Circumventing Forensic Live-Acquisition Tools On Linux

How to harden against rootkits intervening with the acquisition of data from a live Linux system

Yonne de Bruijn
Master in System and Network Engineering, UvA
June 2015

Abstract

This project investigated the current state of forensic live data acquisition when dealing with Linux operating systems. It identified current common tools and techniques, disseminated from different sources. Flaws within the current process and useable tools that could form the basis of anti-forensic techniques have been laid down. Some of the shortcomings were exploited by utilising a ring zero rootkit, offering an effective way of performing anti-forensics against acquisition applications, like *dd* and *tar*. Two kinds of detection mechanisms are functional, namely a command based detection and a linear variant. The command based detection monitors certain system calls aimed at blacklisted locations in the file system. A suggested solution against such a form of anti-forensics could be found in simply encrypting the communication between user-space and kernel-space The linear based detection mechanism monitors the amount of consecutive reads and writes performed by an application. When the threshold is exceeded, an alarm is raised and the corresponding data streams are intercepted and changed. Hardening the digital forensic process against such a technique can prove difficult, but it could be achieved by moving the non-volatile data acquisition process to the kernel. The main conclusion of this project is that the anti-forensic techniques presented by the author should be deemed a serious risk to the digital forensic process and that dedicated solutions must be engineered and adopted by law agencies to prevent losing ground to technologically skilled suspects.

Contents

1 Introduction

Perhaps slowly, but cybercrime seems on the rise. A survey published by Ponemon shows that in 2014, a 10% increase in cybercrime over 2013 was experienced, hitting almost every industry (Ponemon, 2014). The increase in computer related crime comes with a need for sophisticated forensics aimed at those crimes. The need for those procedures became apparent during investigations regarding the alleged Silk Road owner operating under the pseudonym Dread Pirate Roberts. Roberts, appearantly technologically aware, utilised full disk encryption, making the work of the forensic investigators difficult. Arresting him would only be useful if they could catch him with his laptop turned on and logged in. When Roberts was apprehended, this appeared to be the case. The investigators, in turn, quickly acquired evidence from the machine, which eventually lead to the conviction of Ross William Ulbricht. However, can the data extracted from Ulbrichts' running machine be trusted?

Although not initially applicable, Shannon states that: *"The enemy knows the system"*(Shannon, 1949, pp. 662). This principle is originally aimed at cryptography, i.e. one should make sure the system is secure even if its internals are uncovered. However, the same could apply to the tools used by forensic investigators to acquire evidence from a running computer under investigation. If Ulbricht would have been aware of the ongoing investigation and the tools used during the evidence acquisition, could he have influenced the acquired data? Can a suspect who is aware of the procedures create tools to actively deceive the forensic investigator by presenting different data when an acquisition process is started? This project will set out to develop such a tool, called the Anti-Forensic Toolkit (*aftoolkit*), to investigate if the presented scenario is realistic.

1.1 Research Question

The following research questions were formulated:

What tools can a forensic investigator use to perform a live-acquisition of a Linux computer?

What methods could a Linux system employ to defend against the identified live-acquisition tools?

How could live-acquisition tools be protected from the presented techniques?

1.2 Related Work

Jones (2007) already questioned the legitimacy of live-acquisition tools, stating that forensically sound created images could be *'slurred'*. This means that there is metadata inside the image that points to files that have been changed during the acquisition (Jones, 2007, pp. 4). Jones suggests a solution in the form of an acquisition tool that applies *write command policies*, effectively preventing any write from occurring during the acquisition process.

Rekhis *et al* (2012) suggest a method for forensic investigators that is hardened against the use of anti-forensic toolkits (Rekhis & Boudriga, 2012). They propose changing the standard model for forensic investigations, by including three extra steps concerning the finding, identifying and cancelling of anti-forensic attacks. Garfinkel (2007) presents the, at that time, current techniques used to perform anti-forensics and how to counter them. He discusses how anti-forensic toolkits can, for example, hide themselves when they detect that the system is running from an imaged disk (Garfinkel, 2007, pp. 82). Garfinkel claims that most anti-forensic techniques presented in his work could be overcome with the help of improved monitoring systems (Garfinkel, 2007, pp. 82).

Significant work in this area has been performed by Bilby (2006). Bilby developed a Windows rootkit, called DDefy, which has the ability to hide files from forensic acquisition tools. By redirecting file accesses that have to be hidden towards other (clean) files, the acquisition tool can not acquire the protected content (Bilby, 2006).

Bunden (2009) held a talk at the Blackhat conference in Las Vegas, already warning for the risks of rootkits to the anti-forensic process. He identified different attacks for different purposes, like the destruction or fabrication of data. He concludes that state of the art anti-forensics can defeat live disk analysis (Blunden, 2009) and that the detection of those techniques is difficult.

Related to Bilby and Bunden is the work of Stuttgen and Cohen (2013), who investigated anti-forensic resilient memory acquisitions (Stuttgen & Cohen, 2013). They identified the current issues with memory forensics and created some simple anti-forensic methods by hooking the system API used by the memory acquisition tool. Their technique works for different tools, on different platforms (Linux, Mac OS X and Microsoft Windows). Stuttgen and Cohen provide a solution which does not use the API methods and therefore talks directly to the hardware, effectively circumventing anti-forensic tools. Furthermore, they suggest encrypting the payload of system calls used by the acquisition tools to further deceive possible anti-forensic rootkits.

2 Theory

The following section will explain the relevant theory and background information regarding this project. Time will be spend on the common notion of digital forensics and live data acquisition. Additionally, the field of anti-forensics will be explained and what possibilities lie within it.

2.1 Digital Forensics

Rekhis *et al* (Rekhis & Boudriga, 2010) state the following regarding digital forensics: "[...] aims to conduct a post-incident analysis on compromised systems and make inquiries about past events." Compromised systems could encompass personal computers, network switches, mobile devices, or any other device holding data interesting to an investigation. This data is analysed to reconstruct a chain of events, which in turn could be used in, for example, a court case. Most commonly, it is used by incident-response teams or law enforcement to investigate, amongst others, data breaches, child pornography cases, online drug trafficking, large scale DDoS attacks or, more generic, cybercrime. As an example, two businesses could hire a forensic investigator to acquire evidence in regard to a copyright infringement claim, in which both companies are holding files for which they claim they are the initial creator. An investigator would set out to investigate the creation and access dates of the presented files and reconstruct a timeline offering a valid, forensically sound, conclusion. A different example would be the usage of digital forensics to convict suspected distributors of child pornography by collecting evidence from their systems hopefully indicating who created or spread the aforementioned illegal content.

Within digital forensics, many different types of data are deemed interesting and important as evidence. Which type of data to collect depends on the type of investigation conducted. Acquiring digital evidence from suspects is subject to the same procedure as a normal acquisition, which means a search warrant is needed to perform certain acquisitions (as data stored on a computer is usually considered private). When the correct warrants are in place, data could be collected from network logs (portraying internet activity of the suspect), files located on his or her system, data contained within the memory, archives contained on remote servers, smart phones pertaining to the subject, and more. However, different, uncommon sources are increasingly interesting for forensic investigators. For example, data could also be collected from smart watches, televisions, public transport travel logs or even thermostates.

Just as with normal forensics, the process of digital forensics is heavily bound by general practice and standing procedures. Evidence collected and presented in court must be forensically sound, which means it has to be acquired in a correct way, and prove of this must be available when necessary. The general principles adhered by digital forensics are, for example, laid out in the Advanced Data Acquisition Model (ADAM) (Adams, Hobbs, & G. Mann), n.d.). This model describes the general thoughts and ideas a forensic investigator should follow to deliver forensically sound evidence. The following is contained within ADAM, describing best practices for forensic investigators:

1. The activities of the digital forensic practitioner should not alter the original data. If this is not possible, the effects must be minimal and documented.
2. Handling of the original data, and subsequent copies of it, must be documented. Failing to do so will invalidate the chain of custody.
3. The forensic investigator should only undertake procedures within its own capabilities and knowledge.
4. The forensic investigator must always take personal and equipment safety into consideration while performing an investigation.
5. At all times, the legal rights of suspects affected by the work performed by the forensic investigator should be considered.
6. The investigator must be aware of all policies and organizational procedures pertaining to the activities performed during their investigation.
7. A forensic investigator should communicate discretely and appropriately with the affected or included parties.

As one can observe, the utmost care must be taken by a forensic investigator to guarantee that evidence is not modified by the applied acquisition methods. Furthermore, it is important that proof regarding the forensically soundness of the procedure can be delivered when requested. These standards are perfectly valid for regular data acquisition, in which a device can be switched off to prevent it from changing any data. However, the same principles become unreliable when performing an offline analysis becomes difficult due to, for example, full disk encryption (FDE).

2.2 Live Data Acquisition

Live data acquisition is the process of acquiring data from a system which is online and preferably logged in. Where normal acquisition procedures would simply remove the storage device and perform an offline analysis of the data, live acquisition is performed on the evidence as such. Digital forensics and data acquisition is normally not performed on a live system when it could just as well be performed on an offline system. Performing a live analysis comes with the serious risk of changing the evidence stored on the device under investigation. But, as there are scenarios where normal acquisition is not possible (i.e. FDE), live acquisition of data is becoming increasingly popular. If there are clear indicators that evidence has been encrypted (by looking at running processes or the discovery of encryption software), turning of the device would render it useless without the necessary decryption keys. However, as long as the system is logged in, the underlying data is decrypted and can be retrieved.

There are several ways of performing a live acquisition of data stored on a suspect machine. The acquisition can be performed with tools present on the system under investigation. This comes with the risk that the binaries might have been changed to present faulty data to deceive possible forensic investigations. Alternatively, the investigator could bring a USB drive, containing verified binaries, which in turn are loaded onto the evidentiary system. By doing so, *alien* content is being presented to the system, possibly destroying valuable evidence during the process, which again illustrates the risk of performing live data acquisitions.

Live acquisition is not only performed on the storage devices holding the non-volatile data, like the hard-drive. For example, another possibility is the retrieval of volatile information stored in the RAM. As soon as the system under investigation is shutdown, the RAM starts losing its content. This mean that, commonly, RAM is acquired while the system is logged in and preferably before other data has been acquired (minimising the negative effects of a live acquisition on the evidence stored in the RAM).

Live data acquisition presents us with some important issues The destructive nature of such an acquisition makes it almost impossible for a forensic investigator to not change any data on the evidentiary system. As soon as an acquisition tool is executed, memory contained within the system is changed, in turn changing possible evidence. However, adhering to the *order of volatility* should minimise the destructive effect. This principle states that a forensic investigator should first secure the most volatile storage, to preserve the most evidence. Doing so should minimise the changing of evidence, but it does not completely negate it. Even if the memory footprint of a certain acquisition tool is minimal, evidence is still lost. This does not mean that live data acquisitions are not deemed forensically sound. Performing a proper live acquisition can serve as valid evidence in court.

2.3 Anti-Forensics

Anti-forensic attacks are aimed at increasing the difficulty of, or completely preventing, the work performed by a forensic investigator. What it essentially comes down to is that an attacker tries to outlast the forensic investigator by intervening with the digital forensic process (Blunden, 2009). There is a wide range of possibilities to influence or deceive the investigator. The possible attacks can be, roughly, divided into six categories (Rekhis & Boudriga, 2010, pp. 2):

- Encryption: make evidence unusable when the decryption key is not available.
- Steganography: hide data inside files. An example is the hiding of messages in standard PDF files.

- Data Hiding: hide data inside the file system, for example in slack space.
- Data Wiping: wipe data in such a way that recovery is hard or impossible.
- Tail Obfuscation: obfuscate, for example, the source of an attack by spoofing e-mail headers.
- Attack the forensic tools: infect the tools or systems used by the forensic investigator to directly influence the evidence or to render the process not-forensically sound.

Most anti-forensic attacks aimed at the live acquisition process start with encrypting the data, forcing the investigator to perform a live analysis of the system. This means the investigator is depended on the software running on the system, and in turn means the system is now in control. The investigator is able to identify tools that could possibly influence acquired evidence, for example, by looking at the currently running processes or loaded modules. However, all the information presented can already be influenced (sophisticated rootkits are capable of hiding themselves from commonly used forensic techniques).

Generally, anti-forensics adheres to the *Intruder's Dilemma*, as pitched by Richard Bejtlich (Bejtlich, 2009). A commonly known dilemma is the *Defender's Dilemma*; an intruder only needs to exploit one victim to compromise the entire business. The intruder's dilemma is the opposite; "a defender only needs to detect one of the indicators of the intruders presence in order to initiate incident response within the enterprise"(Bejtlich, 2009). This is specifically the case for anti-forensic techniques. As soon as an investigator has one indicator that the evidentiary system is intervening with the digital forensic process, counter measures can be taken to prevent any damage to evidence or to completely neutralise the anti-forensic tool. This means that, when developing anti-forensic techniques, close attention must be paid to detectability and stealth mechanisms. For example, the presence of certain binaries or control scripts could lead an investigator to further inspect the evidentiary system on possible malicious tools.

2.4 Forensics and Linux

Most forensic procedures are aimed at the Windows flavour of operating systems. Information on how to perform forensics, and live data acquisition in particular, on Linux environments is scarce. Luckily, court transcripts from the Silk Road lawsuit describe several steps taken by the forensic investigators in regard to data acquisition and triage of the, alleged, Linux laptop of Dread Pirate Roberts (*Ross Ulbricht Day 5 Court Transcript*, 2014; *Ross Ulbricht Day 6 Court Transcript*, 2014).

2.4.1 Case Study: Silk Road investigation

In 2013, Ross William Ulbricht was arrested and accused of running the large scale online drug market Silk Road. Based on evidence acquired from, amongst other sources, his laptop during a live acquisition led him to be sentenced to life imprisonment on 29th May 2015. The following information comes from the court transcripts describing day four, five and six of the trial (transcripts of other days are not made publicly available).

There are several steps depicted in the transcripts, starting with the triage phase, that describe how data was acquired. The officer that seized the laptop of the suspect performed an initial acquisition of interesting files by archiving them with the *tar* utility and storing the created archives on a USB storage device pertaining to the forensic investigator (*Ross Ulbricht Day 6 Court Transcript*, 2014, pp. 1217). By doing so, the officer in charge allegedly changed all the access times of the archived files. However, the court accepted the archives as legitimate evidence. Also, an initial run of the *tar* program returned an error indicating a wrongly executed command. After the initial files were acquired, the investigator proceeded with creating a block level copy of the hard-drive inside the laptop. By utilising the *dd* tool (and not the commonly used forensic variant *dcfldd*), the data was then stored on a storage device supplied by the forensic investigator (*Ross Ulbricht Day 6 Court Transcript*, 2014, pp. 1234). There is no mention of how the investigator acquired root access to execute the *dd* command. The created image has been handed over to a different forensic investigator, who analysed the image with FTK Imager to extract the needed evidence. The integrity of all the files and images has been checked and

verified with MD5 checksums. During every step in the procedure, images have been taken with an FBI issued camera, which was running 40 minutes slow.

Clearly described in the transcripts is the issue of encryption to a forensic investigator. As there was no guarantee untill when the laptop would be in a live state (i.e. there might be an auto-shutdown mechanism, or the laptop simply runs out of battery), it was decided not to adhere to the order of volatility. Not adhering to this principle means that, the tools used to acquire the data, have tainted the memory which was retrieved three days later (*Ross Ulbricht Day 6 Court Transcript*, 2014, pp. 1236).

Comparing the described techniques with the aforementioned ADAM model shows that several crucial mistakes have been made. Access times have been altered, contrasting with the principle of changing as little evidence as possible (instead, the *cp -pr* command would have left all timestamps in tact). Apart from that, the execution of the *tar* program initially failed (possibly changing evidentiary error logs). Judging by the error message, the investigator in charge possibly tried to archive a directory which was also storing the created archive. Overall, it looks like the forensic process was not strictly adhered.

2.4.2 Field Expertise

The Dutch High Tech Crime Unit has provided information regarding the data acquisition process when considering Linux systems. They state that, when performing a live acquisition, there is a trade-off to be considered. Either, the investigator makes use of the binaries presented on the evidentiary system, effectively not introducing new content to the system. However, the investigator is therefore subject to possible changes to the binary. When this is deemed a serious risk (i.e. the suspect is perceived to be technologically aware enough to perform such an attack), the investigator will present trusted binaries. In regard to the tools that they use, they stated that mostly common tools are used to perform live acquisitions. For example, the *cp*, *dd* and *tar* utility are commonly used.

Not every system is checked for the presence of anti-forensic tools. When a suspect is deemed technologically skilled enough for such techniques, or other indicators are found during the forensic process, extra steps are taken. For example, the */proc* and */sysfs* could be checked for the presence of certain kernel modules. Commonly perceived anti-forensic techniques are, for example, the remote locking of a computer as soon as a paired smartphone leaves its near vicinity. However, more advanced anti-forensic techniques, like DDefy which is actively deceiving the forensic tools, have not yet been encountered.

2.4.3 Non-Volatile Data Acquisition

Non-volatile data is all the data present on, for example hard disks and other storage media. When possible, a block-level copy of a disk is created, ensuring a one to one comparison with the original disk. The remainder of the digital forensic process will be carried out on the created image, reducing the risk of changing the original evidence (CERT-EU, 2012).

Most of the tools created and used within the forensic world are aimed at Windows environments. However, there are tools that can be used to perform a live-acquisition of a Linux system. As presented in the previous court case, a commonly used tool is *dd*, which can be used to create a block-level copy of a storage device. As *dd* is not commonly seen as a forensically sound tool, *dcfldd* was developed, which is the forensic counterpart to *dd*. FTK Imager and EnCase, which are commonly used on Windows to create and analyse forensic images, have command line variants written for the Linux operating systems. Also mentioned throughout the Silk Road court case is the usage of the *tar* archive utility during the triage stage to create a quick copy of interesting files. It bundles specified files in one archive, without applying any form of compression (although flags could be set to compress the files with, for example, *gzip*).

2.4.4 Volatile Data Acquisition

Volatile data, in contrast the non-volatile data, comes with the serious risk of disappearing when the system is switched off. Amongst others, data contained within the memory, certain logs and

network connections are considered volatile (CERT-EU, 2012).

Acquiring data from volatile sources (like the RAM) is more complicated. Where, in a Linux system, internal storage is exposed to the user-space by the */dev/sda* device, newer systems do not supply a similar exposure of the RAM. Older linux systems allowed access to the ram through the */dev/mem* device, which means data could be acquired by simply using the *dd* utility. However, for newer systems, a loadable kernel module (LKM) has to be used to expose the memory. The available options are, amongst others:

- Linux Memory Extractor (LiME): LKM that allows the acquisition of volatile memory on a Linux-based system.
- Linux Memory Grabber: utilises the previously mentioned LiME module, but creates a file that can be directly analysed in Volatility.
- fmem/pmem: LKM that both expose the memory in such a way the it can be acquired with common tools, like *dd*.

2.5 Rootkits

Before describing the possibilities that rootkits create, it is first important to nail down a definition. Commonly, rootkits are seen as malicious and made to create, for example, persistent backdoors in systems. However, Greg Hogland (2005) gave a more accurate description, stating the following: *"A rootkit is a tool that is designed to hide itself and other processes, data, and/or activity on a system"*(Blunden, 2013). In essence, a rootkit is a piece of (malicious) code, hiding in difficult to find places in a target system (like the kernel). The x86 hardware architecture is designed in a layered fashion, called *privilege rings*. At the core of the system is the kernel, which is running in ring zero. The kernel has the most direct access to the underlying hardware, and is thus the most privileged. Ring one and two contain device drivers, needed by the applications running in the final ring. Ring three is hosting the user-space applications, and is the least privileged of all rings.

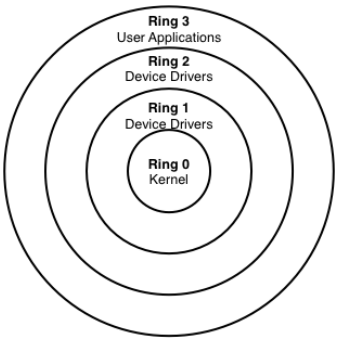

Figure 1: The 4 security rings commonly perceived within in operating systems. Ring 0 has the highest privileges and ring 3 is the least privileged.

A rootkit can hide within these privilege rings to form either a user-space rootkit or a kernel-space rootkit. The user-space rootkit runs next to the other user-space applications and adheres to the same privileges. On the other hand, a kernel-space rootkit mimics itself as being a device driver, gaining high privilege access to the system. Kernel-space rootkits are especially powerful, as all commands coming from applications in user-space can be intercepted within ring zero. Apart from that, by hiding in kernel-space, detecting the rootkit becomes especially hard (detection from a lower privileged ring is very difficult).

Rootkits in the field of anti-forensics are not a new phenomena. As already described, Bilby (2006) developed a rootkit to be used for anti-forensic purposes. Although generally perceived as being an attack tool, it could just as well be utilised to defend a system. Getting back to the previously stated definition of rootkits, hiding data or activity on a system would be perfectly suited for anti-forensic purposes.

3 Approach

This section will describe the approach followed throughout this research. The project has been divided into several stages, which combined deliver the information needed to create a proof-of-concept anti-forensic tool. First, information will be gathered on commonly used forensic tools. The following are under investigation:

- *dd/dcfldd*: dd has been chosen for its appearance in the Silk Road court case. Dcfldd is commonly perceived as the forensic counterpart to dd.
- *FTK Imager*: this is, next to EnCase, the most commonly used tool to acquire and analyse data in Windows environments. It offers a command line version for Linux operating systems, which can be used to perform a live acquisition of data.
- *Tar*: selected due to its appearance in the Silk Road court case.

Due to the previous work performed by Stuttgen and Cohen (2013), it has been decided not to focus on memory acquisition.

3.1 Environment

The anti-forensic scenarios will be developed for the following system, however, should function on most newer Debian based kernels and operating systems:

- Operating System: Ubuntu 12.04.05 LTS (Precise Pangolin)
- Kernel: 3.13.0-32-generic

3.2 Information Gathering

By using the *strace* tool, which lists system calls that are used during execution of an application, information has been gathered regarding the inner workings of the tools under investigation. Strace is not the only way of investigating a programs internals. Most tools under investigation are open source, which means their source code could be analysed to yield important attack vectors. However, as FTK Imager is a proprietary application, and the pertaining source is not released to the public, the decision has been made to follow a black box approach in which the internals of the application are unknown.

3.2.1 dd & dcfldd

The *dd* tool functions relatively easy. The kernel executes the dd-binary by issuing the *execve* command. After the binary has been loaded, the application starts executing. It begins by acquiring relevant information from */etc*. When the application is ready, it opens the block device (*/dev/sda*) by using the *open* command. If the device has been opened correctly, the associated file descriptor is copied by issuing the *dup2* command. The application performs the same routine for the newly created output image, which stores the acquired data. When both the block device and the new image are open, *dd* starts reading (*read* data from the block device and consecutively writing (*write*) it to the newly created output image. When all the data has been written (or the specified amount of bytes has been reached), *dd* closes the block device and the image. A shortened trace created with the *strace* application can be found in Appendix A.

Dcfldd behaves more or less the same as the *dd* utility, the only exception being the lack of duplication of the associated file descriptors. Where *dd* creates a copy of the file descriptors corresponding to the opened block device and newly created image, *dcfldd* uses them as is. Apart from that, the reading and writing of data is performed in the same manner. A trace corresponding to the execution of the *dcfldd* utility is listed in Appendix B

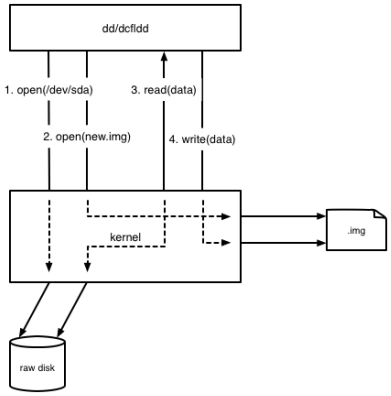

Figure 2: Schematic overview of the interaction between the kernel and the *dd*/*dcfldd* utilities.

3.2.2 FTK Imager

FTK Imager comes in several forms. The variant under investigation is the FTK Imager Command Line version for 64-bit Debian based systems (versions 3.1.1) [1].

When the binary is executed, it starts by acquiring information, needed during execution, from files located in the */etc* directory (similar to *dd* and *dcfldd*). When FTK is ready with setting up, it opens the specified block device (*/dev/sda*) and performs an *llseek* to get the start and end sectors of the device. When the size-parameters are known, *stat* is executed to get general information about the specified block device. As FTK identifies it as being a block device, several *ioctl* calls are executed to acquire parameters pertaining to it (like the block size used by the device). If it is not identified as being a block device (i.e an image has been specified as input instead of a block device), the *ioctl* calls are not executed. After the block device has been opened, and the relevant information has been acquired, the output file will be created and consecutively data will be read from the block device and stored in the newly created file. Where *dd* and *dcfldd* utilise regular *read* commands to acquire the data, FTK Imager is multi-threaded, thus utilises *futex* to dispatch the reads and writes to the correct threads. When all the data has been transferred, the output file and block device are closed, after which execution is completed and the binary is unloaded. A trace pertaining to a run of FTK Imager can be found in Appendix C.

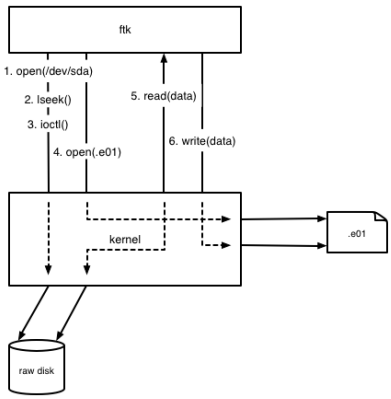

Figure 3: Schematic overview of the interaction between the kernel and FTK Imager. Not every system call is included, namely the *stat* command has been negated to avoid clutter.

[1] FTK Imager CLI can be freely downloaded from: `(http://accessdata.com/product-download/digital-forensics/debian-and-ubuntu-x64-3.1.1`

3.2.3 Tar

The previously presented tools are used to create copies of raw block devices (or images). Tar, on the other hand, is not used to create a block-level copy of the data. This means that *tar* does not use the */dev/sda*, but takes regular folders as input. A run of the *tar* application starts, like the previous tools, by acquiring information from files located in */etc*. When the information has been retrieved, *tar* creates the output archive by using the *creat* system call (see Figure 4. As *tar* interacts with the actual filesystem (and not the underlying raw data), to acquire files, it has to traverse the directory tree. It does this by using the *openat* system call, which takes an already opened folder, and traverses the content. The traversal process starts with the specified target to archive. If the target is a directory, it traverses the corresponding entries (and in turn their entries). When it encounters a file, *tar* writes it to the output archive file. This process continues until all the data contained within the target has been written to the archive file, after which all opened files are closed. A trace pertaining to the *tar* program can be found in Appendix D.

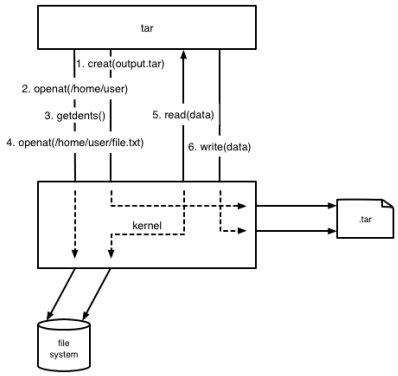

Figure 4: Schematic overview of the interaction between the kernel and the *tar* utility. Not every system call is included, namely the *stat* command has been negated to avoid clutter.

3.3 Anti-forensic Scenarios

A simple scenario could be found in the loading of binaries. The system under investigation could, for example, hold a malicious *dd* binary. When the system raises an alarm regarding the execution of a non-malicious *dd* application, the loaded binary could be killed and in turn the malicious one is executed. This means that, even if the investigator came well prepared with own binaries, the actual acquisition is performed with the malicious binary running on the system.

However, there might be a simpler, generic solution as following the aforementioned approach means that for every available imaging application, the attacker should hold a malicious counterpart, with the corresponding detection mechanisms to facilitate the execution. In contrast, a more simple approach would be to create a generic method to intercepting these tools, by utilising a ring zero rootkit. An interesting feature is that, because they run in kernel space, they can intercept system calls coming from higher level rings (like ring three, which houses user processes). As the live-acquisition tools are running in ring three, the system calls they use can be intercepted and changed by the rootkit running in ring zero.

Intercepting the system calls is more commonly referred to as *hooking*. A hook is simply a method attached to the original system call. For example, by hooking the *open* system call, all uses of this call can be traced and influenced. However, later versions of the Linux kernel no longer expose the system call table, which holds the addresses pertaining to the system calls, in a writeable fashion. Luckily, the chosen rootkit-framework offers mechanisms to retrieve the table and make it writeable.

The rootkit will deliver the needed detection and interception mechanisms. By hooking system calls generically used by imaging and archive utilities, hopefully a detection mechanism can

be created. When such a mechanism is in place, a vast list of anti-forensic scenarios become apparent. Instead of, as earlier presented, switching the requested binary for a malicious one (by, for example, hooking the *execve* command), the data presented to the applications can be changed without touching the internals of the acquisition tools.
Hiding data from tools, or destroying evidence all together, will inhibit the work from a forensic investigator. However, presenting data from a different user changes the entire forensic process. By injecting known clean files, and maybe even include a few bad ones, the investigator might fall for such a decoy (Blunden, 2013). By effectively framing a different user on the computer (which in turn does not have to be a real user, but could just as well be an account created by the suspect). This makes *framing* a more advanced form of hiding or wiping data, by presenting back structured, valid data.

Changing data streams is not the only possibility when it comes to detecting acquisition tools and performing anti-forensics. Judging by the information presented in the Silk Road court transcripts, the forensic investigator utilises a USB stick to store the acquired data. The plugging of this USB stick can easily be detected and in turn be exploited. However, this is outside of the scope of this project.

Apart from the previously presented scenarios, an anti-forensic rootkit could just as well perform the already known anti-forensic techniques. It can hide data by changing flags pertaining to the underlying inodes, or simply intercept the system calls used to fill directories and filtering out files that must not be visible. If no effective anti-forensic can be performed in the form of deception or obfuscation, it could simply detect the presence of a acquisition tools and shut down the system. Alternatively, data destruction could be performed by shredding metadata or complete files. As a final scenario, it could fabricate data and thus leaving false trails for the forensic investigator.

4 Implementation

The following sections described how some of the aforementioned anti-forensic scenarios will be implemented. First, the necessity for a rootkit will be described. Afterwards, information will be gathered on how the identified behaviour of the applications under investigation can be exploited.

4.1 Ring Zero Rootkit

The ring zero rootkit will allow access to the system calls used by the applications under investigation. As previously explained, the ring zero rootkit will mimic itself as a device driver and because of that gain a high level of privileges. Instead of creating a rootkit from scratch, a framework will be utilised.[2]. The chosen framework already offers the capabilities needed to hide files from the general directory listings, which will be used to hide the fake image and home directory from the investigator.

The rootkit will be extended to a functional application. Control will be exercised by creating certain files in the */tmp* directory present on Linux systems. By hooking the *open* system call and parsing the requested filenames, certain patterns can be picked up and coupled to command and control of the rootkit. Doing so offers a form of control that does not need any other applications, again reducing the chance of detection. The effectiveness of an anti-forensics application is based on how well it is hidden and any indicator that something malicious could influence the data acquisition process could seriously inhibit its effectiveness. The presence of an undefined control application could be seen as a serious indicator, leading the investigator to take extra precautions. Luckily, the */tmp* directory is often flushed, effectively leaving no traces. In turn, the *open* system calls could just as well be changed to prevent the actual creation of the file, as control is exercised through the *filename* field in the *open* method, and not the actual presence or creation of the requested file.

By hooking the system calls, code executed by the rootkit is perceived by the system as coming from the currently running process. This means that the solution as pitched by Jones (Jones, 2007) will no longer be valid; writes are perceived to be coming from the imager itself, which means the write policies must allow it. Apart from that, by instigating hooks in to the methods used to fill */proc* and */sysfs*, the rootkit can hide its own existence.

A signature based detection would be specifically hard, as most of these techniques have to be specifically crafted for the system they run on. This means that the signature corresponding to the LKM will not be identified as malicious by most antivirus products. However, adhering to the order of volatility, a skilled forensic investigator could detect its presence within the memory dump created before starting the acquisition of non-volatile data.

4.2 Imager Deception

The imagers under investigation all utilise more or less the same method. The first stage of performing anti-forensics is detecting the activity performed by the acquisition tool. The simplest procedure would be to hook the *execve* method, and check if, for example, the *dd* binary is loaded. However, this detection mechanism can be easily circumvented by renaming the binary. A more solid approach would be to intercept the *open* system calls and parse them to check if they are aimed at the raw block device (*/dev/sda*). Under normal operation, a Linux system usually does not try to open the block device, which makes this behaviour rather specific to the presented imagers.

Opening the device happens with a standard *open* system call. Given a pathname for a file, the *open* call returns a file descriptor pertaining to the opened file, which in turn is used by the read methods to retrieve data. To deceive the imagers, it has been decided to simply open a different file or location, instead of the specified raw block device, leading the imager to retrieve data from an unintended source (see Figure 5).

[2]The Suterusu rootkit (`https://github.com/mncoppola/suterusu`) has been used as a basis for the work presented in this project.

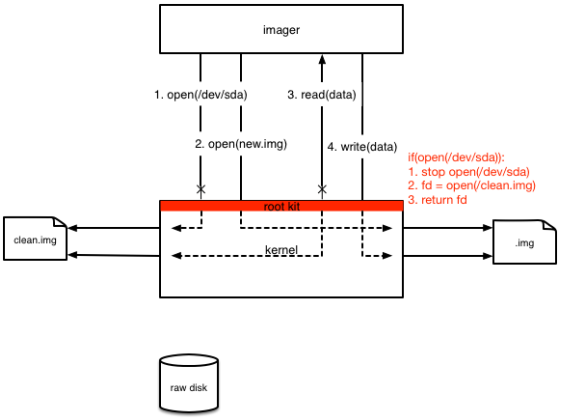

Figure 5: Schematic overview of the interaction between the rootkit and the imagers. Depicted in red is the behaviour, in pseudo-code, of the rootkit.

```
1  hook read(file descriptor):
2    if(previous command == write()):
3      and(if(file descriptor == previous file descriptor):
4        linear_read_counter ++;
5
6  hook write(file descriptor)
7    if(previous command == read()):
8      and(if(file descriptor == previous file descriptor):
9        linear_write_counter ++;
10
11 if(linear_read_counter >= threshold) or
12 if(linear_write_counter >= threshold):
13   alert
```

Figure 6: Simple detection algorithm for the linear behaviour exhibited by most forensic imagers.

Although parsing the *open* system call is relatively robust, simply changing the methods used by the applications would break this detection mechanism. Luckily, imagers exhibit a notably linear behaviour in their read and write commands. By looking at the straces in Appendix A, B and C, it is clearly visible that a read is followed by a write. Apart from that, the file descriptor used by the *read* command, is the same as the previous one, and the one before that, etc. The same holds for the *write* command. This linear behaviour can be detected by simply storing the corresponding file descriptors and counting the amount of read and writes (see Figure 6) initiated by the same process.

4.3 Tar Deception

For the tar utility, it has been decided to try and extend the known anti-forensic techniques by implementing a form of framing. The *tar* utility, judging by the Silk Road court case, is used to quickly archive interesting files. As a scenario, the archiving of a users home directory has been chosen. *Tar* makes extensive use of the *openat* system. To achieve the aforementioned framing behaviour, the *tar* process will not be pointed to an arbitrary file within the filesystem like performed when deceiving the imagers. When *tar* is executed to archive the contents of the home-directory, the corresponding openat system call is changed towards a different directory effectively returning files from a different user.

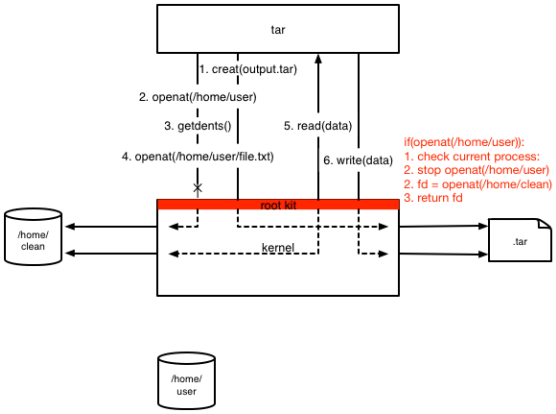

Figure 7: Schematic overview of the interaction between the rootkit and the *tar* utility. Depicted in red is the behaviour, in pseudo-code, of the rootkit. As can be seen, any call towards the */home/user* directory will be pointer towards */home/clean*

5 Results

This section will present the results of implementing the aforementioned anti-forensic scenarios. The imagers have had their behaviour changed in such a way that they return data stored in a disk image instead of the raw block device. Although less advanced, the *tar* application is intercepted to present a form of framing a different user on a computer under investigation. The corresponding code has been released to the public through Bitbucket [3].

5.1 Command Based Detection

The command based detection proved successful. By intercepting and parsing the *open* and *openat* commands, all tested imagers and *tar* could be deceived. In essence, every open command initiated by an application is parsed by a rootkit. When the specified filename is in a blacklist contained in the rootkit, the corresponding system call is stopped. Afterwards, the rootkit opens a different file, containing some fake data, and returns the corresponding file descriptor to the running application (either an imager or *tar*).

5.1.1 Imagers

The command based detection, which means intercepting the open commands aimed towards the raw block device (*/dev/sda*) has proven to be successful. Under normal system operation, no false positives have been perceived in which an application would have been labeled incorrectly as an imager.

When the *filename* parameter pertaining to this command is assigned to */dev/sda*, the command is intercepted and stopped. In response, the rootkit opens a raw disk image located in the file system (*/clean*). When there are no errors, the corresponding file descriptor is returned to the imager and the imager proceeds with its normal operations as if it was copying from a raw block device.

However, for this approach to work for the FTK Imager application, several return values have to be spoofed. As FTK is expecting a raw block device, it probes it with some *ioctl* commands. If the commands return an error, FTK stops its operation and kills the process. Luckily, simply intercepting the commands and returning non-erroneous values is enough to deceive FTK. Apart from the *ioctl* commands, all imagers initiate a *stat* to acquire information from the opened block device. These commands have to be intercepted and in turn changed to be executed towards the fake image presented in the previous *open* interception. Following this approach, every imager under investigation could be deceived (see Listing 8).

[3] `https://bitbucket.org/yonne/aftoolkit`

```
1  ioctl counter = 0
2  ioctl threshold  = 4
3
4  hook open(filename, parameters):
5    if(filename == /dev/sda):
6      move from kernel space to user space
7      fake filedescriptor = open(/clean, parameters)
8      imager boolean = 1
9      return to kernel space
10     return fake filedescriptor
11
12 hook stat(filename, parameters);
13   if(imager boolean == 1 && filename == /dev/sda)
14     return stat(/clean, parameters)
15
16 hook ioctl(filedescriptor, parameters):
17   if(imager boolean == 1):
18     ioctl_counter++
19   if(ioctl counter > ioctl threshold):
20     return true
21   else:
22     ioctl counter = 0
23     imager boolean = 0
```

Figure 8: Deception algorithm for a command based detection mechanism.

```
1  hook openat(pathname, parameters);
2    if(current process == tar): #check seems obsolete as openat() is not an often
         used command
3      if(pathname == contained in blacklist):
4        move from kernel space to user space
5        return openat(/home/clean, parameters)
```

Figure 9: Hook used to intercept and change the *tar* command.

5.1.2 Tar

Command based interception proved successful in deceiving *tar*. By intercepting and parsing the Although the detection mechanism is easy to circumvent by simply changing the binary, the exhibited behaviour by the archive utility is found interesting. When the *tar* process is intercepted, the data presented back to the forensic investigator through the command line represent the original directory structure, with the data from a different home directory. This means that, to the investigator it will look as if files are being retrieved from the specified directory, when in essence they come from a different user, effectively framing this account. *openat* system call the target specified to *tar* could be changed (see Listing 9).

5.2 Linear Based Detection

The linear based detection mechanism has been implemented as presented in listing 6. Internally, the rootkit stores a counter that, based on the current process being executed and the corresponding file descriptors, decides if a process is exhibiting linear behaviour. The threshold was found to be best at 100 consecutive read and writes by the same process. Lowering the threshold means the imagers are more quickly detected, however, they are not the only applications exhibiting linear behaviour. By specifying a threshold of, for example, 50, many other applications were flagged as being linear and thus have their system calls intercepted and changed. This means that normal applications, like *ls* or *dmesg* were rendered useless. Even though a threshold of 100 negates most false positives, a blacklist was still implemented to prevent certain applications from being picked up, namely *dmesg* and *hexdump* for debug purposes. This procedure has only been tested on the imaging applications.

```
1  total offset = 0
2  cache struct lp;
3
4  hook read(file descriptor, count, parameters):
5    if(previous file descriptor != file descriptor):
6      linear read counter = 0
7      previous file descriptor = file descriptor
8    else:
9      linear read counter++
10     cache this read(file descriptor, counter, current process)
11   if(linear read counter >= linear read threshold):
12     for(i in open cache):
13       cr = open cache[i]
14       if(kill the application)
15         kill
16       else if(total offset > size of the fake image):
17         count = size of the fake image - total offset
18         kill the application = 1
19       if(cr->mode == 438 && cr->flags == 577 && cr->process == current process)
             :
20         if(first time detected):
21           lp = cache this open #probably the output file
22           total offset  = parse read cache
23           wipe the output file
24           move from kernel space to user space
25           ret = read n(total offset) bytes from the fake image
26           write ret to the output file
27           flag the process
28           first time detected = 0
29           move from user space to kernel space
30           return the read count
31         else
32           move from kernel space to user space
33           ret = read n(count) bytes from the fake image
34           total offset += count
35           lp->count = count
36           lp->buffer = buffer #cache the created buffer
37           move from user space to kernel space
38           return the read count
39
40 hook write(file descriptor, count, parameters):
41   if(lp->process == current process && lp->buffer != current buffer && lp->file
         descriptor == file descriptor)
42     drop this write #flag this write as being from the original data stream
```

Figure 10: Simplified interception procedure for a linear based detection algorithm.

5.2.1 Imagers

Linear based detection proved relatively easy, but actually intercepting and changing the pertaining commands came with a whole range of issues. The command based detection intercepts the process at the absolute beginning, before any data has been retrieved or written, which means the interception starts with a clean environment. A linear interception leads to the situation in which several bytes of data have already been retrieved from the block device before the rootkit is aware of its presence. This means that the rootkit performs a clean up stage, in which it wipes the content of the output file created by the imager. Afterwards, it transfers the so called *missing bytes* from the fake image. It transfers the amounts of bytes previously retrieved from the raw block device. To facilitate this, the rootkit holds several caches, which store read, write and open commands. As soon as an application has been flagged as being an imager, the corresponding open cache is parsed to find the input and output files. Afterwards, the read cache is parsed to determine the total amount of previously transferred bytes. The same amount of bytes is retrieved from the fake imager, after which control is handed back to the imager. Every consecutive read performed by the flagged application is intercepted, stopped and replaced by a read operation targeted towards the fake image. The writes command executed by the application are blocked off if they do not contain our changed buffer, effectively preventing original data from being written to the output file.

A simple test shows the effectiveness of this interception. Executing, for example, *dd* with a count of one (i.e. one block of 512 bytes is retrieved), the master boot record (MBR) of the original drive ends up in the output file. As soon as the *dd* process runs with a count higher than the threshold, the MBR of the fake disk image is stored. The corresponding debug information (displaying the rootkits internals) can be found in Appendix E.

```
root@ubuntu: ~$ hexdump -n 512 /dev/sda
0000000 63eb 1090 d08e 00bc b8b0 0000 d88e c08e
[...]
root@ubuntu: ~$ hexdump -n 512 /clean
0000000 48eb 1090 d08e 00bc b8b0 0000 d88e c08e
[...]
root@ubuntu: ~$ dd if=/dev/sda of=output.img count=512
512+0 records in
512+0 records out
262144 bytes (262 kB) copied, 0.0015403 s, 170 MB/s
root@ubuntu: ~$ hexdump -n 512 output.img
0000000 63eb 1090 d08e 00bc b8b0 0000 d88e c08e
[...]
root@ubuntu:~$ dd if=/dev/sda of=output.img count=512
512+0 records in
512+0 records out
262144 bytes (262 kB) copied, 0.0219535 s, 11.9 MB/s
root@ubuntu ~$ hexdump -n 512 output.img
0000000 48eb 1090 d08e 00bc b8b0 0000 d88e c08e
[...]
```

However, this procedure is far from stable. As a start, at point of writing, the proof-of-concept only works for *dd* and *dcfldd*, due to FTK Imager being multi-threaded. The usage of *futex* in combination with different process threads prevents the rootkit from detecting its behaviour. Apart from this, the output file created by an imager that has been intercepted does not yield a one to one comparison with the fake input image. When comparing the MD5 checksums of the two images (the fake input image and the output file created by the imager), they only seem to compare up to a randomly observed point. No original data seems to be present in the output file, which is of course the wanted behaviour, but the structure of the image is corrupted.

6 Prevention

Demonstrated were two different forms of detecting and intercepting forensic acquisition tools. This section will theorise how these anti-forensic techniques can be prevented and how the acquisition process could possibly be improved.

6.1 Encryption

First demonstrated was a simple interception method in which the payload of certain system calls is parsed for a so called *magic string*. The destructive nature of it could be seen in the *tar* utility. However, harnessing for such a technique is relatively simple. Encrypting the payload when passing it down to the kernel will effectively prevent the rootkit, in its current form, from detecting or parsing the content. The Linux kernel is already equipped with the necessary libraries to support such a feature, namely the *crypt.h* implementation.

However, simply encrypting the payload will render it useless to the kernel as such. Therefore, an improved system should come with a trusted kernel module to which the imager talks. When adhering to the order of volatility, in most cases an investigator should first acquire the memory before it attempts to retrieve stored data. This means that, as soon as the memory is acquired, kernel modules can be loaded without destroying valuable evidence. Of course such a system would deliver a significant amount of overhead due to the fact that every system call coming from the imager has to be decrypted by the kernel module, before it is passed to a the actual kernel (see Figure 11. A different solution would be to encrypt the entire command, instead of just the

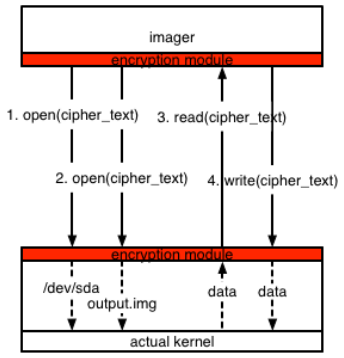

Figure 11: Theorized hardening against rootkits intervening with the live data acquisition process. By creating a trusted environment in the form of a kernel module and a trusted binary, communication can possibly be obfuscated enough to deceive anti-forensic rootkits.

payload. This means that even linear based detection becomes harder, as no semantics can be coupled to the intercepted commands. The downside to this approach is the fact that encrypted commands would stand out from normal commands, delivering an easy detection mechanism to be abused by anti-forensic techniques. Simply encrypting the payload would be harder to detect, although arguably an attacker would still be capable of performing some form of entropy calculation on the payload and draw conclusions from that.

Apart from preventing detection, the changing of retrieved data or intercepted commands becomes near to impossible. If the evidentiary system does not contain the key used by the decryption module in the kernel, any changes to buffers or parameters pertaining to these commands would fail to decrypt. This failure to decrypt offers a perfect opportunity to inform a forensic investigator that something is trying to change commands used by the imagers, increasing the detection rate of such techniques.

6.2 Kernel-mode Acquisition

Although against most common practices, the acquisition process could be performed entirely within the kernel. One of the main weaknesses in live acquisitions is the fact that commonly known, easy to intercept, system calls are used to retrieve data. However, again adhering to the

order of volatility, the acquisition process could just as well be performed with a loadable kernel module (after the RAM has been acquired).

As demonstrated, regular imagers heavily interact with the system through system calls. Kernel-mode acquisition would require just the initialisation to be performed with a regular system call. The LKM could be controlled by a simple application. The application will send a authentication token through a hooked system call (for example the *ioctl* system call). When the LKM parses the system call and finds the authentication token, it sends back an acknowledgement to indicate to the application that it is ready to receive encrypted commands. The application in turn will send the initialisation parameters needed for the LKM to start acquiring the data (like the input, output and count). As soon as the LKM is initialised, no more communication between user-mode and kernel-space will be needed (reducing the chances of interception). The rest of the acquiring process takes place in the kernel, directly aimed at the virtual file system layer (VFS).

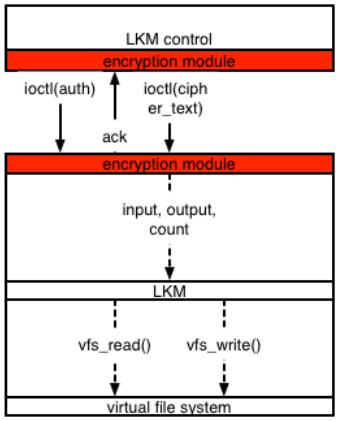

Figure 12: Suggested kernel-acquisition LKM. Communication and control is exercised through an encrypted channel, the acquisition process is carried out within the kernel.

6.3 Hiding

When bad practice is not an issue nor harms the evidence, the entire acquisition tool could be hidden within other processes. Although heavily inspired by the functioning of rootkits, such an approach would be hard to detect. By hooking generic system calls and intercepting them for random applications, the procedures to read and write data could be injected in different processes. When deployed correctly, all the reading and writing of data could be spread out over different applications.

6.4 Hardening Against Anti-Forensic Kernels

The previously presented solutions are aimed at the presented rootkit. However, a suspect could run the same code just as well directly from within the kernel. Doing so would create an anti-forensic kernel, which would still be able to function with the previously presented solutions. The solutions run in kernel-space and by doing so would effectively be hidden from most anti-forensic tools. However, the presented code can be inserted deeper in to the kernel, calling for different prevention techniques.

Apart from that, a custom kernel (or anti-forensic rootkit) could simply block the loading of new kernel modules or applications. This would mean that the modules needed to perform encryption or a kernel-mode acquisition would not be useable. However, simply blocking new content will instantly reveal the existence of anti-forensic techniques in place and doing so would not be the preferred route for an attacker to take.

To offer resistance against anti-forensic kernels, or when the previously presented mechanisms are not applicable in the given situation. a non-linear acquisition could be performed. For ex-

```
1  abuse_applications[] = {ls}
2  counter = 0
3  block size = 512
4  buffer
5
6  hook_stat:
7    if(current application == abuse_applications[i])
8      if(counter == disk size):
9        indicate ready
10     buffer = read data(/dev/sda, block size)
11     write data(buffer)
12     count += block size
13
14 hook_ioctl:
15   if(current application == abuse_applications[i])
16     if(counter == disk size):
17       indicate ready
18     buffer = read data(/dev/sda, block size)
19     write data(buffer)
20     count += block size
```

Figure 13: Hiding of the acquisition process within other, trusted applications, like *ls*

.

ample, acquisition could be performed on the file system level. By performing seemingly random reads through the file system, detection by possible anti-forensic techniques becomes difficult. Simply masking the reads as normal system behaviour would require a very sophisticated algorithm to still be detected. Such a tool would be a useful replacement for the *tar* utility. However, when a block-level copy is required, accessing files through the file system will not lead to all the data. Data could be hidden within slack-space, the master boot record, or other, more innovative, locations.

To create a legitimate, trusted, block-level copy, dedicated hardware solutions should be developed. The acquisition must be performed without utilising parts of the operating system running on the evidentiary system. Previously, imaging hardware has been aimed at extracting the RAM from a running system. For example, the WindowsSCOPE PCIe card can be used to perform a memory acquisition on a running Window systems [4]. Similar devices could be used to acquire data from non-volatile sources by directly communicating with the hardware.

[4] `http://www.windowsscope.com/`

7 Discussion

The presented work raises the following question: how feasible is such a defense mechanism in real-life? Is this technique a realistic threat to the current digital forensic investigation process?

Although unstable, the proof-of-concept as such could already wreak havoc on a forensic process, given that a forensic investigator is not paying close attention. As such, there are plenty of indicators that the evidentiary system is intervening with the acquisition process. For example, linear intercepted imagers do not always create a correct output file, or create files that are smaller than expected. On the other hand, the rootkit is hidden from most places where an investigator would commonly check, like */proc* or */sysfs*. An attacker, or from a different perspective a defender, with enough time and knowledge about the digital forensic process can create rootkits that actively and accurately deceive forensic acquisition tools. As an example: the presented rootkit does not spoof information presented back to the forensic investigator during the acquisition process (i.e. FTK Imager displays the size of the shadow image being copied, not the actual disk size), but implementing the needed procedures is not impossible nor hard. As soon as mechanisms are in place to spoof the displayed information, detection becomes next to impossible. Apart from that, the read and writes speed decrease heavily, dropping from 150 MB/s to approximately 10-15 MB/s, which is something a forensic investigator could notice. However, this could just as well be prescribed to a faulty hard-disk, or other programs influencing the overall read and write speeds.

Recent developments in malware and backdoors showed the increased interest of hiding malicious code in the firmware of devices. For example, malware hiding in the firmware of solid-state drives is slowly moving away from being a myth to real life threats. If the presented rootkit would be combined with malware running in firmware which is controlling a storage device, the regular acquisition process could just as well be deceived. The author believes that it is only a matter of time before more advanced anti-forensic techniques are created and possibly running from within firmware, making detection and acquisition incredibly difficult.

Judging by the available documentation on anti-forensic prevention and the corresponding anti-forensic techniques, the author believes that the digital forensic world is slowly falling behind. There is a vast list of tools and techniques that can be deployed to deceive investigators, but a proper way of dealing with them seems to be absent. Of course, digital forensic models try to incorporate defense mechanisms, which is the basis of solving this problem. However, the real issue is the fact that the process of live data acquisition as such is neither resilient nor reliable, mainly due to the insecure environment it is performed in. An investigator has to deal with the fact the evidentiary system is not trusted, but the resulting evidence is treated as legitimate. A technologically aware suspect could harden his system in such a way, that data can be retrieved by the forensic investigator, but could enforce completely different investigatory tracks.

Even with the weaknesses present in the live data acquisition process, it is still a necessity. With the rise in popularity of full disk encryption, offline analysis is becoming harder, calling for an online analysis. Therefore, the author believes that solution must be created to provide secured environments within a non-trusted system. Within normal forensics, it is common practice to use specifically marked tape to indicate a crime scene and prevent bystanders from possibly tainting evidence. The same should apply to performing live data acquisitions; an investigator should first create a secure environment, before an acquisition can be performed. This could be achieved by the previously mentioned kernel-mode acquisition, in which a forensic investigator first deploys a kernel module that is hard to intercept, and talks to that module through an encrypted channel. Of course breaking with the, in this case, Linux and kernel development conventions is not the most beneficial road to take. However, research should be aimed at finding different techniques of acquiring data, even if that means changing current practices for those unique situations.

A bigger risk to the digital forensic process would be an actual anti-forensic Linux kernel. As previously mentioned, the code included in this rootkit could just as well be ran from within a kernel. Doing so would call for a different solution to still acquire correct data. A file system

level non-sequential acquisition tool could be a perfect replacement to the *tar* utility, hopefully effectively negating detection possibilities. However, block-level copies are almost impossible to perform when the entire kernel could have been patched. The development of hardware based solutions might be to expensive and the usage of such a device comes with the serious risk of crashing the system. Although, when the presented forensic techniques would implemented on a firmware-level, for example the disk controller of a storage device, even hardware based acquisitions become shaky Apart from this, new tools and techniques come with the lengthy process of getting them accepted in court. All the aforementioned combined makes the author believe that, as soon as an anti-forensic kernel is in place, it might be more beneficial to negate the necessity of a live-acquisition. If it encompasses a FDE system, a pre-boot execution environment might be able to extract the needed decryption keys. On the other hand, if the acquisition is performed by law enforcement, they might be able to retrieve the keys and data through general *"rubber hose decryption"*; the suspect might be willing to release the keys, negating the need for a live-acquisition.

The fact that forensic investigators are still using old and proven acquisition techniques is not unexpected; in the end, the evidence has to be presentable to court and as such known techniques are more easy judged as delivering valid results. This means that delivering improvements to the digital forensic process is hard. Getting new tools to be accepted in court becomes near to impossible when not carrying the appropriate certifications. This makes the author believe that the anti-forensic issues will not be solved or even counter efficiently in the upcoming years, due to the fact that new and experimental methods are needed to counter it in an effective way. The academic world can, and already does, provide solutions in regard to anti-forensics techniques, but it is up to the law enforcement area to adopt them.

Even if it has not yet been observed in the wild, the presented techniques is deemed feasible and a realistic threat to the current live data acquisition process. The development of the presented anti-forensics kit is neither impossible nor hard due to the vast list of rootkits out there. Any technologically aware suspect could download a working rootkit framework and patch it in such a way that it can hide, counterfeit or obfuscate data, as most of the needed information is readily available on the Internet.

8 Conclusion

This project investigated the current state of live data acquisitions when dealing with a full disk encrypted Linux machine.

To show the current problems with live data acquisitions, a rootkit for the Linux operating system was developed. This rootkit hides itself within ring zero of the operating system (i.e. the kernel), and takes control over the data stream pertaining to one of the aforementioned acquisition tools. It offers two forms of detecting the presence of such an application. By running a command based detection mechanism, *tar* and the imaging applications can easily be tricked in to returning different data. The alternative comes in the form of a linear detection mechanism, which at point of writing only detects *dd* and *dcfldd*. By monitoring the reads and writes executed by an application, the rootkit is capable of detecting imagers without creating system inhibiting false positives.

When a forensic acquisition tool has been detected, several scenarios can be exploited. The anti-forensic technique presented in this paper returns different data than requested back to the acquisition tools. Imagers return data from a prefabricated disk image, instead of the */dev/sda* raw block device. The *tar* utility, when executed with a directory present in a blacklist (held by the rootkit), retrieves data from a different home directory, thus framing a different user on the computer.

Hardening against these techniques is not impossible, if forensic investigators are willing to part with the common used tools. By encrypting the communication between user-space (the acquisition tool) and kernel-space, command based detection becomes near to impossible. A different solution can be found in moving the entire acquisition process to the kernel. By doing so, a secured environment is created in which a non-volatile data acquisition can be safely executed.

However, the aforementioned solutions are not applicable when the anti-forensic techniques have been coded deep in to the Linux kernel. Such a kernel calls for different acquisition solutions. The *tar* utility could be replaced with a non-sequential file system acquisition tool, which would be difficult to detect with the presented algorithm. Block-level copies become seriously risky when dealing with a patched kernel, calling for dedicated hardware solutions, not unlike those developed for RAM acquisition, to acquire non-volatile storage without using the evidentiary operating system. Such a device could be beneficial in most scenarios, unless the anti-forensic techniques are implemented on a firmware-level, which would effectively negate the possibilities of a live-acquisition.

To conclude, the presented techniques are deemed a serious risk to the digital forensics process. Although not yet observed in real life situation, the usage of rootkits for anti-forensic purposes is not new and is getting more advanced. More research must be performed aimed at improving and securing the live data acquisition process. If not, the arms race might eventually be won by the defending party, instead of the forensic investigator. The main conclusion is that the presented techniques should be deemed a serious risk to the digital forensic process and that dedicated solutions must be build and adopted by law enforcement to prevent losing ground to technologically skilled suspects.

9 Future Work

9.1 Stability and Improvements

First, time should be spend on creating a stable variant of the solution presented by this paper. Normal system behaviour is sometimes inhibited or could even fail due to a kernel panic. For research purposes, the kit should be stable so it can be used to test and improve forensic acquisition tools. Next to this, some parameter optimisation could be beneficial, specifically the thresholds used to label an application as being malicious to the evidentiary system. Apart from obvious stability improvements, a linear detection algorithm for *tar* could be implemented.

If the code is running stable, tests can be performed with an actual anti-forensic kernel. Implementing code inside the kernel calls for different prevention and detection techniques. Research must be performed on how to counter against such a kernel.

9.2 Prevention

Presented in Section 6 are several solutions to improve the live acquisition process of non-volatile data. Specifically, time must be spend on investigating the possible positive and negative effects of performing a kernel-mode acquisition. Doing so is against Linux kernel development standards, as file interaction should take place in user-space. However, when there is a serious indication that an evidentiary system is deploying advanced anti-forensic techniques as presented by this project, such a technique could be justified. Apart from that, although more generic, an encryption and decryption kernel module would be beneficial to facilitate secure communication between user-space and kernel-space. Such a tool would be helpful to forensic investigators, as well as secured environments trying to harden against rootkits.

9.3 Firmware-level Anti-Forensics

The presented techniques offer effective defence against live-acquisitions. On the other hand, ss soon as the storage device has been removed, and an offline analysis is performed, the techniques in place are no longer effective. However, implementing the code on a firmware-level would offer anti-forensic techniques effective during an offline analysis. Time could be spend on, for example, creating anti-forensic techniques hiding in the disk controller of a solid-state drive.

References

Adams, R., Hobbs, V., & G. Mann), y. . . v. . . n. . ., booktitle = Journal of Digital Forensics, Security and Law. (n.d.). The advanced data acquisition model (adam): A process model for digital forensic practice..

Bejtlich, R. (2009). *Sans ir and forensics sumit 2009 keynote.* (`https://digital-forensics.sans.org/summit-archives/2009/1-richard-bejtlich-keynote-incident-response-forensics.pdf` Accessed 25/06/15)

Bilby, D. (2006). *Low down and dirty: Anti-forensic rootkits.* (Ruxcon)

Blunden, B. (2009). *Anti-forensics: The rootkit connection.* (`http://www.blackhat.com/presentations/bh-usa-09/BLUNDEN/BHUSA09-Blunden-AntiForensics-SLIDES.pdf` Accessed 26/06/15)

Blunden, B. (2013). *Rootkit arsenal: Escape and evasion in the dark corners of the system.*

CERT-EU. (2012). *Incident response - data acquisition guidelines for investigation purposes.* (`http://cert.europa.eu/static/WhitePapers/CERT-EU-SWP_12_004_v1_3.pdf` Accessed 26/06/15)

Garfinkel, S. (2007). Anti-forensics: Techniques, detection and countermeasures. In *2nd international conference on i-warfare and security.*

Jones, R. (2007). Safer live forensic acquisition.

Ponemon. (2014). 2014 global report on the cost of cyber crime.

Rekhis, S., & Boudriga, N. (2010). Formal digital investigation of anti-forensic attacks. In *Ieee fifth international workshop on systematic approaches to digital forensic engineering.*

Rekhis, S., & Boudriga, N. (2012). A system for formal digital forensic investigation aware of anti-forensic attacks. In *Ieee transactions on information forensic and security* (Vol. 7).

Ross ulbricht day 5 court transcript. (2014). (`http://www.scribd.com/doc/253356971/Ubricht-Trial-Day-5`, Accessed 08/06/2015)

Ross ulbricht day 6 court transcript. (2014). (`http://www.scribd.com/doc/253475008/Silk-Road-Trial-Day-6-Transcript`, Accessed 08/06/2015)

Shannon, C. (1949). Communication theory of secrecy systems. In *Bell system technical journal* (Vol. 28).

Stuttgen, J., & Cohen, M. (2013). Anti-forensic resilient memory acquisition. In *Digital investigation.*

A DD Strace

```
execve("/bin/dd", ["dd", "if=/dev/sda", "of=dd.img"], [/* 16 vars */])
....
open("/dev/sda", O_RDONLY)
dup2(3, 0)
close(3)
....
open("dd.img", O_WRONLY|O_CREAT|O_TRUNC, 0666)
dup2(3, 1)
close(3)
....
read(0, ..., 512) = 512
write(1, ..., 512) = 512
read(0, ..., 512) = 512
write(1, ..., 512) = 512
....
close(1)
```

B DCFLDD Strace

```
execve("/usr/bin/dcfldd", ["dcfldd", "if=/dev/sda", "of=new.img"], [/* 16 vars */]) = 0
....
open("new.img", O_WRONLY|O_CREAT|O_TRUNC, 0666)
close(0)
open("/dev/sda", O_RDONLY)
....
read(0, ..., 32768) = 32768
write(3, ..., 32768) = 32768
read(0, ..., 32768) = 32768
write(3, ..., 32768) = 32768
....
close(0)
close(3)
```

C FTK Imager Strace

```
execve("./ftkimager", ["./ftkimager", "/dev/sda", "dd.e01"], [/* 16 vars */])
....
open("/dev/sda", O_RDONLY)
lseek(3, 0, SEEK_CUR)
lseek(3, 0, SEEK_END)
lseek(3, 0, SEEK_SET)
lseek(3, 0, SEEK_SET)
read(3, "\353c\220\20\216\320\274\0", 8)
close(3)
stat("/dev/sda", {st_mode=S_IFBLK|0660, st_rdev=makedev(8, 0), ...})
stat("/dev/sda", {st_mode=S_IFBLK|0660, st_rdev=makedev(8, 0), ...})
open("/dev/sda", O_RDONLY)
ioctl(3, 0x301, 0x7fff85082440)
close(3)
open("/dev/sda", O_RDONLY)
ioctl(3, BLKSSZGET, 0x7fff8508230c)
ioctl(3, BLKGETSIZE64, 0x7fff850822f0)
ioctl(3, 0x301, 0x7fff85082300)
open("/dev/sda", O_RDONLY|O_NONBLOCK)
ioctl(4, 0x30d, 0x7fff85082020)
close(4)
....
open("dd.e01.001", O_WRONLY|O_CREAT|O_TRUNC, 0640)
write(4, ..., 262144)
write(2, "0.00 / 20480.00 MB", 180.00 / 20480.00 MB)
write(2, "        \r", 9)
....
futex(0x2227cd8, FUTEX_WAIT_PRIVATE, 2, NULL <unfinished ...>
<... futex resumed> )
futex(0x2227cd8, FUTEX_WAKE_PRIVATE, 1 <unfinished ...>
<... futex resumed> )
futex(0x2227cd8, FUTEX_WAKE_PRIVATE, 1)
futex(0x2227c58, FUTEX_WAIT_PRIVATE, 2, NULL <unfinished ...>
<... futex resumed> )
futex(0x2227c58, FUTEX_WAKE_PRIVATE, 1 <unfinished ...>
<... futex resumed> )
futex(0x2227c58, FUTEX_WAKE_PRIVATE, 1)
lseek(3, 1310720, SEEK_SET)
read(3, ..., 262144)
futex(0x2227cac, FUTEX_CMP_REQUEUE_PRIVATE, 1, 2147483647, 0x2227c80, 10) = 1
futex(0x2227d04, FUTEX_WAIT_PRIVATE, 5, NULL <unfinished ...>
<... futex resu[pid  4428] <... futex resumed> )
med>
futex(0x2227c80, FUTEX_WAKE_PRIVATE, 1)
futex(0x2227cac, FUTEX_WAIT_PRIVATE, 11, NULL <unfinished ...>
<... futex resumed> )
write(4, ..., 262144)
....
close(4)
```

D Tar Strace

```
execve("/bin/tar", ["tar", "cvf", "main.tar", "/home/yonne"], [/* 35 vars */])
....
creat("main.tar", 0666)
fstat(3, {st_mode=S_IFREG|0664, st_size=0, ...})
newfstatat(AT_FDCWD, "/home/yonne", {st_mode=S_IFDIR|0755, st_size=4096, ...}, AT_SYMLINK_NOFOLLOW
    ↪ )
openat(AT_FDCWD, "/home/yonne", O_RDONLY|O_NOCTTY|O_NONBLOCK|O_NOFOLLOW|O_CLOEXEC)
fstat(4, {st_mode=S_IFDIR|0755, st_size=4096, ...})
fstat(4, {st_mode=S_IFDIR|0755, st_size=4096, ...})
fcntl(4, F_GETFL) = 0x28800 (flags O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_NOFOLLOW)
fcntl(4, F_SETFD, FD_CLOEXEC)
getdents(4, /* 51 entries */, 32768)
getdents(4, /* 0 entries */, 32768)
....
newfstatat(4, "Desktop", {st_mode=S_IFDIR|0755, st_size=4096, ...}, AT_SYMLINK_NOFOLLOW)
openat(4, "Desktop", O_RDONLY|O_NOCTTY|O_NONBLOCK|O_NOFOLLOW|O_CLOEXEC)
fstat(5, {st_mode=S_IFDIR|0755, st_size=4096, ...})
fstat(5, {st_mode=S_IFDIR|0755, st_size=4096, ...})
fcntl(5, F_GETFL) = 0x28800 (flags O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_NOFOLLOW)
fcntl(5, F_SETFD, FD_CLOEXEC)
getdents(5, /* 6 entries */, 32768)
getdents(5, /* 0 entries */, 32768)
....
newfstatat(5, "test", {st_mode=S_IFREG|0755, st_size=264, ...}, AT_SYMLINK_NOFOLLOW)
openat(5, "test", O_RDONLY|O_NOCTTY|O_NONBLOCK|O_NOFOLLOW|O_CLOEXEC)
fstat(6, {st_mode=S_IFREG|0755, st_size=264, ...}) = 0
write(1, "/home/yonne/Desktop/test\n", 26/home/yonne/Desktop/test)
read(6, "cd /mnt/hgfs/code/c/suterusu_roo"..., 264)
fstat(6, {st_mode=S_IFREG|0755, st_size=264, ...})
close(6)
....
```

E Linear Interception

```
[aftoolkit] linear imager detection (creates slurred images)
[aftoolkit] detecting linear read and writes: proc=dd
[aftoolkit] printing read cache
[aftoolkit] read: counter=100, fd=0, count=512, comm=dd
[aftoolkit] read: counter=99, fd=0, count=512, comm=dd
[aftoolkit] read: counter=98, fd=0, count=512, comm=dd
[aftoolkit] read: counter=97, fd=0, count=512, comm=dd
[aftoolkit] read: counter=96, fd=0, count=512, comm=dd
[aftoolkit] printing open cache
[aftoolkit] open: fd=1, filename=new.img, flags=577, mode=438, comm=dd
[aftoolkit] open: fd=0, filename=/dev/sda, flags=0, mode=0, comm=dd
[aftoolkit] printing write cache
[aftoolkit] write: counter=0, fd=1, count=512, comm=dd
[aftoolkit] write: counter=0, fd=1, count=512, comm=dd
[aftoolkit] write: counter=0, fd=1, count=512, comm=dd
[aftoolkit] write: counter=0, fd=1, count=512, comm=dd
[aftoolkit] write: counter=0, fd=1, count=512, comm=dd
[aftoolkit] dropping to user-space
[aftoolkit] read count till now: 52224
[aftoolkit] clearing contents of the output file
[aftoolkit] fake image: size=268435456
[aftoolkit] read bytes from fake image: fd=0, offset=0, count=52224
[aftoolkit] intercept read: count=52224, buffer=\xffffffebH\xffffff90\x10\xffffff8e\xffffffd0\
  ↪ xffffffbc
[aftoolkit] wrote missing bytes: ret=52224
[aftoolkit] moving back to kernel-space
[aftoolkit] silently dropping write: comm=dd, fd=1, buf=0
```