malicious network traffic logs on analysis programs.

Every aspect related to input that can be controlled from the outside can somehow be vulnerability-affected. This is why a broad adoption of the testing method and publication of every test case created will help the whole forensic community.

4 Implementation of test cases

In this chapter the creation process of our example test cases is explained. Every highlighted node from Figure 3.4 is presented in a section on its own and called a *test set*. Every test set is in turn subdivided into the *test cases* that were set up in Section 3.3. The chapter is ordered in a bottom-up manner, with Section 4.2 starting with the low-level test cases related to file systems and partition tables. Section 4.3 then covers every test case dealing with files that are managed automatically by the operating system. Finally, Section 4.4 explains the test cases related to files controlled by the computer user. Before diving into the technical explanations, some basic tools that turned out to be very helpful are presented in Section 4.1.

4.1 Tools Used During the Implementation Phase

Operating Systems

In our case, the general setup for all implementation tasks consists of a computer running *Mac OS X 10.6.8*. Also, a Virtual Machine (VM) running *Kali Linux 1.0.5* [Off13a] is used for various tasks. The choice of the distribution *Kali* is somewhat arbitrary, but it comes in useful in Chapter 5, because many open source forensic tools are already preinstalled on it. The Windows test cases in Section 4.3.1 are created with the help of a Windows 7 VM. In both cases *VirtualBox* in version 4.2.18 [Ora13a] is chosen as the virtualisation software and its *Shared Folder* feature provides an easy way to have a common workspace throughout all three operating systems.

Tools Within the Operating Systems

Apart from *VirtualBox* only the hexeditor *Synalyze It! 1.6* [Peh13] is used on the host system. A great feature making the binary view and edit of various file types easier is its support for different *grammars*. These are files describing the structure of a data format and after applying them *Synalyze It!* offers the possibility to highlight and directly manipulate the different fields of a file. Figure D.1 in the appendix shows an example usage of the PNG grammar to highlight the header checksum.

Inside the Linux VM the following command line tools are used. For further information on them the reader is advised to consult the corresponding manual pages.

dd This tool is used for low-level copy operations on files. If the option `if=FILE` is provided, dd uses this file to read from. If `of=FILE` is present as an option, the output of the command is stored there. Otherwise, standard input and output are used. The options `bs=BYTES` and `count=SECTORS` specify how big the sectors to be read at a time are and how many of them are to be processed. `skip=SECTORS` and `seek=SECTORS` can be used to determine the offset into the files, where the former relates to the input file while the latter describes the output. [RMK11]

dc3dd *dc3dd* is a patched version of *dd* containing various enhancements. The most important ones are the support of using a repeated pattern as input via `pattern=HEX` respectively `textpattern=TEXT`, and a live command line output informing about the progress [Kor11].

xxd *xxd* provides a simple, non-interactive hexdump of its input. For our purposes it is most helpful to use it for displaying the output of a *dd* command to view small pieces of a file [Wei98].

mount *mount* is used to mount a file system specified by the first parameter to a folder inside operating system, specified by the second parameter. We take advantage of the fact that *mount* automatically creates a loopback device and mounts it when given a file instead of a block device. Via `-o`, one can specify options to modify the mount process. `ro,noload` is used to mount the file system read only and with the `debug` option useful information is written to the system log.

mkfs.* *mkfs.** are multiple tools to create file systems. In our case `*` is either `vfat`, `ntfs`, `hfsplus` or `ext4` [Hud12] [ARSS12] [App] [Ts'12].

fdisk *fdisk* is used to create MBR partition tables. When used with only a device as input parameter, *fdisk* asks interactively for the configuration options. If `-o` is provided as option, a list of the current partition table on the device is printed to the standard output [fdi].

gdisk *gdisk* is the version of *fdisk* for GPT partition tables. It is also mainly used in an interactive version, but the variation *sgdisk* exists, which has various command-line options and can be used for automating tasks. In this thesis, we mainly use *gdisk* [Smi12].

Another tool, *radamsa 0.3*, is also used on the Linux command line, but instead of the other tools that are more or less typical for Linux systems, it is somewhat special. *radamsa* is a general purpose fuzzer developed at Oulu University. More precisely, it is a collection of different mutators that are applied on an input file. Appendix E shows the output of `radamsa -l`, which lists all available mutators. If none is given, *radamsa* chooses one of them on its own. The same holds for the mutation patterns,

which mutate once or many times. The latter can either happen scattered over the input or in a very close range. The developer of *radamsa* states that it is best to let the program choose all mutator options automatically, with the options shown in Listing 4.1. In that example, *radamsa* creates 100 different output files, each time choosing a random file from the *sample* directory as a starting point and numbers the output from 1 to 100 (through the use of %n). For logging purposes, we also specify the `-M path/to/meta.log` option to obtain detailed information about what happens during a program run. The choice for a fuzzing tool fell on *radamsa* because it is usable with any file type and provides good success rates at the same time. In the last two years, at least 26 vulnerabilities in major software projects were found by using *radamsa* [Hel13].

```
1 # radamsa -o output-%n.foo -n 100 samples/*.foo
```

Listing 4.1: The recommended use of *radamsa*, as stated in the FAQ on the project homepage.

4.2 Test Set 1: File System

This section groups all test cases related to low-level manipulations of a disk image. As file systems and partition tables are complex data structures, it is hard for forensic software to interpret them correctly. The provided test cases are aimed at checking wether the support is forensically sound or just a quick implementation with incorectnesses in the details. If not otherwise stated the work on all disk images starts with the creation of a zero-filled file of 100 MiB size by using *dd* as shown in Listing 4.2. The file may seem quite large but it is reasonable to do so because the file system algorithms are optimised to work on typical storage devices, which today have at least 265 MiB in case of a (very) small USB stick. As the file mostly contains zero bytes it can very efficiently be compressed after the work is done.

```
1 # dd if=/dev/zero of=imagefile.dd bs=1M count=100
2 100+0 records in
3 100+0 records out
4 104857600 bytes (105 MB) copied, 5.70686 s, 18.4 MB/s
```

Listing 4.2: The creation process of a blank disk image file.

After creation, the file is then manipulated somehow to either introduce a benign partition table or simply a single file system. Instead of viewing it in a hexeditor, a combination of Linux commands can be used to have a quick look at single interesting sectors inside the file, as shown in Listing 4.3. It is important to also fill the file systems with sample data that should be found by a software program analysing the image file. Otherwise, malfunction would be very difficult to detect. The sample data has to be reasonably benign, such that a cross-triggering of vulnerabilities in

the forensic software is made very unlikely. For that reason we decide against online reference sets of data and create our own sample files that only contain plain text.

```
# dd if=image.dd bs=512 skip=4711 count=1 | xxd
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.000558177 s, 917 kB/s
0000000: eb58 906d 6b64 6f73 6673 0000 0201 2000 .X.mkdosfs.... .
0000010: 0200 0000 00f8 0000 2000 4000 0000 0000 ........ .@.....
0000020: 0020 0300 2806 0000 0000 0000 0200 0000 . ..(...........
0000030: 0100 0600 0000 0000 0000 0000 0000 0000 ................
0000040: 0000 29c7 e603 bb20 2020 2020 2020 2020 ..).... 
0000050: 2020 4641 5433 3220 2020 0e1f be77 7cac FAT32 ...w|.
[...]
```

Listing 4.3: In this example, *xxd* is used to visualise sectors inside a disk image, read out by *dd*.

4.2.1 Directory Loops

The concept of choice to manipulate the various file system types are directory loops. This is a manipulation technique that has already been described by Newsham et al. [NPSB07], as well as by Wundram et al. [WFM13]. The latter work was conducted in early 2013 and shows that the issue is up-to-date. Figure 4.1 depicts the most simple

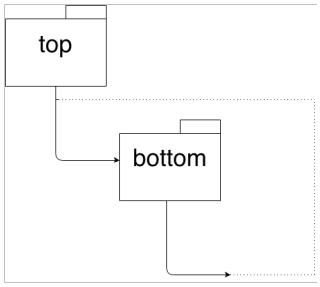

Figure 4.1: A schematic view of a directory loop.

example of a directory loop that can be created. The on-disk structure of the file is manipulated in such a way that the data structures belonging to the *bottom* directory point back to its parent, the *top* directory. As all modern file systems disallow the creation of such *directory hardlinks*, we use a hexeditor to directly modify the data by hand. A detailed description of the task, different for all file system types, is given in the following.

FAT32

FAT is the oldest of the file systems covered, originally designed for Microsoft DOS. It is extensively covered in "File System Forensic Analysis" by Brian Carrier [Car05], where all basic information in this section is taken from.

The on-disk layout of FAT consists of three parts, a *reserved*, a *FAT* and a *data area*. Figure 4.2 shows a simplified version of this layout, omitting unnecessary details.

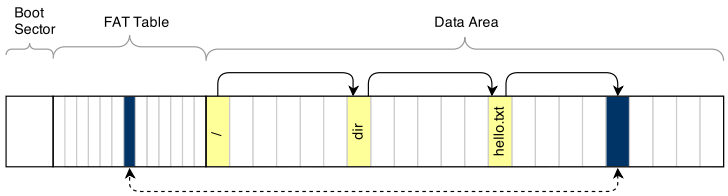

Figure 4.2: A simplified overview of the FAT file system.

Inside the reserved area the boot sector is found that describes the location of the *root directory*, typically at the beginning of the data area. The data area is separated into *clusters* and every cluster has an entry in the *FAT*, which is an abbreviation of *file allocation table*. The size of the FAT entries depends on the version of the file system used. In this thesis we use *FAT32*, resulting in an entry size of 32 bit. Directories and files are represented using so-called *directory entries*, which basically group metadata such as timestamps, attributes and names together with a cluster number pointing to the content of this entry. In Figure 4.2 we can see the root directory that points to the *dir* directory entry, that in turn points to the directory entry for a file *hello.txt*. This entry now points to a darker coloured cluster somewhere inside the data area. At this point the FAT becomes relevant. The number in a directory entry only shows the starting cluster. Bigger files possibly consist of more than one cluster, in which case the FAT entry belonging to the starting cluster points to the next cluster in the chain, whose FAT entry again points to the next cluster and so on. The final cluster however is marked by writing a value greater than 0x0fff fff8 in the corresponding FAT enry.

After having figured out the internals, a directory loop can easily be created. At first, the empty image file *FAT.dd* prepared as in Listing 4.2 is populated with a FAT32 file system via `mkfs.vfat -F 32 FAT.dd`. It is then mounted and the directories */top/bottom* are created, together with two evidence text files. The first one is called */top/cotton.txt*. It contains the string "evidence_top" and its name is chosen because it is alphabetically sorted behind bottom, which will contain our directory loop. Forensic software not correctly handling the loop could miss this file. The second file is named */top/bottom/evidence.txt* and has "evidence_bottom" as content. It tests

if a forensic software tool detects the cluster containing it. As we now rearrange the pointer structure one must follow from the root directory to reach the file, it can not be read out directly and is technically deleted. It may be that only clusters marked as unallocated are searched for deleted data by forensic software, which would in that case not detect our evidence.

Reading out the first sector with the command from Listing 4.3 we obtain a sector size of 512 B and a cluster size of 1 sector. The start of the FAT and the beginning of the data area, which is in sector 3184, can as well be found. With the same command, sector 3184 containing the root directory can now be read and the cluster address of the top directory is found to be 0x0003. This corresponds to sector 3185 (a cluster number of 1 does not exist, 2 is the root directory in sector 3184), where we find another directory entry for the bottom directory, pointing at cluster 0x0004. To change this value, we compute its byte offset as $3185 \cdot 512 + 122 = 1630778$, the latter value being the offset of the cluster number in a directory entry. Finally, we can open the *FAT.dd* file in a hexeditor, navigate to the byte offset and change its value to 0x0003, the cluster address of the top directory. When the image file is now mounted, one can navigate indefinitely into the directory structure *mountpoint/top/bottom/bottom/bottom/...* The directory loop has succesfully been created.

As FAT offers an extremly easy possibility for a second test case, we also describe it very briefly. In another FAT32 file system, we only create a small text file, containing the text "This is not very big". Instead of letting the corresponding FAT entry point to an end-of-file value, we change this value and point it to itself, thereby creating a never ending FAT loop. To further support the impression of having a very large file we also change the metadata part describing the file size to the maximum value, which is 4 GiB.

NTFS

The New Technology File System (NTFS) file system is, compared to FAT, much more complex. Again, Brian Carrier [Car05] provides great information about the on-disk structure, based on research of the *Linux NTFS Project*. However, one should note that not all details must necessarily be correct, as the specification of NTFS is not public and so, all research is based on reverse engineering techniques. Nevertheless, the free reimplementations available are today considered robust enough for productive use.

Fortunately, to implement a directory loop only a basic understanding of the structure is needed. Figure 4.3 shows a simplified version of the on-disk layout. In contrast to FAT, NTFS only contains a huge *data area*, separated into clusters. Every data in NTFS is a file and as such described by a *file record*. All these records are grouped together in the so called MFT –the **M**aster **F**ile **T**able. The MFT also is a file and

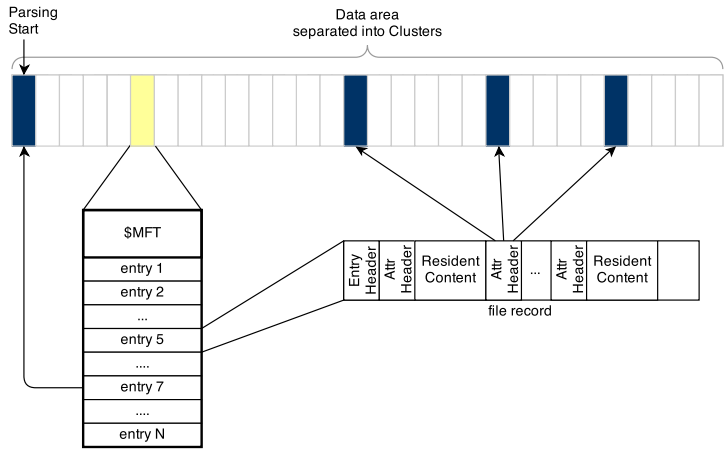

Figure 4.3: A schematic overview of the NTFS file format.

therefore contains an entry for itself. As all meta files needed by NTFS, its filename starts with a $-sign. To have a starting point into the structure, NTFS contains a file called *$BOOT*. It is always located in the first sector of the file system and contains valuable meta information, such as "Bytes per Sector", "Sectors per Cluster" and "Starting Cluster of MFT". After having located the MFT, the file records can be read out. These are structures consisting of an *entry header*, and multiple *attributes*. Every attribute itself has an *attribute header*, specifying that it is either *resident* or *non-resident*. A resident attribute is very small and therefore resides directly inside of the file record. Non-resident attributes instead are the larger ones. Their attribute headers contain pointers to the clusters containing the attribute content. Typically, this holds at least for the *$DATA* attribute, containing the actual file content.

To make things more complicated, a file record containing the information for a directory often has the two attributes *$INDEX_ROOT* and *$INDEX_ALLOCATION*. These contain a lookup tree for files and sub directories to improve access time, which was one of the biggest drawbacks of FAT. Interested readers are pointed to Carriers book [Car05], where all these structures can be studied in detail. Their coverage is out of scope of this thesis.

To practically implement a directory loop, we prepare the image file as described in Section 4.2.1, with the difference of the file system being NTFS. We then make use of the command *ls* as shown in Listing 4.4 to find out the relevant "inode numbers", which happens to be its MFT entry number on NTFS file systems.

```
# ls -lai /mnt/NTFS/top/
total 5
64 drwxrwxrwx 1 root root 248 Dec 9 14:22 .
5 drwxrwxrwx 1 root root 4096 Dec 9 14:22 ..
65 drwxrwxrwx 1 root root 160 Dec 9 14:22 bottom
66 -rwxrwxrwx 1 root root 13 Dec 9 14:22 cotton.txt
```

Listing 4.4: Usage of *ls* with the option -i to find out the inode number.

With knowledge of these numbers we can read out the file record 64, belonging to the *top* directory. This file record contains the aforementioned lookup tree structure and as expected, one of the entries therein points to file record 65, belonging to the *bottom* directory. With use of a calculator to compute the byte offset and a hexeditor for editing, we can now change this value to point back to its own MFT entry, thereby creating an indefinite loop.

In this section, only the most basic parts of NTFS were dealt with. With more research and time it is very likely to find other *exotic* features that can be manipulated, for example looping directly into a lookup tree. Also, MFT entries are to date always found to be 1 KiB in size. However, there is no restriction for them to be so and the size is directly stated inside the first sector of the file system. By patching a version of *mkfs.ntfs* to use a different size, one could trick poorly implemented versions of NTFS into not finding all entries or even crashing due to unexpected data in read operations.

HFS+

The complexity of HFS+ is comparable to the one of NTFS. HFS+ is used by all recent Apple computers. As Mac OS X is partially built on open source software, Apple has released some of the source code, including the HFS part [App13]. At the same time, a technical note explaining the format and giving advices about implementing it has been published [App10a], which is the main source for the following recapitulation of the internals.

After 1024 unused byte, an HFS+ partition starts with the *volume header* containing meta information on the file system, just as in the case of FAT and NTFS. In HFS+ the most interesting informations are the *blocksize*, which in our example setup is 4 KiB, and information on the size and the location of five *special files*. These are called the *catalog*, the *extents overflow*, the *allocation*, the *attributes* and the *startup* file and they provide most information needed to run manage the file system. For our purposes, the catalog file storing information about the directory tree is the most interesting one. From the volume header we find that it starts in block 202 of the filesystem and we compute its byte offset as $4096 \cdot 202 = 827392$.

The inner structure of the catalog file again is a balanced tree format, slightly comparable to the lookup trees found in NTFS folder attributes. However, the catalog file contains a single tree for the whole folder structure and can therefore become much bigger. In our known test case example with two folders and two files, it is still quite small, which dramatically increases its readability. The first block of the catalog file contains the so-called *header node*, which is the root of every balanced tree found in HFS+ (there are a few of them in different places of the file system). This node contains solely meta information, stating that a single leaf node containing data follows. Inside this node, occupying the second block of the catalog file, there is nevertheless a whole tree-like structure on its own. A schematic overview of it is given in Figure 4.4. As one can see, most of the information seems very similar and

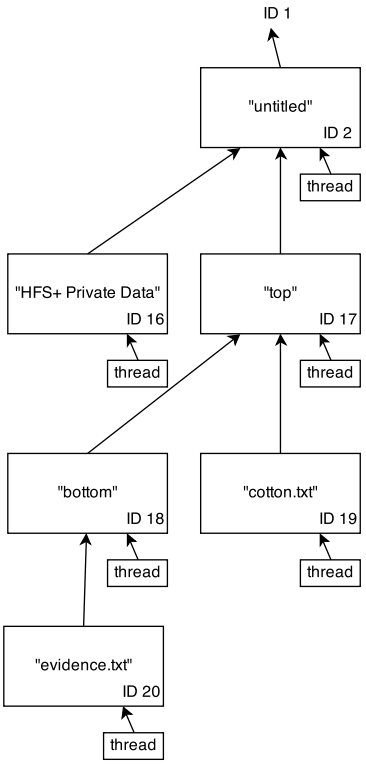

Figure 4.4: A graphical view of the HFS catalog file.

just a few additions are made. Every of the entries is called a *record* of different form. They are identified by an *ID* value and instead of pointin downwards as one would maybe expect, the lower nodes contain the idea of their parents, thereby pointing upwards to the root. The highest level node, the root directory, has a dummy ID of 1 in its parent field. The *thread records* contain only the name of the node they belong to, its ID, and t he ID of the parent of this node. They are used by the file

system to efficently traverse the tree structure.

To create a directory loop, we locate the record entry of the *bottom* folder as well as the one of its corresponding *thread record* and compute their byte offset from the start of the file system. In our example, the values are 0xcb1e4 for the bottom folder record and 0xcb364 for the thread record. Finally, we change the ID value of the bottom folder from 18 to 17 in both records. At that point we encounter the first irregularity in different file system driver implementations. When the image file is mounted in Linux to test the directory loop, *ls* only returns an "input/output-error" and "hfs: walked past end of dir" is written into the system log. However, no further information on the error can be found inside the short time frame of the thesis, not least because when mounted inside the Mac operating system of our host computer, the image file behaves as expected and presents a directory loop both on the command line and in the graphical file explorer. Interestingly even those implementations seem to work different as the command line shows the loop we already know from FAT and NTFS, where the contents of *bottom* are shown again when accessing the bottom folder. The graphical file explorer however shows the top folder again, right after clicking on it.

In this case, we have not only created a directory loop but we have also hidden the files *bottom.txt* and *evidence.txt* from an investigator that either uses a Linux operating system or a Mac operating system without the command line.

Ext4

The last file system to look into is *ext4*. It is the most recent one of the extended file systems created for Linux operating systems and is closely related to its predecessors *ext2* and *ext3*. The best and most complete information on the file system is found in the Linux kernel wiki, in an article written mostly by Darrick J. Wong under his username *djwong* [Won13].

When trying the usual approach as already described multiple times, one finds out that the display of the crafted directory loop simply fails in Linux and the following error message is written to the system log:

```
ext4_lookup:1376: inode #12: comm ls: 'bottom' linked to parent dir
```

After some research one finds a small patch commited to the repository of ext4 that prevents a direct link of a subdirectory to its parent [Dil12]. To circumvent this patch and nevertheless create a loop we change our test setup slightly. After having created an empty file system with *mkfs.ext4* we create two directories *top_1* and *top_2* on the root level, in each case filled with a subdirectory

*bottom_* with the respective number attached. We also create the text files *cotton_1.txt*, *cotton_2.txt*, *evidence_1.txt* and *evidence_2.txt* in similar locations as above.

In the ext4 case we want to take another approach on analysing the file system. We therefore use the command line tool *debugfs*, a debugging utility written from the developers working on the extended file system. To begin with, the inode numbers of all relevant directories are gathered with a recursive *ls* command as shown in Listing 4.5.

```
# ls -liaR /mnt/EXT4

[...]

/mnt/EXT4/top_1:
total 4
12 drwxr-xr-x 3 root root 1024 Dec 10 00:23 .
2 drwxr-xr-x 5 root root 1024 Dec 10 00:23 ..
13 drwxr-xr-x 2 root root 1024 Dec 10 00:24 bottom_1
15 -rw-r--r-- 1 root root 15 Dec 10 00:23 cotton_1.txt

[...]

/mnt/EXT4/top_2:
total 4
1977 drwxr-xr-x 3 root root 1024 Dec 10 00:24 .
  2 drwxr-xr-x 5 root root 1024 Dec 10 00:23 ..
 14 drwxr-xr-x 2 root root 1024 Dec 10 00:24 bottom_2

[...]
```

Listing 4.5: Usage of *ls* to recursively find inode numbers.

As we can see from the output, *top_1* has an inode number of 12, *bottom_1* is inode number 13, *top_2* is assigned to inode 1977 and *bottom_2* resides in inode 14. The *debugfs* command can be used with the options `# debugfs -R "stat <inodeNum>" EXT4.dd` to show information about every inode. We are most interested in the *EXTENTS* entry at the end of the output. Extents are the blocks in which the information about an inode are stored. The system is comparable to a mix of NTFS and FAT, where the inodes are equal to the entries inside an MFT, but they do not have resident attributes. Instead, every such entry points to an extent comparable to the clusters containing directory information in FAT. We find that the extents for *top_1* and *top_2* are 3510 and 3512. With the blocksize from the so-called *superblock* at the beginning of the file system (1 KiB) we can compute the byte offset of the extents block and look at them. Listing4.6 shows the output for the extent belonging to the *top_1* directory.

```
# dd if=EXT4.dd bs=1024 count=1 skip=3510 | xxd
0000000: 0c00 0000 0c00 0102 2e00 0000 0200 0000 ................
0000010: 0c00 0202 2e2e 0000 0d00 0000 1000 0802 ................
0000020: 626f 7474 6f6d 5f31 0f00 0000 d803 0c01 bottom_1........
```

Listing 4.6: The shortened hexdump of an extent block.

We find three entries pointing to directories. The first one belongs to inode 12, which is stated at offset 0x00. It has a length of 12 byte (0x04), with a filename of 1 byte (0x06). The filename follows at offset 0x08 and translates to ".", which is the current directory on a Linux system. The next entry is again 12 bytes in size (0x10) and belongs to inode 02 (0x0c), referencing it as "..", which is the parent directory. The most interesting entry starts at offset 0x18, because this is the entry pointing to the directory *bottom_1*. At 0x18, we see a value of 0x0d, which is the corresponding inode number. Finally, we can create a directory *cross-loop* by changing this value to 0xb907, the inode number 1977 in little-endian, and doing similar for the extent of *top_2*, chaging the value to 0x0c which in turn points back to *top_1*.

We conclude this section with the result that although efforts have been made to patch ext4, thereby making it more resistant to corruptions, it is still extremely easy to trick the file system into a directory loop.

4.2.2 Partition Tables

The *containers* around file systems on a typical computer drive are called partition tables. They enable a storage device to have multiple *partitions*, each of them possibly containing different file systems. Additionally, they contain extra information that helps the computer find the correct operating system on disk [Car05].

In the following, the older MBR is presented, followed by a newer technology, called GPT, which are both very relevant for forensic investigators. The latter has been standardised in the UEFI specification [Uni13], a modern firmware type used to manage the booting process of a computer. As more and more computers ship equipped with EFI firmware, GPT partitions will become even more widespread in the future. MBR instead is a historical product introduced in early DOS computers that until recently had a wide dispersion [Sen95].

Master Boot Record

In Chapter 5 of his book Brian Carrier [Car05] describes the MBR as *DOS style partitions*. We use the technical implementation details explained there and in the technical note describing PC DOS v7 [Sen95] to create loops and other inconsistencies inside the partition table.

Figure 4.5 shows an example hard disk, equipped with an MBR partition table at the start. The MBR always resides in the first sector of a storage device, using 512 B of disk space. Its largest part contains so-called *boot code* information used by computers to locate the operating system. We let this part of the MBR untouched and solely concentrate on the following partition entries. Every MBR can contain a maximum of four of those entries, thereby heavily limiting the possibilities of partitioning a

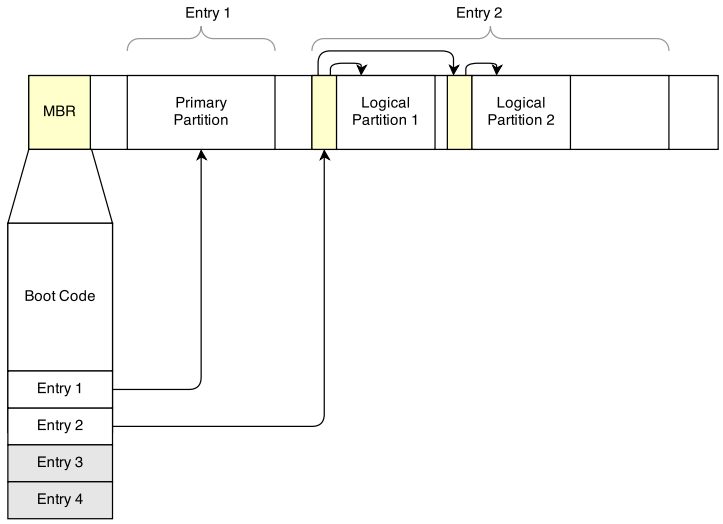

Figure 4.5: A schematic view of an example disk with an MBR partition table.

storage device. To circumvent these limitations, two types of partitions exist, which are both depicted in the figure. The first type is called a *primary partition*, shown by entry 1. Apart from that, one can also create an *extended partition*, as we see in entry 2. Both types of partitions are specified by a starting sector and a sector size, stored inside the MBR partition table. The difference is that extended partitions can internally consist of multiple *logical partitions*. These are described by *extended boot records (EBR)*, structures that are comparable to the MBR but without boot code at the beginning and with only two partition entries that can be used. The first partition entry points to the logical partition it belongs to, the second entry can be used to list the next EBR. Via this technique, one can theoretically have unlimited partitions. A few limitations are given in the specification of MBR. The first is that from the maximum number of four partition entries in the primary table, only one may be an extended partition. Second, although the EBR have the same structure as the MBR they may only use two of the partition tables, where the first one must always be a logical partition and the second one, if it exists, must be a reference to the next record.

To create malicious MBR partition tables we violate these specifications in the following ways. At first we create the loop that has already been described by Wundram et al. [WFM13]. We therefore create a partition table with a primary partition filling half of our 100 MiB sample disk image file and an extended partition filling the other half with the command line tool *fdisk*. We format the primary partition as FAT similar to as we did in Section 4.2.1, but let the formatting tool decide on the size of the FAT entries on its own. We then fill the partition with a textfile *evidence.txt*

containing the string “evidence”. In a hexeditor, we then modify the partition entry of the extended partition to have a starting sector of 0, thereby effectively pointing back to the MBR itself. A program that parses the partition table will find the partition entry for the extended partition, jump to sector 0 and search for an EBR. Due to the similarity to the MBR, a program not checking for loops in the partition table can thus be tricked in reading the MBR as EBR, continuously jumping back to sector 0 when parsing it.

In a variation of this attack we create four logical partitions inside of the extended partition. We format all of them with a FAT file system and fill them with different evidence text files. Instead of changing the offset of the initial extended partition entry we add an additional entry in the last real EBR belonging to the fourth logical partition. This entry points back to the third EBR, which effectively creates a loop around the last two logical partitions.

The third test case is more an extreme example of a valid MBR than a real corruption. It exists of a single extended partition containing 199 logical partitions in it, each one filled with a unique text file. To create the high amount of partitions the bash script shown in Listing 4.7 is used.

```
1  parted FS_PC_MBR_3.dd --script mklabel msdos
2  parted FS_PC_MBR_3.dd --script -- mkpart extended 0 -1
3  for i in {1..199}
4  do
5    parted FS_PC_MBR_3.dd --script mkpart logical $(($i * 25)) $(( $(($i + 1)) * 25))
6  done
7  parted FS_PC_MBR_3.dd --script print > layout.txt
```

Listing 4.7: The automated creation process for disk partitions.

After creation, the Python script shown in Listing4.8 is used to format each partition, mount it and create a unique text file.

```
1  #!/usr/bin/env python
2  import os
3
4  with open("partitions_prepared.txt", 'r') as f:
5     i = 0
6     for line in f.readlines():
7         i += 1
8         offset = int(line.split()[1])*512
9         sizelimit = int(line.split()[3])*512
10        print "Setting up partition %s with offset=%s and sizelimit=%s" % (i, offset,
              sizelimit)
11        os.system("losetup /dev/loop0 many_partitions.dd -o %s --sizelimit %s" % (offset,
              sizelimit))
12        os.system("mkfs.vfat /dev/loop0")
13        os.system("mount /dev/loop0 /mnt/FAT")
14        os.system("sleep 0.5")
15        os.system('echo "evidence_%s" > /mnt/FAT/evidence_%s.txt' % (i, i))
16        os.system("umount /mnt/FAT")
17        os.system("sleep 0.5")
18        os.system("losetup -d /dev/loop0")
```

Listing 4.8: The Python script for automated formatting and creation of evidence.

Finally, the fourth and fifth test case directly target the somewhat *artificial* rules of having no more than one extended partition in the MBR, as well as only a single pointer to a follow-up entry in every EBR. We handcraft two partition layouts shown in Figure 4.6 that violate these limitations. Carrier [Car10] already did some basic research on this as well and found that an additional partition entry in an EBR is mounted in both Linux and Windows, making it very easy to use even for technically unversed persons. In MBR partition tables, offsets are always given both in Logical

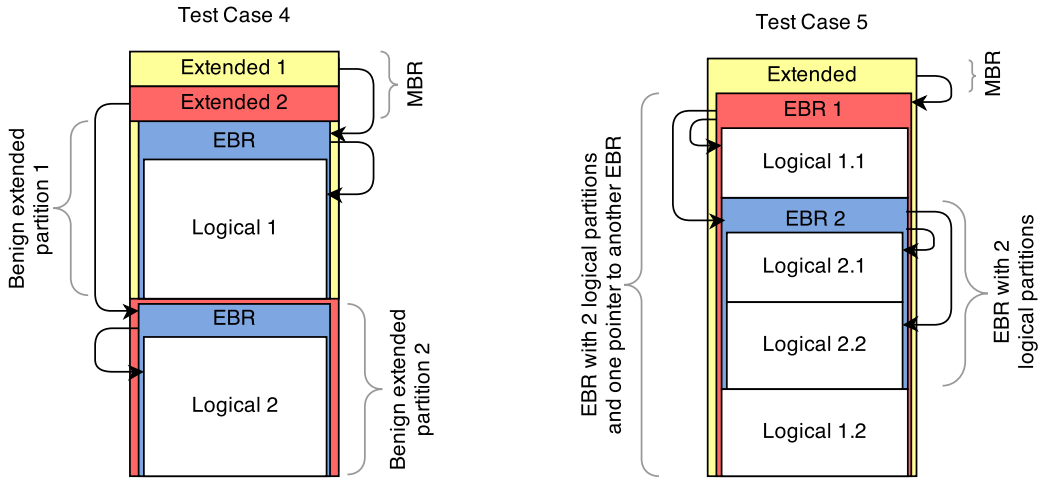

Figure 4.6: The handcrafted partition layouts of two test cases.

Block Address (LBA), that is, sectors relative to either the volume start or the start of the EBR, and in CHS values. The latter refers to cylinder, heads and sector counts and specifies a certain point on the physical hard disk using the disk geometry (heads for example refers to the different read/write heads present on a disk). Although modern computers prefer the use of LBA [fdi], we aim to make the artificial partition layouts as authentic as possible. As LBA are easy to compute but the CHS values are more difficult to generate correctly by hand, we create multiple intermediate partition layouts and use a hexeditor and *dd* to merge them.

If one wants to intensify research in the field of partition tables, it will be beneficial to patch *fdisk* in a way that it ignores the restrictions and directly creates the desired partition layouts, a task that is not difficult but considered too time-consuming for this thesis.

GPT

In modern computers, GPT is the partition table layout of choice. It has been designed many years after MBR and therefore has certain advantages, the most impor-

tant ones being the use of 64 bit LBA to support bigger hard disks and a more flexible size of the partition table, which eliminates the need for extended partitions and increases the simplicity. GPT is specified entirely in the UEFI specification, where all information used in the following section is taken from [Uni13].

In its first sector, volumes partitioned with GPT are identical to MBR partitioned volumes. This sector contains the so-called "protective MBR", which must be present to prevent programs that do not understand GPT from accidentally overwriting data. The protective MBR contains only a single partition entry of type 0xEE that indicates the whole GPT part of the disk, including both partition table information and partitions. The GPT header is located in the second sector of the disk. It specifies metadata about size and positioning of various elements and includes a checksum of itself, as well as of the partition table, to increase robustness against unintended data corruption. The following sectors are filled with the partition entries. Each one of them is typically 128 B in size, but the value is variable and must be indicated in the header. Additionally, the first LBA usable for partition data is stored there and usually has a value of 34, leading to a total of 128 possible partition table entries. Again, to increase robustness of the partition scheme, header and partition table are backed up in the last sectors of the volume in reverse order, that is, with the back-up GPT header being in the very last LBA.

Due to its flat and simple design, loops as created in the previous section are not feasible in GPT partition tables. However, there are certain specification details a forensic software should verify when analysing GPT disks. In our first test case we make use of the redundancy feature of GPT and create a volume with two partitions, each spanning 50 MiB. We then create a second volume with another partitioning and copy the last sectors containing the legacy information of the second volume over the legacy information of the first one. This leads to an image file with a contrasting primary and secondary header that specify different partition labels and locations inside the volume. A forensic tool should not only analyse the first header, but realise that the information contradicts and warn the investigator.

The second test case is comparable to the many partitions created in the third MBR example. Again, 199 partitions are created on a volume with a modified version of the shell script from Listing 4.7 that uses *gdisk*, a GPT version of fdisk, instead of *parted*. To format and fill all partitions with data, a python script comparable to Listing 4.8 is used.

At third, we modify the protective MBR preceding the GPT header. Therefore, an intermediate image file with an MBR partition table containing a partition half the size of the GPT data is created and its first sector copied over the protective MBR of the GPT image file. This leads on the one hand to a false specified size value and on the other hand to a false partition type. Tools that do not properly check for precedence of GPT could interpret such image files as being purely MBR partitioned, thereby missing information.

Test cases four and five are very similar and deal with the checksums contained in the GPT header. In one case, the header checksum is modified only in the primary header while the legacy header still contains the original value. Such behaviour could indicate an accidental corruption and should be detected by software. In the other case, both the primary and secondary value of the partition table checksum are modified. It is very unlikely that the values are modified identically if an error occurs. More probably, wrong checksums at both places indicate a targeted corruption, which is why it is even more important to inform the forensic investigator.

4.3 Test Set 2: OS Specific Files

After the forensic software has succesfully mounted a disk image it is often confronted not only with many loose files, but with a folder structure that was created and managed by an Operating System (OS). Files that are directly manipulated by the user of a computer are only a subset of this structure. In this section, we have a look at those files that are instead managed by the OS itself. These files belong to the second highlighted leaf shown in Figure 3.4 and a first introduction into them is given in Section 3.3. As the files that one encounters vary greatly between different OS, we divide our test cases into the Subsections *Windows (4.3.1)*, *Mac OS X (4.3.2)* and *Linux (4.3.3)*. Often, OS managed files have a file structure that is hard to read for humans and forensic software is used to automatically analyse them and create better readable output, for example *CSV exports* that can be imported into spreadsheet programs or *HTML reports* that are rendered and viewed inside a web browser. The work of Wundram et al. [WFM13] has already shown that the reporting engines are not always secured against injection attacks. In the following, we pick up this idea and extend it to other file formats. Additionally, we attack the analyis process itself and manipulate files such that they might possibly crash a software tool that tries to parse them.

4.3.1 Windows

As Windows is commercial software, a lot of its components use proprietary file formats that are not, or only partially, documented [Rus99]. The documentation is given in a way that developers of software are able to use certain features of the OS. Therefore, a detailed insight into the file structure is often not considered necessary. To tackle the needs of a better understanding for forensic analyses, people have started to dissect many formats, sometimes publishing the results as open source. The work of Joachim Metz is especially outstanding in this field, as he not only concentrates on a single application, but has done research on many of the file formats used in Windows [Met13a].

It is difficult to find information on how commercial forensic software implements the analysis of supported file formats, but it is likely that the developers did their own reverse engineering work, possibly building upon publicly available information. However, reverse engineering is always subject to incertainties and it might be that details are misinterpreted or simply overseen. This makes the resulting software products potentially vulnerable to attacks, which is why these components must be tested even more strictly. For Windows OS, we have a look at the file formats *edb*, *NT Registry*, *Jump List* and *evtx*. From a forensic perspective, these are very interesting, because they can contain many hidden evidence artifacts.

Windows Search

The Windows Desktop Search is one of multiple applications on a Windows computer that, since Windows XP, makes use of the Extensible Storage Engine Database (EDB) format. It uses the database to store information collected during indexing operations. Metz points out, that not only the file name and location, but extensive metadata and sometimes even part of a files content are stored inside the *Windows.edb* [Met10]. Apart from his forensic research on the particular case of Windows Search, Metz has also collected a lot of information on the EDB file format in general [Met12]. The test cases built in this section all concentrate on these general features, so that they can also be applied to forensic software that only analyses the EDB format without a deeper understanding of Windows Search characteristics. At first, we give a short overview of the file format, based on Metz' work [Met12], before explaining three manipulations of it.

EDB files start with a header that contains meta information, for example the *page size*. This value is used to split the remainder of the file into equally sized *pages* that are containers for the actual database. Each page consists of a page header of either 40 or 80 bytes, followed by the page content. Pages are organised in a balanced tree structure, comparable to those used in NTFS or HFS+. In EDB pages do not store the childnodes directly. Instead, a metadata table called the *space tree* is used. A page with childnodes has a pointer to an entry inside the space tree, where the actual values for the children are stored.

The first test case targets the level of detail with which a forensic application parses the database. In the database header, a field called *last object identifier* is specified, which indicates the index of the last page in use. By changing this value to 0, we create a simple modification that could trick badly programmed software into thinking that the database is empty.

Another possibility is to manipulate the extensive use of pointer values for navigation inside the tree structure. We analyse a *Windows.edb* with the command line tool *esentutil* which is found by default on Windows computers. We find that page 16 has a child that is specified in the space tree entry located at page 5. This

entry has a value of 15, which we change to 16, creating a loop inside the tree. Additionally, the *next Page* field in the header of page 16 is also changed to point to itself.

The previous test cases only cover a small part of the complex data structure of *edb*. In order to enlarge the coverage while not spending too much time on this single file format, the third test case is chosen to be a fuzzing of a *Windows.edb* file collected from a sample Windows machine. The command shown in Listing 4.1 is used to create many fuzzed versions of the file. Ideally, forensic investigators conducting this test case will use many different versions of *Windows.edb* found during their everyday work as input to *radamsa*. In that way it is ensured that the test is as realistic as possible.

Windows Registry

The Windows Registry is described short and clear by Mark Russinovich, an employee of Microsoft, in the first sentences of his magazine article "Inside the Registry" [Rus99].

> The Registry is the centralized configuration database for Windows NT and Windows 2000, as well as for applications. The Registry stores information about tuning parameters, device configuration, and user preferences.

The information Russinovich talks about includes installed software, connected USB devices and much more data that is of high interest for a forensic investigator. Therefore, parsing registry files is already a supported function in the major commercial forensic programs *Encase Forensics*, *FTK* and *X-Ways Forensics*. Additionally, lots of free software exists that is created especially for viewing registry files. As with the EDB file format, the structure of registry files is not completely known and must to some extent be reverse engineered. Harlan Carvey, creator of the important software tool *RegRipper*, has created a detailed documentation covering the important aspects [Car11]. It is important to note that the Windows registry is not just a single file, but a collection of multiple files found in different places on the computer. The correlation between registry paths and files is described by Sigel and Geschonneck [SG11]. For the test cases, we limit ourselves to the *NTUSER.DAT* file that holds information that is specific to a single user of a computer. In forensic investigations this file will often be the most important one.

A registry file, a so-called *hive*, contains header information, followed by equally sized *hive bins* that each contain multiple *cells*. A cell is a single element inside the registry that can be of different types. The relevant ones used during our manipulation approaches are cells of type *nk*, called "named key", and the types *lf*, *lh*, *li* and *ri* that are grouped together as "sub keys list". A typical *NTUSER.DAT* file that we dissected during our research starts with an *nk* cell as its first element. This cell points to a sub key list via an offset. The sub key cells in turn are associated to a named key, that can again contain multiple subkeys and so on. In that way, a tree structure is created that resembles the one that can be seen when the registry is viewed in the *regedit* software found on Windows computers, as shown in Figure 4.7. We find that named keys correspond to the folders seen in the screenshot which, at their final level, contain *vk* (value key) cells that represent the actual information in the registry.

In an attempt to force a denial of service the tree structure of a registry file is manipulated to create a loop. Therefore, we locate the sub key list of the first named key. We then search for the offset inside the first list element that points to the associated child named key. This offset is modified to point back to its parent named key.

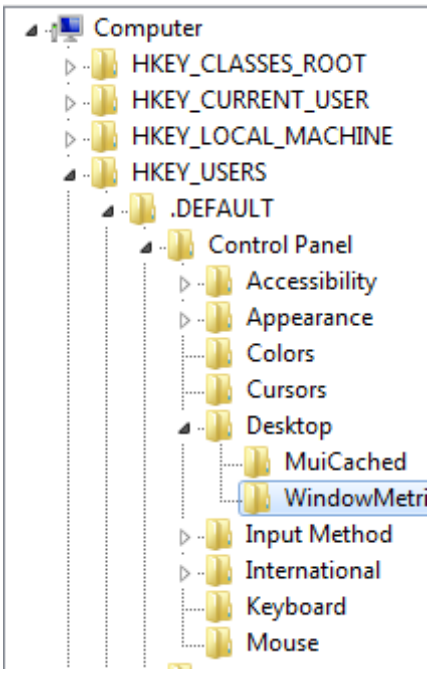

Figure 4.7: A part of the Windows Registry, visualised with the *regedit* software.

The second test case aims at crashing software by making it read more data than actually present. Named keys contain a field that specifies the *number of sub keys*, and a sub key list also contains a field indicating the *number of elements*. We modify both values of the very first named key inside a *NTUSER.DAT* to hold a value of 37 instead of the original number 11. Badly programmed software will try to read 26 more sub keys than existing and might instead encounter data that causes a program crash. If programmed correctly, the parser will check whether a read out sub key has the expected format.

As Windows registry analyses often result in reports and as Wundram et al. have already discovered flaws in exactly that feature it is important to also have a test case dealing with injection attacks. We manipulate the *sCountry* value key to contain the values `:"',;|` and space, characters that are often used in CSV exports to delimit fields from each other. We also inject an HTML comment into the *Username* value key. Finally, the value keys *sLongDate*, *sNativeDigits* and *sTimeFormat* are changed to contain a tabulator, a newline character and a nullbyte. The choice of the value keys is arbitrary but limited to values that are of interest during an investigation. When testing for vulnerabilities, one can use the search feature included in most software to locate the interesting fields.

Fuzzing is again chosen as last test case in this subsection. The complex format is a promising input to a fuzzer and the same command line options as before are used to

generate the output files. In this case, all registry files found on a Windows system are chosen as input.

Jump Lists

While search and registry are old features available for many years, *Jump Lists* are a feature that were introduced with Windows 7. Although this Windows version has been released four years ago [Sin09] the forensic work on Jump Lists is not as complete as it is for the other file formats. The forensicswiki lists some basic information on Jump Lists [Jum13a], but a detailed low level description is hard to find. Therefore, we choose a more practical approach to work with this format. On a sample Windows machine we prepare a file by visiting four websites with the *Internet Explorer* and pinning each of them to the Jump List. Jump List files are stored in a common location underneath the *AppData* folder inside a user profile. After uninstalling an application it might be that the Jump List of it remains, which is why these files can be of interest for an investigator. However, the file names are cryptic alphanumeric combinations that can only be associated to an application with hard work, for example by manipulating a Jump List and tracing the changing files inside the Jump List directory. An extensive list has been collected on the forensicswiki, built upon different sources [Jum13b]. With this list, we locate the prepared file needed for our work. Even without knowledge of the internal format, the websites are clearly visible as UTF-16 encoded strings when looking at the file in a hexeditor. For injection purposes, we manipulate these strings in the following way.

At first we inject javascript that will be triggered if a forensic software reports the contained websites as clickable hyperlinks. We test event handler injection by changing one website entry to `google.de/ onmouseover="alert(123456789)"`. Another website entry is completely replaced by `javascrip:alert(123456789)` to test for a lack of escaping the javascript pseudo protocol handler.

In a copy of the file we test for CSV delimiter vulnerabilities by injecting the characters `,;|`, along with tabulator, newline, space and nullbyte into the four website entries. We omit the characters `.` and `:`, as these are regular parts of a URI and should not cause any problems.

Finally, we decide to also fuzz this file format for a third test case, because the structure is not yet completely researched and might be parsed incorrectly. For a broader fuzzing input, we use all Jump List files that are present on our sample Windows machine.

Windows XML Event Logs

In Windows Vista the old *evt* event log file format has been replaced by the newer *evtx* format. The additional *x* in the file extension stands for XML and indicates that the new event log format is based on a binary XML structure. Again, it is a working documentation of Metz that is very helpful for the understanding and therefore used as a base for the following details [Met13b]. His findings partially based on the eventlog parser work of Andreas Schuster [Sch07]. A relevant event log file for an investigation is the *System.evtx* file that stores many kinds of system related information messages. Normally, this file grows large and becomes confusing for analysis purposes. To gain a better understanding of the file format and to produce smaller test cases we delete the logfile on our sample Windows machine and let the OS create a new one. We then use this freshly created file that only contains about 90 entries for further work.

The inner structure of *evtx* is somewhat comparable to *registry* and *edb* files. The file starts with a header and is then split into chunks. Each chunk has a size of 64 KiB and contains an array of event records, preceded by a chunk header.

The latter is subject to manipulation in our first test case. As *evtx* files normally contain multiple chunks, each chunk header contains fields for the first and last *event record number* it contains. In case of our freshly created file only a single chunk is present and the *first event record number* is 1. By swapping the fields, two things are achieved. The first number field being greater than the last number field provokes badly programmed software to ignore the event records, for example because it computes how many records have to be read by using these numbers. On the other hand, keeping the original values and only swapping them tests the ordering algorithm used for the records. If all records are parsed correctly it has to be seen whether they are ordered in a temporal ascending order or the other way round because the *first event record number* field specifies the last record to come first.

The second test case is a variation of the injection test already known from the other file formats. At the beginning of the file, `" attr='asd' />` is injected into the *xmlns* attribute. In many cases a succesful injection will only affect everything that comes after it in the file and by injecting into the first attribute the damage is as big as possible. Additionally we inject javascript in record 9 by manipulating the event message to contain `<script>alert(1)</script>\00\00`. The injection ends with two nullbytes that also test whether these can be used to terminate a message before it really ends (there is more text after the nullbytes). To further test the interpretation of special characters a newline and a tabulator character are injected into the record numbers 10 and 11.

As with all Windows file formats so far, we choose to fuzz Windows event logs as a third test case. As ususal we use the command line options from Listing 4.1 and take

all event records found on our sample Windows machine as input files.

4.3.2 Mac OS X

In contrast to Windows, *Mac OS X* is based on the open source OS BSD [App04]. Therefore, the core structure of Mac OS X is well documented and mostly consists of plain text files for configuration and logging, which is a typical aspect of *UNIX*-like operating systems. However, many proprietary additions have been built upon the core system that, if not based on open source techniques, remain mostly undocumented. In this respect Apple is even more secretive than Microsoft and avoids every information that needs not to be given. Together with the not so widespread use of Mac OS X as operating system - different sources report varying market shares that all stay below ten percent (see, e.g., [Sta13]) - this makes OS X a not so well researched OS with a relatively bad support in terms of forensic software.

The following test cases pick out some of the better researched file types and give an outlook to what one must think of when implementing support for more formats.

Spotlight Database

The equivalent of Windows Search on Macintosh computers is called Spotlight. One of the few resources on this feature has been published on the Digital Forensic Research Workshop in 2008 [JPA08]. Unfortunately, the description remains only high-level and not much information can be taken from it. The Spotlight index is stored in a file *store.db* that is located in a folder underneath the root level of a mounted volume. Each volume has its own database that contains both metadata and partial file contents.

As no detailed file format description could be found we create only two test cases of which one of them is a fuzzing approach identical to those already described multiple times in Section 4.3.1.

The other test case is an implementation of both javascript and CSV delimiter injection. During our research we found that the layout of the Spotlight database has changed in *Mac OS X 10.8*, it is now located in a folder *Store-V2*. On a test system with *Mac OS X 10.6*, the folder is called *Store-V1*. Due to a lack of availability, no information could be retrieved for *Mac OS X 10.7*. To guarantee compatibility in the future we create the injection test in the modern v2 format. An empty USB stick is filled with sample data of the digital corpora project [GFRD09]. On the root level of the USB stick, we rename a file to "<script>alert(1)</script>". The / character is replaced when listing the files in a terminal and to be sure that the injection is created correctly we implement a second variant of it. One of the folders on the

volume is renamed to "<script>alert(1)<", a file inside this folder to "script>". If forensic software lists the complete paths stored inside a Spotlight index, the injection will be pieced together from its two parts. For the *CSV delimiter* part, three folders called ,, ; and / are created on the drive. The USB stick is then attached to a Macintosh computer that automatically starts indexing it and creates the malicious *store.db*.

Binary Property Lists

Property Lists contain information comparable to the Windows registry, but instead of collecting it in a database-like structure stored in only a few files, many single files that each contain preferences of a single application or part of the OS exist. For example, our sample Macintosh computer used almost daily for two years contained 338 files under the path *Library/Preferences* inside the main users folder and 86 objects in the same path on the root level of the hard drive, where preferences common to all users of the system are stored.

Property lists are XML documents and can exist in a plain text, as well as a binary variant. The latter is a more modern version which has the advantage of reducing the overhead of plain text XML [App10b]. Macintosh computers contain a command line tool called *plutil* to convert the two different formats into each other. The test cases related to property lists target the XML parsing component and check whether the document type definition (dtd) of property lists [App10b] is considered during analysis of the file.

Therefore, we create a file that contains a *<plist>* xml element with a *<true/>* xml element inside. The header information with the XML version and a link to the dtd file are omitted. This file is considered the most minimal property list possible and should be valid in regard to the *dtd*.

In a second test case, the *dtd* is violated in multiple ways. The file *recentitems.plist* containing the recently used server addresses, documents and applications is used as a base. A *<string>* element that contains a server address is changed to hold a text that does not have the form of a *URI*. This does not violate the *dtd* as such, but can be detected only if the expected content of the file is known and checked. Additionally, other violations are created, for example an element *<true>Not so true</true>* is injected into the XML structure, violating the fact that *true* is defined as a primitive element without content. For a complete overview on the manipulations in this test case, the user is referred to Appendix F.

The third test file consists of a handcrafted property list that is very deeply nested. The file contains about 200 levels of nested *<dict>* xml elements with the element *<string>This is a dummy string</string>* in the deepest level. Software analysing this file must not crash due to the deep nesting or show the file incompletely.

Finally, a fuzzing test case is created for the binary versions of the files *recentitems.plist* and *sidebar.plist* that contains information on mounted volumes and can therefore give information on what software has been installed on the computer. This is because Macintosh software installers mostly come packaged as raw images with the file extension *dmg*, which are mounted by the OS. We consider these two property list as most valuable to an investigator and fuzz them to test the binary parsing capabilities of software that are more complex to implement than simple text parsing.

Log Files

A third important source of information on Macintosh computers are logfiles. *Mac OS X* uses an approach that is close to the one of *Linux* and stores logging information in plain text files. These naturally provide less possibilities to manipulate the file structure and have a lower potential of revealing flaws in the analysing software.

An interesting plain text test case is again an injection example. As < and > are regularly used in log entries, we omit a javascript injection and only add the characters *;,/!?*"'*, tabulator, newline and nullbyte to line 100 of a *system.log* file acquired from our sample *Mac OS* machine. Apart from the CSV delimiters it is tested whether newline and tabulator are interpreted in a log entry. This is indeed not a critical behaviour but by using them, an attacker could make the log very hard to read, so that important information could be overseen. Additionally, the use of a nullbyte could be interpreted as end of string and cause a loss of information.

The two other test cases check for a deeper understanding of the logfile format. Therefore multiple entries are reformatted such that the order of log message, issuing application and timestamp information is changed. Additionally the entries are reordered to break the temporal ascending order. At last, the sample *system.log* file is enriched with multiple lines that, when read one after another, form the texts “This is another text to see what happens if random text is inserted somewhere in the file.”, “Random text in between lines” and “Random text at the end!”.

4.3.3 Linux

While *Mac OS X* only has an open source core that for example leads to the use of plain text for the *system.log* file, the *Linux* OS is entirely open source and we cannot think of as many useful test cases as for the other two operating systems. The good readability of all configuration files and probably also the low market share of Linux have kept the coverage in forensic tools so far very low. We nevertheless

show two different approaches of manipulation. Although publicly available software does not cover these file types they are easily analysable by small scripts that a forensic investigator can write for herself to faciliate the everyday work. Such tiny programs are especially endangered because no public review of them is conducted.

Locate Database

An example of Linux file formats where automatic parsing would be very helpful is the *mlocate.db* file used by the command line search software *locate*. In contrast to *Windows Search* and *Macintosh Spotlight*, *locate* only indexes the file names without looking into their content. The *mlocate.db* file format is explained in a Linux manpage that is used as resource for this section [Trm]. The file contains plaintext and basically lists all file and folder names found during its indexing attempt. The single entries are separated by different delimiters that specify whether a file or folder entry is following. Additionally, there is a third delimiter indicating the end of a directory. All test cases are based on a *mlocate.db* acquired from the *Kali Linux 1.0.5* virtual machine used during this thesis. The folder structure mostly reflects a freshly installed OS with only few additional files.

The first test case targets the header information, more precisely the timestamp contained in it. The nanoseconds field is filled with the maximum value 0xFFFFFFFF, which clearly is greater than the possible 1,000,000,000 states that this field could normally have. Software reading out this field should check the plausibility of the analysed information and must not crash due to a buffer overflow.

The second and third test case alter the delimiters between single entries. In one test file, the byte 0x02 indicating the end of the root directory is changed to 0x00, which means that a file entry is to follow. Badly programmed software could produce wrong output because it tries to read on entries in the directory where there are none.

The other test file contains delimiters that are not specified at all. 0x03, 0x04, 0x10 and 0xFF are injected at various places in the file. For the exact locations, see Appendix F.

Shell History File

Depending on the shell used when operating on the Linux command line the typed commands can be logged in a plain text format. The *Kali Linux* used by us is configured with the standard shell *bash* that is found on most Linux operating systems. The commands are logged in the file *.bash_history* that is stored in the home folder of a user. We create a single test case for this file containing various injections.

Every printable character can possibly be injected in the file by simply typing it as a command and should therefore not cause any problems during parsing. As investigators might forget about these characters when quickly writing an analysing script they are nevertheless tested. The string *<script>alert(1)</script>* is injected into the third line of the file, one line later the characters *+;:/,.* are inserted. By manipulating the file in a hexeditor, tabulators and various other non-printable characters are added to the first two lines of the file. These are not found in regular *.bash_history* files and are more likely to be escaped insufficiently.

4.4 Test Set 3: User Files

User Files are the complement to the OS managed files from Section 4.3. They constitute the third highlighted leaf shown in Figure 3.4 and thus are the last group of input data considered in this thesis. User files are a vast group of different file formats that one typically deals with when using a computer. In this section we have a look into *multimedia files* (4.4.1) and *office file formats* (4.4.2). Additionally we look at other file types that cannot be grouped elsewhere in Subsection 4.4.3.

4.4.1 Multimedia Files

Modern life can hardly be imagined without multimedia files. Many people have at least one of *digital camera*, *smartphone* and *MP3 player*. The majority of music sold today is bought online [Gus11] and stores like *iTunes* also increase the amount of films distributed digitally. In context of home photography analog pictures became almost extinct in favor of digital files that can easily be stored on a small device instead of lots of photo albums. It is obvious that every forensic investigator will sooner or later have to deal with multimedia files. In the following subsections, we target the viewing capabilities of forensic software with different multimedia file formats. While pictures are most of the time supported, it is hard to find support for audio and video formats. At least a preview capability seems however a reasonable feature for future implementations and therefore these formats are considered by making quick fuzzing attempts.

Picture File Formats

Digital pictures can be stored in many different file formats. One of the most famous ones has the extension *jpeg/jpg*, but *png*, *gif* and *tiff* are also found quite often. The latter is a high quality format often used in printing environments and has the ability to store multiple pictures in a single file. It is also the native format in which Mac OS X stores screenshot data. The main use of *gif* is instead its

support for animated pictures. Examples range from slowly changing single pictures to files that show short video clips. The formats *png* and *jpg* are used for storing *normal* pictures, that is, those that are taken with digital cameras or entirely created digitally.

All analysed file formats store the resolution information in their file header. The first test case therefore consists of three manipulated files where the picture height has been reduced to half of the real value by changing the corresponding field in the header with a hexeditor. In a forensic context, software must not only rely on the header information but check for additional data inside the file that indicates that part of the file will remain unshown if the specified size is used.

The second test case targets the preview capability of *gif* files. We use a sample animated *gif* taken from the wikipedia [Mar] and change its metadata. In *gif*, every single frame shown has its own metadata stating size, location and display time of it. We use a hexeditor and enlarge the display time value of the first frame to a high value. The exact amount should be ten seconds, but testing has shown that the exact time varies in different viewer programs and goes up to twenty seconds. Forensic software should not only preview this first frame but instead hint the user at the animated content of the file.

The structure of *png* files is subject to a third test case. In short, *png* files consist of multiple chunks that can have many different types. All files contain the chunk type *IHDR*, where header information is stored, and one or more *IDAT* chunks that store the picture content. If multiple of those are present they are interpreted sequentially and put together when displayed. We use the chunk type *tEXt* that is not mandatory but often used to store metainformation about the file. The *tEXt* chunk has a key and a value, where the key describes the name of the meta tag. A file found by doing a google search for the *png* file format [Anob] is used as base and a "comment" meta tag is added with the command line program *ExifTool* [Har13] that is preinstalled on Kali Linux and used for viewing and editing meta information of pictures. The content of this comment is the javascript string *<script>alert(1)</script>*. A tag *additionalComment* is manually inserted into the file with a hexeditor at its end. It contains the value *hidden text;,.:"'|*, followed by a tabulator and a newline character and has a handcrafted checksum value matching the chunk content. This second tag both tests forensic software for injection and showing the presence of unexepected tags that do not conform to the usual exif tag types.

Test cases 4 and 5 deal with *tiff* files. This format is quite complex and has, as already stated, the possibility to contain multiple pictures in a single file. Each picture is described by a so-called *ifd* entry comparable to the chunks in *png*. An *ifd* entry contains the offset pointing to the next *ifd* inside the file. The fourth test case is a *tiff* file that contains two pictures showing the texts "frong" and "bottom". It is not further modified but aims at testing software for its *tiff* viewer quality.

In the last test case, this file is further modified. The offset pointing to the *ifd* entry with the *bottom* image is manipulated so that it points back to the *front ifd*. Parsing software is thus tricked into an endless loop and can crash if programmed incorrectly.

Audio and Video Files

Audiovisual media comes in many different complex formats, which cannot be analysed completely in the timeframe of a bachelor's thesis. There exit however a multitude of papers, for example the one of Thiel [Thi08], that show that fuzzing is a common testing mechanism for media player software. We follow this approach and create two fuzzing test cases with the *radamsa* software. We use different audio files of the types *mp3*, *ogg (audio)* and *wav* collected from different sources on the internet as a base in one test case, and video files in the formats *mp4* and *ogg (video)* in another one. By taking the files from the internet we assure a widespread use of different encoding algorithms, softwares and meta tags used that would be hard to reach when creating all files ourselves. It is advised to stick to a comparably large variation when reproducing the test cases at a later time.

4.4.2 Office Files

A second group of important files are office related formats. Many people use an office suite like *Microsoft Office* or one of the open source equivalents. In professional environments these are used for writing letters, doing spreadsheet calculations and many other tasks, and most private computer users also have such software installed. Another file category we talk about in this section are *emails* in various formats. The categorisation of these is a border case but we feel that they are better grouped to office files than to the next section containing otherwise ungroupable file formats. Additionally, emails are a very important evidence source in investigations conducted by larger companies that for example want to know whether corporate secrets have been sent out of the company by an employee.

Open Document File Format

The *Open Document File Format* is a file standard that was invented during the development of the open source office suite *Open Office*. It is defined in an OASIS standard [OAS11], where all information used in the following is taken from. We limit our work with the file format to *odt* files that are used to store text documents. All features from the more than 200 pages long standard can certainly not be covered, instead a variation of four different test cases is presented. Open Document files are a bundle of different XML files describing content and metadata. The files are stored in a zip container that contains the mimetype as first uncompressed element so that

the file format can be recognised. All other files can be stored in a compressed way to save disk space. The blogger Tanguy Ortolo explains how to un- and repack such container based files on his website [Ort11]. To manipulate *odt* files on a low level, we make use of this in some of the test cases.

At first we make use of a feature called *conditional text*. Conditional text is text that uses a trigger value to either be shown or hidden. This allows the creation of interactive documents. We use the software *Libre Office* in version 4.1.3.2, a variant of *Open Office* and bind the text "The secret key to my hacker admin panel is admin:admin" to a variable text value that must be set to "True" so that the sentence is shown. In every other case, the text remains unseen in the viewed document. By using this advanced feature it is tested whether the preview component of forensic software implements the complete *Open Document* standard or only parts of it.

The second test case also makes use of a feature provided by *Libre Office*, but manipulations are necessary to make it more malicious. The *floating frame* element can be used in *Libre Office* to include other documents embedded in a small, scrollable frame inside a document file. We manipulate the target of this frame in the XML structure of the file *content.xml* and let it point to ".", which is the document itself. When viewing the document in *Libre Office*, a loop occurs and the file embeds itself recursively. To multiply this effect we embed the same floating frame 17 times in the document.

Images that are not embedded via a *floating frame* but directly into the file, are included into the zip container in a subfolder called *Pictures*. The file *manifest.xml* contains a list of all files included into the container and also lists every image included in this subfolder. If a document is unpacked and repacked with an additional picture added to the folder *Libre Office* only shows a warning message and refuses to display the file. However, if it is added to the list in *manifest.xml*, the repacked document now containing a hidden image is considered valid and shown without problems. The third test document uses this fact and shows that forensic software must not only preview the document content itself but also look for additionally inserted files in the container.

The last *odt* test case is again created only by using *Libre Office*. If a picture is included into a document via drag and drop it is not copied to the aforementioned subfolder but only included through a hyperlink. The content of the picture is loaded from the internet everytime the document is viewed. In general, forensic investigations should take place in a secured laboratory environment without internet access but if an investigator only wants to take a quick look at some files she might forget to turn off her connection. In such cases the forensic software must prevent the loading of the file and only show a placeholder or a warning message that the document wants to load content from an external source. Otherwise it would remain unclear when and how forensic software communicates with the internet, which is not a flaw on its own but faciliates attacking attempts.

Microsoft Office

Since Office 2007 Microsoft uses an XML-based file format that is very similar to the *Open Document Format*. We decide to omit this file format as the test cases will likely give comparable results to those from the previous section. Instead, we take a look on the older, proprietary binary format that can be identified by the missing letter x in the file extensions.

Due to newer formats being present, the research level on the older Windows format is rather low. Microsoft has released detailed documentation on the file formats [Mic13], but these documents have an enormous amount of pages, making them useless as source for a quick dive into the format. We decide to instead use the binary nature of the format as input to *radamsa* and create a test case that contains 40 fuzzed versions of each *doc*, *xls* and *ppt* files. The original files have been taken from the digital corpora project [GFRD09].

Email

Emails are an important communication method in both private and professional environments. If software like *Mozilla Thunderbird* or *Microsoft Outlook* is used to retrieve emails from the internet they are stored on the computer. Depending on the software and the OS, different file formats are used and many store their data in plain text. One exception is the *pst* format used by *Microsoft Outlook*. The file contains a database where the complete mailbox of a single user is stored. It uses a proprietary format which is one of the lesser researched Microsoft file types. The libpst project has published a document describing its findings [Byi], but the information is not complete and subject to active development. By using this documentation we learn that *pst* files are build in a tree-like structure with nodes linking to other nodes. Making things more complicated, there are two different tree strcutures in a *pst* file called *Index1* and *Index2*. These two indexes contain different information and links exist in between both structures. In our first test case, we locate two nodes, each from one of the indexes, that describe the same tree element. Both node types contain a backpointer indicating the parent node and an offset, which is a binary address to the structure holding information about possible child nodes. By changing the backpointer to the ID of the nodes themselves and the offset to the binary offsets of the nodes we create a loop in both trees that, when parsed, should lead to an endlessly running or crashing program.

The three other formats we take a look at, *eml*, *emlx* and *mbox* are much more simple. *mbox* files store complete mailboxes for an email account by simply chaining the plain text emails, separated by an empty line. Our test *mbox* has a manipulated layout with many newlines inserted at various positions in the file. It tests the parsing capability of software which must recover all files to give forensically sound results.

The *eml* file format is used to store single email messages, for example when exporting them out of *Mozilla Thunderbird*. The file is plaintext and contains only the transmitted email message itself, without further information. From a technical perspective it is interesting that the complete email header is stored in this file type. It is interesting to filter emails by various header fields such as the receiver of the message and therefore likely that software analyses and interprets these fields. In our third test case we inject the string "<script>alert(1)</script>" into the *To:* header line and an HTML comment, commenting out the first additional recipient into the *Cc:* line.

The last two test cases deal with *emlx*, the file format for emails used by *Mail.app*, the standard email application found on Macintosh computers. *emlx* messages contain plain text messages such as *eml* and additional metadata information in form of an embedded property list (see Section 4.3.2 for more details). The start of the property list is indicated by storing the size in bytes of the original message in the first line of the file. Test case four is an email taken from our sample Macintosh machine, where this bytesize has been changed to a bigger number. Badly programmed software can therefore interpret the property list as part of the message and show wrong or irritating results.

The last test case is an example of an XML bomb [Bil03]. It is based on the assumption that parser software for *emlx* files does not implement a complete property list parser but relies on the XML information found inside the file. The link to the external document type definition is removed and the *dtd* is included internally into the file. It is manipulated to contain many entity definitions that recursively include themselves for twelve levels, resulting in 10^{12} "lol" that use 3 TB of memory space if the software deflates them completely.

4.4.3 Various

The goal of this last subsection is to group file formats that do not fit in another category. To avoid creating two sections with a single file format each, we describe compression bombs and PDF files together in here. Both formats have a high relevance and should therefore not be left unconsidered.

Compression Bombs

The concept of compression bombs is to build files that expand to a very large size, comparably to the XML bomb explained above. Two of the test cases use the typical compression format *zip*, but there is also a test case in the file type *png* that one would not associate with compression at first sight. Compression bombs sometimes use the fact that the analysing software unpacks even recursively compressed files to look at the information found in the lowest level. The attack is known from anti

virus software that was crashed in that way a few years ago and applies exactly in the same way to forensic software that almost always has the option to extract compressed files automatically.

The first compression bomb is a handcrafted zip file consisting of many layers. Therefore a single file of 4 GiB is created that contains the string "random" repeated many times. It is compressed with the zip command line tool using the best compression algorithm. The compressed file is then copied ten times and these ten copies are compressed again to a single file. Due to the redundant information the file size does almost not grow and we repeat the process six times without getting a large file at the last level. When deflating the file completely, 4 PiB of disk space are used, which should be too much for every forensic workstation in the next few years.

The second test case is created by reading from */dev/zero* on a Linux command line and piping the result to the *zip* command. A first test showed that the resulting file grew very large after only a few gigabyte of data, as *zip* could not apply the best possible compression to its input. In a second attempt we create a file *layer_ 0.zip* by piping 1 TiB of zeroes to *zip*. We zip this file that still has a size of more than 1 gigabyte again two times, creating the files *layer_ 1.zip* and *layer_ 2.zip*. At the final level we reach a file that only has a size of a few kilobytes.

The third test case is not created by ourselves but found on the internet during our research on compression bombs [AER09]. It is a crafted *png* picture showing a uniform red color with a width and height of 19,000 pixels each. The information is stored in the file by making use of the compression feature of the *png* format. Therefore, the compressed file size is only about 44 KiB but the file deflates to 1 GiB when viewed in 24-bit color mode.

PDF Files

PDF files are used whenever content is shared that should not be manipulated by its recipients. It is found as a frequent file format for publishing documentation digitally and can include features such as interactive documents to fill out surveys. In this thesis we use PDF as the format to raise the awareness of files having different types at the same time. We therefore use the files crafted by Ange Albertini [Alb12].

The first test case consists of the Linux version of the file while the second test case is made up of the Mac OS X version. As Albertini states on his website, the files are "respectively valid Windows, Linux and OS X binaries, and also a working PDF document, Jar (Zip + Class + manifest), and HTML + JavaScript files". Depending on the file extension and on the application used to view the files they behave like different file types. In our example we present the files to the forensic software as being of type *PDF*. The interested reader is refered to the other work of Albertini, who has

released a more complex example of these files as PDF slides to a talk that contain the demo material of the talk when being unzipped and can be used as a PDF viewer application to display themselves in presentation mode.

Without further testing them in the next chapter, we want to also refer the user on some other PDF security issues found during our research ([Ste09], [Val08], [Fin09], [Dec13]). The last one is the most recent coverage of PDF file issues that we could find (updated in november 2013) and it presents an extensive roundup of tools, techniques and software related to PDF (in)security.

5 Testing forensic software (Evaluation)

In this chapter the test cases implemented in Chapter 4 are evaluated by applying them to selected digital forensic software products. The exact software tools and versions are given in Section 5.1 before the test cases related to the three schema leaf nodes (see Section 3.3) are evaluated separately in Sections 5.2, 5.3 and 5.4. Section 5.5 finally summarizes the results and gives an overview on the efficiency of the implemented test cases.

A complete list of all test cases is given in Appendix F. Each test cases has a unique alphanumeric identifier of the form *A_ B_ C_ D*. The value at position *A* indicates the basic categorisation in *FS* (File System), *OS* (Operating System) and *UF* (User Files) related test cases. Positions *B* and *C* further refine the categories in the same way they are presented in Chapter 4. For example, the prefix *OS_ W_ REG* is common to all test cases that deal with *Reg*istry files on *W*indows operating systems. The last value at position *D* is used to enumerate test cases in a single subcategory. It does not necessarily reflect the order in which the test cases are presented in the sections of Chapter 4, but often this will be the case. Furthermore, a list of files belonging to each test case is given in the appendix. *File name*, *md5 hashsum* and *file size* of at least a single file making up the test case and often additional files, for example the evidence that must be found by forensic software during analysis, are listed.

The intention of a test case is explained by specifying *obligatory*, *ideal*, *alternatively allowed* and *bad* behaviours. To classify a behaviour into one of these categories the "Basic Concepts and Taxonomy of Dependable and Secure Computing" of Aviženis et al. [ALRL04] are used. In their work, the authors define, among others, the following terms. Apart from their definition we give a short explanation on what every term means in context of forensic software and how we define the relevance of every single aspect.

availability "readiness for correct service." Forensic software is able to start processing evidence and can be used for investigations. It is ready to work at the time when it is needed. Relevance: medium

reliability "continuity of correct service." Forensic software continues to process evidence once it has started and finishes the task. It does not start to behave incorrectly in the middle of an investigation. Relevance: medium

confidentiality "absence of unauthorized disclosure of information." Forensic software does not leak information from a processed case to people that are not allowed to view it, e.g., the accused person. Relevance: low to medium

integrity "absence of improper system alterations." Forensic software works absolutely sound and produces correct results. Relevance: maximal

maintainability "ability to undergo modifications and repairs." Forensic software can be updated and repaired after a crash. Relevance: medium

It is important to note that the the chosen relevance levels do not indicate how important a certain aspect is for forensic investigators. It is in contrast very important, not least from a legal perspective, that information found out during the research remains confidential. However, a forensic laboratory environment correctly secured against external access (see, e.g., [Ges11]) has a lower risk of an information leak. Nevertheless, internet access can not be ruled out completely and in that case, the risk becomes much higher. Finally, one must also not forget highly specialised malware such as *Stuxnet* that originally came into nuclear plants without internet access via USB device vulnerabilities. All in all, we rank the confidentiality level as medium. Availability, reliability and maintainability all have to do with the working state of forensic software. It is indeed annoying if a test case can successfully prevent software from working but in general this is not a critical thing, as long as extensive logging information is present that shows which parts of the evidence have been already analysed and which parts are still missing. One must however keep in mind that forensic investigations may be bound to a strict deadline at which information must be reported. In those cases a continuing crash of software can become an important factor. Integrity is the most important concept. Forensic soundness is absolutely required and correct results are crucial for the work of investigators. Therefore, integrity is given a maximal relevance.

With those definitions in mind, we chose the naming *obligatory behaviour* for all things a software *must* do. This is influenced by the *integrity* aspect and mostly states that software must correctly discover evidence aspects that are further specified. *Ideal behaviour* is an additional category on top of the obligatory part. Here, we describe what software *should* do to be a model example with well-documented behaviour. Behaviour in this category is very welcome but absence of it will not be considered a flaw. If an analysed software tool can for any reason not accomplish the obligatory part it has the possibility of an *alternatively allowed behaviour*, that is, behaviour that is present as a fallback strategy. For example, if a file cannot be analysed completely because it is altered beyond recognition software must show a warning message after terminating the analysis. At last, *bad behaviour* describes everything that software should not do when analysing the test cases. Behaviour in this category is marked with either *[c]* or *[p]*, separating it in *c*ritical and *p*otentially critical behaviour. The latter includes behaviour that does not directly lead to false results, for example wrong error messages, but could irritate an investigator and lead to wrong conclusions. Most importantly, *software crashes* are also located in

this category. These will often not be critical, but **could** be based on errors like *buffer overflows* that, when exploited more sophisticatedly, can lead to a whole new area of vulnerabilities, namely the complete compromise of the forensic software. The work of Newsham et al. [NPSB07] shows that every crash must be researched on its own with help of debugging software to find out whether it is extremely dangerous or just annoying. Such a level of detail goes beyond the scope of this thesis. Readers must keep in mind that an error located in the potentially critical category can in the worst case alter the evidence itself.

The evaluation overview in every section is given in tables correlating the results of test cases to the respective programs. We use three different types of correlation to abbreviate the detailed test case descriptions given in Appendix F.

optimal An optimal test result means that the software behaves not only correctly in terms of obligatory or alternative behaviour, but additionally fulfils the ideal behaviour part.

acceptable An acceptable test result is one that either fulfils the obligatory or alternative behaviour. Improvements need to be made to reach an optimal result.

flawed When the result is flawed, at least one of the bad behaviours specified occurs, or neither the obligatory nor the alternative behaviour are reached.

5.1 Evaluated Software

Before we can present the evaluation results we have to define which software we use to gather the results. The amount of available software is vast and we therefore take a subset of it to get a cross section through the forensic sector. From the commercial sector we test *Software A* from Company A, a company that sees itself as market leader in forensic software. *Software B* is chosen as a widespread competitor. As open source *allround* alternative, *Software C* is tested and the results are compared with command line tools as *mount* and *ls* found on a standard Kali Linux 1.0.5 installation. The choice falls on Kali Linux because it has an advertised forensic mode for use as a live system [Off13b]. However, the tests are conducted in the normal operating mode after installing Kali permanently to a (virtual) hard drive, because we are interested in the general behaviour and let the evaluation of its live system qualities to others. While we make sure to test the most recent versions of the first three mentioned softwares, the command line tools are used in the respective version found in the Debian package repositories. For some tasks with very specialised requirements, for example the ability to analyse Macintosh Spotlight databases, additional software is considered because the other competitors do not include this function. In these cases, the exact naming and version of the program is given in the corresponding subsection dealing with the test cases.

5.2 File System Based Evaluation

In this section we present the evaluation results of file system related test cases, that is, those that are identified by the prefix *FS_* . For the sake of clearness the results are presented in a graphical way in Figures 5.1 and 5.2. As one can see from the figures, different software tools are used for the subclasses of *FS_DL* and *FS_PC* test cases. This is because the open source software tools used do not possess the features to analyse multiple types of files.

When dealing with directory loops, single partition images are analysed that are understood by *tool_A* and *tool_B*, two tools that are part of *Software C*. After *mount*ing the files, they can also be viewed with the command line tools *ls* and *find*. Here, the two commercial tools only do an **average** job. *Software A* is always able to find both evidence files inside the images, but does in no case detect that the file system structure is somehow corrupted by a loop, and neither does *Software B*. The latter even misses the file *evidence.txt* in the first and last DL test case, which leads to a **flawed** result. *tool_A* is not completely comparable because it only lists meta information about the used file system. In four out of five test cases, information can be taken therefrom that hints at the **correct file and folder number** that should be found by other tools. For example, in the FAT test cases the program shows the number of used FAT entries that should be equal to the number of recovered files and folders. The companion to *tool_A* is *tool_B*, also from *Software C* and executed with the *-r* option to recursively show all files and folders inside the disk image. **Only in case of NTFS** *tool_B* is **able to list** the *evidence.txt* **file**, in all other cases it is **missed**. Not even the *notbig.txt* file from test case *FS_DL_FAT_2* is shown in the output. When viewing the mounted file systems, the evaluation criteria differ a bit. *ls* and *find* do not need to list the hidden *evidence.txt* files, because they are designed to only show files that are not deleted. *find* is executed with the option *-ls* to produce an output comparable to *ls* itself. However, both tools do not use the same code base, which can be seen in test case *FS_DL_HFS_1*, where *ls* produces an error, **warning** that it could not read the contents of folder *top*, but *find* only shows *top* **without stating that something went wrong**. In the first, third and fifth test case both tools behave **ideal** by warning the user of a directory loop and indicating its position.

	Software A	Software B	tool_A	tool_B	ls -R	find . -ls
FS_DL_FAT_1	acceptable	flawed	acceptable	flawed	optimal	optimal
FS_DL_FAT_2	acceptable	acceptable	acceptable	flawed	acceptable	acceptable
FS_DL_NTFS_1	acceptable	acceptable	unrelated	acceptable	optimal	optimal
FS_DL_HFS_1	acceptable	acceptable	acceptable	flawed	acceptable	flawed
FS_DL_EXT4_1	acceptable	flawed	acceptable	flawed	optimal	optimal

Figure 5.1: The evaluation result of the DL test cases.

For the PC test cases, other open source software is used. *tool_C* and *tool_D* are *Software C's* volume image equivalents to *tool_A* and *tool_B*. They are comparable to those and have in common that *tool_C* only lists very basic information, more precisely only the partition table type. If it does this correctly the result is considered *acceptable*. However, both *Software C* programs **fail** in the same three test cases. The first two MBR test cases implement two variations of partition table loops, which **both programs do not detect**. Instead, they run endlessly and start to slowly consume memory space. The increase in memory is indeed very low and does not reach the critical point to crash the system, even if the programs run for about an hour. In the third GPT test case, they both **fail** to detect that the GPT structure is present after the manipulated MBR header. This is a problem shared with both commercial tools. **Neither** *Software A* **nor** *Software B* **detect** that a GPT partition table is present after the first sector and instead present the MBR layout to the user. In test case *FS_PC_MBR_1* out of the first four programs **only *Software B* correctly** shows both partitions but fails to warn the user of a loop. *Software A* on the other hand presents the disk image as being completely unallocated and **is in this test case completely useless**, a behaviour that we **also** observe in the second MBR test case. Here, *Software B* even **crashes reproducibly**, producing no log message or error report at all. In all other cases the first four tools behave **at least acceptable**. *FS_PC_MBR_3* and *FS_PC_GPT_2* are the partition tables with 199 partitions each, where both *Software A* and *tool_D* can score an **optimal** result by showing all data. The behaviour of *Software B* is **acceptable**, because it warns the user that the "maximum number of internally supported partitions [is] reached" after partition 128.

Figure 5.2 reflects that we choose *fdisk* and *gdisk* as command line tools for printing the partition tables. *fdisk* only supports MBR partition tables and we therefore complement it by *gdisk*. The two programs have the same capability as *parted*, which is the last program tested. For these three programs the criteria are again a bit different. As all of them only print out partition tables they can score acceptable and even optimal results without showing any evidence file, simply because they do not support looking into the partitions. The combination of *fdisk* and *gdisk* works **nearly perfect** and achieves **optimal results in 8 out of 10** test cases. In the MBR test cases 2 and 3, *fdisk* **fails** at its limit of only showing 60 partition entries. Detecting 60 partitions in test case 2 clearly means that *fdisk* does not correctly detect the loop and tries to endlessly look at the file, but the result is still acceptable because after finishing at partition 60 the user is **warned** that the partitions are not shown in the same order as they are on disk. This, together with the displayed partition offset, makes it possible to detect the loop. The results of *parted* are not as robust as of *fdisk/gdisk*, but it can also reach an **optimal** evaluation in five test cases by giving detailed error messages correctly indicating the manipulation. Nevertheless, two test cases are only finished with a **flawed** result. In test case *FS_PC_MBR_5* the program shows a **wrong error message**, which is considered a bad behaviour. In the GPT test case 2, the one with 199 partitions, *parted* again shows a **wrong error**

message and states that "both the primary and backup tables are corrupt", probably because the increase of the maximum partition amount is not understood. The three remaining test cases are finished **acceptably**, because the error messages shown are not sufficiently clear to be considered optimal.

	Software A	Software B	tool_C	tool_D	fdisk -l	gdisk -l	file/path print
FS_PC_MBR_1	flawed	acceptable	flawed	flawed	optimal	unrelated	optimal
FS_PC_MBR_2	flawed	flawed	flawed	flawed	acceptable	unrelated	acceptable
FS_PC_MBR_3	optimal	acceptable	acceptable	optimal	acceptable	unrelated	optimal
FS_PC_MBR_4	acceptable	acceptable	acceptable	acceptable	optimal	unrelated	acceptable
FS_PC_MBR_5	acceptable	acceptable	acceptable	acceptable	optimal	unrelated	flawed
FS_PC_GPT_1	acceptable	acceptable	acceptable	acceptable	unrelated	optimal	acceptable
FS_PC_GPT_2	optimal	acceptable	acceptable	optimal	unrelated	optimal	flawed
FS_PC_GPT_3	flawed	flawed	flawed	flawed	unrelated	optimal	optimal
FS_PC_GPT_4	acceptable	acceptable	acceptable	acceptable	unrelated	optimal	optimal
FS_PC_GPT_5	acceptable	acceptable	acceptable	acceptable	unrelated	optimal	optimal

Figure 5.2: The evaluation result of the PC test cases.

5.3 Operating System Based Evaluation

The second group of test cases is the one related to operating system files, which are therefore indicated by the prefix *OS_*. Again, we present the results in two different figures because the tested software differs depending on the files that are of interest. An evaluation of the Linux based test cases (*OS_L_*) is completely left out. This is because we could not find any open source software specialised in analysing these formats and none of the commercial software tools supports an automated analysis of Linux files, at least not of system logs and bash history.

Figure 5.3 shows the evaluation result for anything related to Windows OS files. Apart from *Software A* and *Software B*, *Software D* in version XYZ on Linux, the Jump List parser *Software E* on Mac OS X, *Software F* in version ABC on Linux, a registry editor (*Software G*) on Linux and *Software H* on Windows are evaluated. From these, *Software E* is the only tool that is not open source but instead tested in a free demo version with reduced functionality in comparison to the commercial product. It is important to note that none of these missing functions limit the usability of the demo version. They are instead all comfort functions. In comparison to the FS test cases, one can clearly see that the overall result on OS test cases has a lower quality. Fewer optimal results are present and the relative amount of flawed scorings is also much higher. Another eye-catching attribute of Figure 5.3 is that no

	Software A	Software B	Software D	Software E	Software F	Software G	Software H
OS_W_EDB_1	flawed	unrelated	acceptable	unrelated	unrelated	unrelated	unrelated
OS_W_EDB_2	flawed	unrelated	flawed	unrelated	unrelated	unrelated	unrelated
OS_W_EDB_3	flawed	unrelated	flawed	unrelated	unrelated	unrelated	unrelated
OS_W_JL_1	acceptable	unrelated	unrelated	acceptable	unrelated	unrelated	unrelated
OS_W_JL_2	acceptable	unrelated	unrelated	flawed	unrelated	unrelated	unrelated
OS_W_JL_3	acceptable	unrelated	unrelated	flawed	unrelated	unrelated	unrelated
OS_W_EVTX_1	unrelated	acceptable	unrelated	unrelated	optimal	unrelated	unrelated
OS_W_EVTX_2	unrelated	acceptable	unrelated	unrelated	optimal	unrelated	unrelated
OS_W_EVTX_3	unrelated	flawed	unrelated	unrelated	acceptable	unrelated	unrelated
OS_W_REG_1	optimal	acceptable	unrelated	unrelated	unrelated	flawed	acceptable
OS_W_REG_2	acceptable	acceptable	unrelated	unrelated	unrelated	flawed	unrelated
OS_W_REG_3	acceptable	acceptable	unrelated	unrelated	unrelated	flawed	acceptable
OS_W_REG_4	flawed	flawed	unrelated	unrelated	unrelated	optimal	acceptable

Figure 5.3: The evaluation result of the W test cases.

single software supports all test cases. Admittedly, the free software alternatives are selected due to their specialisation in one area but even *Software A*, advertised as all-in-one solution, could **not be setup to work** on the *OS_W_EVTX* test cases. This is remarkable as parsing of event logs is one of the more common tasks. We can therefore not rule out that our *Software A* test installation behaved incorrectly during the tests and that other installations will be able to complete them. The **bad EDB evaluation** results for the program are based on the fact that *Software A* reports all files to be Windows email databases. As Outlook supports this form of email storage, this is not a completely random choice. However, the software fails to detect that another version of edb database is present and could therefore set an investigator on the wrong track. When dealing with Jump Lists and registry files, *Software A* does a **good job** and even detects a corruption in test case *OS_W_REG_1*. Nevertheless, it shows **flaws** when working with the fuzzed test cases and often **misses to report** obviously corrupt files. For example, **37 out of 100** files produce entirely empty output, even though they have a file size greater than zero and a hexadecimal look into the files shows that they are not fuzzed beyond readability of at least text fragments. The same behaviour can be observed in *Software B*, which otherwise works **acceptable** on registry files and does not produce display errors. The outcome of testing fuzzed event logs has to be evaluated as **flawed**, because the program interprets **42 out of 100** files as event log, sometimes **showing clearly wrong results**, e.g., only 25 displayed log entries in a file that is reported to have 2963 entries.

The result on the free software examples is mixed. In the second EDB test case, *Software D* somehow interprets the injected loop, but at least stops after having

produced twice as **much output** as on the original file. This is not critical, but also **not acceptable** without printing an error message. The fuzzed edb files only produce an **output in 10%** of all files. In these cases the output looks **acceptable**, but missing error messages and verbose output that in some cases result in over 200 MiB big text files **blur** the otherwise **good working**. For forensic purposes, *Software E* is **useless** and one better resorts to *Software A* if possible when analysing Jump Lists. *Software E* often produces empty results **without warning** and **fails** to handle strings with inserted newlines correctly. In test case *OS_W_JL_2* this results in a terminated string in the middle of a URI. Another software in a non-working state is *Software G*, which is **even worse** because the program is directly targeted at digital forensics. On our first manipulated registry file, *Software G* crashes with a **segmentation fault**, the second file is simply shown as containing no registry keys at all. In the third test case an injected nullbyte terminates the displayed string, which enables malicious users to hide data. Alone the behaviour on the fuzzed registry files is **outstanding**. 19 out of 100 files are displayed and **look good** from what we can see. Due to the deeply nested structure it is complex to give an overall evaluation. **Even more remarkably**, all other files are rejected with detailed error messages, a **behaviour that is commendable**. *Software F* and *Software H* both show **good performances** in their respective specialisations. The former behaves **best of all tools** in this category by printing warning messages that the analysed file is corrupt. The latter is more difficult to compare with the other programs due to its different design. However, none of the exporting actions was affected by our manipulated files.

	Software A	Software B	Spotlight Inspector	plist Viewer
OS_M_SLDB_1	unrelated	unrelated	acceptable	unrelated
OS_M_SLDB_2	unrelated	unrelated	flawed	unrelated
OS_M_BPL_1	optimal	unrelated	unrelated	flawed
OS_M_BPL_2	optimal	unrelated	unrelated	acceptable
OS_M_BPL_3	flawed	unrelated	unrelated	flawed
OS_M_BPL_4	acceptable	unrelated	unrelated	acceptable
OS_M_LOG_1	flawed	acceptable	unrelated	unrelated
OS_M_LOG_2	acceptable	acceptable	unrelated	unrelated
OS_M_LOG_3	acceptable	acceptable	unrelated	unrelated

Figure 5.4: The evaluation result of the M test cases.

Aside from Windows the amount of supported file formats drops quickly. For Mac OS X there are still software tools that can be found to facilitate the analysis but when dealing with Linux, investigators are pretty much left on their own. The best coverage on Mac OS X files is achieved by *Software A*. Plug-ins enable *Software A*

to handle property list and system log files of Mac OS X, so that only the Spotlight Database test cases remain uncovered. The property list plug-in works **quite good** and produces an **optimal** result in the first test case by reading the XML file **flawlessly** and warning that the binary file could not be read. Test case 2 is also processed quickly and with a **complete** result. However, the misuse of XML elements in the third test case causes a **major problem** for the plug-in. The expected structure is applied to a file in any case, which, if malicious content is present, leads to a shifted display of keys and values in the following and thereby produces **wrong output**. Surprisingly, the same result can be observed in the *plist viewer* that is preinstalled on Macintosh computers. The tested version on Mac OS X 10.6 **simply stops** displaying the file after the first unexpected element and gives the impression that no more content follows. When viewing the minimal file from test case 1, the viewer shows the content of both binary and XML version, but the file preview shows that the missing header information is inserted **without informing** the user about that. Changing evidence without notice is **unacceptable** and leads to a flawed evaluation. Although the results of the *plist viewer* are not the best we do not consider it bad software per se. One should only keep in mind that it is not intended for forensic purposes and our test shows that one does better not rely on it during investigations. The log file support in both *Software A* and *Software B* is pretty rudimentary. Both simply provide a text digest of the unreadable file format and show the entries as is. **Neither** of the programs **has problems** with injected characters of any kind, but the *Software A* log file plug-in **produces errors** while parsing the misformatted log entries from test case *OS_M_LOG_1*, in which case it tries to regroup multiple entries and ignores line breaks in between to find a format that looks as expected. This leads to an **unacceptable** result, but it is the only one along the log file test cases in both software tools. The last tool we have a look at is *Software I* of *Company I* that supports both live view and export of Spotlight databases. In test case 1, **neither** of both features is **affected** by the injected characters and the only thing that prohibits an optimal rating is a **missing warning message** that something unusual has been detected. Nevertheless, the result is worse when analysing the fuzzed databases. **36%** of them are **interpreted** as Spotlight database and the other ones are **correctly rejected** with an error message that prints the whole stacktrace of the catched exception. The live view functionality seems **largely unaffected** by the fuzzing but when exporting the analysis to a CSV document, the **output grows very large** in multiple cases. Spotlight files that have a size of less than 6 MiB sometimes produce almost 200 MiB CSV output that crash *Libre Office* when trying to view them. In that cases, something must clearly **work wrong** inside the program and we consider the test case **flawed**.

5.4 User File Based Evaluation

In Figure 3.4, the last highlighted node of file types are the user files. The related test cases, prefixed with *UF_*, are evaluated in this section. Figure 5.5 gives an overview of the results and shows that the general situation is even worse than in the previous section dealing with OS files. More than half of all results cannot be rated better than flawed. When dealing with user files forensic software tools often ask the investigator to analyse them in an external viewer application. This is why we choose the commonly used software tools *VLC Player 2.1.2* and *Libre Office 4.1.3.2* on Mac OS X, as well as *Software J* from *Company J* and *Software K* from *Company K* on Windows. The latter two softwares were found by an internet search looking for viewer applications especially designed for forensic purposes. The results however show that labelling a software *forensic* does not mean that it works forensically sound. In contrast, *Software J* is the **worst working program** tested throughout the whole thesis. It blindly shows only the reduced resolution pictures in *PIC_1* and even shows resolution metadata that has neither to do with the real, nor with the fake size of the pictures. When displaying the animated gif from the second test case, the program **only** shows a **black picture**, which is even less than the first frame of the file. In test case *UF_MM_PIC_3* it does **not show the meta fields** that are present in the file and finally, test cases 4 and 5 result in a **flawed** evaluation because the software does not even support multiple pictures inside of tiff files. One could argue that this can also be seen as acceptable in the latter test case, as the lack of multi-picture support also prevents the software from looping endlessly, but at the same time the secretely available second picture stays hidden in the file and so we decide for the **unacceptable** result. The evaluation of *Software K* is **not much more satisfying**. When trying to open the mbox file from test case *UF_OF_M_1* the software simply **shows nothing** without giving any warning that something is corrupt. In test case 2 it does show such a warning ("maximal line length exceeded"), although we do not see a reason for this message, as we left the line lengths in this test case untouched. The other three email test cases deal with formats that the software **does not even support**, a thing we did not expect from a dedicated email viewer and that we only found out after installing the program. From Figure 5.5 the results of *VLC Player* and *Libre Office* look not much better, but we want to remind the reader that their primary use case is not a forensic investigation. On the fuzzed audio files, *VLC Player* behaves **ambiguously**. On the one hand it shows a **very robust** performance on the MP3 files, where it can play every single file, but on the other hand the result is **not so good** in the case of ogg files, where meta information is often missed and no sound is played in multiple examples. This can either mean that the codec *VLC Player* uses for ogg is not as robust as the MP3 codec, but it can also mean that ogg files can much more efficiently be fuzzed to a degree at which they become unreadable. *Libre Office* in turn **misses the hidden evidence** in the ODF test cases 1 and 3. This is interesting, as we created test case 1 with exactly this program. At evaluation time, we now mount the hard disk containing the test

cases read-only, which is the reason why the hidden conditional text can no longer be seen by *Libre Office*. It does also **not show information** on a hidden file included in the document in test case 4, which is understandable because of the intended use case of the program. However, we defined this behaviour as being **unacceptable** for a forensic viewer program.

The commercial software tools *Software A* and *Software B* are designed to be as complete as possible, which is achieved by implementing the viewer component *Outside In* of *Oracle* [Ora13b]. Therefore, the results of both programs are comparable in many cases. Surprisingly, our installation of *Software A* had problems with showing metadata of audio and video files, as well as with previewing odt files. As already stated in the previous section, we cannot rule out that the installation we had access to during our research was incomplete in some way. It is in any case not unlikely, as we also had to install the *Outside In* viewer component for *Software B* additionally. However, the viewer component is not completely absent. It can display the files of the email test cases and produces **simple but robust** output that is **not affected** by any of the manipulations. The pst file format is instead processed by the forensic software itself. *Software A* here shows a **major misbehaviour** by repeatedly terminating its execution without an error message. It is indeed possible to preview the file using a text or a hexadecimal view, but the crash leads to an evaluation result of **flawed**. The behaviour of *Software B* is **exactly the opposite**. After the program is told to export emails out of archives it starts working and produces 2315 single output files. Additionally, it marks the pst file with a **warning** message, stating that it could not completely export the file and **giving more details** on where the error occured. The *UF_ V_ PDF* test cases are also displayed by the *Outside In* viewer, which **fails on both files**. The acceptable result of *Software A* is explained by the very **prominent** placing of the hexadecimal view, where the *ELF* header of the file indicating its ambiguous nature can easily be seen. *Software B* **at least** shows a file format of *ELF* on one of the two files after *refining* the evidence, an option where additional deeper analyses are done. The picture preview of *Software A* is mostly based on the viewer component, which **does not recognise** the wrong resolutions, the animated gif and the metadata in the third test case. When viewing tiff files, *Software A* previews them with only the first page visible but asks the user to open the file in an external viewer when clicking on them. On test case *UF_ MM_ PIC_ 5 Software B* crashes, which results in a **flawed** score, but warns the user in detail which file caused the crash after it is relaunched. To value this good behaviour, we mark the result with a * in the overview. In test cases 2 and 4 the software confirms the **positive impression** by both **detecting** multi-page tiff and animated gif and **flagging** them for manual review in a program that supports the formats.

Finally, we evaluate compression bombs and come to the result that they are hard to handle for forensic software. *Software A* does simply **not show anything** when trying to view the very high resolution picture from test case *UF_ V_ CB_ 1*. When decompressing the zip bomb from test case 2, it **runs for a long time** and starts

consuming a **lot of disk space**. We stopped this test manually after an hour and consider it failed. In test case 3, *Software A* does **not decompress** the zip bomb to its actual value of 1 TiB but instead only decompresses about 1 GiB. However, as it does not indicate any problem but presents the output as if it was the correct content of the zip file, this test case also results in a **flawed** evaluation. *Software B* can **successfully** show the image from test case 1, after it becomes unresponsive for a few minutes. In test case 2, it **crashes continuously** after reaching the lowest compression level and always warns the user that *random_0_0.zip*, the first file in the lowest level, should be excluded from the analysis. When excluding it via a filter, *Software B* **fails** on the next file in the same level. Test case *UF_V_CB_3* is very similar to one that has already been reported to the developer of *Software B*. This is why when extracting the third zip file test case, the software only **automatically** extracts two levels of data. When we want to manually extract the lowest level, *Software B* **warns** us that the file might be a compression bomb and **does not start** the extraction **without** our **explicit command**.

5.5 Evaluation Results

In this section we summarise the results from this chapter. Overall we found 66 cases in which the programs did **not** behave as they should. We also detected 33 cases of **ideal** behaviour. These are mostly found when analysing the file system related test cases. After having worked with all programs mentioned in this thesis we think that this distribution is not primarily based on the design of our test cases but on the varying quality of the software regarding the different classes of tests. Forensic software developers today want to include as many features as possible, so that potential customers decide for their product. They therefore use third party components as *Outside In* or open their software for community based plug-ins, which lowers the overall quality. The test cases that we created for OS and user files are hard to apply to all softwares found on the market, because the functions differ very much. Here, targeted test cases would lead to more solid results. Additionally, the choice of test cases might not have been perfect, which is shown by the difficulty to find free software alternatives for the commercial products. A low amount of available software can mean that the real world needs of forensic investigators are located at different file formats. On the other hand, all test cases are valid examples that could be found during investigations and often it is possible to hide evidence by using the presented manipulations. Therefore, the lack of software also means that more research has to be done on some file formats.

The results in Section 5.4 also show two other things. Often forensic investigators use normal software as *Libre Office* or *VLC Player* to have a look into files. By doing so, they might miss information that is hidden in special ways. If it seems likely in an investigation that a suspect has something to hide, more detailed solutions must be applied. It would for example be possible to find the hidden jpeg image in the odt

	Software A	Software B	Software J	VLC Player	Libre Office	Software K
UF_MM_PIC_1	flawed	flawed	flawed	unrelated	unrelated	unrelated
UF_MM_PIC_2	flawed	optimal	flawed	unrelated	unrelated	unrelated
UF_MM_PIC_3	flawed	acceptable	flawed	unrelated	unrelated	unrelated
UF_MM_PIC_4	acceptable	optimal	flawed	unrelated	unrelated	unrelated
UF_MM_PIC_5	flawed	flawed*	flawed	unrelated	unrelated	unrelated
UF_MM_A_1	unrelated	flawed	unrelated	flawed	unrelated	unrelated
UF_MM_V_1	unrelated	flawed	unrelated	acceptable	unrelated	unrelated
UF_OF_ODF_1	unrelated	acceptable	unrelated	unrelated	flawed	unrelated
UF_OF_ODF_2	unrelated	acceptable	unrelated	unrelated	acceptable	unrelated
UF_OF_ODF_3	unrelated	flawed	unrelated	unrelated	flawed	unrelated
UF_OF_ODF_4	unrelated	acceptable	unrelated	unrelated	flawed	unrelated
UF_OF_MSO_1	flawed	flawed	unrelated	unrelated	flawed	unrelated
UF_OF_M_1	acceptable	acceptable	unrelated	unrelated	unrelated	flawed
UF_OF_M_2	acceptable	acceptable	unrelated	unrelated	unrelated	acceptable
UF_OF_M_3	acceptable	acceptable	unrelated	unrelated	unrelated	unrelated
UF_OF_M_4	acceptable	acceptable	unrelated	unrelated	unrelated	unrelated
UF_OF_M_5	flawed	optimal	unrelated	unrelated	unrelated	unrelated
UF_V_CB_1	flawed	acceptable	unrelated	unrelated	unrelated	unrelated
UF_V_CB_2	flawed	flawed	unrelated	unrelated	unrelated	unrelated
UF_V_CB_3	flawed	optimal	unrelated	unrelated	unrelated	unrelated
UF_V_PDF_1	acceptable	acceptable	unrelated	unrelated	unrelated	unrelated
UF_V_PDF_2	acceptable	flawed	unrelated	unrelated	unrelated	unrelated

Figure 5.5: The evaluation result of all UF test cases.

file with the help of *file carving*, a technique where files are located by their header information. The other thing shown is that the detection of compression bombs is extremely difficult, if not impossible. Compressed files can by design contain huge amounts of data that is legitimate, but they can as well contain garbage data to consume resources on the investigator's computer. The example of *Software B* shows that even after implementing a compression bomb detection heuristic, not all variants of them are detected.

6 Conclusion

This chapter recapitulates the work done in the thesis. An overall explanation on what was done is given in Section 6.1. We show the advantages of having a schema where structured test cases can be derived from and summarise the results that were found out while working on their implementation. Furthermore we give ideas on additional work that infers from our work in Section 6.2.

6.1 Advantages of Structured Testcases and Summary

At the beginning of our thesis a new schema was developed. We designed it with the special needs of forensic tool testing in mind and built it around the many different types of input that can be given to a forensic software product. The goal of the schema is to be a complete overview of all possible input types, although three specific ones were selected to work with. We defined them as *File System Related*, *OS Related* and *User File Related* and chose them because they represent the for us most common type of forensic analysis, the *post mortem* analysis of data. In Chapter 4 we gave test case implementation examples for these types of input. These test cases were then evaluated in Chapter 5.

The advantage of having a structured schema as base for test case derivation over just implementing what comes to mind is shown by the ability of the tested tools to process the test files, or rather by the lack of doing so in some examples. As we chose our test cases to represent a cross-section through all different types of data we also selected file formats that are not so well supported. Our work shows in detail where the functionality of forensic software must further be improved. This is especially the case when dealing with OS and user files and more related to the allround software products as to the smaller specialised programs. When using our schema one can easily decide on the types of user input that one faces most often during investigations. Every person working in digital forensics can thereby find their own subset of test scenarios that tests his or her forensic laboratory setup in the most effective way.

As we could see in Chapter 5, flaws can be found easily in a relatively fast way when using our prepared test cases. This is not only good for investigators to do a quick survey of their software but also for developers that just start securing their software against anti-forensic risks. The earlier errors are detected, the cheaper and easier

to conduct the error correction is. Of course, our approach cannot only target new software products but also longer existing ones. Errors can exist in every phase of the lifecycle and one must always keep in mind that the absence of flaws does not provide one hundred percent security. Our test cases are designed to be a starting point. They are for example well suited to test the free software products on the market. Here, many flaws could be found, probably because not all developers have thought of anti-forensic attacks so far. Due to the broad range that we covered, every topic is only treated as deeply as it was possible in the short amount of time. When the robustness of software rises in the future our test cases need to be refined by other persons, which leads us to the next section.

6.2 Future Work

The test cases presented in this thesis provide a starting point in structured forensic tool testing. We believe that a structure is very helpful to coordinate the development of tests between multiple people. Therefore, we encourage everybody to use our schema and propose improvements to it. The more forensic investigators give input, the more realistic it will become. In terms of test cases, everybody with a deeper knowledge of a certain topic is welcome to produce more detailed ones. By sharing the test cases with the forensic community, everybody can profit and the overall quality of the discipline will probably rise.

We limited ourselves to test cases dealing with *post mortem* data. In future it will be necessary to expand the coverage to all kinds of input. A growing threat to forensic software are not only prepared test cases but also active attackers that penetrate into networks that are thought to be secure. The writeblocker example depicted in Section 3.3 underlines that even components that are designed as barriers securing the evidence might contain vulnerabilities. It is even more disastrous that the mentioned writeblocker is specifically designed to be connected to a potentially dangerous network to gather evidence from it and that it anyhow was trivial to penetrate it via this interface.

In continuation of this thesis, the flaws have yet to be reported to the developers. After this has been done, the test cases along with the schema will be presented to the forensic community. They will be submitted to NIST or CFReDS to become part of standardisation processes. Additionally, we will report the test cases to the unofficial forensic tool testing mailing list mentioned in Section 1.3 once we get admitted to it. The response from any of these actions will most likely result in additions and changes to both test cases and schema.

We conclude this section and thereby the thesis with the invitation to test more tools, especially the tools that one uses normally during work. It is specifically important to repeat the testing with every new version of a software release, because flaws can not only be patched away but also be introduced inadvertently.

Information security is a fast-growing industry and new threats that we do not yet think of now can emerge quickly. It is important for everyone to stay up-to-date, both with software and knowledge. The security mindset in the forensic community must further increase and everybody should spread the word that evidence may at the same time be helpful and dangerous.

A Acronyms

CFTT Computer Forensic Tool Testing Project

NIST National Institute of Standards and Technology

CFReDS Computer Forensic Reference Data Sets

CWE Common Weakness Enumeration

CVE Common Vulnerability Enumeration

StPO Strafprozessordnung

PLOVER Preliminary List of Vulnerability Examples for Researchers

RAM Random Access Memory

afr anti forensic rootkit

MBR Master Boot Record

GPT GUID Partition Table

VM Virtual Machine

LBA Logical Block Address

NTFS New Technology File System

OS Operating System

EDB Extensible Storage Engine Database

B Test Cases On CD-R

This CD contains all test cases that have been implemented in this thesis. The test cases are ordered in a folder structure that matches the test case identifiers. They are provided in a compressed (tar.gz) form to save space.

C Complete Schema Tree

The next page shows a full schema overview printed as big as possible. However, as the schema is very broad the tree structure is difficult to fit on a single page. For better readable excerpts, see Section 3.2.

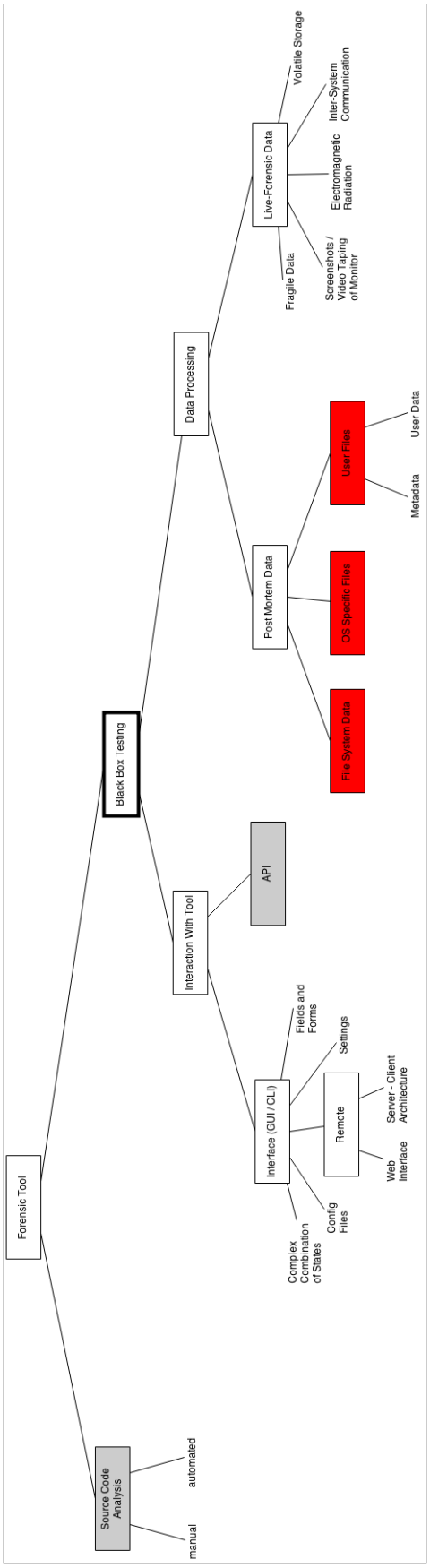

Figure C.1: A complete overview of the schema.

D Screenshots

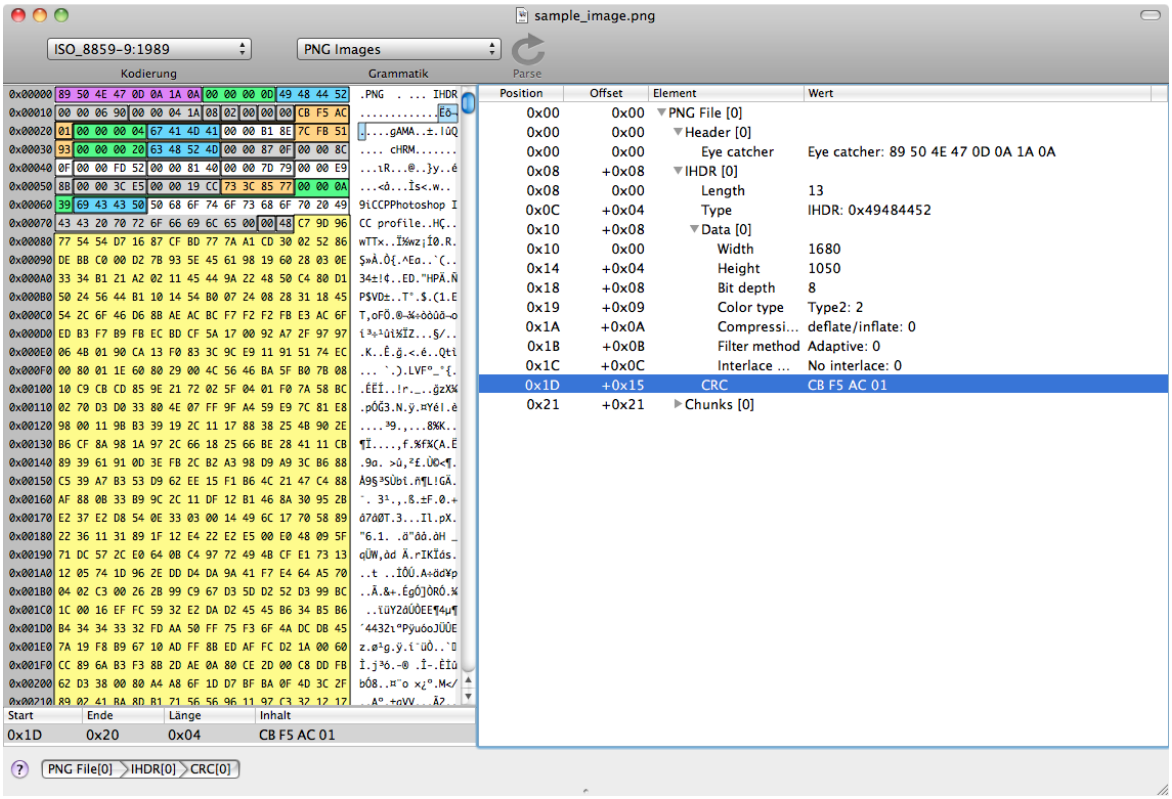

Figure D.1: The interface of *Synalyze It!*, showing the use of a grammar for PNG files.

E Radamsa Mutators

If `radamsa -l` is executed, the following list, showing options to configure the fuzzing process, is printed. If no option is given, which is the recommended use, radamsa randomly chooses mutations and mutation patterns for the input.

```
Mutations (-m)
  bd: drop a byte
  bf: flip one bit
  bi: insert a random byte
  br: repeat a byte
  bp: permute some bytes
  bei: increment a byte by one
  bed: decrement a byte by one
  ber: swap a byte with a random one
  sr: repeat a sequence of bytes
  ld: delete a line
  lr2: duplicate a line
  li: clone and insert it nearby
  lr: repeat a line
  ls: swap two lines
  lp: swap order of lines
  td: delete a node
  tr2: duplicate a node
  ts1: swap one node with another one
  ts2: swap two nodes pairwise
  tr: repeat a path of the parse tree
  uw: try to make a code point too wide
  ui: insert funny unicode
  num: modify a textual number
  ft: jump to a similar position in block
  fn: likely clone data between similar positions
  fo: fuse previously seen data elsewhere

Mutation patterns (-p)
  od: Mutate once
  nd: Mutate possibly many times
  bu: Make several mutations closeby once
```

F Test Sets

FS_DL_FAT_1

file name	md5	file size
FS_DL_FAT_1.dd	3451c9e7de48d53248b536e12b01938d	104 857 600 B
cotton.txt	b2c4c8e1c7da08143ec21ba088b3ac53	13 B
evidence.txt	e314e89d4e3ea2806be9a179f3161045	16 B

Description:
The image contains a FAT-32 file system with a directory *top*, therein a directory *bottom* and a file *cotton.txt* with text "evidence_top". *bottom* contains a file *evidence.txt* with text "evidence_bottom", but when opening the directory it points back to top.

To be used on:
Tools that read and analyse single partition images

Obligatory behaviour:
Detect files *cotton.txt* and *evidence.txt*

Ideal behaviour:
Detect loop
Scan file system and ensure that nothing was missed due to the loop
Display that *evidence.txt* probably belongs inside of *bottom*
Show correct disk usage (5 blocks, 2.5 KiB)

Alternatively allowed behaviour:
Stop processing after having reached a fixed level of depth and warn user that not everything might have been analysed
List *evidence.txt* as deleted without stating that its cluster is not unallocated
Show inconsistent size of occupied space and warn about that

Bad behaviour:
Show wrong error messages [p]
Show wrong size of occupied space without warning [c]
Crash / Become unresponsive [p]
Ignore *cotton.txt* [c]
Ignore *evidence.txt* [c]

FS_DL_FAT_2

file name	md5	file size
FS_DL_FAT_2.dd	1920368c1377a98159d6f782d8e1580d	104 857 600 B
notbig.txt	4ffd2964ee762ab57000512b61b22ed5	17 B

Description:
The image contains a FAT-32 file system with a file *notbig.txt* containing the text "This is not big.". The file's FAT table entry has been modified to point to itself, causing a loop. The file's size has been modified to be the maximum in FAT-32, 4 294 967 295 B.

To be used on:
Tools that read and analyse single partition images

Obligatory behaviour:
Detect file and show its content

Ideal behaviour:
Detect anomaly within file
Show real file size

Alternatively allowed behaviour:
Stop processing and warn that the file could be bigger than read out

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Skip file without a notification [c]

FS_DL_NTFS_1

file name	md5	file size
FS_DL_NTFS_1.dd	bc2620575af7be3638006335bedf7e1c	104 857 600 B
cotton.txt	b2c4c8e1c7da08143ec21ba088b3ac53	13 B
evidence.txt	e314e89d4e3ea2806be9a179f3161045	16 B

Description:
The image conains an NTFS file system with a directory *top*, therein a directory *bottom* and a file *cotton.txt* with text "evidence_top". *bottom* contains a file *evidence.txt* with text "evidence_bottom", but when opening the directory it points back to top.

To be used on:
Tools that read and analyse single partition images

Obligatory behaviour:
Detect files *cotton.txt* and *evidence.txt*

Ideal behaviour:
Detect loop
Scan file system and ensure that nothing was missed when stopping
Display that *evidence.txt* probably belongs inside of *bottom*
Show correct disk usage (5096 blocks, 2548 KiB)

Alternatively allowed behaviour:
Stop processing after having reached a fixed level of depth and warn user that not everything might have been analysed
List *evidence.txt* as deleted without stating that its cluster is not unallocated
Show inconsistent size of occupied space and warn about that

Bad behaviour:
Show wrong error messages [p]
Show wrong size of occupied space without warning [c]
Crash / Become unresponsive [p]
Ignore *cotton.txt* [c]
Ignore *evidence.txt* [c]

FS_DL_HFS_1

file name	md5	file size
FS_DL_HFS_1.dd	1fd0e8efa0eba5c4175c09085629daeb	104 857 600 B
cotton.txt	b2c4c8e1c7da08143ec21ba088b3ac53	13 B
evidence.txt	e314e89d4e3ea2806be9a179f3161045	16 B

Description:
The image conains a HFS file system with a directory *top*, therein a directory *bottom* and a file *cotton.txt* with text "evidence_top". *bottom* contains a file *evidence.txt* with text "evidence_bottom", but when opening the directory it points back to top.

To be used on:
Tools that read and analyse single partition images

Obligatory behaviour:
Detect files *cotton.txt* and *evidence.txt*

Ideal behaviour:
Detect loop
Scan file system and ensure that nothing was missed when stopping
Display that *evidence.txt* probably belongs inside of *bottom*
Show correct disk usage (4840 blocks, 2420 KiB)

Alternatively allowed behaviour:
Stop processing after having reached a fixed level of depth and warn user that not everything might have been analysed
List *evidence.txt* as deleted without stating that its cluster is not unallocated
Show inconsistent size of occupied space and warn about that

Bad behaviour:
Show wrong error messages [p]
Show wrong size of occupied space without warning [c]
Crash / Become unresponsive [p]
Ignore *cotton.txt* [c]
Ignore *evidence.txt* [c]

FS_DL_EXT4_1

file name	md5	file size
FS_DL_EXT4_1.dd	57587f82a2bf2589819846bf95286d06	104 857 600 B
cotton_1.txt	d4c83337f8d112afbf6705237899f70d	15 B
cotton_2.txt	170aa693f63c380355c6d7e608832e0a	15 B
evidence_1.txt	f21544abfa33976d11dbbfb63d9a15aa	15 B
evidence_2.txt	8df827c6a4e214133acfd60b0d0a0537	15 B

Description:
The image conains an EXT4 file system with two directories *top_1* and *top_2*. *top_1* contains a folder *bottom_1* and a file *cotton_1.txt* with text "evidence_top_1". *top_2* contains a folder *bottom_2* and a file *cotton_2.txt* with text "evidence_top_2". *bottom_1* contains the file *evidence_1.txt* with text "evidence_bottom_1", but instead of showing the file, the folder points to *top_2*. *bottom_2* contains the file *evidence_2.txt* with text "evidence_bottom_2", but instead of showing the file, the folder points to *top_1*.

To be used on:
Tools that read and analyse single partition images

Obligatory behaviour:
Detect the four files *cotton_1.txt*, *cotton_2.txt*, *evidence_1.txt* and *evidence_2.txt*

Ideal behaviour:
Detect loop
Scan file system and ensure that nothing was missed when stopping
Display that *evidence_1.txt* probably belongs inside of *bottom_1*
Display that *evidence_2.txt* probably belongs inside of *bottom_2*
Show correct disk usage (4840 blocks, 2420 KiB)

Alternatively allowed behaviour:
Stop processing after having reached a fixed level of depth and warn user that not everything might have been analysed
List *evidence_1.txt* and *evidence_2.txt* as deleted without stating that its cluster is not unallocated
Show inconsistent size of occupied space and warn about that

Bad behaviour:
Show wrong error messages [p]
Show wrong size of occupied space without warning [c]
Crash / Become unresponsive [p]
Ignore *cotton_1.txt* [c]
Ignore *evidence_1.txt* [c]
Ignore *cotton_2.txt* [c]
Ignore *evidence_2.txt* [c]

FS_PC_MBR_1

file name	md5	file size
FS_PC_MBR_1.dd	3c640766c54a3b570a9c8a5acae9f091	104 857 600 B
layout.txt	248d396f80da5e93c6f2a3c5324a830f	539 B
evidence.txt	fe4d232432a039cec9f8f10f71649f32	9 B

Description:
The image contains an MBR partition table containing a primary and an extended partition. The exact sector level layout is shown by *layout.txt*. The primary partition contains a file *evidence.txt* with text "evidence". The extended partition points back to the MBR itself by having an offset of 0, thereby causing a loop.

To be used on:
Tools that read and analyse volume images (multiple partitions)

Obligatory behaviour:
Detect the primary partition and show the file *evidence.txt* in it

Ideal behaviour:
Detect loop
Scan disk image and ensure that nothing was missed when stopping
Show that the extended partition is in fact empty / non-existing
Warn user that the partition table looks unusual and show it

Alternatively allowed behaviour:
Do not show empty extended partition but log it somewhere
Stop processing after a certain time / partition count and warn user that not everything might have been analysed
Show inconsistent size of occupied space and warn about that

Bad behaviour:
Show wrong error messages [p]
Show wrong size of occupied space without warning [c]
Crash / Become unresponsive [p]
Ignore *evidence.txt* [c]
Ignore primary partition [c]
Ignore extended partition [c]

FS_PC_MBR_2

file name	md5	file size
FS_PC_MBR_2.dd	8bd8e5ef075fe377618a6038099aefb8	104 857 600 B
layout.txt	cc6e45a73b1786f0151431d71fff55ca	706 B
evidence_1.txt	c71b08d050f458e01fe0ea981d3fc1a5	11 B
evidence_2.txt	55c5d4bebc55c35ed51a648bdc13cfb9	11 B
evidence_3.txt	a8dd0472fe2033ad2deb336320948a31	11 B
evidence_4.txt	3d2a965500f4da0d940e5a6c2a410634	11 B
evidence_5.txt	51a8867fe74098d7bb4dadfb817b5237	11 B

Description:
The image contains an MBR partition table containing a primary and an extended partition.The extended partition contains four logical partitions. The exact sector level layout is shown by *layout.txt*. The primary partition contains a file *evidence_1.txt* with text "evidence_1". Every logical partition contains a file *evidence_X.txt* with text "evidence_X", where X is ranging from 2 to 5. The EBR locating the last logical partition contains an additional entry to an EBR with an offset pointing back to the third partition record, thereby causing a loop.

To be used on:
Tools that read and analyse volume images (multiple partitions)

Obligatory behaviour:
Detect all five partitions and show the corresponding text file in every single one of them

Ideal behaviour:
Detect loop
Scan disk image and ensure that nothing was missed when stopping
Warn user that the partition table looks unusual and show it

Alternatively allowed behaviour:
Stop processing after a certain time / partition count and warn user that not everything might have been analysed
Show inconsistent size of occupied space and warn about that

Bad behaviour:
Show wrong error messages [p]
Show wrong size of occupied space without warning [c]
Crash / Become unresponsive [p]
Ignore any of the partitions or the text files within [c]

FS_PC_MBR_3

file name	md5	file size
FS_PC_MBR_3.dd	125a70d7db426b744ef9470b10ba1dae	5 368 709 120 B
layout.txt	ef457d019ef48dad5fc6bf4090c55468	9815 B
evidence_md5.txt	7b9f2ce1fa93a5dd29f42bf779022f37	10 041 B

Description:
The image contains an MBR partition table containing an extended partition. The extended partition contains 199 logical partitions. The exact sector level layout is shown by *layout.txt*. Every logical partition contains a file *evidence_X.txt* with text "evidence_X", where X is ranging from 1 to 199. The corresponding md5 checksums are listed in *evidence_md5.txt*.

To be used on:
Tools that read and analyse volume images (multiple partitions)

Obligatory behaviour:
Detect all 199 partitions and show the corresponding text file in every single one of them

Ideal behaviour:
-

Alternatively allowed behaviour:
Stop processing after a certain partition count and warn user that not everything might have been analysed
Show inconsistent size of occupied space and warn about that

Bad behaviour:
Show wrong error messages [p]
Show wrong size of occupied space without warning [c]
Crash / Become unresponsive [p]
Ignore any of the partitions or the text files within without warning the user [c]

FS_PC_MBR_4

file name	md5	file size
FS_PC_MBR_4.dd	cd5117dd7f383b56fc5a493e21324577	104 857 600 B
layout.txt	25352fd6e2ecadbedb1efcb6fa7bf16a	686 B
evidence_1.txt	c71b08d050f458e01fe0ea981d3fc1a5	11 B
evidence_2.txt	55c5d4bebc55c35ed51a648bdc13cfb9	11 B

Description:
The image contains an MBR partition table containing two extended partitions. Every extended partition contains a logical partition. The exact sector level layout is shown by *layout.txt*. The first logical partition contains the file *evidence_1.txt* with text "evidence_1".The second logical partition contains the file *evidence_2.txt* with text "evidence_2".

To be used on:
Tools that read and analyse volume images (multiple partitions)

Obligatory behaviour:
Detect both logical partitions and show the corresponding text file in every single one of them

Ideal behaviour:
Warn user that the partition table looks unusual and show it

Alternatively allowed behaviour:
Skip the second extended partition entry and warn user that not everything might have been analysed
Show inconsistent size of occupied space and warn about that

Bad behaviour:
Show wrong error messages [p]
Show wrong size of occupied space without warning [c]
Ignore any of the partitions or the text files within without warning the user [c]

FS_PC_MBR_5

file name	md5	file size
FS_PC_MBR_5.dd	17769ea13e6902155d187db87749db2f	209 715 200 B
layout.txt	15c66f3ee0a7d4ad883742e85cb7da8d	828 B
evidence_1.txt	c71b08d050f458e01fe0ea981d3fc1a5	11 B
evidence_2.txt	55c5d4bebc55c35ed51a648bdc13cfb9	11 B
evidence_3.txt	a8dd0472fe2033ad2deb336320948a31	11 B
evidence_4.txt	3d2a965500f4da0d940e5a6c2a410634	11 B

Description:
The image contains an MBR partition table containing an extended partition. The EBR of that partition points to a logical partition in its first entry and to the next EBR in its second entry. The first EBR also contains a forbidden third entry, pointing to a logical partition. The second EBR contains two entries to logical partitions. The exact sector level layout is shown by *layout.txt*. All logical partitions contain files named *evidence_X.txt* with text "evidence_X", where X is ranging from 1 to 4.

To be used on:
Tools that read and analyse volume images (multiple partitions)

Obligatory behaviour:
Detect all four logical partitions and show the corresponding text file in every single one of them

Ideal behaviour:
Warn user that the partition table looks unusual and show it

Alternatively allowed behaviour:
Skip the second logical partition entry in every EBR and warn user that not everything might have been analysed
Show inconsistent size of occupied space and warn about that

Bad behaviour:
Show wrong error messages [p]
Show wrong size of occupied space without warning [c]
Ignore any of the partitions or the text files within without warning the user [c]

FS_PC_GPT_1

file name	md5	file size
FS_PC_GPT_1.dd	43e5af27cf7ce2bf497b667037beb2fc	104 857 600 B
layout.txt	541aca9f1218414e01f238931b47f4ba	544 B
layout_backup.txt	8dc9d5b73b2377e2a4eee74ed9380f78	713 B
evidence_1.txt	c71b08d050f458e01fe0ea981d3fc1a5	11 B
evidence_2.txt	55c5d4bebc55c35ed51a648bdc13cfb9	11 B

Description:
The image contains a GPT partition table containing two partitions. The exact sector level layout is shown by *layout.txt*. The legacy partition table at the end of the disk image does not match the primary one. It describes two different partitions. The exact sector level layout of this partition table is shown by *layout_backup.txt*. The first real partition contains a file *evidence_1.txt* with text "evidence_1". The second real partition contains a file *evidence_2.txt* with text "evidence_2".

To be used on:
Tools that read and analyse volume images (multiple partitions)

Obligatory behaviour:
Detect both actual partitions and show the corresponding text file in every single one of them

Ideal behaviour:
Warn user that the primary and legacy GPT do not match and show both of them

Alternatively allowed behaviour:
Do not show active warning about GPT mismatch but log it somewhere
Refuse to analyse the disk image due to corrupt metadata

Bad behaviour:
Show wrong error messages [p]
Refuse to analyse the disk image without warning [p]
Ignore any of the partitions or the text files within without warning the user [c]

FS_PC_GPT_2

file name	md5	file size
FS_PC_GPT_2.dd	a2bf6a9afe770c327862cdba95565683	5 368 709 120 B
layout.txt	f0bdf0e240fd8ac32f718a1c42b915c2	14 469 B
evidence_md5.txt	7b9f2ce1fa93a5dd29f42bf779022f37	10 041 B

Description:
The image contains a GPT partition table containing 199 partitions. The exact sector level layout is shown by *layout.txt*. Every partition contains a file *evidence_X.txt* with text "evidence_X", where X is ranging from 1 to 199. The corresponding md5 checksums are listed in *evidence_ md5.txt*.

To be used on:
Tools that read and analyse volume images (multiple partitions)

Obligatory behaviour:
Detect all 199 partitions and show the corresponding text file in every single one of them

Ideal behaviour:
-

Alternatively allowed behaviour:
Stop processing after a certain partition count and warn user that not everything might have been analysed
Show inconsistent size of occupied space and warn about that

Bad behaviour:
Show wrong error messages [p]
Show wrong size of occupied space without warning [c]
Crash / Become unresponsive [p]
Ignore any of the partitions or the text files within without warning the user [c]

FS_PC_GPT_3

file name	md5	file size
FS_PC_GPT_3.dd	af5808bd946fd18ac6a835d87ef5e6f7	104 857 600 B
layout.txt	15bb8bf3c663f19cd20754b0df15a0ef	43 B
evidence_1.txt	c71b08d050f458e01fe0ea981d3fc1a5	11 B
evidence_2.txt	55c5d4bebc55c35ed51a648bdc13cfb9	11 B

Description:
The image contains a GPT partition table containing two partitions. The exact sector level layout is shown by *layout.txt*. The first partition contains a text file *evidence_1.txt* with text "evidence_1". The second partition contains a text file *evidence_2.txt* with text "evidence_2". The GPT is enclosed by a false protective MBR. This MBR shows a wrong partition type for the GPT part of the image .It also declares the GPT part only half as big as it really is.

To be used on:
Tools that read and analyse volume images (multiple partitions)

Obligatory behaviour:
Detect both actual partitions and show the corresponding text file in every single one of them

Ideal behaviour:
Detect GPT partition and warn user that the protective MBR and the GPT partition inside do not match
Ask whether to use MBR or GPT for disk analysis

Alternatively allowed behaviour:
Do not show active warning and use GPT struture for analysis, but log the mismatch somewhere

Bad behaviour:
Show wrong error messages [p]
Decide automatically to use MBR without notification [c]
Decide automatically to use GPT without notification [p]
Ignore any of the partitions or the text files within without warning the user [c]

FS_PC_GPT_4

file name	md5	file size
FS_PC_GPT_4.dd	663c74824375a29298c38d840409ce63	104 857 600 B
layout.txt	481eb0f363f5b3d1a3b8459c838e4d3c	542 B
evidence_1.txt	c71b08d050f458e01fe0ea981d3fc1a5	11 B
evidence_2.txt	55c5d4bebc55c35ed51a648bdc13cfb9	11 B

Description:
The image contains a GPT partition table containing two partitions. The exact sector level layout is shown by *layout.txt*. The first partition contains a file *evidence_1.txt* with text "evidence_1". The second partition contains a file *evidence_2.txt* with text "evidence_2". The checksum of the primary GPT header is not correct, but the backup one is intact.

To be used on:
Tools that read and analyse volume images (multiple partitions)

Obligatory behaviour:
Detect both partitions and show the corresponding text file in every single one of them

Ideal behaviour:
Inform user that primary GPT header checksum is corrupt
Inform user that backup GPT header checksum is intact
Show both headers to user and ask which one should be used

Alternatively allowed behaviour:
Warn user that primary GPT header checksum is corrupt and use backup one for disk analysis
Warn user that primary GPT header checksum is corrupt and ignore it for disk analysis

Bad behaviour:
Show wrong error messages [p]
Give no warning about a corrupt header checksum [p]
Ignore the backup header entirely [c]
Ignore any of the partitions or the text files within without warning the user [c]

FS_PC_GPT_5

file name	md5	file size
FS_PC_GPT_5.dd	c3fce5181ff57605ac57e47e5e9b42e4	104 857 600 B
layout.txt	f82235d66d5791d3739baac4008ba7e3	542 B
evidence_1.txt	c71b08d050f458e01fe0ea981d3fc1a5	11 B
evidence_2.txt	55c5d4bebc55c35ed51a648bdc13cfb9	11 B

Description:
The image contains a GPT partition table containing two partitions. The exact sector level layout is shown by *layout.txt*. The first partition contains a file *evidence_1.txt* with text "evidence_1". The second partition contains a file *evidence_2.txt* with text "evidence_2". The checksum of the GPT is corrupt, both in the primary and in the backup header.

To be used on:
Tools that read and analyse volume images (multiple partitions)

Obligatory behaviour:
Detect both partitions and show the corresponding text file in every single one of them

Ideal behaviour:
Inform user that primary GPT checksum is corrupt
Inform user that backup GPT checksum is corrupt
Warn user that analysis results could be incorrect due to corrupt metadata

Alternatively allowed behaviour:
Warn user that GPT checksum is corrupt and ignore it for disk analysis
Refuse analysis and warn user about corrupt disk metadata

Bad behaviour:
Show wrong error messages [p]
Give no warning about a corrupt GPT checksum [p]
Ignore the backup header entirely [c]
Ignore any of the partitions or the text files within without warning the user [c]

OS_W_EDB_1

file name	md5	file size
OS_W_EDB_1.edb	476020edf9a5d5e5519765fb4d36bf78	142 671 872 B
Windows.edb	d301e229392e7d91dc2ed71d01b2172a	142 671 872 B

Description:
The test case is based on a real *Windows.edb* acquired from a Windows 7 machine. The *last_ ObjID* field in the database main header is manipulated and set to 0 to simulate an empty database.

To be used on:
Tools that analyse Windows Search databases

Obligatory behaviour:
Show any information that can be read out using the original *Windows.edb*

Ideal behaviour:
Inform user that the database header incorrectly shows zero entries
Show user the relevant header field

Alternatively allowed behaviour:
Refuse to analyse the file and warn user that it might be corrupt

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore any information in the database [c]

OS_W_EDB_2

file name	md5	file size
OS_W_EDB_2.edb	4dfba8219e169a06fd21c22635cd4944	142 671 872 B
Windows.edb	d301e229392e7d91dc2ed71d01b2172a	142 671 872 B

Description:
The test case is based on a real *Windows.edb* acquired from a Windows 7 machine. The *space tree entry* in database page 5 that describes database page 16 is changed. It now specifies the *childnode* of page 16 to be page 16 itself. Additionally, the *nextPgno* field in the header of page 16 is changed to 16.

To be used on:
Tools that analyse Windows Search databases

Obligatory behaviour:
Show any information that can be read out using the original *Windows.edb*

Ideal behaviour:
Detect loop
Warn the user that the space tree might be corrupt
Warn the user that the header of database page 16 looks corrupt
Show the unnormal parts of the file to the user

Alternatively allowed behaviour:
Stop processing after a certain time and warn user that not everything might have been analysed
Refuse analysis and warn user that the file might be corrupt

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore any information in the database [c]
Produce output that contains much redundant information du to the loop [p]

OS_W_EDB_3

file name	md5	file size
No data associated, use radamsa		

Description:
The test case is based on a real *Windows.edb* acquired from a Windows 7 machine. Various fuzzed versions of it are created using the tool *radamsa*. The test can be reproduced with any real world Windows Search database or even with artificially created ones.

To be used on:
Tools that analyse Windows Search databases

Obligatory behaviour:
Show as much information of the original *Windows.edb* as possible

Ideal behaviour:
Warn user whenever a relevant part of the database might be corrupt
Show the unnormal parts of the file to the user

Alternatively allowed behaviour:
Stop processing after a certain time and warn user that not everything might have been analysed
Refuse analysis and warn user that the file might be corrupt

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Produce wrong / incomplete output without warning the user about problems during the analysis

OS_W_JL_1

file name	md5	file size
OS_W_JL_1.autom...	115bb7ea1c478c4e11c7209025ebe17c	5120 B
iexplore.autom...	2748ed629492103d43251668a4085df0	5120 B

Description:
The test case contains a JumpList configuration of the Internet Explorer. The four websites have been pinned to the JumpList. For exact information see the original file *iexplore.automaticDestinations-ms*. One website entry has been changed to `google.de/ onmouseover=""alert(123456789)""`. One entry has completely be replaced by `javascrip:alert(123456789)`.

To be used on:
Tools that analyse Windows 7 Jump List files and create an HTML report

Obligatory behaviour:
Show both changed website entries as they are, escaping the injection such that it remains visible
Show the two unmodified domains *facebook.com* and *microsoft.com*

Ideal behaviour:
Warn user that an injection attempt has been detected

Alternatively allowed behaviour:
Remove dangerous characters from the output but inform the user about their presence

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Remove the two malicious domains from the output, even only partially, without a notification [c]
Ignore any of the other two domains [c]
Create an HTML report without escaping the injections [c]

OS_W_JL_2

file name	md5	file size
OS_W_JL_2.autom...	f17c722979f4b0f3b3e9b3c3b66d95e3	5120 B
iexplore.autom...	2748ed629492103d43251668a4085df0	5120 B

Description:
The test case contains a JumpList configuration of the Internet Explorer. The four websites have been pinned to the JumpList. For exact information see the original file *iexplore.automaticDestinations-ms*. Several special characters often used for separating CSV exports are injected into the website entries. Additionally, a newline and a tabulator character, as well as a nullbyte are injected.

To be used on:
Tools that analyse Windows 7 Jump List files and optionally create a formatted export

Obligatory behaviour:
Show all four website entries as they are, escaping the injecton such that it remains visible

Ideal behaviour:
Warn user that an injection attempt has been detected

Alternatively allowed behaviour:
Remove dangerous characters from the output but inform the user about their presence

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Interpret the newline or tabulator character [p]
Remove any dangerous character from the output without notification [c]
Interpret the nullbyte as termination of a string literal [c]
Create a CSV export without escaping the special characters used for delimitation of entries [c]

OS_W_JL_3

file name	md5	file size
No data associated, use radamsa		

Description:
The test case is based on real world JumpList files acquired from Windows machines. Various fuzzed versions of them are created using the tool *radamsa*. The test can be reproduced with any real world JumpList or even with artificially created ones.

To be used on:
Tools that analyse Windows 7 Jump List files

Obligatory behaviour:
Show as much information of the original JumpList files as possible

Ideal behaviour:
Warn user whenever a relevant part of the JumpList might be corrupt
Show the unnormal parts of the file to the user

Alternatively allowed behaviour:
Stop processing after a certain time and warn user that not everything might have been analysed
Refuse analysis and warn user that the file might be corrupt

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Produce wrong / incomplete output without warning the user about problems during the analysis [c]

OS_W_EVTX_1

file name	md5	file size
OS_W_EVTX_1.evtx	4ef33ef47ab2ccff997b7c8f2eae8001	69 632 B
small_System.evtx	ec87f1b41f27e1ef79d3c4a56716ceaa	69 632 B

Description:
The test case is based on the logfile *small_System.evtx* with about 90 entries, acquired from a Windows 7 machine. The fields *first event record number* and *last event record number* are swapped in the chunk header of the file. The chunk header checksum is adjusted.

To be used on:
Tools that analyse Windows XML Event Logs

Obligatory behaviour:
Show any information that can be read out using the original *small_System.evtx*

Ideal behaviour:
Warn user that the record numbers might be incorrect
Ask the user what numbers are to be used, possibly making proposals based on the data found in the file

Alternatively allowed behaviour:
Refuse to analyse the file and warn user that it might be corrupt

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore the incorrect chunk header values while correctly reading the rest of the data [p]
Ignore any event record [c]

OS_W_EVTX_2

file name	md5	file size
OS_W_EVTX_2.evtx	a9ff0bdb071ee59fd70e4e2321b276eb	69 632 B
small_System.evtx	ec87f1b41f27e1ef79d3c4a56716ceaa	69 632 B

Description:
The test case is based on the logfile *small_System.evtx* with about 90 entries, acquired from a Windows 7 machine. At the beginning of the file, `"ls attr='asd' />` is injected into the *xmlns* attribute. In record 9, the string `<script>alert(1)</script>\00\00` is injected. In record 10, a newline character is injected. In record 11, a tabulator character is injected.

To be used on:
Tools that analyse Windows XML Event Logs and optionally create a formatted export

Obligatory behaviour:
Show any information that can be read out using the original *small_System.evtx*
Show the manipulated records as they are, escaping the injection such that it remains visible

Ideal behaviour:
Warn user that an injection attempt has been detected

Alternatively allowed behaviour:
Remove dangerous characters from the output but inform the user about their presence

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Interpret the newline or tabulator character [p]
Remove any dangerous character from the output without notification [c]
Interpret the nullbyte as termination of a string literal [c]
Create a CSV export without escaping the special characters used for delimitation of entries [c]
Create an HTML report without escaping the injections [c]

OS_W_EVTX_3

file name	md5	file size
No data associated, use radamsa		

Description:
The test case is based on real world XML event logfiles acquired from Windows machines. Various fuzzed versions of them are created using the tool *radamsa*. The test can be reproduced with any real world *evtx-file* or even with artificially created ones.

To be used on:
Tools that analyse Windows XML Event Logs

Obligatory behaviour:
Show as much information of the original *evtx-files* as possible

Ideal behaviour:
Warn user whenever a relevant part of the logfile might be corrupt
Show the unnormal parts of the file to the user

Alternatively allowed behaviour:
Stop processing after a certain time and warn user that not everything might have been analysed
Refuse analysis and warn user that the file might be corrupt

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Produce wrong / incomplete output without warning the user about problems during the analysis [c]

OS_W_REG_1

file name	md5	file size
OS_W_REG_1.DAT	4d927ae980050fcbb25dc3e7996e80ed	786 432 B
NTUSER.DAT	02f25912300df8e5bac0720c89a1197c	786 432 B

Description:
The test case is based on a real *NTUSER.DAT* registry file acquired from a Windows 7 machine. The subkey list of the first *name key* entry in the file, specifying the registry hive root, is modified to contain a *name key* entry that points back to the root *name key*, thereby causing a loop.

To be used on:
Tools that analyse Windows Registry Files

Obligatory behaviour:
Show any information that can be read out using the original *NTUSER.DAT*

Ideal behaviour:
Detect loop
Scan file and show the remaining registry keys

Alternatively allowed behaviour:
Stop processing after a certain time and warn user that not everything might have been analysed

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore any information in the registry file [c]

OS_W_REG_2

file name	md5	file size
OS_W_REG_2.DAT	d93e5250f5dbeedd9e7a260a63bcfb9c	786 432 B
NTUSER.DAT	02f25912300df8e5bac0720c89a1197c	786 432 B

Description:
The test case is based on a real *NTUSER.DAT* registry file acquired from a Windows 7 machine. The first *name key* entry in the file, specifying the registry hive root, is modified to contain the value 37 in its *number_of_subkeys* field but it really has only 11 subkeys. The *number_of_entries* field in the *subkey list* is also changed from 11 to 37.

To be used on:
Tools that analyse Windows Registry Files

Obligatory behaviour:
Show any information that can be read out using the original *NTUSER.DAT*

Ideal behaviour:
Warn user that unreadable / misformatted entries were found while parsing the subkey list

Alternatively allowed behaviour:
Show unreadable / misformatted registry entries but notify user about the unexpected format

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore any information in the registry file [c]
Show unreadable / misformatted registry entries without notification [p]

OS_W_REG_3

file name	md5	file size
OS_W_REG_3.DAT	04bcae7958a4a354ba8f5305a56284f3	786 432 B
NTUSER.DAT	02f25912300df8e5bac0720c89a1197c	786 432 B

Description:
The test case is based on a real *NTUSER.DAT* registry file acquired from a Windows 7 machine. Various injections are made in existing values. The *sCountry* key is manipulated with various CSV delimiter characters. The *Username* key is manipulated with an HTML comment. The *sLongDate* key is manipulated with a tabulator character. The *sNativeDigits* key is manipulated with a newline character. The *sTimeFormat* key is manipulated with a nullbyte.

To be used on:
Tools that analyse Windows Registry Files and optionally create a formatted export

Obligatory behaviour:
Show any information that can be read out using the original *NTUSER.DAT*
Show the manipuated entries as they are, escaping the injection such that it remains visible

Ideal behaviour:
Warn user that an injection attempt has been detected
Warn user that unlikely characters are present in some registry keys

Alternatively allowed behaviour:
Remove dangerous characters from the output but inform the user about their presence

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Interpret the newline or tabulator character [p]
Remove any dangerous character from the output without notification [c]
Interpret the nullbyte as termination of a string literal [c]
Create a CSV export without escaping the special characters used for delimitation of entries [c]
Create an HTML report without escaping the injections [c]

OS_W_REG_4

file name	md5	file size
No data associated, use radamsa		

Description:
The test case is based on real world registry files acquired from Windows machines. Various fuzzed versions of them are created using the tool *radamsa*. The test can be reproduced with any real world registry file or even with artificially created ones.

To be used on:
Tools that analyse Windows Registry Files

Obligatory behaviour:
Show as much information of the original registry files as possible

Ideal behaviour:
Warn user whenever a relevant part of the registry file might be corrupt
Show the unnormal parts of the file to the user

Alternatively allowed behaviour:
Stop processing after a certain time and warn user that not everything might have been analysed
Refuse analysis and warn user that the file might be corrupt

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Produce wrong / incomplete output without warning the user about problems during the analysis [c]

OS_M_SLDB_1

file name	md5	file size
OS_M_SLDB_1.db	ba623eac2570543f03784c687bba2490	118 784 B

Description:
The test case contains a Spotlight database of a USB drive indexed by Mac OS X 10.8 (Spotlight DB v2). The drive contained a file "<script>alert(1)</script>" and a folder "<script>alert(1)<" with a file *script>* in it. Additionally, three folders called ",", ";" and "|" were on the drive when it was indexed.

To be used on:
Tools that analyse Macintosh Spotlight databases and create a formatted export

Obligatory behaviour:
Show the folder structure of the USB stick as it is, escaping the injection such that it remains visible

Ideal behaviour:
Warn user that an injection attempt has been detected

Alternatively allowed behaviour:
Remove dangerous characters from the output but inform the user about their presence

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Remove any dangerous character from the output without notification [c]
Create a CSV export without escaping the special characters used for delimitation of entries [c]
Create an HTML report without escaping the injections [c]

OS_M_SLDB_2

file name	md5	file size
No data associated, use radamsa		

Description:
The test case is based on real world Spotlight database files. Various fuzzed versions of them are created using the tool *radamsa*. The test can be reproduced with any real world registry file or even with artificially created ones.

To be used on:
Tools that analyse Macintosh Spotlight databases

Obligatory behaviour:
Show as much information of the original Spotlight database files as possible

Ideal behaviour:
Warn user whenever a relevant part of the database might be corrupt
Show the unnormal parts of the file to the user

Alternatively allowed behaviour:
Stop processing after a certain time and warn user that not everything might have been analysed
Refuse analysis and warn user that the file might be corrupt

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Produce wrong / incomplete output without warning the user about problems during the analysis [c]

OS_M_BPL_1

file name	md5	file size
OS_M_BPL_1_BIN.plist	634314bb920287b16957f17e4dc33a68	42 B
OS_M_BPL_1_XML.plist	bbdf3432377888e136bc12d76a24dc2d	29 B

Description:
The test case consists of a handcrafted file that is the most minimal plist possible. The file contains a "<plist>" xml element with a "<true/>" xml element inside.

To be used on:
Tools that analyse Macintosh property lists in XML or binary format

Obligatory behaviour:
Show the contents of the file, in this case a single *true* element, as well as metadata

Ideal behaviour:
Warn user that the *plist* file does not conform to the usual structure

Alternatively allowed behaviour:
Refuse to analyse the file and warn user that it might be corrupt

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore any information in the file [c]
Ignore the missing header structure without notification [p]

OS_M_BPL_2

file name	md5	file size
OS_M_BPL_2_BIN.plist	37af33ae905a787537a37bda3d6a0b50	1132 B
OS_M_BPL_2_XML.plist	43286407df064b194faba4457a3eaade	270 201 B

Description:
The test case consists of a handcrafted file that is very deeply nested. The file contains about 200 levels of nested "<dict>" xml elements with the element "<string>This is a dummy string</string>" in the deepest level.

To be used on:
Tools that analyse Macintosh property lists in XML or binary format

Obligatory behaviour:
Show the contents of the file, most importantly the dummy string, as well as metadata

Ideal behaviour:
Warn user about deeply nested xml elements that could reduce the performance

Alternatively allowed behaviour:
Stop processing after a certain time / level of elements and warn user that not everything might have been analysed

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore any information in the file [c]

OS_M_BPL_3

file name	md5	file size
OS_M_BPL_3_XML.plist	843dfa89624d914aa9c1dd6be1ccf457	16 878 B
com.apple.recentitems.plist	1eab903ef65bb34ac590c65e228fb5f9	9565 B
recentitems_xml.plist	c8c3d381ef1afaf4d3d6663599ac0469	16 900 B

Description:
The test case is based on a real *recentitems.plist* file acquired from a Mac OS X 10.6 machine. The first "string" element normally containing the URL of a recently used server is filled with a string that is not a URL. An element "<true>Not so true</true>" is injected into the XML structure. An "<integer>" XML tag is filled with text instead of a number. An element "<real>asd_not_a_number</real>" is injected into the XML structure. An element "<string>Name</string>" is injected into a "<key>" XML element. An element "<integer><true/></integer>" is injected into the XML structure.

To be used on:
Tools that analyse Macintosh property lists in XML or binary format

Obligatory behaviour:
Show the contents of the file
Show every corrupt part of the file

Ideal behaviour:
Warn user that the *plist* file is corrupt

Alternatively allowed behaviour:
Stop processing after a certain time and warn user that not everything might have been analysed

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore any information in the file [c]
Fix the corruption without notification of the user [c]

OS_M_BPL_4

file name	md5	file size
No data associated, use radamsa		

Description:
The test case is based on real *plist* files acquired from Mac OS X machines. Various fuzzed versions of them are created using the tool *radamsa*.The test can be reproduced with any real world registry file or even with artificially created ones.

To be used on:
Tools that analyse Macintosh property lists in XML or binary format

Obligatory behaviour:
Show as much information of the original *plist* files as possible

Ideal behaviour:
Warn user whenever a relevant part of the *plist* might be corrupt
Show the unnormal parts of the file to the user

Alternatively allowed behaviour:
Stop processing after a certain time and warn user that not everything might have been analysed
Refuse analysis and warn user that the file might be corrupt

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Produce wrong / incomplete output without warning the user about problems during the analysis [c]

OS_M_LOG_1

file name	md5	file size
OS_M_LOG_1.log	65d3815d53456e1c3caa28a9584d46d6	28 718 B
system.log	09a6348d86f1cab3eeb13fbc70c30e87	29 215 B

Description:
The test case is based on a real *system.log* file acquired from a Mac OS X 10.6 machine. Various entries have been reformatted such that the syntax of a single entry was changed. The entries in the file are reordered to break the temporal ascending display of events. For exact differences compare the original *system.log* to the version provided as *OS_M_LOG_1.log*.

To be used on:
Tools that analyse Macintosh system.log files

Obligatory behaviour:
Show all entries in the logfile as they are

Ideal behaviour:
Warn user that the syntax of an entry is not intact
Warn user that the entries are not in temporal order

Alternatively allowed behaviour:
Stop processing after a misformatted entry and warn user that the file might be corrupt

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Automatically reorder the entries without notification [c]
Ignore the misformatted entries without notification of the user [c]

OS_M_LOG_2

file name	md5	file size
OS_M_LOG_2.log	9ad89405ce6940e32e903952ba585844	29 395 B
system.log	09a6348d86f1cab3eeb13fbc70c30e87	29 215 B

Description:
The test case is based on a real *system.log* file acquired from a Mac OS X 10.6 machine. Various lines have been inserted into the text, forming the sentences "This is another text to see what happens if random text is inserted somewhere in the file.", "Random text in between lines" and "Random text at the end!". Addtionally, many newlines have been inserted, thereby producing empty lines in the logfile.

To be used on:
Tools that analyse Macintosh system.log files

Obligatory behaviour:
Show all entries in the logfile as they are

Ideal behaviour:
Warn user that the entry syntax is not correct
Show the unnormal lines and thereby reveal the hidden message

Alternatively allowed behaviour:
Stop processing after a misformatted entry and warn user that the file might be corrupt

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore the misformatted entries without notification of the user [c]

OS_M_LOG_3

file name	md5	file size
OS_M_LOG_3.log	17dc95f76d55a330db5271bd6b5ef2f9	29 223 B
system.log	09a6348d86f1cab3eeb13fbc70c30e87	29 215 B

Description:
The test case is based on a real *system.log* file acquired from a Mac OS X 10.6 machine. The characters ";,|!?*"'", tabulator, newline and nullbyte have been injected in the file at line 100.

To be used on:
Tools that analyse Macintosh system.log files and optionally create a formatted export

Obligatory behaviour:
Show all entries in the logfile as they are, escaping the injection such that it remains visible

Ideal behaviour:
Warn user that an injection attempt has been detected

Alternatively allowed behaviour:
Stop processing after reaching an unexpected character and warn user that the file might be corrupt
Interpret all special characters as long as the output is complete
Remove dangerous characters from the output but inform the user about their presence

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore any information in the logfile [c]
Create a CSV export without escaping the special characters used for delimitation of entries [c]

OS_L_MLDB_1

file name	md5	file size
OS_L_MLDB_1.db	1f0089d7400eef60afe7cb479c26cf79	6 450 487 B
mlocate.db	50c589900d90ee927f3fbce4329b6cdf	6 450 487 B

Description:
The test case is based on a real *mlocate* database acquired from a Kali Linux 1.0.5 machine. The nanoseconds of the *creation_time* field in the database header are changed to 0xFFFFFFFF.

To be used on:
Tools that analyse Linux mlocate databases

Obligatory behaviour:
Show any information that can be read out using the original *mlocate* database

Ideal behaviour:
Warn user that the nanoseconds entry contains invalid information
Show the value of the entry to the user

Alternatively allowed behaviour:
Refuse to analyse the file and warn user that it might be corrupt

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore the wrong nanoseconds field without notification [c]
Ignore any information in the file [c]

OS_L_MLDB_2

file name	md5	file size
OS_L_MLDB_2.db	81bc8b9b4855e1443fa305f88b704385	6 450 487 B
mlocate.db	50c589900d90ee927f3fbce4329b6cdf	6 450 487 B

Description:
The test case is based on a real *mlocate* database acquired from a Kali Linux 1.0.5 machine. The byte 0x02 indicating the end of the root directory is changed to 0x00.

To be used on:
Tools that analyse Linux mlocate databases

Obligatory behaviour:
Show any information that can be read out using the original *mlocate* database

Ideal behaviour:
Warn user that more data than belonging to the directory could have been read
Show user the relevant part of the file

Alternatively allowed behaviour:
Stop processing after reaching the corrupt directory end and warn user that the file might be corrupt

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore any information in the file [c]

OS_L_MLDB_3

file name	md5	file size
OS_L_MLDB_3.db	822f9112e2a4e7bb769a4bb393ae9a47	6 450 487 B
mlocate.db	50c589900d90ee927f3fbce4329b6cdf	6 450 487 B

Description:
The test case is based on a real *mlocate* database acquired from a Kali Linux 1.0.5 machine. Various unexisting delimiters are injected. In the */bin* directory, a delimiter 0x03 is injected. In the */boot* directory, a delimiter 0x04 is injected. In the */dev* directory, a delimiter 0x10 is injected. In the */etc* directory, a delimiter 0xFF is injected.

To be used on:
Tools that analyse Linux mlocate databases

Obligatory behaviour:
Show any information that can be read out using the original *mlocate* database

Ideal behaviour:
Warn user that unknown delimiters have been found
Show user the relevant part of the file

Alternatively allowed behaviour:
Stop processing after reaching an unexpected delimiter and warn user that the file might be corrupt

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore any information in the file [c]
Fix the wrong delimiters without notification [c]

OS_L_BH_1

file name	md5	file size
OS_L_BH_1	9c093fea1e2f432fe149b1957f55ba69	7267 B
bash_history	68961063ce69d1ad41cd1023967fef5a	7407 B

Description:
The test case is based on a real *.bash_history* file acquired from a Kali Linux 1.0.5 machine. In the middle of the file, 10 consecutive newlines are injected. In the first line, tabulators are injected around the &&. In the second line various non-printable characters are injected. In the third line, the string "<script>alert(1)</script>" is injected. In the fourth line, the characters "+;:|,." are injected.

To be used on:
Tools that analyse Linux bash history files and optionally create a formatted export

Obligatory behaviour:
Show all entries in the history file as they are, escaping the injection in a way that it remains visible

Ideal behaviour:
Warn user that an injection attempt has been detected

Alternatively allowed behaviour:
Stop processing after reaching an unexpected character and warn user that the file might be corrupt
Remove dangerous characters from the output but inform the user about their presence

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore any information in the file [c]
Create a CSV export without escaping the special characters used for delimitation of entries [c]
Create an HTML report without escaping the injections [c]
Remove any dangerous character from the output without notification [c]

UF_MM_PIC_1

file name	md5	file size
UF_MM_PIC_1a.gif	8c320ca6fbcb40f32dc53e0b3a47a541	12 137 B
UF_MM_PIC_1b.png	ad3edf512705938c7c5c6453603cea55	163 675 B
UF_MM_PIC_1c.tiff	04c56f1a78b8fae1bc924e5f2cf9afd7	26 251 B
original.gif	d7bd270d9fae6235ef7d12257af2fecd	12 137 B
original.png	e5a116db3fd38d17b8277437b9a3c6d0	163 675 B
original.tiff	65cf94ef413cc934faa83ee4c81e3c12	26 251 B

Description:
The test case consists of three picture files in the formats *gif*, *png* and *tiff*. In every of these files, the resolution information has been changed to show only the upper half of the respective picture. The original information is still present inside the files.

To be used on:
Tools that analyse and display pictures in at least one of the formats gif, png or tiff

Obligatory behaviour:
Show all pixels of the pictures

Ideal behaviour:
Warn user that the header contains wrong information about the image size
Propose alternative image height that use all data present in the file (width cannot be changed without corrupting the displayed image part)

Alternatively allowed behaviour:
Warn user that more data than displayed is present
Refuse to display the image and warn user about a size corruption

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Automatically display the whole image without notification [p]
Ignore the hidden part of the image [c]

UF_MM_PIC_2

file name	md5	file size
UF_MM_PIC_2.gif	0840e6f95c8ceb85e452213a573c4552	1 052 095 B
original.gif	62c7cfb5bf544a122cf720b8939e7e55	1 052 095 B

Description:
The test case is based on a publicly available animated gif image [Mar]. The display time of the first frame is increased to more than 10 seconds to mimic a static picture when not looking long enough.

To be used on:
Tools that analyse and display gif pictures

Obligatory behaviour:
Indicate that the picture is animated
Show the animation

Ideal behaviour:
Show every frame of the image separately
Show detailed information about the display time of every frame

Alternatively allowed behaviour:
Inform user that the picture is animated and warn that this feature is not supported

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Automatically display the animated picture in a single frame without notification [p]
Ignore the hidden part of the image [c]

UF_MM_PIC_3

file name	md5	file size
UF_MM_PIC_3.png	1a87fb983e7a4ff28752009840d7895a	163 770 B
original.png	e5a116db3fd38d17b8277437b9a3c6d0	163 675 B

Description:
The test case is based on a picture found via an online search for *png* files [Anob]. The regular meta tag *comment* is added to the picture. It contains the string "<script>alert(1)</script>". An additional text chunk with meta tag value *additionalComment* is added at the end of the file. It contains the string "hidden text;,.:""'|", followed by a tabulator and a newline character.

To be used on:
Tools that analyse png pictures and optionally create a formatted export

Obligatory behaviour:
Show all text chunks present in the picture file, escaping the injection such that it remains visible

Ideal behaviour:
Warn user that an injection attempt has been detected

Alternatively allowed behaviour:
Stop processing after reaching an unexpected character and warn user that not everything has been analysed
Remove dangerous characters from the output but inform the user about their presence

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Interpret the newline or tabulator character [p]
Remove any dangerous character from the output without notification [c]
Create a CSV export without escaping the special characters used for delimitation of entries [c]
Create an HTML report without escaping the javascript injection [c]

UF_MM_PIC_4

file name	md5	file size
UF_MM_PIC_4.tiff	65cf94ef413cc934faa83ee4c81e3c12	26 251 B

Description:
The test case contains a regular *tiff* file found on the internet, more precisely in a now deleted thread inside the Apple developer forum. It uses the uncommen method of packing multiple pictures in a single *tiff* file using multiple *ifd* entries. The file contains two pictures showing the texts *front* and *bottom*.

To be used on:
Tools that analyse and display tiff pictures

Obligatory behaviour:
Show both pictures contained in the file

Ideal behaviour:
-

Alternatively allowed behaviour:
Warn user that only a basic set of functions is supported when looking at *tiff* files
Warn user that additional information is present in the file that it does not understand

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore the second picture in the file without notification [c]

UF_MM_PIC_5

file name	md5	file size
UF_MM_PIC_5.tiff	a586de2696ed369bbb3d46569616cf9d	26 251 B
original.tiff	65cf94ef413cc934faa83ee4c81e3c12	26 251 B

Description:
The test case is based on the regular *tiff* file used in test case UF_MM_PIC_4. Instead of correctly pointing to the next *ifd* structure in the file, the offset value in the first *ifd* points back to itself, thereby causing a loop.

To be used on:
Tools that analyse tiff pictures

Obligatory behaviour:
Show both pictures contained in the file

Ideal behaviour:
Detect loop
Scan file and show the second *ifd*

Alternatively allowed behaviour:
Warn user that only a basic set of functions is supported when looking at *tiff* files
Stop processing after a certain amount of time and warn user that not everything might have been processed

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore the second picture in the file without notification [c]
Ignore the loop without notifying the user about it [p]

UF_MM_A_1

file name	md5	file size
No data associated, use radamsa		

Description:
The test case is based on multiple files in different audio formats. To conduct this test, sample music has to be collected independently. After gathering a broad range of different files, fuzzed versions are created with the tool *radamsa*.

To be used on:
Tools that analyse metadata of the chosen audio formats and optionally play the files

Obligatory behaviour:
Show any information that remains in the audio files after the fuzzing (sound, meta tags)

Ideal behaviour:
Warn user whenever a relevant part of the audio files might be corrupt
Show the unnormal parts of the file to the user

Alternatively allowed behaviour:
Stop processing after a certain amount of time / after reaching a corrupt part in the file and warn user that not everything might have been analysed
Refuse to start the analysis

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Produce wrong / incomplete output without warning the user about problems during the analysis [c]

UF_MM_V_1

file name	md5	file size
No data associated, use radamsa		

Description:
The test case is based on multiple files in different video formats. To conduct this test, sample videos have to be collected independently. After gathering a broad range of different files, fuzzed versions are created with the tool *radamsa*.

To be used on:
Tools that analyse metadata of the chosen video formats and optionally play the files

Obligatory behaviour:
Show any information that remains in the video files after the fuzzing (video, sound, meta tags)

Ideal behaviour:
Warn user whenever a relevant part of the video files might be corrupt
Show the unnormal parts of the file to the user

Alternatively allowed behaviour:
Stop processing after a certain amount of time / after reaching a corrupt part in the file and warn user that not everything might have been analysed
Refuse to start the analysis

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Produce wrong / incomplete output without warning the user about problems during the analysis [c]

UF_OF_ODF_1

file name	md5	file size
UF_OF_ODF_1.odt	ed3b235913e04cc7d505b2c83831d46f	8907 B

Description:
The test case uses the *conditional text* feature of the open document file format. A trigger variable is connected with the text "The secret key to my hacker admin panel is admin:admin". If the trigger is equal to "True" the text is shown. Otherwise it is hidden.

To be used on:
Tools that analyse Open Document text files

Obligatory behaviour:
Show the normal content of the file
Show the hidden text
Show the value of the normally invisible trigger variable

Ideal behaviour:
Highlight the text to mark it as hidden

Alternatively allowed behaviour:
Warn user that an open document feature is used that is not supported
Inform user that some of the displayed text would normally be hidden without marking it directly

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Automatically display the hidden text without notification [p]
Ignore the hidden text inside the file[c]

UF_OF_ODF_2

file name	md5	file size
UF_OF_ODF_2.odt	2c3354aceabd25eaa8957fbcbf299c52	65 523 B

Description:
The test case uses the *floating frame* feature of the open document file format. Multiple frames are inserted in a text file containing lorem ipsum text. The source of the displayed document inside the floating frames is manipulated in the XML structure of the file. It points back to itself, thereby causing a loop.

To be used on:
Tools that analyse Open Document text files

Obligatory behaviour:
Show the normal content of the file
Show the embedded floating frame
Show the document source of the floating frame

Ideal behaviour:
Detect loop

Alternatively allowed behaviour:
Warn user that an open document feature is used that is not supported
Stop the processing after a certain amount of time, warning the user that not everything might have been analysed

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore any information in the file[c]

UF_OF_ODF_3

file name	md5	file size
UF_OF_ODF_3.odt	7bc408ec059a3e05449e1880e3c94276	678 173 B
hidden.jpg	4f65dea50ea8cd30a7f98698d843091d	87 113 B

Description:
The test case is based on a document created with *Libre Office*. The document is unpacked and an additional picture is added to the picture folder, which is not embedded in the document directly. The picture is also added to the list of files in *manifest.xml*. The document is repacked to look like a unsuspicious *odt* file.

To be used on:
Tools that analyse Open Document text files

Obligatory behaviour:
Show the normal content of the file
Detect the hidden picture and show it

Ideal behaviour:
Show the hidden picture
List every file inside the *Open Document* container file
Warn user that unreferenced files are present in the document

Alternatively allowed behaviour:
Detect the hidden picture and notify the user without directly showing it
Warn user that the file is manipulated and refuse to process it

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore the hidden picture [c]

UF_OF_ODF_4

file name	md5	file size
UF_OF_ODF_4.odt	b74722a07ca2e0d4f61b6d92679f1a46	19 002 B

Description:
The test case consists of a benign *odt* file created with *Libre Office*. A picture is embedded that, when viewed, is directly loaded from the internet [Goo13].

To be used on:
Tools that analyse Open Document text files and are installed on a computer with internet access

Obligatory behaviour:
Show the normal content of the file
Show that an internet picture is embedded without loading it
Show the link to the picture

Ideal behaviour:
Warn user that the file tries to download data from the internet
Show user a link to the image to open in an external browser

Alternatively allowed behaviour:
Warn user that an internet picture is embedded without directly presenting the link
Ask the user whether the connection should be established (discouraged behaviour!)

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Establish the internet connection without asking for user interaction [c]

UF_OF_MSO_1

file name	md5	file size
No data associated, use radamsa		

Description:
The test case is based on multiple files in different Microsoft office formats (old *doc*, *xls* etc. format). To conduct this test, sample documents have to be collected independently. After gathering a broad range of different files, fuzzed versions are created with the tool *radamsa*.

To be used on:
Tools that analyse Microsoft Office documents in the old format

Obligatory behaviour:
Show any information that remains in the documents after the fuzzing (text, embedded objects, meta information, ...)

Ideal behaviour:
Warn user whenever a relevant part of the document might be corrupt
Show the unnormal parts of the file to the user

Alternatively allowed behaviour:
Stop processing after a certain amount of time / after reaching a corrupt part in the file and warn user that not everything might have been analysed
Refuse to start the analysis

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Produce wrong / incomplete output without warning the user about problems during the analysis [c]

UF_OF_M_1

file name	md5	file size
UF_OF_M_1.mbox	27e434092205cf2e71d0e80989246be2	80 314 B
original.mbox	cb93437bf980634e1d088332eb7b754d	80 254 B

Description:
The test case is based on a real world *mbox* file that has been created by converting a pst file that is part of the Enron email data set [Coh09] with *libpst*. The typical structure of *mbox* files is broken by inserting many newlines at various positions in the file.

To be used on:
Tools that analyse mbox email containers

Obligatory behaviour:
Show all messages contained in the *mbox* file

Ideal behaviour:
Warn user about an unusual file structure

Alternatively allowed behaviour:
Stop processing after reaching a newline and warn user that not everything might have been analysed due to an unexpected format

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore any information in the file [c]
Ignore the newlines without notification [p]

UF_OF_M_2

file name	md5	file size
UF_OF_M_2.eml	2e8870a3d86d1091eb376a324218e895	617 110 B
original.eml	1d91a8315a65e89d4556d1a7b9fa5adf	617 153 B

Description:
The test case is based on a real world *eml* file. In the *To:* header line the string "<script>alert(1)</script>" is injected. In the *CC:* header line, a HTML comment is injected, commenting out the first additional recipient of the message, "<Diane.Anderson@ENRON.com>".

To be used on:
Tools that analyse eml files and create a formatted HTML export

Obligatory behaviour:
Show all information present in the file, escaping the injection such that it remains visible

Ideal behaviour:
Warn user that an injection attempt has been detected

Alternatively allowed behaviour:
Stop processing after reaching an unexpected character and warn user that not everything has been analysed
Remove dangerous characters from the output but inform the user about their presence

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Remove any dangerous character from the output without notification [c]
Create an HTML report without escaping the javascript injection [c]

UF_OF_M_3

file name	md5	file size
UF_OF_M_3.emlx	ffa17b4903c944243b2accfad0f9c756	1195 B
original.emlx	984978c7c895cd8be9e5829d8d6684a1	1195 B

Description:
The test case is based on a real world *emlx* file. The first characters in the file, normally containing the message size in bytes, are changed to represent a bigger message than actual.l (765 B vs. 739 B).

To be used on:
Tools that analyse emlx files

Obligatory behaviour:
Show all information present in the file

Ideal behaviour:
Inform user about the wrong bytesize

Alternatively allowed behaviour:
Refuse to analyse the file and warn user that it might be corrupt

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore any information in the file [c]
Ignore the wrong header size and display the file correctly without notification [p]

UF_OF_M_4

file name	md5	file size
UF_OF_M_4.emlx	2e613bbe75206062f3e57f1998edfadf	3086 B
original.emlx	984978c7c895cd8be9e5829d8d6684a1	1195 B

Description:
The test case is based on a real world *emlx* file. The XML structure at the end of the file, containing meta information, is manipulated. The *dtd* document describing the internal XML structure is embedded instead of being referred to as a URI. The *dtd* is then modified to create a huge amount of data when being parsed. Therefore, entities are defined that include each other recursively for 12 levels, resulting in about 3 TB of data.

To be used on:
Tools that analyse the metadata plist embedded in emlx files

Obligatory behaviour:
Show all information present in the file

Ideal behaviour:
Detect XML bomb attempt
Warn user about an unusually embedded *dtd* file

Alternatively allowed behaviour:
Refuse to analyse the file and warn user that an unexpected inline *dtd* file is present
Stop processing after a certain amount of time or used disk space and warn user that not everything might have been analysed

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Escape the XML bomb without notification [p]
Ignore any information inside the file [c]

UF_OF_M_5

file name	md5	file size
UF_OF_M_5.pst	7eb431fd900e2ecedd40863c99264b80	26 428 416 B
original.pst	21c079f7c43daf7e42a1528e287a0ff2	26 428 416 B

Description:
The test case is based on a real world *pst* file. The inner tree structure is modified, such that nodes of the tree point back to themselves, thereby causing a loop.

To be used on:
Tools that analyse pst email containers

Obligatory behaviour:
Show all information present in the file

Ideal behaviour:
Detect loop

Alternatively allowed behaviour:
Stop processing after a certain amount of time and warn user that not everything might have been analysed

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Escape the loop without notification [p]
Ignore any information inside the file [c]

UF_V_CB_1

file name	md5	file size
UF_V_CB_1.png	518602a9c69ff638320e3f0d6a943344	44 024 B

Description:
The test case consists of a picture crafted by *AERAsec Network Services and Security GmbH* [AER09]. It contains the picture information of a monochrome red pixel and deflates to 361 megapixels when viewed. It therefore uses the compression capabilities of the *png* file format.

To be used on:
Tools that analyse and display png pictures

Obligatory behaviour:
Show the contents of the file

Ideal behaviour:
Detect the heavy decompression ratio
Ask user whether or not he wants to continue

Alternatively allowed behaviour:
Stop processing after a certain time / decompressed data count and warn user that not everything might have been decompressed for analysis

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore the file content without asking the user whether he wants to abort the process [c]

UF_V_CB_2

file name	md5	file size
UF_V_CB_2.zip	b2d4891746e649e3fa43e46a0b9d876b	113 712 B

Description:
The test case consists of a handcrafted *zip* file that contains six layers. In every layer, ten new zip files are found, which at the lowest level deflate to a file of 4 GiB in size, containing the repeated string value "random". In total, the file deflates to about 4 PiB.

To be used on:
Tools that automatically deflate compressed files for further analyses

Obligatory behaviour:
Notify the user of a decompression problem after a reasonable amount of time or used disk space

Ideal behaviour:
Detect the heavy decompression ratio
Ask user whether or not he wants to continue

Alternatively allowed behaviour:
Stop processing after a certain time / decompressed data count and warn user that not everything might have been decompressed for analysis

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore the file content without asking the user whether he wants to abort the process [c]

UF_V_CB_3

file name	md5	file size
UF_V_CB_3.zip	22564e6badd1151d2163fb6cb7b1b160	56 387 B

Description:
The test case consists of a handcrafted *zip* file that contains three layers. Every layer contains another zip file and the lowest layer deflates to 1 TiB of zeroes.

To be used on:
Tools that automatically deflate compressed files for further analyses

Obligatory behaviour:
Notify the user of a decompression problem after a reasonable amount of time or used disk space

Ideal behaviour:
Detect the heavy decompression ratio
Ask user whether or not he wants to continue

Alternatively allowed behaviour:
Stop processing after a certain time / decompressed data count and warn user that not everything might have been decompressed for analysis

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Ignore the file content without asking the user whether he wants to abort the process [c]

UF_V_PDF_1

file name	md5	file size
UF_V_PDF_1.pdf	bf4bfb54536a6e260c04bbd747fd30c0	1347 B

Description:
The test case consists of a Linux-compatible *PDF* file that at the same time is a Linux *ELF* binary, an Oracle Java *jar* and an *HTML* with javascript. The file is not selfmade but created by Ange Albertini [Alb12].

To be used on:
Tools that analyse PDF files

Obligatory behaviour:
-

Ideal behaviour:
Detect all file formats

Alternatively allowed behaviour:
Warn user that more information is present in the file, which cannot be displayed easily
Refuse to analysis the file

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Only show the *PDF* file [c]
Ignore any content inside the file [c]

UF_V_PDF_2

file name	md5	file size
UF_V_PDF_2.pdf	9037cc041277d038d24b6eb37e847ff9	1526 B

Description:
The test case consists of a Macintosh-compatible *PDF* file that at the same time is a Macintosh *MachO* binary, an Oracle Java *jar* and an *HTML* with javascript. The file is not selfmade but created by Ange Albertini [Alb12].

To be used on:
Tools that analyse PDF files

Obligatory behaviour:
-

Ideal behaviour:
Detect all file formats

Alternatively allowed behaviour:
Warn user that more information is present in the file, which cannot be displayed easily
Refuse to analysis the file

Bad behaviour:
Show wrong error messages [p]
Crash / Become unresponsive [p]
Only show the *PDF* file [c]
Ignore any content inside the file [c]

List of Figures

List of Tables

List of Listings

Bibliography

[Ado13] Adobe Inc. Adobe Security Bulletins: APSB13-25, 2013. URL: https://www.adobe.com/support/security/bulletins/apsb13-25.html.

[AER09] AERAsec Network Services and Security GmbH. FTP File Index, 2009. URL: ftp://ftp.aerasec.de/pub/advisories/decompressionbombs/pictures/.

[Alb12] Ange Albertini. Mix, 2012. URL: http://mix.corkami.com.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable an Secure Computing*, 1(1):11–33, 2004.

[Anoa] Anonymous. Computer Forensic Tool Testing. URL: http://groups.yahoo.com/neo/groups/cftt/info.

[Anob] Anonymous. Giraffe Animal Gray Wallpaper. URL: http://www.wallchan.com/wallpaper/20159/.

[App] Apple Inc. mkfs.hfs(8): construct a new HFS Plus file system - Linux man page. URL: http://manpages.ubuntu.com/manpages/precise/man8/mkfs.hfs.8.html.

[App04] Apple Inc. Kernel and Device Drivers Layer, 2004. URL: https://developer.apple.com/library/mac/documentation/MacOSX/Conceptual/OSX_Technology_Overview/SystemTechnology/SystemTechnology.html.

[App10a] Apple Inc. HFS Plus Volume Format - Technical Note TN1150, 2010. URL: http://dubeiko.com/development/FileSystems/HFSPLUS/tn1150.html.

[App10b] Apple Inc. Property List Programming Guide: About Property Lists, 2010. URL: https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/PropertyLists/AboutPropertyLists/AboutPropertyLists.html#//apple_ref/doc/uid/10000048i-CH3-SW2.

[App13] Apple Inc. HFS+ Source Code, 2013. URL: http://opensource.apple.com/source/xnu/xnu-1504.15.3/bsd/hfs/.

[ARSS12] Anton Altaparmakov, Richard Russon, Erik Sornes, and Szabolcs Szakacsits. mkfs.ntfs(8): create NTFS file system - Linux man page, 2012. URL: http://linux.die.net/man/8/mkfs.ntfs.

[Bil03] CVE-2003-1564, 2003. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-1564.

[Bru11] Josh Brunty. Validation of Forensic Tools and Software: A Quick Guide for the Digital Forensic Examiner, 2011. URL: http://www.dfinews.com/articles/2011/03/validation-forensic-tools-and-software-quick-guide-digital-forensic-examiner.

[Byi] Carl Byington. outlook.pst - Format of MS Outlook .pst File. URL: http://www.five-ten-sg.com/libpst/rn01re05.html.

[Car05] Brian Carrier. *File System Forensic Analysis*. Addison-Wesley Professional, 2005.

[Car10] Brian Carrier. Digital (Computer) Forensics Tool Testing Images, 2010. URL: http://dftt.sourceforge.net/.

[Car11] Harlan Carvey. *Windows Registry Forensics: Advanced Digital Forensic Analysis of the Windows Registry*. Syngress, 2011.

[Cha09] Ian Charters. The Evolution of Digital Forensics : Civilizing the Cyber Frontier. Technical report, 2009.

[Che09] Brian Chess. Rough Auditing Tool for Security (RATS), 2009. URL: https://code.google.com/p/rough-auditing-tool-for-security/.

[Chr06] Steve Christey. PLOVER - Preliminary List of Vulnerability Examples for Researchers. Technical report, 2006.

[Coh09] William W. Cohen. Enron Email Dataset, 2009. URL: https://www.cs.cmu.edu/~enron/.

[Cor13] MITRE Corp. CWE - CWE List (2.5), 2013. URL: http://cwe.mitre.org/data/index.html.

[Dec13] Decalage. PDF Security Issues, 2013. URL: http://www.decalage.info/file_formats_security/pdf.

[Dil12] Andreas Dilger. Ext4 Patch, 2012. URL: http://git.kernel.org/cgit/linux/kernel/git/stable/stable-queue.git/diff/queue-3.4/ext4-disallow-hard-linked-directory-in-ext4_lookup.patch?id=fc126d9ef33b426f811ab822d1ab751d3d172e71.

[fdi] FDISK(8): Partition table manipulator - Linux man page. URL: http://linux.die.net/man/8/fdisk.

[Fin09] Rodel Finones. Exploit:Win32/Pdfjsc.BI, 2009. URL: http://www.microsoft.com/security/portal/threat/encyclopedia/Entry.aspx?Name=Exploit%3AWin32%2FPdfjsc.BI.

[Gar07] Simson Garfinkel. Anti-Forensics: Techniques, Detection and Countermeasures. In *2nd International Conference on i-Warfare and Security*, pages 77–84, 2007.

[Ges11] Alexander Geschonneck. *Computer-Forensik: Computerstraftaten erkennen, ermitteln, aufklären.* dpunkt.verlag, ix edition, 2011.

[GFRD09] Simson Garfinkel, Paul Farrell, Vassil Roussev, and George Dinolt. Bringing Science to Digital Forensics with Standardized Forensic Corpora. *Digital Investigation*, 6:2–11, September 2009. URL: http://digitalcorpora.org/corpora/files.

[Goo13] Google Inc. Google Logo, 2013. URL: https://www.google.de/images/srpr/logo11w.png.

[Gus11] Sam Gustin. Digital Music Sales Surpass Physical Music Sales For the First Time Ever, 2011. URL: http://business.time.com/2012/01/06/digital-music-sales-finally-surpassed-physical-sales-in-2011/.

[Har06] Ryan Harris. Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem. *Digital Investigation*, 3S:44–49, 2006. URL: http://www.sciencedirect.com/science/article/pii/S1742287606000673.

[Har13] Phil Harvey. ExifTool, 2013. URL: http://www.sno.phy.queensu.ca/~phil/exiftool/.

[Hel13] Aki Helin. Radamsa - ouspg - On fuzzing., 2013. URL: http://code.google.com/p/ouspg/wiki/Radamsa.

[HR10] Prof. Dr. Bernd Heinrich and Dr. Tobias Reinbacher. Freie richterliche Beweiswürdigung – § 261 StPO. *Arbeitsblatt zur Vorbereitung auf Staatsexamen*, 2010.

[Hud12] Dave Hudson. mkfs.vfat(8): create MS-DOS file system under Linux - Linux man page, 2012. URL: http://linux.die.net/man/8/mkfs.vfat.

[ILA02] ILAC. Guidelines for Forensic Science Laboratories. Technical report, 2002. URL: https://www.ilac.org/documents/g19_2002.pdf.

[Int11] Int0x80. Anti-Forensics, 2011. URL: https://github.com/int0x80/anti-forensics.

[ISO05] ISO/IEC. Allgemeine Anforderungen an die Kompetenz von Prüf- und Kalibrierlaboratorien (ISO/IEC 17025:2005), 2005. URL: http://www.uni-due.de/imperia/md/content/water-science/2951wa_0809_r01.pdf.

[IW08] Vinay M. Igure and Ronald D. Williams. Taxonomies of Attacks and Vulnerabilities in Computer Systems. *IEEE Communications Surveys & Tutorials*, 10:6–19, 2008. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4483667.

[JPA08] Rob Joyce, Judson Powers, and Frank Adelstein. Mac Marshal: A Tool for Mac OS X Operating System and Application Forensics. In *Proceedings of the 2008 Digital Forensic Research Workshop*, 2008. URL: http://www.dfrws.org/2008/proceedings/p83-joyce_pres.pdf.

[Jum13a] Jump Lists, 2013. URL: http://www.forensicswiki.org/wiki/Jump_Lists.

[Jum13b] List of Jump List IDs, 2013. URL: http://www.forensicswiki.org/wiki/List_of_Jump_List_IDs.

[Kor11] Jesse Kornblum. dc3dd, 2011. URL: http://sourceforge.net/projects/dc3dd/.

[LBMC94] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and Williams S. Choi. A Taxonomy of Computer Program Security Flaws, with Examples. *ACM Computing Surveys*, 26(3):211–254, 1994. URL: http://www.stormingmedia.us/78/7855/A785564.html.

[Mar] Marvel. File:Rotating earth (large).gif. URL: http://commons.wikimedia.org/wiki/File:Rotating_earth_(large).gif.

[Met10] Joachim Metz. Windows Search Forensics. Technical report, 2010.

[Met12] Joachim Metz. Extensible Storage Engine (ESE) Database File (EDB) Format Specification. Technical report, 2012.

[Met13a] Joachim Metz. File Formats, 2013. URL: http://code.google.com/p/libyal/wiki/Overview.

[Met13b] Joachim Metz. Windows XML Event Log (EVTX). Technical report, 2013.

[Mic13] Microsoft Corporation. Microsoft Office File Formats, 2013. URL: http://msdn.microsoft.com/en-us/library/cc313118.aspx.

[NIS01] NIST. General Test Methodology for Computer Forensic Tools. Technical report, NIST, 2001.

[NIS05] NIST. Digital Data Acquisition Tool Test Assertions and Test Plan, 2005. URL: http://www.cftt.nist.gov/DA-ATP-pc-01.pdf.

[NIS13a] NIST. NIST Computer Forensic Tool Testing Program, 2013. URL: http://www.cftt.nist.gov/.

[NIS13b] NIST. The CFReDS Project, 2013. URL: http://www.cfreds.nist.gov/.

[NPSB07] Tim Newsham, Chris Palmer, Alex Stamos, and Jesse Burns. Breaking Forensics Software : Weaknesses in Critical Evidence Collection. Technical report, iSEC Partners, Inc., 2007.

[OAS11] OASIS Open. Open Document Format for Office Applications (OpenDocument) Version 1.2, 2011. URL: http://docs.oasis-open.org/office/v1.2/OpenDocument-v1.2.pdf.

[Off13a] Offensive Security Ltd. Kali Linux, 2013. URL: http://www.kali.org/.

[Off13b] Offensive Security Ltd. Kali Linux Forensics Mode, 2013. URL: http://docs.kali.org/general-use/kali-linux-forensics-mode.

[Ora13a] Oracle. Oracle VM VirtualBox, 2013. URL: https://www.virtualbox.org/.

[Ora13b] Oracle Corporation. Oracle Outside In Technology, 2013. URL: http://www.oracle.com/us/technologies/embedded/025613.htm.

[Ort11] Tanguy Ortolo. Repacking ZIP-based containers, 2011. URL: http://tanguy.ortolo.eu/blog/article24/repack-zip-container.

[OWA08] OWASP Foundation. OWASP Code Review Guide. Technical report, 2008.

[OWA13] OWASP. OWASP Top 10 - 2013. Technical report, 2013.

[Oxf13] forensic, adj. and n. : Oxford English Dictionary, 2013. URL: http://www.oed.com/view/Entry/73107?rskey=4nKI1g&result=4&isAdvanced=false.

[Peh13] Andreas Pehnack. Synalyze It!, 2013. URL: http://www.synalysis.net/.

[Rid07] Chris K. Ridder. Evidentiary Implications of Potential Security Weaknesses in Forensic Software. Technical report, 2007. URL: http://www.scn.rain.com/~neighorn/PDF/Ridder-Evidentiary_Implications_of_Security_Weaknesses_in_Forensic_Software.pdf.

[RMK11] Paul Rubin, David MacKenzie, and Stuart Kemp. dd(1): convert/copy file - Linux man page, 2011. URL: http://linux.die.net/man/1/dd.

[Rog05] Marcus K. Rogers. Anti-Forensics - Presentation given to Lockheed Martin, 2005.

[Rus99] Mark Russinovich. Inside the Registry. *Windows NT Magazine*, 1999. URL: http://www.microsoft.com/technet/archive/winntas/tips/winntmag/inreg.mspx?mfr=true.

[Sch07] Andreas Schuster. Introducing the Microsoft Vista Event Log File Format. In *Proceedings of the 2007 Digital Forensic Research Workshop*, 2007.

[Sen95] Tim Sennitt. PC DOS 7 Technical Update. Technical report, 1995.

[SG11] Alexander Sigel and Alexander Geschonneck. Goldmine: Spurensuche in der Windows-Registry. *iX - Magazin für professionelle Informationstechnik*, (1):100–105, 2011. URL: https://www.heise.de/artikel-archiv/ix/2011/01/100_Goldmine.

[Sin09] Steven Sinofsky. Our Next Engineering Milestone: RTM, 2009. URL: http://blogs.msdn.com/b/e7/archive/2009/07/22/our-next-engineering-milestone-rtm.aspx.

[Smi12] Roderick W. Smith. gdisk(8) - Linux man page, 2012. URL: http://linux.die.net/man/8/gdisk.

[Sta13] StatCounter. Top 7 Desktop, Tablet & Console OSs from Dec 2012 to Dec 2013, 2013. URL: http://gs.statcounter.com/#os-ww-monthly-201212-201312.

[Ste09] Didier Stevens. Malformed PDF Documents, 2009. URL: http://blog.didierstevens.com/2009/05/14/malformed-pdf-documents/.

[TCM05] Katrina Tsipenyuk, Brian Chess, and Gary McGraw. Seven pernicious kingdoms: A taxonomy of software security errors. In *Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics*, pages 36–43, Long Beach, CA, 2005. URL: http://hissa.nist.gov/~black/Papers/NISTSP500-265.pdf.

[Thi08] David Thiel. Exposing Vulnerabilities in Media Software. In *BlackHat EU 2008*, 2008. URL: http://www.blackhat.com/presentations/bh-europe-08/Thiel/Whitepaper/bh-eu-08-thiel-WP.pdf.

[Trm] Miroslac Trmac. mlocate.db(5) - mlocate database. URL: http://linux.die.net/man/5/mlocate.db.

[Ts'12] Theodore Ts'o. mkfs.ext4(8): create ext2/ext3/ext4 filesystem - Linux man page, 2012. URL: http://linux.die.net/man/8/mkfs.ext4.

[Uni13] Unified EFI Inc. Unified Extensible Firmware Interface Specification 2.4. Technical report, 2013.

[Val08] Javier Vicente Vallejo. Adobe Acrobat Reader Malformed PDF Code Execution Vulnerability, 2008. URL: http://www.zerodayinitiative.com/advisories/ZDI-08-073/.

[WA13] Martin Wundram and Hendrik Adam. Anti-Anti-Forensik. In *3. IT Forensik Workshop FH Aachen*, 2013.

[Wei98] Juergen Weigert. xxd(1): make hexdump/do reverse - Linux man page, 1998. URL: `http://linux.die.net/man/1/xxd`.

[WFM13] Martin Wundram, Felix C. Freiling, and Christian Moch. Anti-Forensics : The Next Step in Digital Forensics Tool Testing. In *7th International Conference on IT Security Incident Management and IT Forensics (IMF)*, pages 83–97, 2013. `doi:10.1109/IMF.2013.17`.

[Wik13] Wikipedia. List of digital forensic tools, 2013. URL: `http://en.wikipedia.org/wiki/List_of_digital_forensics_tools`.

[Won13] Darrick J. Wong. Ext4 Disk Layout, 2013. URL: `https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout`.

[Wun13] Martin Wundram. Kritische Sicherheitslücken in Write-Blocker entdeckt, 2013. URL: `http://heise.de/-2071582`.

[Zet13] Kim Zetter. How a Crypto 'Backdoor' Pitted the Tech World Against the NSA, 2013. URL: `http://www.wired.com/threatlevel/2013/09/nsa-backdoor/`.