Federal Office
for Information Security

Security Analysis of TrueCrypt

November 16, 2015

Authors

This study was executed by the Fraunhofer Institute for Secure Information Technology on behalf of the Federal Office for Information Security. This publication was authored by

Mauro Baluda
Andreas Fuchs
Philipp Holzinger
Lisa Nguyen
Lotfi ben Othmane
Andreas Poller
Jürgen Repp
Johannes Späth
Jan Steffan
Stefan Triller
Eric Bodden

Fraunhofer Institute for Secure Information Technology
Rheinstraße 75
64295 Darmstadt
Germany

The authors wish to thank Andreas Junestam and Nicolas Guigo of the Open Crypto Audit Project for clarifications regarding their own security assessment of TrueCrypt. Thanks go also to Andreas Follner, Technische Universtät Darmstadt, who gave valueable input regarding the exploitability of buffer-overflow vulnerabilities that the OCAP-report mentions.

Federal Office for Information Security
Post Box 20 03 63
D-53133 Bonn
Phone: +49 22899 9582-0
E-Mail: bsi@bsi.bund.de
Internet: https://www.bsi.bund.de

Contents

List of Tables

1 Summary

1.1 Aim

TrueCrypt is an encryption program that was until recently freely available. The development and distribution of TrueCrypt was terminated with version 7.1a without prior notice at the end of May 2014. The reason for this decision is not well known. On the official TrueCrypt website [10] "WARNING: Using TrueCrypt is not secure as it may contain unfixed security issues" is written in red at the top of the page. As parts of TrueCrypt are also contained in the approved product Trusted Disk, vulnerabilities in TrueCrypt could also have an impact on Trusted Disk.

It was for this reason that the German Federal Office for Information Security (BSI) commissioned the Fraunhofer Institute for Secure Information Technology (SIT) to carry out a security analysis of TrueCrypt. This report summarizes the results of that security analysis. As well as uncovering possible vulnerabilities, the aim was also to point out possible areas for improvement during any future developments of the program.

1.2 Procedure

Security issues can have a diverse range of causes such as incorrect design decisions, programming errors and also misleading documentation. Therefore, the project was subdivided into a number of different work packages in order to analyze TrueCrypt from a range of different viewpoints. The results of the individual work packages are summarized in this report in their own chapters.

Analysis of the differences between versions 7.0a and 7.1a The product Trusted Disk that is approved by the BSI is based on the previous version of TrueCrypt – version 7.0a. In this work package, the changes between the current version 7.1a and version 7.0a were recorded and evaluated in terms of their relevance to the security of the program. This information will help to gauge whether vulnerabilities in the current version could also have a potential impact on Trusted Disk.

Evaluation of the OCAP Phase 1 Test Report Before the TrueCrypt project was terminated, a security analysis financed through crowd funding had already been started. The results of the first phase of this analysis were published in a report [14]. The subject of this work package was to evaluate these results and the procedures used for the analysis.

Review of the encryption mechanisms Cryptographic processes for deriving keys and for encrypting data form the key functionality of TrueCrypt. Weaknesses in these functions would have a particularly high potential for endangering the security goals of TrueCrypt. Therefore, the aim of this work package was to investigate whether the cryptographic functions in TrueCrypt were correctly implemented.

Evaluation using automated code analysis An analysis of the complete source code for TrueCrypt for vulnerabilities was completed using different up-to-date tools. All of the vulnerabilities identified by these tools were then investigated and evaluated – either manually or using a tool-based approach.

Evaluation of the code quality and documentation Many security holes are based on errors or incorrect assumptions made by developers or users. Easy to understand, maintainable

source code, as well as complete, well-structured and target-group oriented documentation, are thus very important for avoiding security issues. The code quality and documentation also provides clues as to the importance given to non-functional aspects such as security during the development of the program.

Conceptual evaluation of the architecture In this work package, possible attack vectors that could endanger the security goals of TrueCrypt were determined and tested to evaluate whether suitable decisions relating to the design and architecture of the program had been taken to protect against these threats.

Identification of dispensable parts of the code One strategy to reduce the probability of vulnerabilities is to avoid unnecessary attack surfaces and generally to reduce the complexity of the system. Based on the analysis of the architecture, it was thus tested whether parts of the TrueCrypt code could be removed without losing any important functions.

Evaluation of the OCAP Phase 2 Test Report Before the TrueCrypt project was terminated, a security analysis financed through crowd funding had already been started. The results of the second phase of this analysis were published in a report in March 2015 [2]. The subject of this work package was to evaluate the findings and to derive recommendations for action.

1.3 Results

The subject of the analysis was the last full version of TrueCrypt – version 7.1a. As only small changes were identified between versions 7.0a and 7.1a that were not considered relevant to security, it is possible, however, to also transfer the results to the older version that has been used as the basis for, among other things, the product “Trusted Disk”.

Overall, the analysis did not identify any evidence that the guaranteed encryption characteristics are not fulfilled in the implementation of TrueCrypt. In particular, a comparison of the cryptographic functions with reference implementations or test vectors did not identify any deviations.

Reviews of the source code and the cryptographic functions that were financed by crowd funding were carried out in part simultaneously with this study (Open Crypto Audit Project, OCAP). The 15 vulnerabilities published by OCAP were manually verified. One of the vulnerabilities represents a high practical threat.

The application of cryptography in TrueCrypt is not optimal. The AES implementation is not timing-resistant, key files are not used in a cryptographically secure way and the integrity of volume headers is not properly protected. There are many redundant implementations (sometimes for hardware-optimization) and disused algorithms are still present in a deactivated form in the source code. In particular, this project identified a need to improve the implementation of the random number generator for the Linux version and OCAP has uncovered a potentially dangerous error in the implementation of the random number generator for Windows.

The source code for TrueCrypt was tested for possible errors and weaknesses using three different static code analysis tools. As a result of a careful manual investigation of the tool-based results, it was possible to identify all of the suspected vulnerabilities that were flagged as being potentially critical to the security of the code – such as overflows – as false positives.

Quality defects were identified primarily in the maintainability and documentation of the source code. The static code analysis, various automatically calculated evaluation metrics and the manual review of the source code all revealed numerous indications of deficiencies and deviations from generally accepted practice. Furthermore, there is a lack of suitable documentation for developers. The source code only contains sporadic comments and there is no description of the system architecture. This is not immediately relevant to the issue of security. However,

the above-average effort required for maintenance due to the quality defects and the lack of documentation make any possible continuation of the project by third parties difficult.

The user handbook for TrueCrypt is comprehensive but poorly structured. Much of the detailed information is difficult to find and can only be understood with previous technical knowledge. This creates a problem because the security characteristics of TrueCrypt are partially dependent on user behavior (for instance avoiding sleep modes or the use of hidden volumes).

From a security perspective, the fact that TrueCrypt is a purely software solution means that it cannot in principle protect against all relevant threats. Effective protection only exists when an encrypted disk is lost or stolen in a deactivated state. TrueCrypt does not provide any protection against active attack scenarios such as the installation of a key logger or malware. To protect against these would require hardware-based security measures such as those provided by a TPM or smartcard.

2 Analysis of the differences between versions 7.0a and 7.1a

This chapter describes the procedure used to compare the two versions of TrueCrypt based on their source code. It describes the procedure used and the differences found, as well as highlighting those differences that were classified as being *relevant to security*.

2.1 Procedure

The source code for both versions was used as the basis for finding the differences between the two TrueCrypt versions. Version 7.0a was downloaded from the website https://github.com/DrWhax/truecrypt-archive/blob/master/TrueCrypt%207.0a%20Source.zip and Version 7.1a from https://github.com/AuditProject/truecrypt-verified-mirror?files=1. Version 7.1a was not initially taken from the same archive as 7.0a because the Open Crypto Audit Project had used the version from the above-mentioned link. However, a later comparison of both 7.1a versions from these different sources demonstrated that they were identical.

Versions 7.0a and 7.1a were compared with the help of version 4.1.3 of the program *KDE Kompare*. Kompare processes either individual text files or whole directory trees. The latter were used for the TrueCrypt comparison. Kompare compares every individual file in the first directory tree against those in the second and displays the differences. If a new file was added in the second directory tree, it is compared to an empty file and thus the complete file is marked as being different. The program only displays the parts of the directory tree that contain different files. For the comparison of the different versions of TrueCrypt, the two directory trees that corresponded to one of the TrueCrypt versions were selected in Kompare. We then worked through the differences that were highlighted in the directory tree in alphabetical order and manually examined the changed lines of source code.

Firstly, we attempted to contextualize the different lines of code so that we could evaluate whether the changes would have any potential impact on the security of TrueCrypt. In order to identify the context for a particular change, we used the name of the method in which the changes were found and the source code within it. If it was already possible to ascertain at this stage that the change was highly unlikely to be relevant to the security of the program because e.g. it only dealt with the relocation of a button in the graphical user interface, the change was marked as such and was not examined further. However, if indications were found such as e. g. key words like »password«, we then examined where in the source code this method was called and what impact the change could have.

2.2 Differences relevant to security

There were only two changes between versions 7.0a and 7.1a that were relevant to the security of TrueCrypt:

1. Outdated encryption algorithms are no longer offered as an option on the command line
2. The function for including the contents of a directory in the user password now ignores hidden files

The first of the above-mentioned Changes can be found in the file `CommandLineInterface.cpp` in directory `Main` in line 259. The encryption algorithms AES/Blowfish, AES/Blowfish/Serpent, Blowfish, Cast5 and TripleDES were already marked as being outdated in the source code for version 7.0a, while in version 7.1a they are no longer accepted on the command line should the user attempt to use them.

However, this change only has a small impact on the security of TrueCrypt because experienced users would already have ignored these outdated algorithms and these algorithms were already marked as being outdated in version 7.0a of TrueCrypt and the user is now made aware of this fact in version 7.1a.

The second change can be found e. g. in the file `Keyfiles.c` in directory `Common` from line 333 onwards. This affects the function `ApplyListToPassword`. It now ignores hidden files. The reason for this function is to enable the encryption key for containers to not only be derived from the password set by the user but also to include a list of files. This process increases the entropy of the key because users often tend to select only numbers and/or letters as passwords.

This change is considered just as uncritical to security as the first from a technical point of view. Firstly, the user is able to freely chose whether or not they activate this option at all. If the user decides to activate this option, a message is displayed telling the user that hidden files in the directory selected by the user will be ignored. Should the user ignore this message and, in the worst case scenario, select a directory that only contains hidden files, the user will receive an error message because no files will be found. This change thus makes sense from a usability standpoint because inexperienced users do not usually understand the difference between hidden and non-hidden files and may possibly never have even seen hidden files.

2.3 Other differences

In the following section, we have listed the differences between the two versions that in our opinion do not have any impact on the security of the software. They are mostly small changes to functions in the software such as e. g. changes to buttons in the graphical user interface or fixes to bugs that occurred with some PC BIOS. The differences are presented based on the directory tree for the software and grouped according to the directory in which they are found. Very small differences such as altered comments were ignored.

- Directory `Boot/Windows`
  - The data structure `BootArguments` has had another field `BootDriveSignature` added to it, which was designed to make partitions from which the system is booted easier to recognize
  - Some BIOS reported I/O errors too early. Now, certain actions are attempted more often before an error is reported
- Directory `Common`
  - Methods for parsing arguments of commands on the command line no longer accept an »`-`« or »`--`« before a parameter
  - Small changes to the GUI (sizes, buttons, word selection, etc.)
  - New methods for tooltips via the taskbar icon in TrueCrypt
  - The class`BootEncryption` has a new help method called `SystemdriveContainsNonStandardPartitions`, which tests whether there are partitions on a drive that are not very common, meaning ones that are not e.g. Fat16, Fat32, Extended, etc.

  - `Dlgcode.c` has a method called `MountVolume`, whose parameter `mountOptions` must now according to the comment always be »const« because otherwise problems could occur elsewhere in the source code

- Directory `Driver`
  - Some mutexes were removed
  - Source text for reading the »boot« flag for a partition added.
  - Workaround for sending »I/O Control Requests«, instead of directly aborting
  - In the method`TCCreateDeviceObject`, a »magic number« was removed that only served to recognize the mounted volume
  - Bug fix for unmounting volumes when they are nested
  - Compatibility problems with Windows tools such as diskmgmt, diskpart, vssadmin were resolved

- Directory `Format`
  - In certain circumstances, the operating system should not switch to sleep mode; this is now enforced (see file `TcFormat.c`)
  - Changes to the graphical user interface: Some warnings relating to Windows Vista SP1 removed, warnings added for users who want to save files larger than 4 GB in hidden OS mode to a non-hidden partition
  - Other wizards and dialogs in multiboot operation

- Directory `Main/Forms`
  - Button to show the donation form removed

- Directory `Mount`
  - Mounting of favorites changed slightly and help function added as »Baloon popup« via the TrueCrypt Tray Icon
  - In the case of a crash, the file `(winDir)\textbackslash MEMORY.DMP` is now also analyzed

- Directory `Platform`
  - The method`Erease` in the file `Memory.cpp` now uses the method `RtlSecureZeroMemory` on Windows to delete the memory, which was specially developed by Microsoft to be more secure than the standard method *memset*

2.4 Summary

We compared the source code for both TrueCrypt versions 7.0a and 7.1a with one another with the help of the open source tool *KDE Kompare* version 4.1.3. The tool compares two directory trees with one another at a file level. We then analyzed the changes found to see whether they could have an impact on the security of TrueCrypt. We only identified two changes that could be considered to be »relevant to security«. On the one hand, leaving out hidden files when increasing the entropy for deriving a password for encrypted containers and, on the other hand, ignoring outdated encryption algorithms on the command line. Other changes between the two versions tended to be bug fixes or changes to the graphical user interface. Due to the rather small change in the version number from 7.0a to 7.1a, we also didn't expect to discover any major changes. This mostly occurs in so-called *major releases* in which the initial digit in the

version number increases. Therefore, if there are security-relevant errors in the source code for TrueCrypt, these were already present in version 7.0a because there were no other changes in this context except for the two already mentioned above.

Results of chapter 2

- There were only two small security-relevant changes between versions 7.0a and 7.1a of TrueCrypt and these were not considered critical
  - Outdated encryption algorithms are no longer offered as an option on the command line
  - The function for including the contents of a directory in the user password now ignores hidden files
- The other changes were bug fixes or changes to the graphical user interface

3 Evaluation of the OCAP Phase 1 Test Report

3.1 Comments on OCAP

Junestam and Guido assessed the vulnerabilities of TrueCrypt and reported their finding in the Open Crypto Audit Project[1]. The report was published in February 2014 and is based on an analysis of TrueCrypt version 7.1a. In this chapter, we evaluate the scope of the report, discuss its findings and check whether the reported vulnerabilities can be identified using automated or manual analyses.

3.1.1 Scope of the report

The report assesses only parts of the TrueCrypt project: the *bootloader*, the *Windows kernel driver* and also the *setup process*. In the project, TrueCrypt was assessed for vulnerabilities with a focus on the areas of information disclosure, access rights and other similar security-relevant issues. The assessment was completed using a range of proprietary and publicly available tools, manual tests and direct analysis of the source code.

A total of 11 vulnerabilities were identified in the report, which can be categorized as follows: insecure cryptographic functions, buffer overflows, and memory overflows.[2] The authors of the report did not assign any of the findings a high level of severity. Finally, they provided recommendations for short-term solutions to the individual problems.

The two appendices to this report provide details on the different types of vulnerabilities and areas with poor code quality. We consider the description of the findings, as well as the examples and recommendations provided, to be informative and illuminating, although there is a lack of information that simplify reproducing the findings. There is no clear description of the methods used for the analysis and it is thus not possible to judge whether the results of the analysis are complete in relation to the tools used or whether only partial results have been reported. Another criticism is that the source code for the individual vulnerabilities was, in general, no provided. For example, we did not have enough information to precisely verify Vulnerability 2 in the report. Furthermore, we note–as stated above–that the OCAP report only refers to three of the six sub-projects: `Boot`, `Driver` and `Setup`. The projects `Mount`, `Format` and Crypto were not covered in the report.

3.1.2 Comments on the individual findings of the OCAP report

This section discusses the findings of the OCAP Report.

Finding 1 – Weak volume header key derivation algorithm. TrueCrypt uses the PBKDF2 algorithm to derive keys. In order to guarantee a sufficient level of security, a sufficient number of iterations should be completed when using the PBKDF2 algorithm. The iteration count is set in the method `get_pkcs5_iteration_count` (file `Pkcs5.c` of the subproject `Boot`) in the TrueCrypt project. Depending on the hash algorithm, TrueCrypt selects an iteration count

[1] https://opencryptoaudit.org/

[2] The OCAP classification of the findings was as follows: 1 cryptographic vulnerability, 4 data exposure vulnerabilities. 3 data validation vulnerabilities, 2 denial of service vulnerabilities and 1 error reporting vulnerability.

of 1000 or 2000; the recommended minimum iteration count is actually 1000 [22]. In contrast, NIST recommends an iteration count of 10 000 000 [22] for critical keys. The authors do not directly address the question of which iteration count should be selected for PBKDF2.

Finding 2 – Sensitive information might be paged out from kernel stacks. According to the authors, TrueCrypt could cause the operating system to experience a memory overflow resulting in stack data being paged out to the hard disk. This could mean that attackers could gain access to non-encrypted, confidential data in the memory. Tools for identifying these types of memory overflows in this scenario are rare. Therefore, we have come to the conclusion that this vulnerability is very difficult to exploit.

Finding 3 – Multiple issues in the bootloader decompresser. The authors found buffer overreads and buffer overflows in the file `Decompressor.c` in the `Boot` project. Unfortunately, the report does not provide any details about how this could impact the security of TrueCrypt.

As automated tools such as Coverity experience problems when scanning the Windows version of TrueCrypt (see 3.1.4), we decided to carry out a manual analysis of the implications of the stated weakness. We came to the conclusion that, even if they could be exploited by attackers, they would not endanger the security of TrueCrypt. Detailed information on the manual analysis can be found in section 3.1.3.

Finding 4 – Windows kernel driver uses memset() to clear sensitive data. TrueCrypt uses the function `memset` to overwrite locations in the memory. The authors explain that the compiler can optimize out these function calls. A search for the keyword "memset" in the TrueCrypt project demonstrates that the function is indeed used in multiple places. In general, this method is used to initialize variables before they are used. However, the use of this method is dangerous in some cases, as shown by the authors through two examples in the function `RMD160Final()` in file `RMD160.c`. We confirm that the calls are optimized out by the compiler. [5]. In addition, we also confirm that it is possible as a result that confidential information could be extracted from the memory.

Finding 5 – TC_IOCTL_GET_SYSTEM_DRIVE_DUMP_CONFIG kernel pointer disclosure. The report highlights a vulnerability via which an attacker could identify an address in the kernel address space using a malicious program in the user-space. As correctly recognized by the authors, this could be used to break the kernel space ASLR. Although this would be of relatively minor significance in practice.

Finding 6 – IOCTL_DISK_VERIFY integer overflow. In this finding, the authors reported an integer overflow in the method `ProcessVolumeDeviceControlIrp()` in the file `Ntdriver.c`. In addition, they explained that the error could theoretically occur. We did a manual data flow for the variables that can cause the vulnerability. The content of the variable comes from the method `ExInterlockedRemoveHeadList()`, which is part of Microsoft `Ntoskrnl.lib` library [3]. Thus, this vulnerability dependents on whether the method `ExInterlockedRemoveHeadList()` intercepts an integer overflow. Therefore, the vulnerability is not in TrueCrypt but rather in the Microsoft library. Nevertheless, it is possible under certain circumstances for information to be extracted from the memory. It is thus relevant to the security of the program and should be resolved.

[3] https://msdn.microsoft.com/en-us/library/windows/hardware/ff545427%28v=vs.85%29.aspx

Finding 7 – TC_IOCTL_OPEN_TEST multiple issues. According to the authors, the use of the method `zwCreateFile()` within the function `ProcessVolumeDeviceConrrolIrp()` in file `Ntdriver.c` enables an attacker to derive information, for example, about whether files exist or not (however, the attacker cannot open the files). The function `zwCreateFile()` is also called in the method `TCOpenFsVolue()` within `Ntdriver.c`. Within the function, the method `InitializeObjectAttributes()` with the parameter `OBJ_KERNEL_HANDLE` is called first. This sets the mode for the current caller to *kernel mode*[4]. The function `zwCreateFile()` is then subsequently called. The file is then created in kernel mode. This means that access permissions are no longer checked when accessing the file (even externally). Meta data such as the file path could thus be accessed externally, although viewing the content of the file is not possible externally.

Finding 8 – MainThreadProc() integer overflow. In this finding, the authors report on another vulnerability relating to an integer overflow that was localized to the function `MainThreadProc()` in the file `EncryptedIoQueue.c`. The vulnerability can trigger an integer overflow and disclose information.

Finding 9 – MountVolume() device check bypass. In the method `VolumeThreadProc()` in the file `Ntdriver.c` the authors report that access with `pThreadBlock->mount->wszVolume` is not validated before it is used. This can lead to unintentional behavior of TrueCrypt. The report does not go into more detail on the impact of a possible exploitation of this vulnerability.

Finding 10 – GetWipePassCount() / WipeBuffer() can cause BSOD. In the analysis carried out by the OCAP project, it was possible to produce a *Blue Screen of Death*. This is located within the functions `GetWipePassCount()` and `WipeCount()` in the file `Wipe.c`. The default handler for a switch statement causes the blue screen by triggering an action that requires administrator privileges. We can confirm that better error handling should be considered in this case.

Finding 11 – EncryptDataUnits() lacks error handling. This finding of the report is located in the function `EncryptDataUnits()` in the file `Crypto.c`. If a call within the function fails, the program will nevertheless still continue as normal. In particular, methods that call this function do not check whether the method has been successfully executed. Our analysis showed that the method is called, for example, from `SetupThreadProc()` in the file `DriverFilter.c`. Subsequently, the returned value of `EncryptDataUnits()` is directly written to the device – without even testing whether `EncryptDataUnits()` was successful or not. This can corrupt the files.

3.1.3 Detailed manual analysis of Finding 3

In the following paragraphs, we present our findings relating to Finding 3, Multiple issues in the bootloader decompresser, of the OCAP report.

In principal, the decompressor is only used for full encryption of the hard disk. The first steps in the booting process are then as follows: BIOS is run after switching on the PC. The code in the boot sector is then executed, which consists of the following steps: as the TrueCrypt bootloader is present in a compressed form, it needs to unpacked by the decompressor before it can be executed. The decompressor is firstly loaded into memory for this purpose. The checksum for the decompressor is then calculated to test its integrity. If this is confirmed, the compressed TrueCrypt bootloader is loaded into memory. Checksums are also used during this step to check

[4] https://www.osronline.com/article.cfm?id=257

for integrity. The bootloader is then unpacked using the decompressor and executed. It is only now that the user is asked for their password.

It should be noted that the calculation of the checksums does not guarantee that the relevant file has not been manipulated by an attacker. As it is easily possible for the attacker to change the checksum used for the comparison, it is only possible for this process to determine whether the file has been damaged as a result of hardware problems or similar issues – which was also the intention of the programmers.

We will now describe why neither the buffer overflows nor the buffer overreads in the decompressor represent a threat to the security of the program. In any case, it would be necessary for an attacker to manipulate or totally replace the compressed bootloader that is saved on the hard disk because this is the only input for the decompressor. This implies that the attacker must have physical access to the hard disk.

Buffer Overflows The threat represented by buffer overflows is generally that an attacker could, under certain circumstances, manipulate the program flow and thus run malicious code that had been previously introduced. Time consuming analyses would be required to identify whether a vulnerability could actually be exploited by an attacker, which are not necessary in this case for the following reason: as already mentioned, an attacker must have replaced the original, compressed bootloader with a manipulated version in order to exploit this vulnerability. This would exploit the vulnerability in the decompressor to divert the program flow to other code that the attacker would also have added to their manipulated bootloader. However, this is obviously not expedient because it is easier for the attacker to change the bootloader so that it runs this additional code as standard. Exploiting the buffer overflow is thus superfluous.

Buffer Overreads The threat represented by buffer overreads is generally that an attacker could, under certain circumstances, read a memory area. If keys or passwords are located in these areas as plaintext then they could fall into the hands of the attacker. The size of the memory area that the attacker is able to read differs from case to case and would require a time-consuming, manual analysis. However, this is not necessary in the case of the two buffer overreads found in the decompressor because as previously described the decompressor is run before the user enters their password. Therefore, there is nothing of interest in the memory at this point in time and any exploitation of this vulnerability is pointless.

3.1.4 Checking the findings using Coverity

In order to additionally check the findings of the OCAP report, we analyzed TrueCrypt for Windows using the automated code scanner software Coverity (see Chapter 5 for more details) and compared its results to those of the report. It is important to note here that TrueCrypt for Windows utilizes three different Microsoft compilers. In particular, the Microsoft Visual C++ 1.52c compiler from 1994 is used[5]. Coverity no longer supports this old compiler. The compiler is used in the project `Boot`, therefore this project cannot be analyzed using Coverity. The analysis of the results revealed that none of the vulnerabilities identified in the OCAP report were found by Coverity. These vulnerabilities are thus viewed by Coverity as false negatives.

[5] http://support.microsoft.com/kb/145669

3.2 Summary

Results of chapter 3

- The OCAP report describes a total of 11 vulnerabilities in TrueCrypt.
- The report has clear deficiencies when describing the underlying methods and techniques used for the analysis. It is thus not possible to a large extent to reproduce the findings precisely.
- The automated code analysis could not identify the vulnerabilities presented in the OCAP report.
- For some of the vulnerabilities highlighted in the report, we were able to prove by means of a manual analysis that they would be difficult to exploit by attackers in a real TrueCrypt environment.

4 Review of the encryption mechanisms

4.1 Encryption algorithms

Version 7.1a of TrueCrypt supports the encryption schemes AES, Serpent and Twofish for newly created volumes. The hash functions RIPEMD-160, SHA-512 and Whirlpool are used for these volumes. The encryption process is always carried out here in XTS mode. In the first part of this review, internal interfaces and data flows in the TrueCrypt implementation were investigated in order to identify areas that may enable a comparison of the implemented crypto functions with analogue functions in open source implementations. A call graph for the TrueCrypt software compiled on Linux was generated in order to plan what tests could be carried out. The evaluation and generation of the graphs was done with the help of the scripting language Ruby in combination with the tools cscope [6] and rtags [1]. In order to support the analysis, abstractions of the graphs (e. g. interfaces between classes) were also implemented. Checking the entry points in the call graph revealed that these functions were part of the user interface or test functions. The functions `aes_encrypt_key`, `aes_encrypt_key192` and `aes_encrypt_key128` were not used. In the version under investigation, only the function `aes_encrypt_key256` was used. Figure 4.1 illustrates the complexity of the calculated call graph, Figure 4.2 shows a part of the call graph that contains all of the call paths to the basic function `aes_encrypt` which is implemented in assembler.

In order to implement the tests for the review, the available algorithms and the corresponding interfaces of the standard open source libraries were investigated. Table 4.1 lists the available algorithms, as well as the availability of XTS mode for the algorithms to be tested which are implemented in TrueCrypt, OpenSSL and Libgrypt.

Algorithm	**TrueCrypt**	**OpenSSL**	**Libgcrypt**
AES	+	+	+
Twofish	+	-	+
Serpent	+	-	+
AES-XTS	+	+	-
Twofish-XTS	+	-	-
Serpent-XTS	+	-	-

Table 4.1: Available algorithms

The table shows that Libgrypt can be used to test the implementation of all symmetric encryption algorithms. The open source alternative tcplay[12] that is compatible with TrueCrypt was selected for this test. It contains a generic XTS implementation and can be compiled in combination with Libcgrypt. tcplay was modified to provide one library with the functions required for the test.

The green nodes in Figure 4.3 show a part of the call graph used for the AES encryption in XTS mode. This part does not contain any interfaces that could be used for a direct comparison with tcplay/Libcrypt. For this purpose, the function *EncryptionAlgorithm::EncryptSectors* from the generic part of the TrueCrypt implementation was selected (blue nodes). The tests implemented for the comparison were initially carried out with test vectors defined in accordance with the

Figure 4.1: TrueCrypt Call Graph

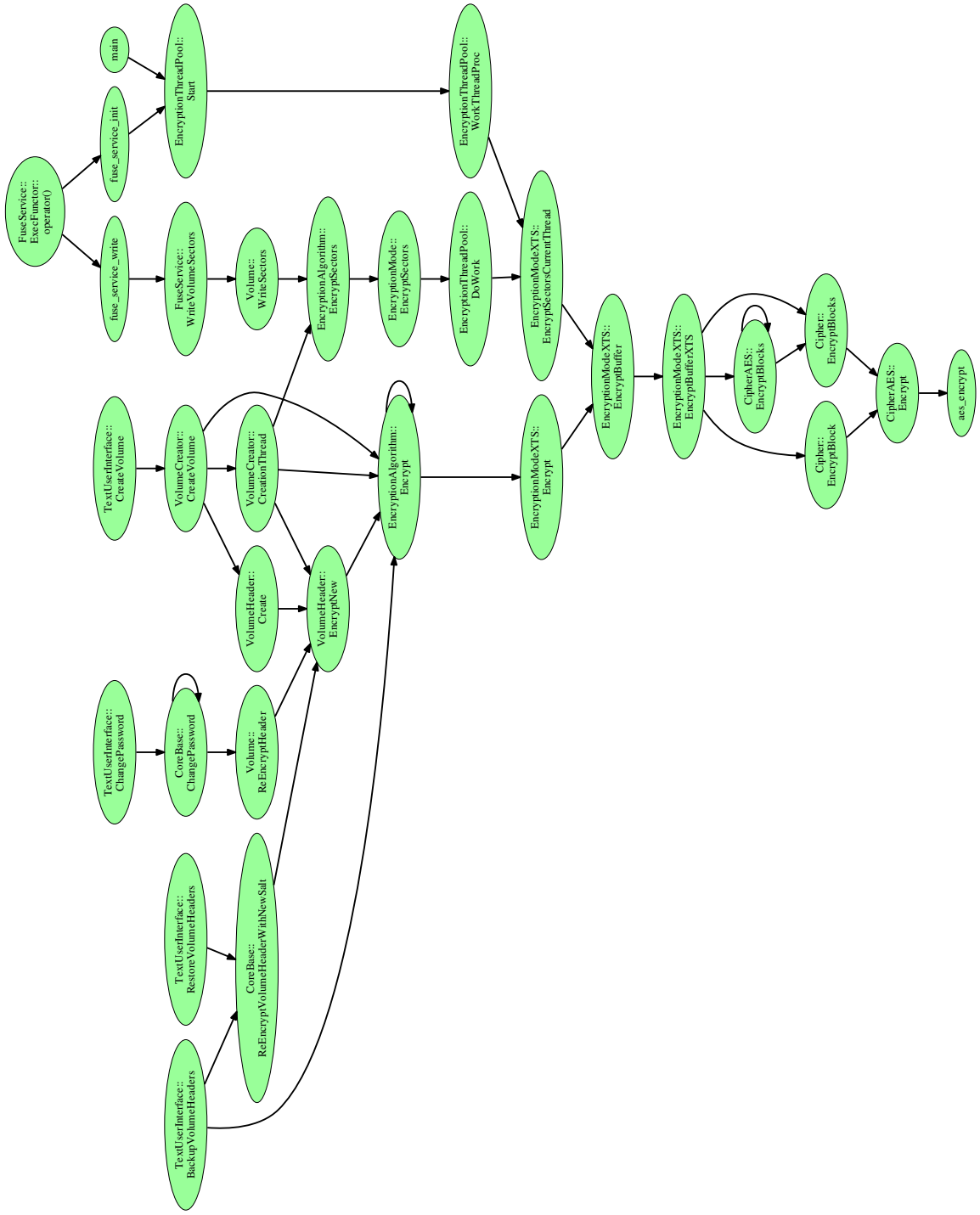


Figure 4.2: Call graph to function `aes_encrypt`

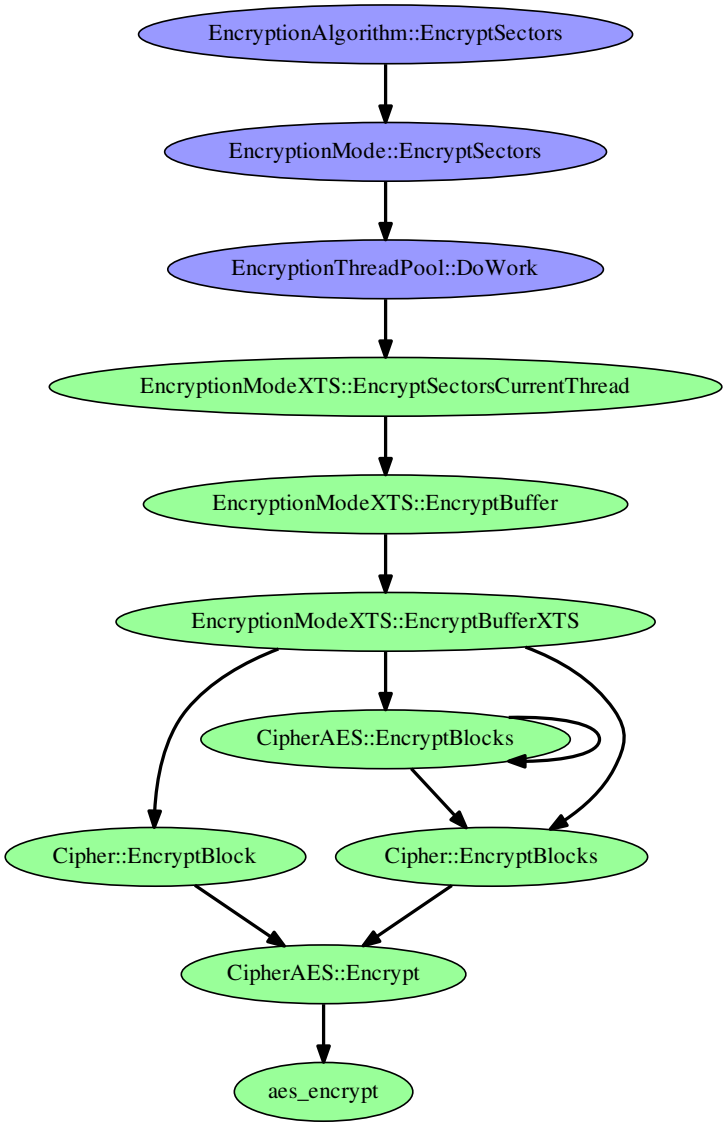


Figure 4.3: Call graph for the test of the AES encryption

IEEE P1619TM/D16 standard [13] in order to check the validity of the approach. This test was successfully completed with the modified tcplay library. The data for further tests were created by random:

- 256 bit AES key for the data encryption
- 256 bit AES "tweak key" for incorporating the position of the data in the encryption.
- 512 byte data blocks

As Serpent and Twofish also use the same key lengths as AES, it was possible to carry out the tests for all algorithms in parallel using the same test data. The generated test data were written to a CSV file (delimiter = ,) in accordance with the format defined inTable 4.2 . The data was represented as hex string format (e. g. 4 bytes: CAFFEE02).

Type	**Algorithm**	**Length**
Data key	Random	32 bytes
Tweak key	Random	32 bytes
Data	Random	512 bytes
Cipher	AES, Serpent or Twofish	512 bytes

Table 4.2: Format of the test data for symmetric encryption

10 million tests were carried out. There was no difference identified in the calculated cipher in any of the tests.

4.2 Key derivation

A 256 bit key was used for all of the tested encryption processes. Both of the keys required for XTS mode were generated using the PBKDF2 method, which is specified in PKCS5 v2.0. Figure 4.4 shows an extract from the TrueCrypt call graph with the functions for deriving the keys as the end points.

OpenSSL was utilized in accordance with table 4.3 for testing the key derivation functions used.

TrueCrypt	**OpenSSL**
pkcs5HmacRipemd160.DeriveKey	PKCS5_PBKDF2_HMAC digest: EVP_ripemd160
pkcs5HmacSha1.DeriveKey	PKCS5_PBKDF2_HMAC_SHA1
pkcs5HmacSha512.DeriveKey	PKCS5_PBKDF2_HMAC digest: EVP_sha512
pkcs5HmacWhirlpool.DeriveKey	PKCS5_PBKDF2_HMAC digest: EVP_whirlpool

Table 4.3: TrueCrypt / OpenSSL key derivation functions

The generated test data were written to a CSV file (delimiter = ,) in accordance with the format defined inTable 4.4 . Except for the length values, the data was was represented as hex string format (e. g. 4 bytes: CAFFEE02). The lengths were given as a normal number in text form. A CSV file was created for every algorithm.

The length values were selected in accordance with the existing TrueCrypt implementation. A key length of 240 represents the maximum possible key length multiplied by two because two keys are required for some modes. Both keys were each taken from the longest derived key without overlapping, which was not a problem because during the derivation the maximum value

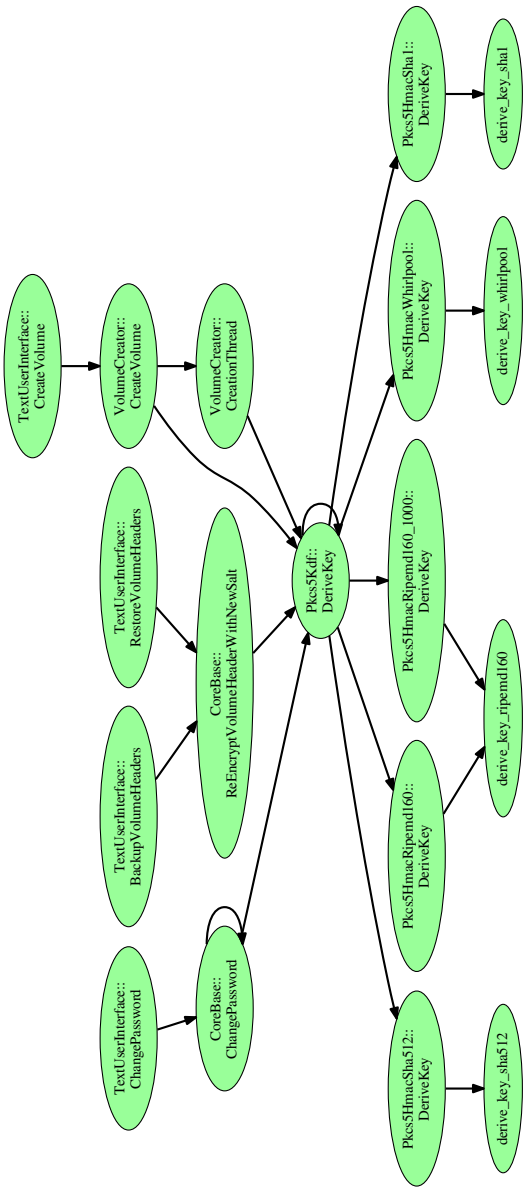


Figure 4.4: Call graph for the key derivation functions

Type	Algorithm	Length
Password length	Random	Variable
Password	Random	Password length (bytes)
Salt length	-	4
Salt	Random	4 bytes
Key length	Ripemd160	240 bytes
Key	Sha1, Sha512, Whirlpool or Ripemd 160	Key length in bytes

Table 4.4: Format of the test data for the key derivation

is always used. The computed call graph was used to investigate whether this maximum value was used in all cases. 10 million tests were also carried out here and no differences between the calculated keys were identified.

4.3 Random number generation

The improvements presented in this code review, related to random number generation in the file Core/RandomNumberGenerator.cpp, should be verified in a practical test. The function `RandomNumberGenerator::AddSystemDataToPool` adds greater entropy to the pool of random numbers for the random number generator. This can apparently take place in two modes: `fast` and non-`fast` – encapsulated by the calls `RandomNumberGenerator::GetDataFast` and `RandomNumberGenerator::GetData` in the file Core/RandomNumberGenerator.h. This process for generating random numbers is only used by the Linux or MacOS variants of TrueCrypt. On the Windows platform the generation of random numbers in the file Common/Random.c is carried out based on the Windows crypto functions. Vulnerabilities were also identified here within the OCAP-2 report, which will be discussed in Chapter 9.2. The `fast` variant is only found in the functions `CoreBase::ChangePassword`[1] and a `FatFormatter::Format` application. These applications are not very critical but could nevertheless be handled using a better schema (see below). All other calls are carried out with the non-`fast` variant.

However, the non-`fast` variant is implemented in such a way that it falls back to the `fast` variant in certain cases. In both cases, the entropy pool is initially filled along its entire length with data from `/dev/urandom`. Then the pool is enriched with as much data from `/dev/random` as is currently available. The problem is caused, if there is no data currently available in `/dev/random`. In this case the level of entropy is not greater than the `fast` variant. This is particularly true for the first initialization of the entropy pool. Furthermore, all of the entropy added to `/dev/random` by the kernel is also added to `/dev/urandom`. Therefore, the output from `/dev/urandom` has at least the same level of entropy as added to the pool in this schema by `/dev/random` and, most importantly, this is the same entropy.

The reason for using `/dev/random` at all lies in the fact that the Linux kernel only responds to requests when it has collected sufficient entropy. Yet this type of call typically takes a long time because it requires that every byte is filled with a corresponding minimum level of entropy. However, requests are already initialized using `/dev/urandom` before. In the use case for automated deployment, embedded systems and virtualisation (which generally display a lack of entropy), the implementation used does not help because there are no requirements for a minimum level of entropy imposed on `/dev/random`. In addition, once the minimum level of entropy had been assured in the Linux kernel, the data from `/dev/urandom` could be used directly and further calls `/dev/random` would be unnecessary[2].

[1] for the first n calls; followed by a non-`fast` call
[2] Note: Too much Entropy is not a bad thing

Instead, a correct implementation would ensure that RandomNumberGenerator will read a minimum level of entropy from `/dev/random` during initialization before delivering the first random numbers. No further calls to `/dev/random` would be necessary. From this point in time, either the data from `/dev/urandom` could be directly used or added to a pool, or this precursory-entropy could supply a deterministic random number generator with a seed and no further calls to `/dev/random` or `/dev/urandom` would be necessary. Since Linux 3.17, there has also been the SysCall `getrandom`[3]. This tackles the problem of ensuring the correct initialization of the Linux kernel without having to wait for results from `/dev/random`.

This statement based on a code review and should be verified in a practical test, for example, on the basis of a QEmu setup or corresponding debugging points in the source code.

4.4 Summary

Results of chapter 4

- In order to identify suitable areas that would allow an investigation of the cryptographic functions used in TrueCrypt, automated callgraphs were implemented.
- Tests to compare the currently used cryptographic functions for symmetric encryption and the derivation of keys based on standard open source libraries were carried out. The test data was generated randomly. Long-running tests (8*times*10 million test cases) revealed no differences to the TrueCrypt implementation.
- An investigation of the random number generator for Linux revealed that in low entropy scenarios there is no guarantee for a sufficient entropy level because the use of the high entropy `/dev/random` requires significant improvement.

[3] http://man7.org/linux/man-pages/man2/getrandom.2.html

5 Evaluation using automated code analysis

Automated static code analysis of software has established itself as an effective tool for software development and provides a comprehensive and objective assessment of a software product. Research over the last few years has enabled the high precision analysis of large-scale projects in terms of the quality of their code and their security. The code scanner initially derives a static model of the source code, which can then be examined for known patterns – so-called *checkers*. In this process, the software does not need to be explicitly executed; some analysis tools do not even require compilable code.

However, these tools still have great scope for improvement in the area of *false positives*. These are warnings issued by the scanner that do not correspond to actual deficiencies but often occur because the assumptions made by the scanner are too broad based. A large part of the scientific research in this area currently focuses on drastically reducing the proportion of false positives.

In the last few years, some automated tools – both those under commercial license and open source tools – have been launched on the market for a variety of programming languages. In this project, we used three of the most well-known and most advanced static code analysis tools for C and C++ : the Clang Static Analyzer, Coverity and Cppcheck. In the following sections, we describe the features of these tools and focus, in particular, on their strengths and weakness, as well as on the differences between them.

Clang

Clang Static Analyzer is an analysis tool for C, C++ and Objective-C that requires the software's source code. The tool is part of the Clang project, a front end for the LLVM compiler infrastructure that is available for all standard platforms. This analysis tool specializes in the high precision analysis of software errors in general – meaning it has a low false positive rate. In order to keep false positives to a minimum, the *symbolic execution* technique is used. This technique involves systematically scanning the program along all feasible paths and abstracting symbolic values from concrete values in the program. Clang integrates several specialized checkers and verifies these along the detected program paths. In our analysis, we activated all – stable and experimental – standard checkers and also the security-relevant checkers. The command we used for the analysis was thus:

```
scan-build -enable-checker core -enable-checker alpha.core \
           -enable-checker security -enable-checker alpha.security
```

Many analysis tools based on the symbolic execution of a program experience problems with *path explosion*. The number of all possible paths grows exponentially as the program get larger. In order to produce a workable analysis, it is generally necessary to limit the depth of the analysis to realistic programs. This can lead to incomplete results.

Coverity

Coverity is a commercial static code analysis tool and enables software developers to analyze their programs for software errors. This tool is interesting because all possible paths – particularly inter-procedural paths[1] – in the program are analyzed. Coverity assumes that the software being

[1] The effect of the way methods are called from within other methods is taken into account. This makes it possible to form a complete model of the Software.

analyzed can be compiled. The tool then taps into the build process and transforms the bytecode into a static model that can then be analyzed by Coverity for typical patterns – checkers . The patterns cover different categories of software errors. For example, Coverity reports deficiencies in the code quality and also security holes.

An analysis using Coverity involves three steps:

1. Configure the compiler for Coverity. (once per compiler)
2. Compile the project within the Coverity environment. This transforms the code into an intermediate representation that is used internally by Coverity.
3. Analyze the intermediate results – the desired checkers can then be activated for this purpose.

We used the static code analysis tool on Linux and Windows builds. We have described the actual command sequences used for scanning with Coverity below:

Details for Linux:

1. On Linux, TrueCrypt can be compiled using the compiler `gcc`[2].

   ```
   cov-configure gcc
   ```

2. Coverity expects a build command here. A directory in which the intermediate results will be saved must also be entered as an argument. The complete build command for TrueCrypt is given as a further argument. It is important to ensure here that a `make clean` command is called in advance so that all previously compiled files are discarded and recompiled.

   ```
   cov-build --dir ... make NOGUI=1 WXSTATIC=1
   ```

3. Analysis of the intermediate results with specific checkers.

   This is carried out with: `cov-analyze --dir ... --all --aggressiveness-level medium`

   The files in the specified directory in the `cov-build` command are analyzed here.

The parameter `aggressiveness-level` controls the intensity of the search. On the one hand, it can increase the length of the analysis, while on the other hand, it can also return more false positives.

For example, a total of 1382 results were found for the configuration `aggressiveness-level high`, while in comparison a total of 58 errors were found for the setting `medium`. In terms of the further manual inspection of the results, we limited the search to `aggressiveness-level medium`.

Details for Windows: A TrueCrypt build on Windows uses a total of three different compilers, which causes problems when carrying out the Coverity analysis. The TrueCrypt project comprises the projects `Boot`, `Driver`, `Crypto`, `Mount`, `Format` and `Setup`. The subproject `Boot` uses the Microsoft Visual C++ 1.52c compiler from 1994. Coverity does not support this compiler and cannot be configured for this compiler (step 1). Therefore, the standard TrueCrypt build process for a Visual Studio project cannot be used for the Coverity analysis. It was thus necessary to break down the build process into its individual subprojects and their corresponding build processes.

With the exception of `Crypto` and `Driver`, all other projects are directly dependent on `Boot`. As Coverity requires a functioning build command, it is important to ensure that `Boot` has already been compiled in advance. The following steps must be completed (see step 2):

[2] https://gcc.gnu.org/

1. Configure the Visual Studio compiler:

   ```
   cov-configure --msvc .../cl.exe
   ```

2. Compile the project `Boot` without Coverity:

   ```
   msbuild  /p:Configuration=Boot .../TrueCrypt.sln
   ```

   Analysis of all other projects with Coverity:

   ```
   cov-build --dir  ... msbuild /p:Configuration={project} ../TrueCrypt.sln
   ```

   The projects should be compiled here according to their dependencies, preferably in the following sequence `Driver`, `Mount`, `Format` and `Setup`. The project `Crypto` cannot be compiled on its own as the subprojects reference it directly.

3. Start the Coverity analysis:

   ```
   cov-analyze --dir ... --all
   ```

The Coverity analysis uses the build for `Boot` previously created in step 2. Therefore, Coverity does not analyze this project and thus no results can be expected within `Boot`. Using the default value `aggressiveness-level normal`, the analysis returned 158 findings, which we will focus on in the results section later.

Cppcheck

Cppcheck is a tool for finding software errors in C and C++. The tool directly analyzes the program's source code. The analysis searches for patterns to recognize memory leaks, uninitialized variables or null dereferences. One advantage of scanning at a source code level is that the semantics of the language can also be taken into account.

Cppcheck promises a 0 percent false positive rate. However, this also means that some potentially interesting findings are either discarded by the analysis or not reported. In the case of TrueCrypt 7.1a, most of the results from Cppcheck referred to bad programming practices, which could but do not necessarily lead to security holes.

As the individual results are highly dependent on the underlying technology, it is necessary when completing an automated security analysis to utilize a broad spectrum of tools. This was also confirmed by a comparison of the (Linux) findings with one another. We were able to link 8 of the 209 results from Cppcheck with the 58 findings from Coverity. (Coverity groups error messages of the same type in the same file as one result. Without this grouping, it would be possible to link a total of 17 warnings.) In contrast, we were unable to identify any overlap between the 71 results from Clang and those returned by the other tools.

5.1 Overview of the results of the analysis

In this section, we will provide an overview of the warnings that were reported by the different tools. In order to make the results comparable, we classified them according to the taxonomy *Seven (plus one) Pernicious Kingdoms*, which was developed by the Fortify Software Security research group. This classification system is defined as CWE-700 and aims to categorize software errors as concrete and specific problems rather than abstract and theoretical issues.

The software errors that were reported by the different analysis tools can be mapped to the eight "Kingdoms". The eight Kingdoms are (sorted in descending order of importance)

- Input Validation and Representation
- API Abuse

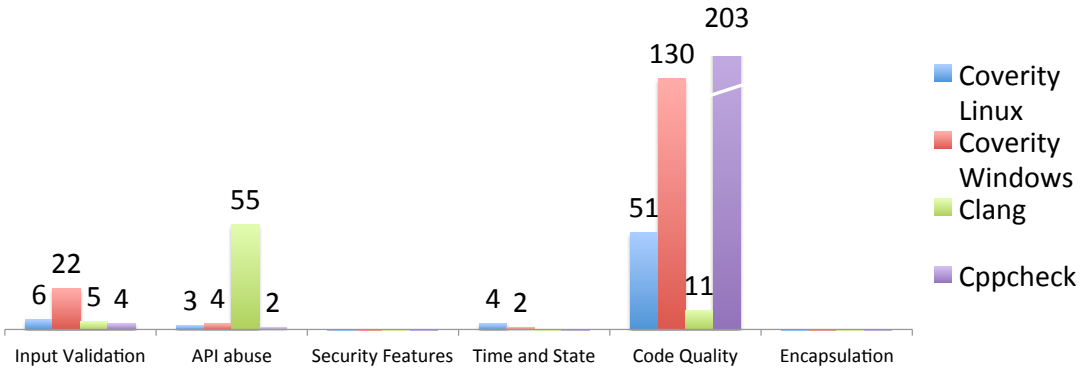


Figure 5.1: Classification according to the Seven Pernicious Kingdoms [21]. The software errors in the category "Input Validation" correspond to the security-relevant software errors

- Security Features
- Time and State
- Errors
- Code Quality
- Encapsulation
- Environment

In this report, we integrated the categories "Errors" and "Environment" into the category "Code Quality" because only a small number of software errors were reported in the first two categories and we were able to attribute these to bad programming practices.

The classification of the results from the Clang, Coverity and Coverity scanners according to CWE-700 can be found in Figure 5.1. Most of the errors found by Coverity and Cppcheck can be attributed to deficiencies in code quality. It is also clear here that Cppcheck specializes in identifying deficiencies in the code quality. In contrast, the Clang Static Analyzer primarily reported errors that we attribute to incorrect API usage.

5.2 Details on the error reports and recommendations

In this section, we report on the results of the three analysis tools in relation to TrueCrypt.

We focus on each tool individually and discuss the the security-relevant classes of warnings. The security-relevant software errors mainly fall into the first four Kingdoms (Figure 5.1).

By initially classifying the warnings, we were able to identify potentially security-relevant software errors. We then subsequently analyzed every one of these software errors in detail and came to the conclusion that they were false positive warnings. Therefore, we have merely covered the interesting, non-trivial examples in this section.

5.2.1 Clang Static Analyzer

The analysis of TrueCrypt using the Clang Static Analyzer delivered a total of 71 warnings. Following a manual analysis of the results, we are firmly convinced that all of the warnings are false positives. The majority of the results (55) deal with data type conversions for pointers, which could in principle lead to corrupt data but are actually unproblematic in this case. The

ID	File name	Method	Type	Stated risk level
CLANG-1	`/Crypto/Sha2.c`	sha_end1	Out-of-bounds access	High
CLANG-2	`/Crypto/Sha2.c`	sha_end2	Out-of-bounds access	High

Table 5.1: Risk level for the Clang results

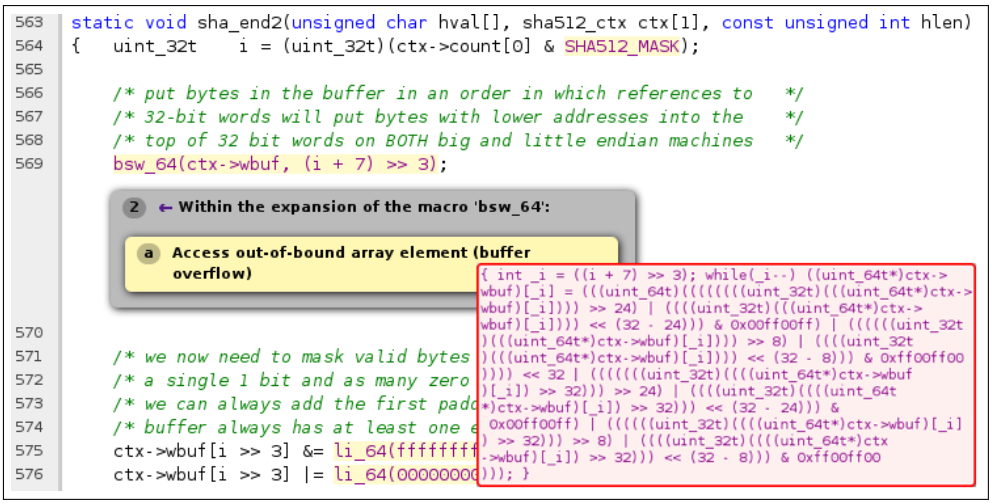


Figure 5.2: SHA512 out of bound array access.

high number of false positives can be explained by the fact that these types of operation are extremely difficult to recognize using an automated static code analysis because, for example, no precise semantic handling of integer arithmetic (in concrete terms: semantics at a bit level) is carried out, as is otherwise standard with symbolic execution.

The 11 results returned by Clang that related to code quality are not relevant to security. Therefore, we have not included them here and refer you to Chapter 6. Nevertheless, we would like to point out that it is possible to resolve most of the warnings with small improvements to the source code. This would drastically reduce the number of results returned by the three tools.

The remaining six findings refer to potential out-of-bounds accesses. In Table 5.1, two interesting warnings are listed that required further analysis before it was ultimately possible to categorize them as false positives. Both warnings are caused by two similar functions. They both carry out so-called digest calculations for the cryptographic hash functions SHA256 and SHA512.

Figure 5.2 shows a possible out-of-bounds access of the array `ctx->wbuf` within the function `sha_end2`. An out-of-bounds access could be an exploitable security hole. Therefore, we continued to investigate the circumstances under which the array access could occur.

The red box in Figure 5.2 shows the complex expression that calculates the index with which the array is accessed. In this example, indices are calculated in a particular way that involves complex binary operations such as shifts, OR and AND expressions. As already mentioned, it is precisely these expressions that are extremely difficult to analyze – either automatically or manually.

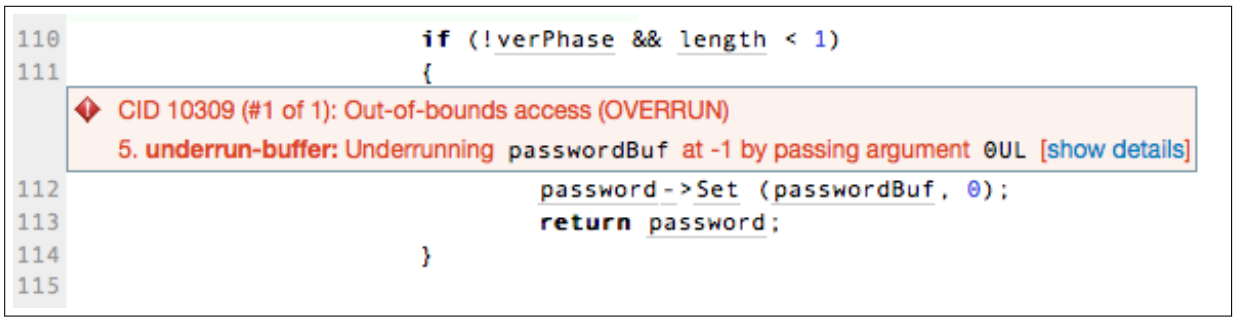
In this case, the fact that the observed array has a fixed size is helpful. Therefore, it can be expected that only a limited number of execution paths exist in the program that can access this array. It is thus sensible to use a symbolic execution tool that focuses on generating test cases.

KLEE is one such tool: It can analyze C and C++ and generate symbolic test cases. Thanks to its integrated STP theorem prover, it can precisely handle expressions that involve bit-vector arithmetic [4].

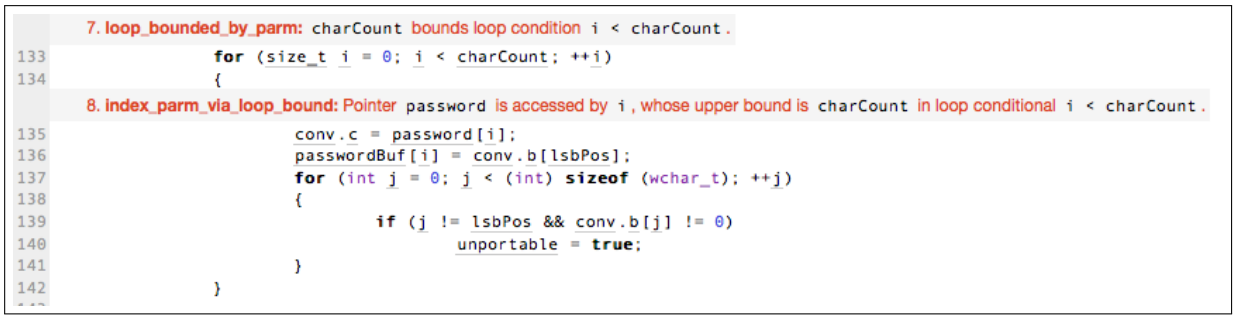
KLEE was able to complete a full symbolic analysis within less than a minute and identify

ID	File name	Method	Type	Stated risk level
COV-1	`/Main/TextUserInterface.cpp`	AskPassword	Out-of-bounds access	High
COV-2	`/Main/CommandLineInterface.cpp`	ToKeyfileList	Various	High
COV-3	`/Platform/Unix/File.cpp`	GetPartition-DeviceStartOffset	Overflowed return value	Medium

Table 5.2: Risk level for the Coverity results for Linux



(a) The method is called with parameters that lead to a buffer underun according to Coverity.



(b) Details for the method `Set`, which is called in the Figure above.

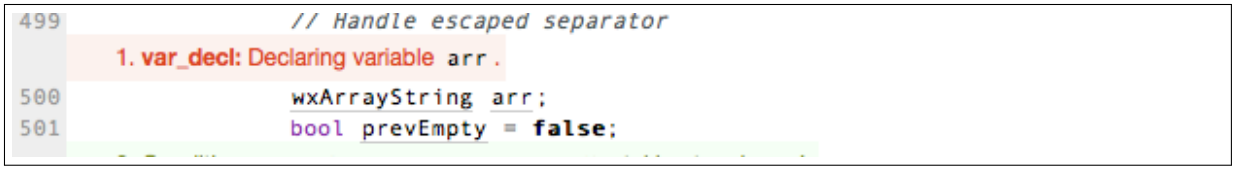
Figure 5.3: Details about the result COV-1.

17 different execution paths in the analyzed subprogram . The index for the array access was generated for each individual program path. None of these indices led to an out-of-bounds access to the array.

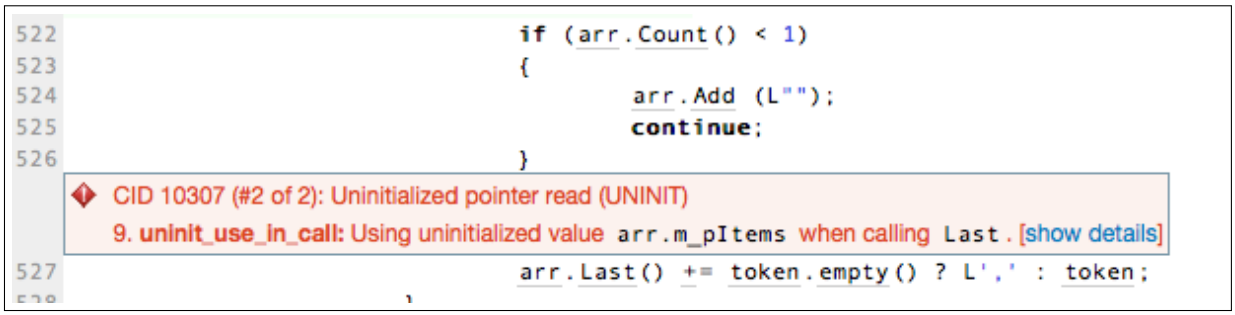
5.2.2 Coverity

Linux scan In total, the scan of the Linux version of TrueCrypt using Coverity returned 58 results. The proportion of the results relating to code quality was remarkably high. In particular, these deal with some non-initialized variables in constructors. We were able to categorize these software errors as not being relevant to security after a manual analysis, although it would nevertheless be sensible to fix these errors in TrueCrypt. The results from Coverity are extremely helpful for this purpose.

This section concentrates on three of the results that were categorized by Coverity as being security-relevant (category: Input Validation) (see Table 5.2). Our analysis revealed that these three results were false positives. We will present our findings below.



(a) The variable `arr` is initialized with the default constructor.



(b) The last element of the array is read later.

Figure 5.4: Details about the result COV-1.

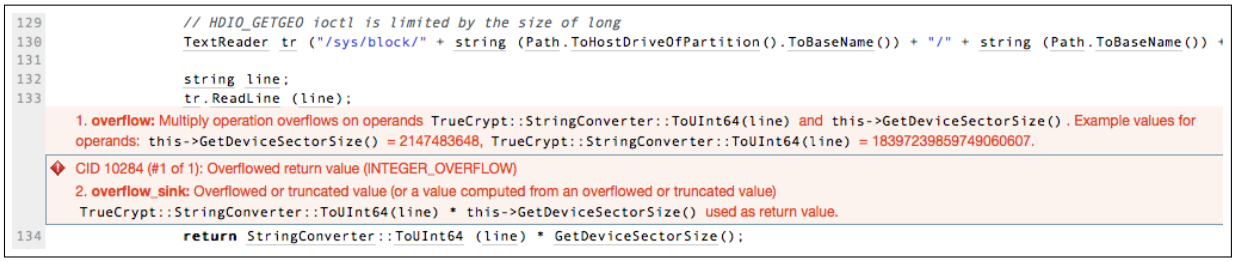


Figure 5.5: Details about the result COV-3.

COV-1 In Figure 5.3, the details about the first result returned by Coverity are presented. The method `Set` is called, whereby the second parameter is 0. Within the method `Set`, the second parameter is the variable `charCount`. Coverity returns a warning in line 135 because it assumes that the array `passwordBuf` is dereferenced with index `i`. However, it is clear that the value of the variable `i` is limited and cannot be less than 0 or more than `charCount`. Furthermore, the variable `charCount` is actually 0 in this call context and the loop is thus immediately skipped. A buffer underrun cannot therefore occur.

COV-2 The second result (Figure 5.4) is due to the insufficient initialization of a variable within the constructor for an included library (wxWidgets). The non-initialized variable is accessed at a later point in time. However, the variable is a field of an object within the library. The initialization is thus the responsibility of the developers of the library and not TrueCrypt.

COV-3 Figure 5.5 shows an integer overflow. A line is read from a file and then converted into a unsigned integer variable with 64 bits. The subsequent multiplication with another integer variable can cause it to exceed the maximum range for a variable of type `uint64` of $-2^{63}+1$ to $2^{63}-1$.

In any case, the variable `line` is limited by the memory capacity of a `long` variable (of $-2^{31}+1$ to $2^{31}-1$) (see comment). The result from `GetDeviceSectorSite()` is also of type `long`. A multiplication of two variables of type `long` cannot, however, exceed the maximum range of `uint64`.

ID	File name	Method	Type	Stated risk level
COV-4	`/Common/Keyfiles.c`	KeyFilesApply	Out-of-bounds write	High
COV-5	`/Mount/Mount.c`	DismountIdleVolumes	Insecure data handling	Medium
COV-6	`/Format/InPlace.c`	FastVolume-HeaderUpdate	Overflowed return value	Medium

Table 5.3: Risk level for the Coverity results for Windows

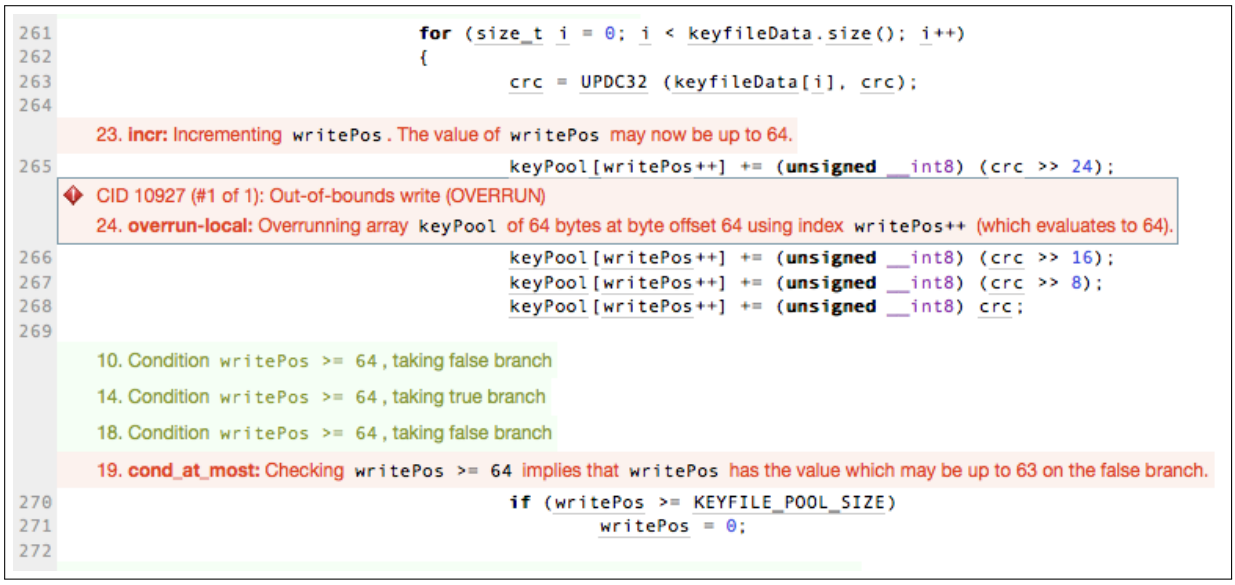


Figure 5.6: Details about the result COV-4.

Windows scan The scan of the Windows version of TrueCrypt returned a total of 158 warnings. It is also important to note here that the majority of these warnings once again referred to code quality. The 22 findings in the category "Input Validation" were of particular interest. We will demonstrate the process for the manual analysis based on three (see Table 5.3) selected examples that are relevant to security. We proceeded in the same manner with the other findings.

COV-4 This finding (see Figure 5.6) from Coverity is a false positive: Coverity returns here an out-of-bounds write access. The array `keyPool` has a length of 64 and the static analysis demonstrates that the array is written to with an index of 64. This would lead to a write access outside the array. However, the analysis does not recognize the fact that the operation `writePos++` is called precisely four times within the loop. As `writePos = 0` initializes to 0, the invariant `writePos % 4 == 0` is true after the loop. In the `if` statement in line 270, `writePos` thus has a value that is a multiple of `4`. The value is either precisely `KEYFILE_POOL_SIZE = 64` or is less and thus has a maximum of `60`. A further pass of the loop thus does not lead to any write access outside the length of the array.

COV-5 The result in Figure 5.7 shows a so-called taint flow. The displayed code is within a loop. The function `DeviceIoControl` sends a control sequence to a driver that carries out the corresponding operation. These operations could, among other things, be read and write accesses to a data volume. The static analysis assumes here that after the first pass through the loop the

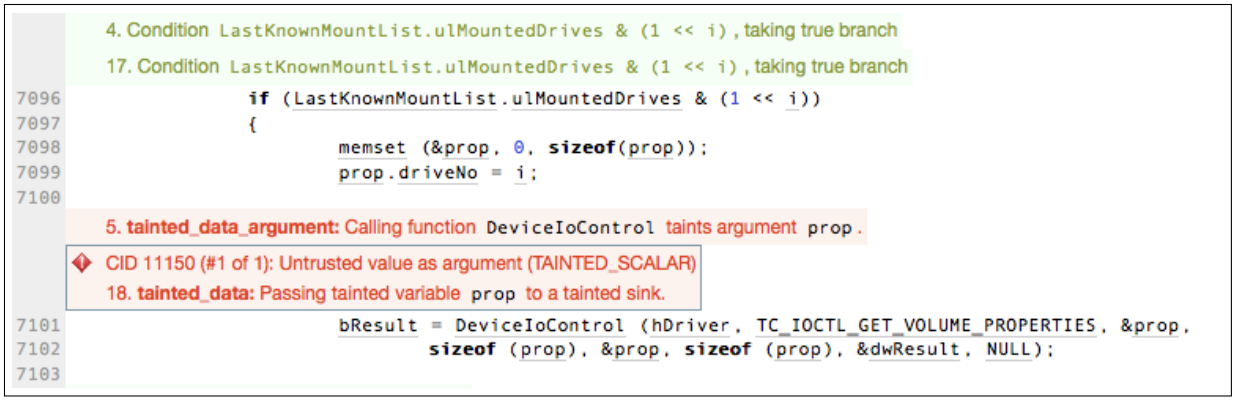
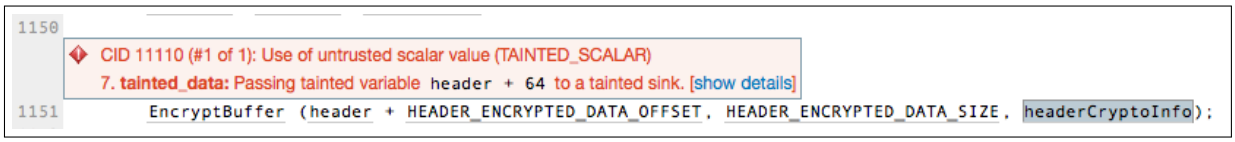
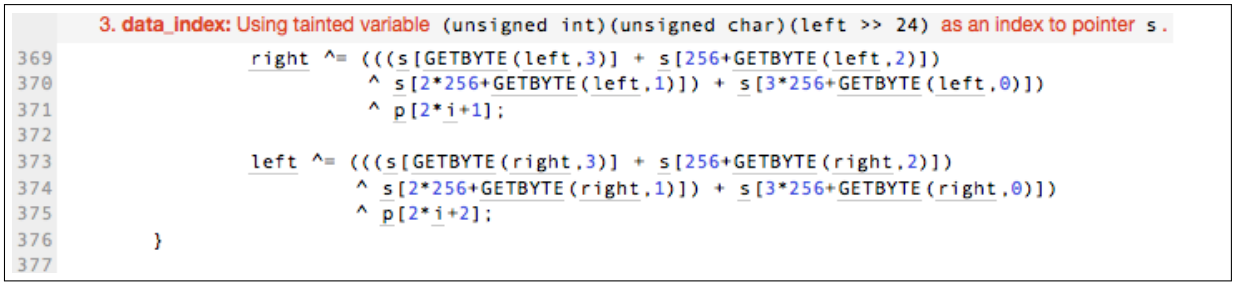


Figure 5.7: Details about the result COV-5.



(a) The variable `header` contains information about a file header.



(b) An integer representation of `header` is used later for encryption.

Figure 5.8: Details about the result COV-6.

argument `prop` could receive sensitive data e.g. through a read access. In the second pass, the function `DeviceIoControl` is called once again. This could then write the data to another data storage device. In this actual case, it can be seen that only data from the driver is loaded during each pass through the loop. In addition, the values saved to `prop` are additionally deleted by the call from `memset` (line 7098). Therefore, this ensures that `prop` does not store any data from the last pass through the loop.

COV-6 This result (see Figure. 5.8) is interesting because Coverity identifies a taint flow that uses a crypto function. The variable `header` contains information on a file header. This information then flows into the function `EncryptBuffer(...)`. Via multiple function calls, the encryption algorithm BlowFish (depending on the settings) is ultimately used to encrypt the header information. Part of the BlowFish algorithm can be seen in Figure 5.8 (b). The variable `left` contains parts of the information from the variable `header`. Of course, an encryption algorithm must use the (sensitive) incoming data in order to generate the encrypted data. Nevertheless, the variable `left` is only used as an index to access the array `s`. Therefore, we do not consider this result to be critical from a security standpoint.

5.2.3 Cppcheck

This static code scanner primarily delivers results in the area of code quality. The tool only checks the syntax of the source code and thus does not model any method calls. It was thus possible to easily categorize the results. Four other array out-of-bounds accesses proved interesting, although it was possible to quickly confirm these as false positives following a manual analysis.

5.3 Summary

As part of this work package, TrueCrypt 7.1a was analyzed in more detail for security holes with the aid of static code analysis tools. The use of three different tools demonstrated that a variety of different warnings could be found.

A precise analysis of the security-relevant warnings returned by the tools showed, however, that these were false positives that either could not occur during the runtime or did not represent a problem.

We exported the results from the scanners and have submitted them additionally alongside the report.

Results of chapter 5

- It is sensible to use a broad spectrum of tools: Clang, Coverity and Cppcheck all deliver different results.
- No security holes could be found in TrueCrypt 7.1a with an automated code analysis, although it was possible to exclude some potential holes in the program .
- It was only possible to categorize some results as false positives after a comprehensive analysis using additional tools.
- The results could make it possible to easily and efficiently resolve quality deficiencies in the implementation of the program.

6 Evaluation of the code quality and documentation

6.1 Evaluation of the code quality

The ideal quality assessment of software artifacts would deliver a formal proof of the correctness of the program. In the case of large and complex software such as TrueCrypt, formal verification still remains impractical. The most common method for verifying software is thus via functional tests. The thoroughness of these functional tests is mostly assessed based on their coverage of the source code. Unfortunately, almost no suitable test cases exist for TrueCrypt, which means that it is not possible to carry out a meaningful assessment of the quality of the software on this basis.

Evaluating the code quality based on the software's internal structure and source code is an indirect but nevertheless helpful indicator of the overall quality of the software. In particular, it has been shown that the code quality has a major influence on the long term maintainability of a software project. Particularly in view of the termination of the official TrueCrypt project, we believe that an evaluation of the maintainability of the code base is especially important. Those institutions that decide to continue using TrueCrypt will necessarily be responsible themselves for maintaining the code base.

6.1.1 Programming guidelines and best practices

The documentation for TrueCrypt does not contain any information on programming guidelines. A set of programming guidelines is very common for large software projects because languages such as C and C++ allow an extremely broad range of different programming styles. Defining programming guidelines enables project managers to ensure that contributions by multiple programmers to a common source code all have a uniform programming style. The reason for the lack of programming guidelines for the TrueCrypt project is presumably the small number of original developers. Nevertheless, the TrueCrypt source code displays a mix of different conventions. For example, compound variable names are written both with an underscore (`file_name`) and also using camel case (`fileName`). Equally, complex data structures are defined using a mixture of the C++ constructs `class` and `struct`.

Violation of generally accepted rules

The use of the `goto` command was already criticized in the 1960s and 1970s by advocates of structured programming such as E. W. Dijkstra [7]. In the TrueCrypt source code, we were able to count 388 occasions when `goto` statements are used. However, the `goto` statement is generally considered appropriate when used to deal with exceptional cases in the program because the C programming language does not have its own construct for this purpose. Recent studies have shown that programmers now primarily use the `goto` command only in a sensible manner [20]. This is also the case with TrueCrypt. The use of `goto` statements is limited to exceptional cases. In the normal control flow, `goto` is not used.

Listing 6.1: GetSystemPartitions

```
1 static bool GetSystemPartitions (byte drive) {
2     size_t partCount;
3     if (!GetActivePartition(drive))
4         return false;
5     // Find partition following the active one
6     [...]
7     return true;
8 }
```

Listing 6.2: File::Read

```
1  DWORD File::Read (byte *buffer, DWORD size) {
2      DWORD bytesRead;
3      if (Elevated) {
4          DWORD bytesRead;
5          Elevator::ReadWriteFile(false, IsDevice, Path,
6                                  buffer, FilePointerPosition,
7                                  size, &bytesRead);
8          FilePointerPosition += bytesRead;
9          return bytesRead;
10     }
11     throw_sys_if(!ReadFile(Handle, buffer, size, &bytesRead, NULL));
12     return bytesRead;
13 }
```

Multiple return statements

Similar to the use of `goto`, multiple `return` statements within a function are also generally considered to be bad style, although there are also exceptions to this rule. In his influential book Code Complete [16], Steve McConnell wrote the following on this subject:

> "*Minimize the number of returns in each routine.* It's harder to understand a routine if, reading it at the bottom, you're unaware of the possibility that it returned somewhere above.
>
> *Use a return when it enhances readability.* In certain routines, once you know the answer, you want to return it to the calling routine immediately. If the routine is defined in such a way that it doesn't require any cleanup, not returning immediately means that you have to write more code."

TrueCrypt contains a large number of functions with multiple exit points. Listing 6.1 shows an example of an acceptable early exit from a function (abridged). The exit condition in line 3 can be calculated directly from the function parameters. An immediate exit has no side effects with respect to the rest of the code in the function.

In contrast, listing 6.2 shows an example of a questionable use of a `return` command (line 9). The calculation of the exit condition covers the majority of the body of the function and could be easily overlooked.

Application logic within header files

Another noticeable feature is the violation of the standard rule that header files with the ending `.h` should only contain definitions for interfaces and data structures. In the programming language C, splitting interface definitions and implementations is common in order to allow them to be compiled separately. In contrast, constructs such as template classes in the more modern

Listing 6.3: WasHiddenFilePresentInKeyfilePath

```
1 static bool WasHiddenFilePresentInKeyfilePath() {
2     bool r = HiddenFileWasPresentInKeyfilePath;
3     HiddenFileWasPresentInKeyfilePath = false;
4     return r;
5 }
```

programming language C++ are only created at the time the code is compiled. For this reason, the implementation of the class templates must be available during compilation. This can be achieved more simply by implementing the class templates in the header file. Better and more elegant solutions for this problem exist and are preferable, although they are often dependent on the compiler and are not portable.

In the TrueCrypt source code, there are a number of header files that contain application logic. One example is the definition for function `MountVolume` in the file `Core/Unix/CoreServiceProxy.h`, which contains more than 70 lines of complex code. However, `MountVolume` is a function template and this justifies it being embedded in the header file.

A clearer violation of standard modularization is the presence of multiple short fragments of code in header files that do not serve to define templates. Listing 6.3 from the file `Volume/Keyfile.h` shows a typical example of this type of short function definition within a header file.

6.1.2 Complexity of the source code

One thing that intuitively makes sense and that has been substantiated by a large amount of scientific work is the close interrelationship between the complexity of the source code and the maintainability of the software. The simplest and most common metrics for measuring the complexity of the source code include the length of the functions and the complexity of the control flow. In particular, cyclomatic complexity is used to measure the complexity of the control flow. Values larger than 15 indicate that refactoring would be sensible. Values over 30 are often associated with error prone code [15].

In order to evaluate the maintainability of TrueCrypt, we used the analysis tool Lizard[1]. Lizard measures the length and cyclomatic complexity of C, C++ and Java functions and issues warnings when various thresholds are exceeded. The results for TrueCrypt are worrying. There are 170 functions with a cyclomatic complexity greater than 15. 75 functions had a cyclomatic complexity greater than 30 and 9 even had a value over 100. The 75 functions with the highest values are listed in Appendix A.

In terms of their length, five functions stand out due to the fact that they contain more than 500 lines of code. The three longest functions contain more than 1500 lines of code but implement the user interface of TrueCrypt and are presumably not critical to the core functionality of TrueCrypt. In contrast, the two functions *ProcessMainDeviceControlIrp* and *TCOpenVolume* each with a length of over 500 lines of code are part of the device driver that TrueCrypt integrates into the Windows operating system and therefore safety critical.

6.1.3 Code Clones

Another typical indicator of the need for code refactoring is the presence of code clones – identical sequences of code that are repeated in different sections of the project. It is obvious that this type of repeated code should be combined in one function to improve the modularity of the code and to avoid the duplicate code drifting apart due to negligent code maintenance.

[1] https://github.com/terryyin/lizard

Listing 6.4: Codeduplikat in Mount.c

```
1 if(!VolumeSelected(hwndDlg)) {
2     Warning("NO_VOLUME_SELECTED");
3 } else {
4     GetWindowText(GetDlgItem(hwndDlg, IDC_VOLUME),
5                   volPath,  sizeof (volPath));
6     WaitCursor();
7     if(!IsAdmin()&& IsUacSupported()&& IsVolumeDeviceHosted(volPath))
8         ...
```

The analysis tool Duplo[2] finds duplicate code in C and C++ source code. In the standard setting, duplicate code with a length of three lines of code or more is identified. In the TrueCrypt code base, Duplo identified 7091 duplicated lines of code in 1155 duplicated blocks of code (see Appendix B).

One example of the duplicate code found by Duplo is shown in Listing 6.4. This code sequence consisting of seven lines of code is present in identical form in four different locations in the file `Mount.c`.

6.1.4 Summary

We would like to emphasize in summary that the problems with the quality of the source code listed here are not necessarily vulnerabilities in TrueCrypt. However, they do justify questioning the reliability of TrueCrypt to some extent. In particular, the high complexity of the code combined with a lack of comprehensive test cases leads us to anticipate that the maintenance work and maintenance costs will be very high. As the project will no longer be maintained by the original developers, the responsibility for the maintenance work must be assumed by individual users.

6.2 Evaluation of the documentation

The security characteristics of IT systems are mostly dependent on the fact that certain assumptions about the environment or the use case apply. If these assumptions are not fulfilled or if the user is not aware of any limitations, security mechanisms can become ineffective or even have the opposite effect due to configuration or user errors. For example, a user could in fact arouse suspicion through the incorrect use of the plausible deniability function in TrueCrypt.

Developers, who join the project at a later point in time, and system administrators, who acquire and set up software for others, also require a complete understanding of the assumptions and limitations of the security functions. Otherwise, there is a danger that the security features will be nullified by programming or configuration errors due to ignorance.

In the case of a security product such as TrueCrypt, complete, up-to-date and target-group specific documentation of all security-relevant features is thus an indispensable component.

Available documentation

An English language “User’s Guide” [9] for end users was available for TrueCrypt until the termination of the project, as well as a project website [10] that had identical content to a large extent. The content available on the website before the project was terminated has been archived and can be accessed under [24].

[2] http://duplo.sourceforge.net/

The source code for TrueCrypt 7.1a contains a file "Readme.txt" with instructions on how to build the source code for the platforms Windows, Linux and MacOS. One paragraph in this file is aimed at software developers who want to contribute to the source code. It is generally recommended here that software developers get in contact with the TrueCrypt developers.

There is no other publicly available documentation to our knowledge. In particular, there does not appear to be any publicly available information for software developers who want to contribute to the further development of TrueCrypt or who want to use parts of the program in their own projects.

Quality of the content and completeness of the documentation

The "User's Guide" [9] describes the security features of TrueCrypt at a conceptual level. However, the reader is already assumed to have an understanding of the basic functionality. For example, it does not explain what encryption is and what purpose it serves.

The functionality of TrueCrypt is described in great detail, such as calculating keys from keyfiles or the structure of volume headers in Chapter "Technical details". The level of detail provided in the document enables security experts to make a meaningful evaluation of the security concept offered by TrueCrypt. It thus provides developers with a first point of entry for understanding the design of the software.

In principle, the description appears to be sufficiently precise for all core functions of TrueCrypt to be reimplemented based on it. However, the text contains some errors that were discovered by the developers of the compatible product "tcplay" [12, README.md]:

> "The TrueCrypt documentation is pretty bad and does not really represent the actual on-disk format nor the encryption/decryption process.
>
> Some notable differences between actual implementation and documentation:
>
> - PBKDF using RIPEMD160 only uses 2000 iterations if the volume isn't a system volume.
> - The keyfile pool is not XOR'ed with the passphrase but modulo-256 summed.
> - Every field except the minimum version field of the volume header are in big endian.
> - Some volume header fields (creation time of volume and header) are missing in the documentation.
> - All two-way cipher cascades are the wrong way round in the documentation, but all three-way cipher cascades are correct."

The "User's Guide" does not provide end users with sufficient information. For example, the user is left to select the hash functions and encryption algorithms to be used. However, there is no recommendation at all about which selection or selection criteria are most sensible.

A list of the basic assumptions and limitations can be found in Chapters "Security model" and "Security requirements and precautions". A lot of additional information about possible threats and insecure ways of application is given throughout all of the chapters depending on the relevant context. Therefore, it is necessary to read the whole document in order to be able to take into account all of the information on the secure use of the program.

Comments within the source code

The TrueCrypt source code only has sporadic comments. In most cases, they are in the form of shorthand notes at the end of lines, such as to describe the function of a variable.

Warning messages for developers can be found in approx. 40 places labeled as “WARNING” or “IMPORTANT”. These make developers aware of possible sources of error if they change the code, such as:

> “IMPORTANT: Modifying this value can introduce incompatibility with previous versions”
>
> “WARNING: ADD ANY NEW CODES AT THE END (DO NOT INSERT THEM BETWEEN EXISTING). DO *NOT* DELETE ANY”

A few comments are more comprehensive and also provide reasons for the warnings being given. This type of information is very important for subsequent development work.

Overall, the comments focus on local implementation details. There is no description of the overarching design or architecture aspects. There is not even any summarized descriptions of the functions implemented in the individual source code files. One exception here is when parts of the source code have been adopted from third party sources.

6.3 Summary

Results of chapter 6

Evaluation of the code quality:

- A lack of comprehensive test cases.
- No written program guidelines. The style of the source code is inconsistent. Failure to observe best practices.
- Poor maintainability and thus high maintenance costs (partially due to the high complexity, duplicate code)

Evaluation of the documentation:

- The documentation is primarily limited to the “Users’ Guide”. There is no documentation for developers.
- The “Users’ Guide” contains lots of information on the functionality and use of TrueCrypt but is nevertheless poorly structured and contains mistakes.
- The source code is supplemented sporadically with short comments.

7 Conceptual evaluation of the architecture

7.1 Introduction

This chapter is dedicated to a threat analysis of TrueCrypt and based upon this a subsequent conceptual evaluation of the TrueCrypt architecture. The basis for these evaluations is a series of assumptions that cover the intended usage context and the resulting requirements. This is necessary because an all-encompassing evaluation of the attack scenarios and suitability of the system architecture can never be carried out in absolute terms but instead is always linked to concrete, situation-specific factors. In the following section, we will firstly document these assumptions and define a set of intended security goals. We will then present various potential attack strategies and evaluate their relevance in relation to the TrueCrypt architecture.

7.2 Context and use cases

The usage context for TrueCrypt is to protect sensitive data from unauthorized access in a private, official or company setting. These three areas of application are viewed as being broadly similar for the purpose of this evaluation, although this report will touch on any special features at certain points where required. We have assumed that those systems protected by TrueCrypt in an official or company setting are at least logically embedded in a computer network and maintained by administrators. Furthermore, we have assumed that at least stationary systems (PCs) are generally operated at the premises of the relevant employer.

Use cases for TrueCrypt that are relevant for the further evaluation of the software are derived from the description of the functionalities that we have taken from the TrueCrypt User's Guide. This resulted in the following three use cases:

ID: UC1

Title: Protecting information using system encryption

Description: The user utilizes a computer to process sensitive data. The user always switches the computer off when it is not being used. The user utilizes TrueCrypt to protect sensitive data on non-volatile memory (generally a hard disk) against unauthorized access. TrueCrypt prompts the user to authenticate themselves when starting the system. TrueCrypt prevents the booting of the operating system and thus access to the sensitive data if the authentication process fails and only permits access when the authentication process is successful. If the authentication process is successful, the user can work with the data on the computer in the usual manner. This system encryption functionality is only available for certain Windows versions.

Variants:

a) **Encryption of the system disk**
TrueCrypt encrypts the entire hard disk on which the system partition is located. TrueCrypt asks the user to enter a password as authentication immediately after the computer is started.

b) **Encryption of the system partition**
TrueCrypt only encrypts the system partition and asks the user to enter a password as authentication immediately after the computer is started. Partitions other than the system partition can remain unencrypted.

ID: UC2

Title: Protecting information using encrypted volumes

Description: The user processes sensitive data on a computer or wants to transfer a copy of the data to an external non-volatile data storage device (e. g. as a backup copy, or to share the data with project partners for official business use). The user wants to protect the sensitive data against unauthorized access on the transport device and/or the non-volatile data storage device. For this purpose, TrueCrypt is used to save the data requiring protection in an encrypted volume. Instead of the unencrypted readable data, the user transfers/saves the volume. In order for the user to be able to read, write and edit the encrypted data again, TrueCrypt creates a virtual drive in which the contents of the volume are displayed in readable form (»mounting«) after the successful authentication of the user. As is the case with regular drives, the user can save, delete and edit files on the virtual TrueCrypt drive. The changes are automatically mapped to the volume by TrueCrypt.

Variants:

a) **File-hosted volume**
TrueCrypt provides the option of saving an encrypted volume to file (also known as »file-hosted (container)«). This container is a regular file that can be saved, deleted and otherwise edited like other files. In the context of this use case, the user could, for example, save this file as a protected backup copy to an external hard disk or also send it via email to a project partner.

b) **Partition/device-based volume**
Alongside a file-hosted volume, TrueCrypt also provides the option of so-called partition/device-hosted volumes. Here, the volume is directly saved to a partition on a data storage device or it utilizes the entire memory of a data storage device. If the volume is saved within a partition on a data storage device, it is not permitted, however, to use the system partition of the system that is currently running.

ID: UC3

Title: Maintaining confidentiality and protecting information using hidden volumes

Description: The user wants to save sensitive data that he/she has processed on a computer in such a way that, on the one hand, it is protected against unauthorized access and, on the other hand, its existence cannot be ascertained without the appropriate prior knowledge. Therefore, the user cannot be forced to reveal the data because he/she can plausibly claim that it does not even exist. The user utilizes TrueCrypt here to create a hidden volume within a different TrueCrypt volume. From a user perspective, access to the data in the hidden volume is achieved in the same way as access to data within a regular volume. The user is firstly asked to authenticate themselves to TrueCrypt and then receives access to a virtual drive on which the hidden data can be read.

Variants:

a) **Hidden volumes**
TrueCrypt can be used to save a hidden volume within a different TrueCrypt volume, which can be accessed in the same way as a regular volume. Both file-hosted and partition/device-hosted volumes are suitable as the »outer« volume.

b) **Hidden operating system**
TrueCrypt can also be used to hide a complete system partition (»hidden operating system«). At least three TrueCrypt volumes are required in total for this purpose. The first TrueCrypt volume contains a complete system partition, whose existence cannot be plausibly denied (known as the »decoy operating system«). The user could thus be forced to reveal a password and hence the volume should not contain any really sensitive data. In addition, there is a second TrueCrypt volume whose existence also cannot be plausibly denied. However, this second volume serves as the »outer« volume for the third hidden volume, which contains the hidden operating system. When the computer is started, TrueCrypt asks for a password to be entered. If the user enters the password for the decoy system, only the operating system for the first partition is booted. However, if the user enters the password for the hidden operating system, this system will be booted. This is a form of system encryption and is also only available for certain Windows versions.

7.3 Security goals and requirements

In the following section, we will describe the security goals that we consider to be relevant in the context of the use cases outlined above. They form the basis for a number of requirements (R1-R7) that can be used for evaluating the system architecture.

Primary security goal

Based on the usage scenarios and the intended purpose of TrueCrypt, the primary security goal is the protection of confidential data in a TrueCrypt volume or TrueCrypt container against access or perusal by unauthorized persons (confidentiality of data in a container or volume).

R1: Protecting the confidentiality of data must be achieved through the use of suitable cryptographic processes. The sole use of a simple authentication mechanism for restricting access, such as the request for a password to enable a user to log in that is typical of operating systems, is completely inadequate because this can be circumvented using the simplest of means.

R2: Data whose confidentiality is to be protected by TrueCrypt must not be stored as plaintext on a non-volatile data storage device or transferred to third parties by TrueCrypt. In particular, this means that the cryptographic processes that are used to protect the data must be carried out locally on the user's computer. They are not, for example, permitted to be outsourced to an external service provider.

Secondary security goals

Alongside the confidentiality of the saved data, there are other security goals that can be applied to TrueCrypt to a limited extent. Thus, limited protection of the integrity of the data encrypted by TrueCrypt is desirable: It should be impossible to make any changes to the ciphertext with the aim of forcing certain plaintext to be revealed during the decryption process. Yet there is nothing in the description of the TrueCrypt software indicating that the recognition of changes to the ciphertext is a relevant security goal.

Special features of the software are the modes »hidden volume« or »hidden operating system«. In these operating modes, the deniability of the existence of the encrypted data is a relevant security goal. This is a special feature of TrueCrypt in comparison to other encryption software.

R3: The integrity of the data whose confidentiality is protected by TrueCrypt should be protected. A user should thus be able to recognize whether protected data has been changed by any unauthorized person since the last time it was worked on with authorized access.

R4: The integrity of TrueCrypt itself should be secure. An authorized user should be able to recognize when using TrueCrypt whether it has been changed by any unauthorized person since it was last used.

R5: TrueCrypt must not save any information on a non-volatile data storage device that indicates whether a TrueCrypt volume contains another hidden volume or not. This is true for both hidden volumes and also for hidden operating systems.

Derived security goals

The features of an encryption software reveal the fact that the confidentiality of the data in a TrueCrypt volume or TrueCrypt container is directly dependent on the confidentiality of the key, keyfiles or passwords used for the decryption. Therefore, the confidentiality of these metadata is also a security goal of TrueCrypt, at least to the extent that TrueCrypt is able to influence the security of this data.

R6: TrueCrypt is not permitted to save secret keys or passwords in plaintext on a non-volatile data storage device at any time.

R7: TrueCrypt is not permitted to transfer secret key or passwords to third parties at any time.

7.4 Attack strategies

7.4.1 Preliminary considerations

In terms of the security provided by TrueCrypt, IT systems protected by TrueCrypt primarily face a threat from potential attackers who want to gain knowledge about confidential information stored on the system and then attempt to extract data from this IT system for this purpose. In principle, it is also conceivable that the system may face a threat from attackers who want to alter, add or delete data on the system. However, these threats have a lower level of significance, see section 7.3.

As an attacker requires access in some form or other to the IT system protected by TrueCrypt, it is possible at the beginning of this analysis to make a general differentiation between two different situations (see Table 7.1). An attacker can gain both

physical access to an IT system protected by TrueCrypt, as well as

logical access to an IT system protected by TrueCrypt.

The access can be temporary or also permanent. Physical access is possible in situations where the attacker gains unauthorized entry to the premises or locations in which the protected IT system is operated or situated (offices, server room), or a protected IT system can be left exposed in an environment to which the attacker has authorized access (public areas, non-public areas with public traffic). Logical access is possible if an attacker can influence the data handling processes in the IT system, without having to physically access the system e .g. through unauthorized

use of the remote access capabilities of a legitimate user or with the help of malware that has infiltrated the system. Physical access often also enables logical access, although the reverse is not true in most cases.

Another important differentiation is the ability of the attacker to gain either just single access or repeated access to the IT system (see Table 7.1). Correspondingly, it is also possible to differentiate here between

single access in which the attacker has access to the protected IT system or the data storage device containing the protected data on a single occasion and

repeated access in which the attacker has access to the IT System or the data storage device containing the protected data on multiple occasions, whereby the legitimate user of the IT system or the data storage device regularly uses them between these attacks.

Cases of single access are not usually directly linked with targeted attacks that specifically focus on the confidential data stored on a particular IT system. The threat to the confidential data is more often a side effect, whereas a single access attack targets the appropriation of physical entities, especially the IT system's hardware. The most common example is access to hardware in public areas or areas with public traffic, or break-ins to closed premises with the objective of stealing expensive hardware such as laptops or servers.

In contrast, it can be assumed that targeted attacks often utilize possibilities for achieving repeated access to the IT system. These possibilities are typically open to insiders (regular employees or service personnel) through authorized access to IT systems or data storage devices that are operated or stored without supervision at certain times. In premises with relatively poor security (e.g. private houses or apartments), there is also the potential for repeated access. The possibility of repeated access also always includes the possibility of single access; while logical access enables repeated logical attacks on the IT system in many cases.

Attack scenario	Description	Examples
Single access	Attacker gains single access to the data storage device or protected IT system	Theft
Repeated access	Attacker gains repeated access to the data storage device or protected IT system	»Evil maid attacks«, targeted attacks on individual people
Physical access	Attacker gains physical access to the data storage device or protected IT system	Theft, entry to premises with the IT system
Logical access	Attacker gains logical access to the data storage device or protected IT system	Malware, circumventing logical access controls

Table 7.1: Types of attack scenarios and examples

7.4.2 Overview of attacks with physical access

In line with the security goals from 7.3 and taking into account the use cases from 7.2, those attacks in which the attacker gains at least partial physical access to the system protected by TrueCrypt for a short period of time are particularly important. In terms of this type of access, there are a number of different possibilities for making an attack on the confidential data

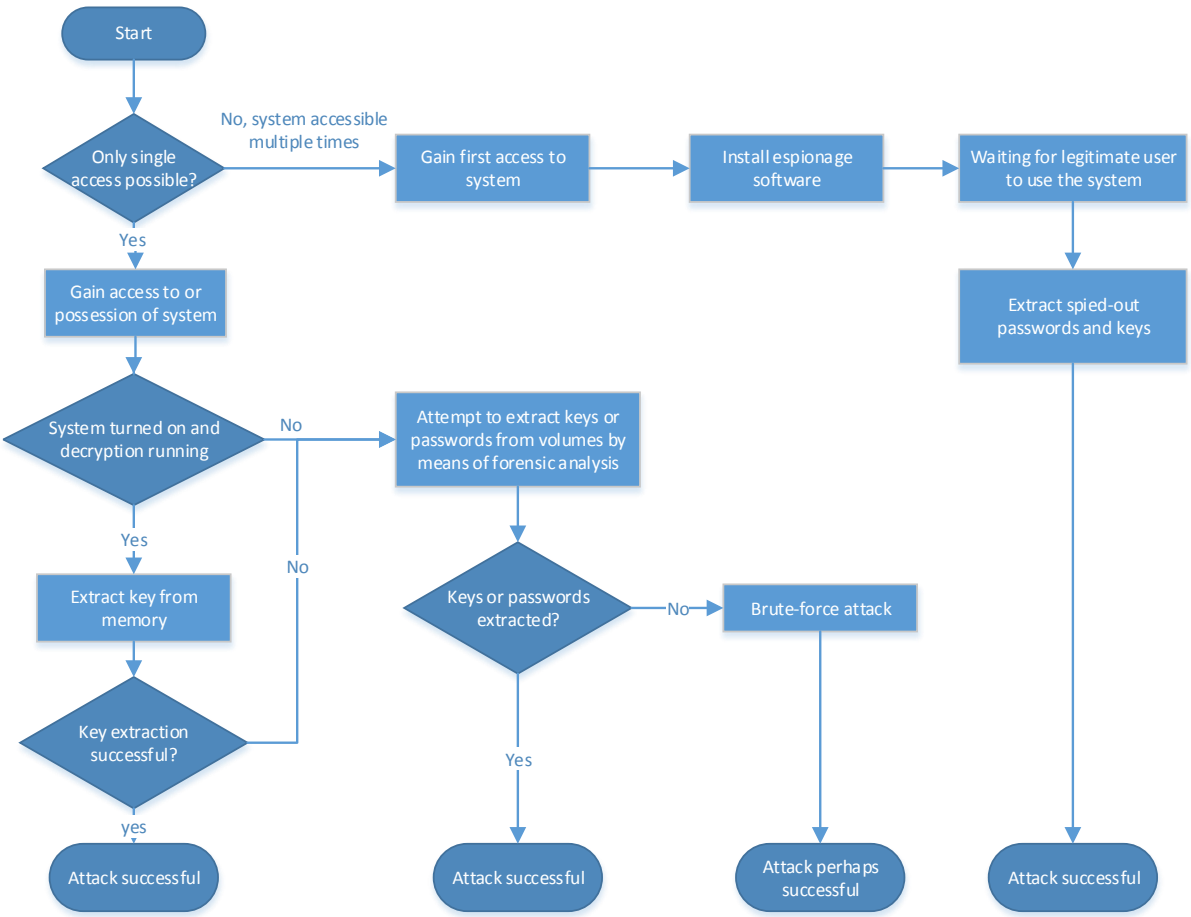


Figure 7.1: Options for attacking TrueCrypt with physical access

protected by TrueCrypt in these scenarios. Figure 7.1 shows a process flowchart that describes in principle all of the attack strategies that will be explained in detail below.

As shown in Figure 7.1, it is always initially decisive whether in principle only single access or also repeated access is possible. If repeated access is possible, the detection of passwords and keys through cyber espionage is, depending on the technical capabilities of the attacker, the method of choice because this attack strategy promises the greatest probability of success. At the same time, there are also suitable attack vectors for a case of single access. If only single access is possible, the attack vectors and thus the associated probability of success are dependent on the condition of the compromised system. If the system is active and TrueCrypt's decryption is running, the attacker can attempt to read the available encryption key from the volatile memory. Depending on the system being attacked, this requires an advanced or high level of technical expertise; while it also promises a successful outcome in principle. If the system is inactive or the attacker only gains access to the data storage device, the only options are a forensic search for passwords and keys or brute-force attacks.

7.4.3 Detailed overview of the attack strategies

Brute force attacks

A brute-force attack involves the attacker attempting to guess a TrueCrypt password or a cryptographic key by trying to decrypt the TrueCrypt volume header with randomly selected keys or passwords. This strategy can lead to success if the user of the system has selected their

passwords or keys from a small set of all possible passwords or keys. If this is not the case, the probability of the attacker succeeding is very unlikely.

Brute-force attacks are made more difficult by »Key stretching« in the key derivation function, meaning the repeated use of a hash function in an access key or the derived intermediate key in a loop. When testing possible passwords, the attacker must then also run through these iterations every time for each password, as well as the associated effort involved in the data processing.

Brute-force attacks can generally be considered when the attacker gains possession of encrypted date (ciphertext) through

- Permanent possession of the IT system through theft or
- Possession of data storage devices containing the encrypted data (e.g. due to the withdrawal from service of data storage devices) or
- Copying encrypted data

but cannot extract the key from the system or data storage device in any other way; the attacker has no method of interaction with the original owner of the system (e.g. through social engineering) or the possession of the system or data storage device has already been noticed by the victim. Brute force attacks are thus a typical strategy when only one single physical access to the IT system or data storage device is possible.

Forensic analysis of unencrypted data in the IT system

Instead of attempting to break the TrueCrypt encryption by testing potential keys or passwords, an attacker can also attempt to examine the unencrypted data stored on the IT system to forensically search for clues to the keys and passwords used. Potential targets for this type of forensic analysis are primarily non-volatile memory but also volatile memory if the attacker has access to a running system.

In the forensic analysis of non-volatile memory, the focus is mainly placed on files in which the operating system or TrueCrypt itself could have temporarily stored or logged keys and passwords. Examples include swap files, temporary files, log files or input buffers. The analysis does not only have to focus here on files that are referenced in the existing file systems, but it can also cover areas of memory on the data storage device that are no longer referenced but have not yet been otherwise overwritten. If TrueCrypt is used to not only encrypt data partitions but also the partition containing the operating system, the probability of success for this type of attack is low.

If the IT system is still in an active state when the attacker gains access, forensic attacks of the volatile memory are also possible in principle e. g. so-called cold boot attacks in which the attacker attempts to read content on the main memory with their own software, either by removing memory modules that have been cooled for this purpose or booting their own system. In certain circumstances, an attack is also possible via external interfaces e. g. via DMA.

Detection of passwords and keys using cyber espionage

If an attacker has the opportunity to gain repeated access to the IT system between periods of use by the legitimate user, the detection of passwords and keys using cyber espionage will be the first choice of attack. The precise form of the attack is dependent on the technical capabilities of the attacker, how inconspicuous the attack should be and the time available for accessing the IT system. Overall, the range of conceivable attack vectors is extremely diverse and thus it is only possible to outline selected ones at this point:

A relatively simple but thoroughly effective measure is to install a hardware keylogger between the keyboard and the computer to record all keyboard inputs, including TrueCrypt passwords.

However, this is only possible without too much effort on desktop computers or on laptop docking stations. If the operating system partition is not encrypted, the attacker can also install a software keylogger instead of a hardware keylogger. In some cases, the attacker may need to temporarily remove the data storage device and install the cyber espionage software on it using another system, which would take more time. Manipulating software on an encrypted operating system demands a higher level of technical knowledge because it requires changes to be made to the boot loader. If the attacker possesses the appropriate level of technical expertise, he or she can complete the manipulation themselves within a few minutes – as Türpe et al have already demonstrated using the example of Microsoft BitLocker [23].

Rather more conspicuous but just as possible in principle is for the attacker to replace the computer with one that looks identical. The »duplicate« would only run the Trojan horse and then cause the boot process to fail with a realistic error e.g. an alleged data storage device error that makes the system unusable. As the data storage device is completely encrypted, in practice it is initially difficult to detect that it is actually a different device. Closer examination for any signs of use, serial numbers or an attempted decryption with a backup of the master key could reveal the attack. However, the attacker may then already be in possession of the password.

It is also conceivable that an attacker could attempt to directly introduce malicious code onto the data storage device via the data processed by TrueCrypt and then execute it e.g. via a buffer overflow. For this purpose, the attacker could either alter header data or ciphertext. In contrast to the other types of attack, this would, however, require a vulnerability in TrueCrypt. Apart from the relative simplicity of the attack, this type of attack does not offer any advantages over the others already described.

An attacker does not necessarily need to read the key and password gained through cyber espionage via a second physical access to the computer. In principle, it is also possible for the attacker to utilize a wired or wireless network or a broadband connection. If the attacker has already copied the encrypted data storage device on-site in advance, they can directly decrypt the data storage device once they have gained the key or password through cyber espionage.

In summary, it should be noted that the »repertoire« of cyber espionage attacks is almost limitless. Should an attacker have repeated access to the TrueCrypt-encrypted system – whereby only the first access needs to be of a physical nature in some circumstances – a multitude of different cyber espionage attacks are possible. The attacker can adapt almost any number of different factors: The level of risk the attacker is prepared to accept with respect to the attack being discovered, the logical accessibility of the system through network interfaces, the time available for the physical access and the gullibility of the legitimate user. It is thus plausible to assume that there is a high probability that a cyber espionage attack carried out by a technically sophisticated attacker with corresponding criminal intent will prove successful.

Side-channel attacks

Side-channel attacks on TrueCrypt could play a role if an attacker gains physical or logical access to an IT system on which a decryption process for a data volume encrypted by TrueCrypt is already running. In this type of situation, it is conceivable that an attacker could record physical parameters from the system, e. g. the power consumption of certain components, in order to be able to draw conclusions about the key used for processing the data. If logical access is possible, other information could also serve this purpose such as program execution times or memory usage.

However, it should be noted that side-channel attacks generally require a high level of technical expertise and that the attacker is well equipped with the necessary material resources, at least if the attacker wants to gain access through the monitoring of physical parameters. Nevertheless, side-channel attacks remain a realistic scenario even though an attacker, for economic reasons alone, would prefer to utilize other types of attack insofar as the situation allows.

Attacks with logical access to the IT system

Even if an attacker only has limited logical access to the IT system on which the TrueCrypt decryption process is running, there are nevertheless conceivable attack scenarios that exploit vulnerabilities in the security of TrueCrypt. These scenarios could arise when logical access is limited due to access rules on the data storage device, partitions or parts of the file system on the IT system so that an attacker is not permitted to read all of the encrypted TrueCrypt volume or TrueCrypt container. For example, a vulnerability in the TrueCrypt driver could enable a privilege escalation through which the attacker can gain far-reaching, additional access rights. An example of this type of vulnerability in a FAT driver can be found under [17]. Just recently, two vulnerabilities of this kind were found in TrueCrypt [18, 19]. As part of the privilege escalation, it could also be possible to gain access to encrypted data that is secured by access controls.

However, this type of attack is a fringe scenario: firstly because in principle any other driver, or the kernel itself, could have a vulnerability that would enable a privilege escalation. TrueCrypt is thus impacted by this security risk because it is a software that runs in the kernel and not because of its encryption software characteristic. Secondly, this attack scenario requires the attacker to already be capable of exchanging data with the TrueCrypt driver in order to exploit this vulnerability. Essentially, this would only be the case if the attacker could already interact with TrueCrypt's administrative tools or if the attacker already had (limited) access to a TrueCrypt encryption or decryption process. In both cases, the attacker would already possess far-reaching privileges for the system being attacked e. g. user access to certain program execution privileges and read privileges for the data storage device.

Indirect attacks without physical access to the system

Alongside the aim of expanding his or her own privileges, if an attacker already has direct, logical access to the IT system, it is also conceivable that he or she would »plant« maliciously compromised data within that of the legitimate user for processing in TrueCrypt. This could include e. g key files, TrueCrypt containers or plaintext data or files for encryption by TrueCrypt at a later point in time. An attacker could specifically exploit a vulnerability in the testing and processing of input data in TrueCrypt e. g. provoking a buffer overflow by executing malicious code.

As a result of the unauthorized execution of malicious code on the IT system running TrueCrypt, an attacker could attempt to weaken or nullify the security mechanisms of TrueCrypt. However, it is necessary to question what form this scenario would actually take in reality and what cost-benefit effect the attacker could achieve in comparison to other attack strategies. In order to compromise the IT system with malicious code, other methods for exploiting vulnerabilities would be more suitable e. g. via attacking rendering algorithms for web browsers, in Adobe Flash software, in PDF software, etc. If an attacker could achieve far-reaching logical access to the system with these types of attack, he or she would generally already be capable of reading confidential data from unlocked TrueCrypt containers or partitions. However, this type of attack becomes interesting from a practical standpoint for preparing for a possible physical attack; the attacker could, for example, use malicious code to save keys in plaintext in non-volatile memory. Then, when the attacker subsequently gains physical access to the system, he or she simply has to steal the data storage device and the key required for the decryption process could then be read directly from it.

Attacks via backdoors

Another special attack strategy is the exploitation of so-called »backdoors« in TrueCrypt, which are only known to the attacker. In order for an attacker to integrate a backdoor into TrueCrypt, he or she must have had access to the software's source code that has gone unnoticed or have

worked as a developer on the software. A backdoor can also find its way into the software through other means e. g. through manipulation of the source code in a source code repository that remains unnoticed by the developers. Alternatively, an attacker could have manipulated the source code or compiled sections of the program before they are transferred from the developer to the end user, maybe even including the required public key for certifying the code.

The backdoor can implement a variety of functions, whereby it is often important for the attacker that manipulations remain undiscovered. Hiding the backdoor is particularly important if the attacker has directly compromised the original source code of the developers. Alterations to the original source code could and should be very subtle e. g. a deliberate weakness in the implementation of a cryptographic algorithm to simplify cryptographic analyses. Large-scale manipulation is more suitable for already compiled code, such as a targeted alteration before or while the end user downloads the TrueCrypt program package.

Altering encrypted data

In contrast to the previously described attacks on the *confidentiality* of data encrypted with TrueCrypt, an attacker could in principle also wish to compromise the integrity of the protected data. An attacker could alter the ciphertext of data encrypted with TrueCrypt, whereby the real goal is to alter the program flow in a system encrypted with TrueCrypt. For example, an (uncontrolled) alteration to an encrypted configuration file could be used by the attacker to make certain access controls in the IT system ineffective.

However, it should be noted that even if this attack strategy appears to be possible in theory, the associated effort required compared to the probability of success and the principal benefits remains questionable. Firstly, an attacker must know the location of the data or files to be compromised within the encrypted TrueCrypt volume or container. This assessment is associated with a not insignificant level of uncertainty, requiring more extensive alterations to the ciphertext and thus making the detection of the change more probable. Furthermore, this attack scenario assumes that the attacker has physical access to the system or comprehensive logical access. This then raises the question of why an attacker would decide on such an uncertain, partially uncontrollable and conspicuous attack strategy. It would be easier in these circumstances to attack the unencrypted data, achieving the desired malfunction in the system in an easier and more targeted way.

7.4.4 Preliminary conclusions

The analysis of the conceivable attack strategies raises the question of which countermeasures could be taken in TrueCrypt to combat which of these described attacks that would disable or hinder this kind of attack strategy. In conclusion, it can be firstly stated that TrueCrypt cannot as a matter of principle implement any effective measures against attacks in repeated-access scenarios. As Türpe et al. have already demonstrated, it is hardly possible to deliver this type of protection even on systems with TPM-based secure boot mechanisms [23]; in the case of Microsoft Bitlocker Drive Encryption, for example, there is no effective protection. For technical reasons, TrueCrypt cannot deliver a higher level of security in this area.

The second important conclusion is that should an attacker with above-average technical expertise gain access to an *active* IT =system running TrueCrypt decryption, the risk of a successful extraction of key or plaintext information should always be considered high. This risk is system and situation imminent – at least the required key for enabling the decryption is stored in the volatile memory. The software architecture used for TrueCrypt can at best provide flanking measures for protecting this key, such as e. g. protecting against side-channel attacks or the »economical« storage of passwords and key material in the volatile memory. In contrast, there are, however, a broad range of attack strategies, such as cold boot attacks and other attacks on the hardware, or other methods for exploiting possible vulnerabilities in the operating system,

network services and other software running on the system. Whether these types of attack are successful or not does not primarily depend on the security mechanisms in TrueCrypt but on the overall state of the IT system and the expertise and material capabilities of the attacker. Viewed conservatively, it can be concluded that an active IT system running TrueCrypt decryption must be considered per se as easy to compromise.

Attacks involving single access to inactive IT systems with TrueCrypt are also a relevant part of the discussion. In particular, those vulnerabilities in TrueCrypt in which information about the keys and passwords used by the system are written in plaintext on non-volatile data storage devices or in which the encryption process itself exhibits algorithmic weaknesses could enable attacks. It is even possible here that attackers who have previously had write access to TrueCrypt's source code could have introduced vulnerabilities to this source code in a targeted manner as »backdoors«. Once a vulnerability exists, it can be exploited by the attacker themselves through the (temporary) theft of the data storage device and subsequent forensic analysis.

7.4.5 Comparison with the security model used by TrueCrypt

The security model used by the TrueCrypt developers envisages a very narrow scope of application for the software [9, Page 83ff.]: TrueCrypt encrypts data before writing it to a non-volatile data storage device and then decrypts it again after it has been read.

In addition, the developers explicitly point out that TrueCrypt does *not* generally possess the following characteristics and functions:

- Protection of data on an IT system that an attacker can manipulate or control in some manner, or which the attacker can monitor
- Protection of data on an IT system to which an attacker has gained access before, during or directly after TrueCrypt has been executed
- Protection of the integrity or authenticity of data
- Encryption of data in volatile memory
- Protection of information about changes to data on encrypted data storage devices or in encrypted volumes.
- Protection of data flows outside of the TrueCrypt encryption, e.g. the transmission of data over computer networks.

If we take this security model as a basis, the developers have already categorically excluded the provision of protection against many of the attack strategies mentioned in 7.4.3. In particular, this also includes cases of repeated access to systems, single access to systems with active TrueCrypt decryption, attacks with logical access to IT systems and cyber espionage attacks. This analysis is consistent with the conclusion drawn in the preliminary conclusions in 7.4.4 that these attacks represent a major challenge for system security in general and it is hardly possible to mitigate them with encryption software. Overall, the security model appears to have been conservatively designed with relatively few assurances and lots of references to residual security risks.

In particular, the focus of the security model is placed on the unintentional saving of keys, passwords and plaintext files in unencrypted areas of the data storage device. These unintentional »outflows« of plaintext data are especially problematic if an attacker has the opportunity to forensically analyze a data storage device containing data encrypted by TrueCrypt. The developers refer here to security-relevant features of operating systems (page files, sleep mode, etc.) and special hardware features (wear leveling, physically moved sectors, deletion of volatile memory, etc.).

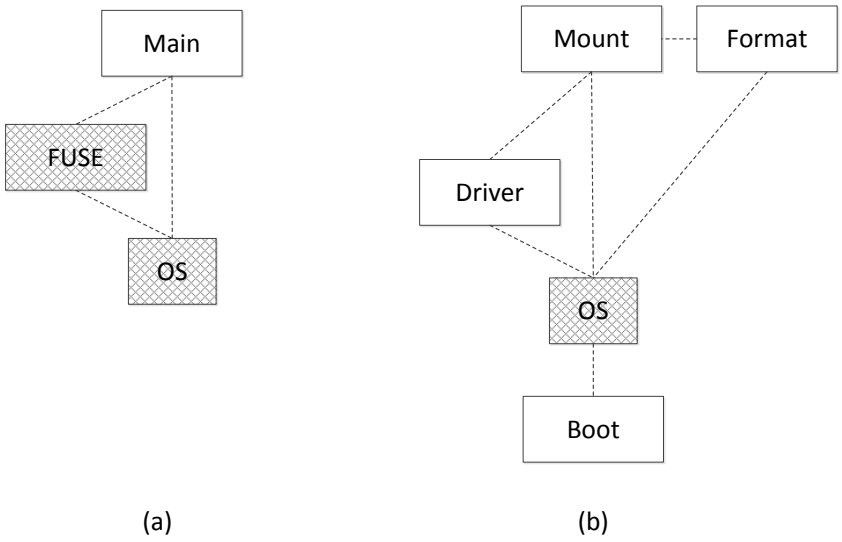


Figure 7.2: Simplified illustration of the runtime components of TrueCrypt; (a) shows the components for a Linux-based system, (b) for a Windows system

7.5 Evaluation of the architecture

In this subchapter, we discuss the suitability of the TrueCrypt architecture for addressing the described attack scenarios. We will firstly provide an overview of the runtime components in TrueCrypt. Afterwards, we will explain how these components interact with one another to implement the functionalities described at the beginning of this chapter in the use cases. We will evaluate here how the requirements R1-R7 are fulfilled and focus on the relevant attack scenarios. Finally, we will consider the topics of maintainability and testability and offer recommendations for improvements in a partial reimplementation.

7.5.1 An overview of the components

In the following section, we will describe the different runtime components that make up TrueCrypt and explain what basic role they fulfill within the overall system. This architectural perspective – which describes the system during runtime – is particularly important in the context of evaluating attack scenarios and the fulfillment of security goals because all of the attacks considered here are carried out when the system is running (or if switched off, it at least remains capable of being run).

We will not provide a detailed model of the structure of the source code. In principle, there are also attack scenarios that target the integrity of the source code e.g. integrating a backdoor into the code that is designed to be as difficult to detect as possible. However, we will not focus here on these scenarios in more detail and the structure of the source code is thus not of central importance at this point in time.

Figure 7.2 shows an overview of the runtime components of TrueCrypt. Depending on the target platform, TrueCrypt is made up of different components. On the one hand, the reason for this is that the functionality offered by TrueCrypt differs across the various platforms, whereby the largest range of functions is available on Windows systems. On the other hand, platform-specific features may require the runtime components to be organized differently.

On Linux-based systems, TrueCrypt only consists of one core component that interacts with the other system components:

Main We describe the main component on Linux-based systems as »Main« because it is also named as such in the source code. It is an executable file that implements both a command line interface and also a graphical user interface. Every direct user interaction with TrueCrypt takes place through one of these interfaces. Any functionalities that are offered by TrueCrypt on this platform are implemented in this component. Main utilizes, on the one hand, operating system functions (e.g. for displaying the user interface) and, on the other hand, FUSE (»Filesystem in Userspace«) for this purpose. FUSE is a kernel module and is used to mount encrypted volumes as virtual drives. FUSE is an independent software that is not part of the TrueCrypt project. It is also used for similar purposes by a large number of other projects on Linux-based systems.

TrueCrypt consists of the following four components on Windows systems:

Format Format is an executable file that can be directly controlled by the user via the command line or graphical user interface. The program is basically a wizard that helps the user to create volumes and/or initiate a system encryption.

Mount Alongside Format, Mount is the second executable file with which the user can interact with the program through a text-based or graphical interface. As the name suggests, this program is used to mount already created volumes as virtual drives. In order to implement this functionality, Mount utilizes the TrueCrypt component Driver. In the graphical user interface, Mount also offers users the option of creating volumes and starting a system encryption, although Format is then directly started for this purpose. This functionality is thus not implemented again within Mount.

Driver The Driver is a kernel driver and a central component of TrueCrypt on Windows systems. On the one hand, it is used to mount encrypted volumes as virtual drives, whereby in principle it takes over the functionality of FUSE in this usage context. In addition, it can also be integrated into the functions of the operating system as an »intermediary« component to such an extent that all write and read accesses to the data storage device are initially routed through it. As a result, the Driver can carry out »on-the-fly« encryption and decryption.

Boot The Boot component consists of a boot loader and implements what would generally be described as »pre-boot authentication«. Boot is only used if a system encryption has been carried out because an encrypted system partition is by implication not bootable. When the computer is started, the component asks the user to enter the password that is required to access the encrypted content. At this early phase of the start-up process, Boot is then responsible for the »on-the-fly« encryption and decryption so that the original boot loader can be loaded and control transferred to it. As soon as the operating system has been started, Driver takes over responsibility for the task of transparent encryption and decryption and Boot no longer fulfills any role.

7.5.2 The core functionality of TrueCrypt

We will describe below how the three core functionalities of TrueCrypt are technically implemented. Then we will evaluate their implementation with regards to the requirements R1-R7. Finally, we will discuss the suitability of the architecture with respect to the relevant attack scenarios.

Encrypted volumes The first step in the use of an encrypted volume is its creation. The user utilizes the TrueCrypt component Format on Windows systems for this purpose. Format is built like a wizard and initially asks the user to enter all of the information required for creating a volume step by step. Format then creates and saves the desired volume. As already explained in section 7.2, this could be a file-hosted or a partition/device-hosted volume. The data structure used for creating volumes is documented in the TrueCrypt User's Guide and is generally the same for both file-hosted and other type of volumes.

In order to access content in an already created volume, the user utilizes the TrueCrypt component Mount. Mount asks the user to enter the required data (e.g. path to the volume, password, password files, etc.) and passes on this information with a command to the Driver. The Driver creates a virtual drive and redirects all write and read accesses to this drive to the volume. During this »redirection process«, the Driver carries out on-the-fly encryption and decryption.

The redirection process is carried out as follows: In regular operation, applications use the so-called Win32 subsystem. The subsystem provides a comprehensive API (including Kernel32.dll, User32.dll, etc.) for standard operations, such as reading and writing data to a data storage device. This type of input/output operation is passed from the Windows subsystem to another Windows system component which then processes the request – the I/O manager. The I/O manager initially creates and initializes a so-called input/output request packet (IRP), which contains all of the information about the desired write/read operation. This IRP is then sent by the I/O manager to the device stack responsible for the respective data storage device. The device stack contains all of the drivers (e.g. bus driver, file system driver, etc.) that are required to talk to the device. In the case of a virtual drive created by TrueCrypt, the TrueCrypt kernel driver (Driver) is part of this stack and thus also receives the IRPs that describe a write and read process. This is precisely the stage at which the Driver »steps in«, carries out the encryption and decryption and, in the event of a write process, creates a non-volatile copy of the new data in the volume. It is thus irrelevant to applications whether the data is read/written from a TrueCrypt volume or from an unencrypted data storage device because the API for the subsytem is always addressed in the same manner.

System encryption The encryption of a system disk or a system partition brings with it a special feature: The encrypted system cannot simply be booted. In order for a TrueCrypt encrypted system to be booted at all, it requires the TrueCrypt component Boot that was installed as part of the initial system encryption by TrueCrypt. The TrueCrypt boot loader is started instead of the original boot loader and firstly asks the user to enter their secret password. The password is used by the Boot component to read the secret key in the volume header that is required to access data on the encrypted system. This component than executes a so-called interrupt hook to redirect subsequent read accesses to the encrypted data through a decryption routine. In detail, the process works as follows: Shortly after the system has been started, it is in so-called real mode. In real mode, the boot loader and any code loaded afterwards use so-called BIOS interrupts for carrying out basic input and output operations. Interrupt 13h (in short: INT 13h) is particularly interesting at this point for our analysis because this interrupt is used for the read and write operations on the hard disk. The TrueCrypt component Boot now uses an interrupt hook to manipulate the functionality of INT 13h so that read and write operations are processed through an encryption and decryption routine. Once this has occurred, TrueCrypt passes on control to the original boot loader. The original boot loader can now use INT 13h, to load the actual encrypted core components of the system and then pass on control. As soon as the operating system has taken over control, the read and write accesses are no longer processed through INT 13h but handled instead by kernel drivers. The TrueCrypt

component Driver takes over the on-the-fly encryption and decryption at this point in the same way as has already been described for encrypted volumes.

Hidden volumes Every volume essentially consists of an encrypted header and encrypted user data. In all cases, a header reserves dedicated space for a second header, which can belong to a hidden volume. If a TrueCrypt volume is not being used as an outer volume for a second hidden volume then this dedicated space for the second header is filled with random numbers. In this case, the unused memory in this volume is also filled with random numbers. The situation is different when a volume is actually acting as an outer volume. In this case, the dedicated space for the second header is actually the encrypted header for the hidden volume. The encrypted user data in the hidden volume is then located in the unused memory for the outer volume. It is not possible to read either the header or the user data for the hidden volume without the corresponding password. An attacker cannot actually tell the difference in this situation between the encrypted data and the random data; the existence of the hidden volume can thus be plausibly denied.

In general, TrueCrypt behaves as follows: If a user attempts to access a volume, TrueCrypt firstly requests a secret password. This process is handled by the TrueCrypt components Boot or Mount, depending on whether a system encryption has been carried out or not. The relevant TrueCrypt component (Boot or Driver) then firstly uses the password entered by the user to attempt to read the outer header. If this is successful then the contents of this outer volume are made accessible. If this process fails, TrueCrypt then uses the password entered by the user in the next step to attempt to read a possible second header. If this process also fails, TrueCrypt notifies the user that the password is incorrect. However, if this process succeeds, the system is now certain that this is an attempt to access an actually existing hidden volume and the user data on this hidden volume is made accessible. From a technical standpoint, access to the user data on the hidden volume is achieved in almost precisely the same way as access to the outer volume. In the case of a hidden operating system, the process is very similar to that described in the section »System encryption«, while in the case of a hidden volume, the process is as described in the section »Encrypted volumes«.

Following this explanation of the basic technical implementation of TrueCrypt, we will now examine the extent to which the requirements R1-R7 are fulfilled:

R1 - »Use of suitable cryptographic processes«: TrueCrypt uses recognized cryptographic methods for protecting the data in a volume. The AES, Serpent and Twofish algorithms are thus available as block ciphers. XTS is used as the operating mode. We thus consider this requirement to be fulfilled.

R2 - »Confidential data must not be saved as plaintext«: An encrypted volume is initially empty when it is first created. All data that is subsequently written to the volume is encrypted as part of the output operation from the component Driver. As part of the initialization of a system encryption, the whole system (depending on the chosen option, either a system partition or a system disk) is encrypted on the data storage device. In any event, the Driver only decrypts the encrypted data during a read operation in RAM and passes it on to the application that sent the request. TrueCrypt thus does not make a non-volatile copy of the data as plaintext in regular operation. The Driver is executed on the user's system and does not make use of any services that are external to the system. Therefore, we consider this requirement to be fulfilled.

R3 - »Ensuring the integrity of the protected data«: The encryption ensures that an attacker can not arbitrarily manipulate data so that certain plaintext is created after it is decrypted.

Yet there is no process for ensuring the integrity of the data that would indicate an unauthorized manipulation of a volume. We thus consider this requirement to be partially fulfilled. Nevertheless, ensuring the integrity of the data has a lower priority in this context than protecting the confidentiality of the data.

R4 - »Ensuring the integrity of TrueCrypt itself«: The officially published binary files for Mount, Format and Driver are digitally signed and thus their integrity can be easily checked by the user. Some Windows versions also require the kernel driver to be signed by default for them to be used at all. However, we are not aware of the publication of a TrueCrypt boot loader that can be used in combination with Secure Boot. Secure Boot is a technical measure that is being used to an increasing extent and which guarantees that only signed boot loaders can be used when the system is started. This makes it more difficult for attackers who want to manipulate the boot loader. This requirement is thus not completely fulfilled.

R5 - »Information about hidden volumes must not be saved«: In accordance with the specifications in the TrueCrypt User's Guide, every volume contains a header with metadata. This header must have dedicated space reserved for the possible existence of a second encrypted header. Without the corresponding password, it is thus not possible for an attacker to reliably decide whether the reserved space is only filled with random numbers or with an actual header. The same is true for the encrypted user data in the hidden volume: This is either found in the unused space of the outer volume or this space only contains random numbers. An attacker is also not able to differentiate between these two possibilities. As there is no further information pointing to the existence of a second volume, we consider this requirement to be fulfilled.

R6 - »Keys or passwords must not be saved as plaintext«: There is no functionality that would require keys or passwords to be saved as plaintext. We could not find any indications that TrueCrypt is doing anything similar. The requirement is fulfilled.

R7 - »Keys or passwords must not be transferred to third parties«: On the basis of the uses cases and the functional description, there is no need for keys or passwords to be transferred to third parties. We could not find any indications that TrueCrypt is doing anything of this sort. The requirement is fulfilled.

At this stage, we will now consider the suitability of the basic technical implementation of TrueCrypt with respect to the described attack scenarios. Firstly, it should be noted that an encrypted volume opened by TrueCrypt on a running system is essentially unprotected. This is true for all applications and functionalities. As a general rule, the opened volume will look like a regular drive to an attacker who has logical access. Therefore, it can be easily read or manipulated. The fact that TrueCrypt does not make a non-volatile copy of the encrypted content of a volume in plaintext is also true for all functionalities. Even in the event of a sudden system crash, the data is not present in an unencrypted form, not even if the volume was open at the time of the crash.

If the system is switched off or the volume is closed then TrueCrypt actually provides effective protection. However, the functionalities offered by TrueCrypt and the use cases in which the these functionalities are used are more or less susceptible to certain attack scenarios.

An attacker has the largest range of options to select from in the case of a file-hosted or partition/device-hosted volume. This includes the following:

- The simplest way to analyze and manipulate the user's system is through a single logical or physical access. For example, an attacker can install malware (such as a keylogger or remote access software) to monitor the future input of passwords or to directly access the data on an opened volume remotely.

- The use of keyfiles can make brute force attacks significantly more difficult. In this area, TrueCrypt offers users the option of linking a number of keyfiles to a volume. This process can be carried out when the volume is created or also for already existing volumes. When mounting this kind of volume, it is then no longer sufficient to simply enter the correct password, the user must also enter the keyfiles linked to the volume. TrueCrypt then uses the content of the keyfiles and password entered by the user to calculate the secret key for the volume header. If either the password entered by the user or the stated keyfiles do not correspond to those linked to the volume by the authorized user, the process for generating the key will fail and access to the volume remains blocked. This increases the level of difficulty for brute force attacks because not only does the password need to be correctly guessed but also the content of the keyfiles used in each case. However, it is important to note for this analysis that TrueCrypt only processes a maximum of the first 1,048,576 bytes ($1024 \cdot 1024$ bytes corresponding to 1MB) in each keyfile for generating the key. If a keyfile is larger, the additional data is ignored. Keyfiles can be stored on external data storage devices such as USB sticks or also on smartcards. The number of keyfiles that can be linked to a single volume is practically unlimited.

- Ensuring the integrity of TrueCrypt itself can be beneficial in some situations. For example: TrueCrypt is used to save an encrypted backup to an external data storage device. In order to make access easier, a portable version of TrueCrypt is saved to the backup device. It is then easy to check the integrity of the TrueCrypt binary files before the volume is accessed. It would be difficult here for an attacker to falsify the signature.

- The use of hidden volumes is also associated with plausible deniability. It is thus more difficult for an attacker to gain access to the encrypted data by threatening the user.

From the perspective of the attacker, the encryption of a system partition needs to be considered a little differently than a regular volume.

- As a general rule, analyzing and manipulating the system requires more effort if the system partition is completely encrypted. Therefore, the possibilities for installing malicious software are limited to some extent.

- Nevertheless, any unencrypted partitions that may exist are still accessible to the attacker and could be used for a forensic analysis. It is possible that exploitable data has been saved here by applications or the operating system that could make further attacks easier. Indeed, it is even possible that the user themselves has inadvertently saved data in unencrypted areas.

- As Secure Boot is not supported, the boot component represents a vulnerability. As has been demonstrated in practice (e. g. Stoned), an attacker can infect this unencrypted component whose integrity is not protected and thus attack the system encryption.

- No keyfiles can be used here and it is thus not possible to make brute force attacks more difficult.

The encryption of the entire system disk imposes the greatest limitations on an attacker in terms of those functionalities being considered here. In contrast to the encryption of the system partition, there are no longer any unencrypted partitions that could provide an attacker with useful data. In the special case of a hidden operating system, the fact that its existence cannot be proved provides additional protection.

Some attack scenarios are nevertheless still possible. For example, the integrity of the hardware is not protected by TrueCrypt. An attacker could, for example, install a hardware keylogger or another monitoring tool within the system in order to gather confidential information via cyber

espionage. In an extreme case, an attacker could even replace the user's entire hardware with a manipulated copy.

In summary, it can be said that the functionalities offered by TrueCrypt permit a whole series of attack scenarios. The basic technical implementation of TrueCrypt is nevertheless appropriate because it does not introduce any serious, avoidable vulnerabilities.

7.5.3 Maintainability and testability

Unit tests should play an important role in the test and development process for TrueCrypt, as well as for any software based on it. There are a series of frameworks available, such as CppUnit[1] or Google Test[2], which can be used to help develop corresponding test cases for C++ code. A comprehensive test suite ensures that certain sections of the code conform to expectations. As part of the further development of any software, they serve as regression tests to prevent new errors being introduced into already existing code when changes are made to the source code.

Microsoft provides documentation and tools[3] that are especially useful for testing drivers on Windows. Alongside comprehensive support in current editions of Visual Studio, designed to simplify the development of drivers, there is also, for example, the Static Driver Verifier (SDV)[4]. The SDV analyzes the driver's source code to identify errors and design issues that could threaten a flawless interaction with the Windows kernel. This could enable serious problems to be found at an early stage of the development process. Alongside specialist solutions for drivers, there are also a series of other tools – generally available commercially – that can find software errors using static program analysis. The capabilities of these products can sometimes vary considerably and introducing this type of analysis software into the development process should thus only be carried out on the basis of a careful examination.

In addition to testing methods based on the source code, a testing method called fuzzing can also be employed. Fuzzing is a technique where the interfaces to the software are tested while it is running with a very high number of random inputs in order to identify any input values that could possibly cause an error in the software critical to its security. This method can be used to identify security issues that cannot be identified with static analyses and which were not taken into consideration when the unit tests were created. In particular, it would be expedient to analyze the interfaces to the TrueCrypt driver component because it generally implements the core functionality of the program. The authors of the OCAP Phase 1 Test Report state that they have tested some interfaces to the driver and the boot loader using fuzzing.

Another elaborate method for verifying certain characteristics of a tested software is formal verification. By applying a mathematical verification method it can be proven whether part of the software conforms to a given formal specification. As this verification generally requires a great deal of effort, the process could not be used on the entire software for TrueCrypt but would need to be limited to particularly critical parts of the system. The authors of the OCAP Phase 2 Test Report recommend verifying the functions `EncryptBufferXTSNonParallel` and `DecryptBufferXTSParallel` in this way because they believe there is potential for error prone »pointer arithmetic« and »bounds checking«.

In order to increase the testability and maintainability of TrueCrypt and other software based on it, deficiencies in the code quality should firstly be resolved and the build infrastructure updated. Last but not least, the maintainability of software is also closely linked to a clear and well-documented software design. Well-defined interfaces and a modular software structure also simplify testing. A modern build infrastructure is required in order to be able to utilize up-to-date analysis software and development tools. Some of these products are integrated into

[1] http://sourceforge.net/projects/cppunit/
[2] https://code.google.com/p/googletest/
[3] https://msdn.microsoft.com/en-us/library/windows/hardware/ff554651%28v=vs.85%29.aspx
[4] https://msdn.microsoft.com/en-us/library/windows/hardware/ff552808%28v=vs.85%29.aspx

the build process for analysis purposes and the use of very old software could cause problems.

7.5.4 Recommendations for improvements

The need to resolve deficiencies in the code quality and to update the build infrastructure has already been emphasized on a number of occasions. Alongside this clear recommendation, there are also a series of other proposals for improvements designed to improve the security and usability of the software.

Alternatives to XTS-AES Firstly, the operating mode XTS-AES, which is used for encrypting volumes, has been generally criticized. As discussed in Chapter 9, the authors of the OCAP Phase 2 Test Report point to possible attack scenarios associated with the insufficient protection of integrity and the small block size (16 bytes) in AES. As a result, an attacker can manipulate a single cipher block on a data storage device for the purpose of altering a particular memory location for their own purposes, without this attracting the attention of the user. Although the attacker cannot predict what effect this manipulation of the ciphertext will have on the plaintext without knowledge of the key, they can certainly guarantee that only the actually manipulated block is changed. An attacker can thus take advantage of this fact by overwriting certain parts of an executable file in order to alter its control flow for their own benefit. Manipulation of the configuration files and the Windows registry is also possible. In order to carry out this type of attack, the attacker must have knowledge of a memory location that would be beneficial if manipulated but would not damage the functionality of the system in a conspicuous manner. The authors of the test report highlight the possibility of utilizing an alternative solution to combat this problem – which was originally developed by Microsoft for Bitlocker and is discussed in [8]. The publication discusses in detail the use of AES-CBC in combination with a so-called diffuser. The diffuser combines the plaintext of multiple AES blocks together in one large block. Before encryption and also after decryption, the diffuser applies a mathematical operation to the plaintext to distribute local changes across the entire combined block. The block size of the diffuser is significantly larger than that in AES and can vary in the range of 512-8192 bytes. The use of the diffuser ensures that any manipulation of the ciphertext has significantly greater effects on the plaintext than would be the case when using XTS-AES. Even the minimum block size of 512 bytes is considerably larger than the 16 bytes in XTS-AES and thus significantly more difficult to attack. However, the authors clearly state in [8] that the use of the described method should firstly be carefully examined and is not necessarily suitable for all use cases. Before a similar method is used as part of the further development of TrueCrypt, a detailed analysis of its effects is urgently required.

A possible alternative approach for guaranteeing the integrity of the plaintext could be to use a special file system that already possesses the necessary functionality. Btrfs is one example of a file system that assigns a checksum to all data and metadata to ensure the integrity of the saved data. In the context of TrueCrypt, even when using XTA-AES, the unauthorized manipulation of the ciphertext would become apparent because the checksum calculated by the file system would no longer be correct. An attacker without knowledge of the key would have no opportunity to correspondingly adapt the checksum – which like the data is itself encrypted – to the manipulation.

Support for multiple users Especially when TrueCrypt is used in an official or company setting, support for multiple users per volume makes good sense. TrueCrypt does not currently provide any direct option for linking multiple individual users to a single volume. This type of functionality could be implemented, for example, by adapting the volume header format so that the secret key for the volume is not just added once but multiple times. Each copy of the key could be encrypted with the password of a different user so that each user could access the

volume using their own individual password. The specific requirements for supporting multiple users is highly dependent on the use case. Therefore, there is no general solution that fulfills the requirements for every case. As explained in the TrueCrypt User's Guide, a similar functionality can be achieved by using smartcards in combination with keyfiles.

Removing the hidden volume functionality In a professional setting, it may be preferable for the employer to remove the option of creating hidden volumes. This would make it more difficult for users to save secret data on a work computer that would even be hidden from the employer themselves. Even if the employer has complete trust in the user, the removal of this functionality would contribute to reducing the software's code base. A small code base is also linked to less testing and maintenance work and could also possibly result in a smaller attack surface. The removal of unnecessary functionalities can thus be useful and should be investigated for specific usage scenarios.

7.6 Summary

Results of chapter 7

- There are three main use cases for TrueCrypt:
  - Protecting information using system encryption
  - Protecting information using encrypted volumes
  - Maintaining confidentiality and protecting information using hidden volumes
- The primary aim is to protect the confidentiality of data in a TrueCrypt volume.
- Alongside the primary security goal, there are other goals such as protecting integrity or enabling deniability.
- The attacker's opportunities for accessing the system can be split into two dimensions:
  - physical/logical access
  - Single/repeated access
- There are no effective measures against attackers that have repeated access.
- An open volume on a running system is easy to compromise.
- The basic technical implementation of TrueCrypt is appropriate because it does not introduce any serious, avoidable vulnerabilities.

8 Identification of dispensable parts of the code

8.1 Aim

The aim of this work package was to investigate the code base in order to identify components, files and functions that are not required for the desired functionality and thus could possibly be removed to reduce the attack surface. After consultation with the client, all possible usage scenarios offered by TrueCrypt in practice should be taken into account. In this respect, a function can only be considered to be dispensable if it is not used in any possible configuration of the software.

8.2 Procedure

Component level In order to identify dispensable parts of the code at a component level, it is helpful to fall back on the description of the software architecture in Chapter 7.5.1. This provides a detailed description of the runtime components and their dependencies.

File level This work package focused at a file level on the comparison of the log files with the Windows and Linux build directories to determine whether every `.c` and `.cpp` file was actually used when compiling the program. In the case of a Linux build, the source code is compiled in files with the ending `.o` and for a Windows build it is compiled in `.obj` files. If neither a `.o` nor a `.obj` file exists for a source code file, it can be concluded that the corresponding source code file is not used in the build for that platform.

Function level In order to determine which functions are used in practice, the call graph from Chapter 4 was utilized on the one hand, while the source code was also investigated manually for call dependencies on the other hand. Further information was provided by the results of the static code scanner from Chapter 5, which have additional checkers for unused functions, branches and variables.

8.3 Results

Component level In terms of the components, the description of the software architecture from Chapter 7.5.1 suggests that all components are required in a general usage scenario (see Figure 7.2). If the usage scenario is restricted, complete components could be omitted. It is possible, for example, to leave out the `Boot` components on Windows if the user has no interest in an encryption of the complete system.

File level Only the files for the user interface in the Linux variant (mainly found in folder `Main/Forms`) were not required by the compilation. However, this can be explained by the fact that we explicitly deactivated the user interface during the build process and the analysis is based on the command line variant of TrueCrypt. Furthermore, the file `Crypto/Aescrypt.c` was also not compiled. This is used for architectures other than x86 that are not covered by this

analysis. Overall, it can be concluded that if there is no restriction of the usage scenario then all of the files are required at a file level.

Function level In general, no large fragments of code were found that are not used in TrueCrypt, although not all of the unused sections of code have been consequently eliminated. For example, the functions `aes_encrypt_key`, `aes_encrypt_key128`, `aes_encrypt_key192` can be activated by setting a preprocessor variable, yet there are no circumstances in which they are called.

The tools Clang, Coverity Scan and Cppcheck (described in Chapter 5) that are used for carrying out static code analyses only returned a few indications of unused fragments of code. Coverity and Clang reported a total of three unused variables in the code; these should definitely be deleted, but furthermore do not point to large unnecessary sections of the code.

Other recommendations In the case of defined usage strategies for TrueCrypt, it is possible to make detailed statements about dispensable sections of the code. The tool `gcov` can be utilized, for example, on Linux for this purpose. However, this is a dynamic tool that delivers statistics about used and unused calls in the code during the runtime of the program. This requires that the desired operations e.g. encrypting and decrypting a partition using TrueCrypt are carried out. The statistics can then be used afterwards to argue which sections of the code are not significant for the functionality executed by the program.

However, it is important during this process to be aware that the tool logs information dynamically, meaning during runtime. Those execution paths that are not reported are thus not necessarily dispensable. For example, paths that are followed in the event of errors may not have been taken into account because the program has possibly followed a different execution path during runtime.

8.4 Summary

Results of chapter 8

- In the case of a general usage scenario for TrueCrypt, all components and files are required to execute TrueCrypt.
- A further restriction of the usage scenarios would make it possible under certain circumstances to dispense with complete components.
- At a functional level, some functions could be identified in the subproject`Crypto` that are not executed.

9 Evaluation of the OCAP Phase 2 Test Report

9.1 Comments on OCAP-2

As part of the Open Crypto Audit Project[1], the second phase of the audit of TrueCrypt in Version 7.1a was completed. The report by Alex Balducci, Sean Devlin and Tom Ritter was published in March 2015 [2]. In this chapter we evaluate the scope of the report, discuss its findings and submit recommendations for software based on TrueCrypt.

Scope of the report

The second phase of the OCAP focused on the cryptographic services of the TrueCrypt software. This included a comprehensive source code review and specific debugging of the software on a Windows platform. Reverse engineering of the assembler code or a comparison between the TrueCrypt binary files and the source code were not carried out. The focus of the source code review was the AES implementation in XTS mode, as well as the random number generators and the SHA-512 hash function. In addition, the *header volume format* was analyzed. In particular, it is notable that there was no analysis of the non-secure deletion of memory.

This analysis revealed 4 vulnerabilities. However, none of them would allow a general attack on any TrueCrypt installations. Two vulnerabilities were categorized as having a high level of severity, one as having a low level of severity and one with an undetermined level of severity. The difficulty of exploiting these vulnerabilities was categorized as undetermined or high in all cases.

9.2 Comments on the results

We will discuss the findings of the OCAP Report in the following section.

Finding 1 – `CryptAcquireContext` may silently fail in unusual scenarios The first vulnerability is related to the fact that the random number generator in TrueCrypt on Windows relies in some scenarios on poor sources of entropy, without causing abortion of the program or at least issuing of a warning message. [2] The relevant source code is located in the file `Common/Random.c` in lines 101-105, 645 and 756.

The vulnerability relates to the fact that calls to `CryptAcquireContext`[3] can fail, if, for example, a Windows group policy has been set that prevents access to the key store of the Cryptographic Service Provider. TrueCrypt exclusively uses this context for generating random numbers via `CryptGenRandom` and thus no access is actually required.

If no context for the Cryptographic Service Provider is available, TrueCrypt uses its own sources of entropy instead, such as the process number or the current runtime of the system. A list can be found in Annex A of the OCAP-2 Report, as well as in the file `Common/Random.c` in lines 614 and 639 or 667-752 via the functions `RandaddBuf` and `RandaddInt32`.

[1] https://opencryptoaudit.org
[2] In this sense, the title of this vulnerability is unfortunately very technically focused and not target-oriented.
[3] See https://msdn.microsoft.com/en-us/library/windows/desktop/aa379886.aspx

The OCAP-2 Report recommends setting of `CRYPT_VERIFYCONTEXT`[4] in the 5th parameter. The result is that no access to the key storage is requested when initializing the Cryptographic Service Provider context. Therefore, `CryptAcquireContext` would also not fail even if access to the key storage were prohibited.

In addition, in case of an error where not context is returned, an error message should be issued and execution aborted.

This is similar to the situation with the Debian-OpenSSL-Bug[5], whereby more sources of entropy are actually used. However, in scenarios involving virtual machines, embedded systems and the automated booting of devices, these sources of entropy (used by TrueCrypt) can be considered almost equally vulnerable as the Debian-OpenSSL-Bug. Accordingly, software based on these sources should urgently implement the sensible recommendations in the OCAP-2 Report and replace old keys that were generated in these types of scenarios.

The implementation of the random number generator in Linux in the file `Core/RandomNumberGenerator.cpp` can also lead to vulnerabilities in the entropy, as described in Chapter 4.3.

Finding 2 – AES implementation susceptible to cache timing attacks The second vulnerability describes the possibility that the AES key could be detected while it is being used through cyber espionage. The report states that the implementation in file `Crypto/AesSmall.c` uses an implementation that is susceptible to cache-timing attacks.

A cache-timing attack on a key exploits the fact that an AES implementation uses look-up tables and optimizes access to these tables in a naive manner. The attack will thus work as follows: an attack program is applied such that the cache lines in the CPU are shared with the victim program. The attack program then uses its scheduler slot to flood the cache with its data. In the next slot (after the victim program just executed), the attack program once again attempts to access its data. Due to the latency of the data access, it can now determine which cache lines were flushed. This information provides an access pattern for all programs that are running in parallel. These access patterns can be used to draw conclusions about possible keys in the AES implementation, which could reduce the potential search area and lead to exposure of the secret key.

In order to be able to carry out this type of attack, the attacker would, however, need access to the same computer and the opportunity to execute code. In this context, the OCAP-2 Report refers to advances in attack techniques, such as those using JavaScript. The report recommends the use of alternative implementations, such as that of Käsper, as well as the use of other mitigation strategies.

In general, this recommendation should certainly be followed. In particular, this would prepare the way for scenarios in which new projects are created based on the TrueCrypt source code, as well as enabling the existing code to still be used for other purposes. There should not generally be a problem when used within TrueCrypt because AesSmall.c is only used for the boot module (see Boot/Windows/Boot.vcproj and Boot/Windows/Makefile). This only represents a problem in a scenario where TrueCrypt carries out full disk encryption in a virtual machine and the attacker has access to another virtual machine. In this type of scenario, the level of difficulty increases even further, although cache timing attacks on the AES key in TrueCrypt would then be possible.

Finding 3 – Keyfile mixing is not cryptographically sound The algorithm for deriving a key from keyfiles (also in combination with a password) is not cryptographically secure. The problem lies in the fact that the method used by TrueCrypt – which is based on CRC and ring

[4] https://msdn.microsoft.com/en-us/library/windows/desktop/aa379886.aspx#crypt_verifycontext
[5] See https://www.debian.org/security/2008/dsa-1571

addition – is not collision resistant in the sense of a cryptographically secure hash function. This means that it is possible for a given set of keyfiles (and/or passwords) to generate another keyfile (and/or password) that results in the keypool having a previously desired value for the calculation of the key. In particular, it is also possible – with knowledge about a keypool formed from a number of keyfiles (and/or password) – to generate an alternative keyfile (and also a password to some extent) that would generate the same key.

The OCAP-2 Report recommends the use of a cryptographically secure hash function and rates the severity of this hole as low.

However, the purpose of this function is to allow the possibility of multi-factor authentication and the four or six-eye principle. The possibility of negating this multi-factor characteristic both during and also after the generation of the keypool and the fact that it is not possible to prove this type of manipulation means that it should be given a medium rating and a medium level of difficulty.

The scenarios could occur as follows: A password and a keyfile (on a USB stick) are used to secure a key. The attacker manipulates the PC on which the keyfile for the password is generated. During the generation process, the keyfile is not randomly chosen but instead selected so that a second password exists that generates the same key without a keyfile. Another scenario could be that three keyfiles are generated for a six-eye principle, whereby two of the keyfiles are selected so that they negate each other. It is then subsequently possible to generate the AES key both in accordance with the six-eye principle as well as for the owner of the non-negated keyfile alone.

According to the OCAP-2 Report, a cryptographically secure hash function should be used. In order to retain the characteristic that keyfiles can be entered in any order in TrueCrypt, the keyfiles could be sorted according to their values before they are hashed. This would ensure that this change does not result in the loss of this feature.

In cases where it cannot be guaranteed that the creator of a keyfile has not manipulated it, or an attacker did not have at least temporary access to the keypool to generate a »MasterKeyfile«, all keys should be regenerated in accordance with this secure method. This is particularly important because this type of manipulation cannot be proven. Even in the case of self-negating keyfiles, it would instead be possible to generate ones that enable access with a password in combination with the third keyfile.

Finding 4 – Unauthenticated ciphertext in volume headers A common requirement in cryptography is to protect the confidentiality and also the integrity of some data using a symmetric key. It is common practice here to add a checksum in the form of a CRC or hash code within the encrypted memory block and to check it for plausibility – whether it is correct based on the actual data – after decryption. Due to the encryption of the memory block, it should not be possible in theory for an attacker to alter the actual data and also the checksum (as well as the magic string) so that the results then again appear plausible.

However, it is difficult to prove this in practice. Instead of the selected approach, it is common practice to secure the integrity of the data cryptographically without indirections. The OCAP-2 Report discusses this issue and makes a good recommendation that a second key – alongside the encryption key – should be derived from the main key and should then be used with a message authentication code (MAC) – this could be both a CMAC or a HMAC. In this way, the integrity of the data could be guaranteed in accordance with best practices in the field of cryptography.

Above and beyond this recommendation, it is also possible to complete the cryptographic encryption in modes for authenticated encryption. The modes CCM and GCM offer the possibility of guaranteeing both the confidentiality and also the integrity of the data in one step.

9.3 Further findings

XTS Mode The OCAP Phase 2 Test Report generally criticizes XTS as an operating mode for hard disk encryption. Consequently, the report believes that the integrity of the data is not guaranteed to a sufficient degree. As is the case with some other operating modes, protection against unauthorized alterations in XTS mode is only based on the fact that an attacker cannot alter the ciphertext in a way that would enable a defined and predictable manipulation of the plaintext without knowledge of the key. As unauthorized manipulations cannot be directly recognized, which would be the case if using a message authentication code (MAC), this form of protection is described in literature as »poor man's authentication«. The test report describes the problem as being that the block size (16 bytes) is very small in XTS-AES, which means that an attacker can target the manipulation of certain areas of the hard disk by manipulating the ciphertext. Although the concrete effects of this manipulation of the ciphertext on the plaintext are not foreseeable without knowledge of the key, it nevertheless guarantees that the alteration only affects the actually manipulated block and the other blocks remain unaffected. An attacker could take advantage of this fact by overwriting certain parts of an executable file in order to change its control flow for his/her own benefit. Manipulation of configuration files or the Windows registry is also possible. A prerequisite for this type of attack is that the attacker must have knowledge of a memory location useful for manipulation but would not damage the functionality of the system in a conspicuous manner. In Chapter 7.5.4 of this report, proposals are discussed for combating this problem.

Alongside this general criticism of XTS, the authors of the test report recommend the formal verification of selected functions in TrueCrypt (`EncryptBufferXTSNonParallel`, `DecryptBufferXTSParallel`) that are used as part of the encryption and decryption processes. In particular, this should be designed to exclude errors in the »pointer arithmetic« and »bounds checking«.

Further recommendations Further recommendations in the report include a continuation of the code review and improvements of the code. In particular, the pointer arithmetic and bounds checking in the XTS implementation on various platforms with different endianness seem to be significant here. The parameters for the encrypted areas and the overall size also appear to be possible trapdoors and should be examined throughout the entire source code – only `DriveFilter.c` was examined for this purpose in the OCAP-2 Report. Finally, the program flow should be specifically reviewed with a focus on how errors are handled and the implact of errors from certain function calls beyond the actual cryptographic functions.

Moreover, the program logic appears very complex and should be simplified. In particular, the diverse range of cryptographic functions and redundant implementations (see also Chapter 4 Figure 4.1) result in a high level of complexity. The report explicitly highlights hardware-optimized implementations as an exception. Another example of the increased level of complexity can be found in Appendix D of the report – which highlights the use of defensive coding practices. Cases such as the described fall-through and switch statements without defaults significantly increase the complexity for programmers and reviewers.

Furthermore, the report recommends better error handling and logging. The case involving the poor handling of errors in random number generators provides a very good example. If the program would have been terminated with a corresponding error message, the programmers would probably already have incorporated the improvements described under Finding 1. Logging would probably have provided the required information in the current implementation, while the termination of the program would have provided the motivation for finding a correct solution.

9.4 Summary

Results of chapter 9

- The random number generator in the Windows version is significantly weakened in case of certain group policy settings. This can go unnoticed and lead to poorly encrypted volumes in many systems. This should be resolved in the code and affected volumes regenerated.
- The implementation of the bootloader for full disk encryption is susceptible to cache timing attacks. However, this is only relevant when used for virtual machine booting and even then only with a lot of effort.
- The mix of multiple keyfiles and/or a password is not cryptographically secure against collision attacks. Consequentually, the system is incapable of actual multi-factor authentication and four-eye principles.
- The volume header in the ciphertext should be secured either by MAC or CCM/GCM instead of via CRC.
- The use of XTS-AES is associated with risks because an attacker can overwrite certain memory locations in a targeted manner without attracting the attention of the user.
- Simplifying the program logic, defensive coding and better error handling and output would make it easier for programmers and reviewers to guarantee security.
- Further reviews should follow, with a particular focus on the pointer arithmetic in the XTS implementation, the handling of header volume parameters and the program flow outside of the cryptographic functions.

Appendix A

Functions with a cyclomatic complexity greater than 15

```
======================================
  NLOC    CCN   token  PARAM  location
--------------------------------------
    1832    440    8579      4 MainDialogProc@5368-7770@./Format/Tcformat.c
    1536    416    8087      4 MainDialogProc@4637-6565@./Mount/Mount.c
    1542    368   12019      4 PageDialogProc@3292-5364@./Format/Tcformat.c
     604    138    3990      3 ProcessMainDeviceControlIrp@805-1553@./Driver/Ntdriver.c
     522    113    2956      5 TCOpenVolume@36-732@./Driver/Ntvol.c
     345    107    1760      1 TestSectorBufEncryption@637-1030@./Common/Tests.c
     306    103    1495      1 TestLegacySectorBufEncryption@1033-1363@./Common/Tests.c
     344    102    1898      0 EncryptionTest::TestXts@473-861@./Volume/EncryptionTest.cpp
     443    102    2374      1 TCFormatVolume@73-675@./Common/Format.c
     377     98    2751      2 CommandLineInterface::CommandLineInterface@20-483@./Main/CommandLineInterface.cpp
     368     87    2064      1 VolumeCreationWizard::ProcessPageChangeRequest@514-972@./Main/Forms/VolumeCreationWizard.cpp
     393     78    2491      4 PasswordChangeDlgProc@1381-1867@./Mount/Mount.c
     282     78    2128      1 TextUserInterface::CreateVolume@477-846@./Main/TextUserInterface.cpp
     332     74    1112      1 AfterWMInitTasks@8423-8914@./Format/Tcformat.c
     232     72    1399      9 MountVolume@5963-6263@./Common/Dlgcode.c
     245     70    1232      1 volTransformThreadFunction@2261-2576@./Format/Tcformat.c
     306     68    1478      2 RestoreVolumeHeader@7668-8074@./Mount/Mount.c
     291     67    1722      5 ReadVolumeHeader@163-575@./Common/Volumes.c
     324     63    1753      4 EncryptPartitionInPlaceResume@638-1102@./Format/InPlace.c
     241     59    1496      4 RawDevicesDlgProc@2869-3180@./Common/Dlgcode.c
     229     59    1123      5 ChangePwd@117-421@./Common/Password.c
     245     58    1550      1 SetupThreadProc@1150-1466@./Driver/DriveFilter.c
     266     57    1878      1 AnalyzeKernelMiniDump@8609-8940@./Mount/Mount.c
     165     56     927      2 MountAllDevices@3687-3915@./Mount/Mount.c
     212     55    1244      2 LoadPage@2578-2850@./Format/Tcformat.c
     120     55     706      2 TCDispatchQueueIRP@199-349@./Driver/Ntdriver.c
     155     54    1067      1 TCTranslateCode@1864-2024@./Driver/Ntdriver.c
     209     53    1638      1 MainThreadProc@482-756@./Driver/EncryptedIoQueue.c
     150     53    1183      4 PreferencesDlgProc@2190-2383@./Mount/Mount.c
     226     53    1300      2 ExtractCommandLine@6567-6845@./Mount/Mount.c
     225     51    1322      4 FavoriteVolumesDlgProc@101-407@./Mount/Favorites.cpp
     258     49    2002      4 VolumePropertiesDlgProc@2624-2968@./Mount/Mount.c
     268     49    2112      4 CipherTestDialogProc@5044-5400@./Common/Dlgcode.c
     287     48    1748      2 LoadDriveLetters@978-1334@./Mount/Mount.c
     229     47    1458      4 PasswordDlgProc@1876-2167@./Mount/Mount.c
      61     46     740      0 EncryptionTest::TestLegacyModes@42-113@./Volume/EncryptionTest.cpp
     161     45    1059      9 Volume::Open@100-308@./Volume/Volume.cpp
     162     45    1010      3 BackupVolumeHeader@7446-7665@./Mount/Mount.c
     115     44     697      3 Mount@3364-3530@./Mount/Mount.c
     415     44    1308      0 UserInterface::ProcessCommandLine@867-1323@./Main/UserInterface.cpp
     207     43    1220      4 HotkeysDlgProc@270-523@./Mount/Hotkeys.c
      54     43     393      1 Hotkey::GetVirtualKeyCodeString@62-127@./Main/Hotkey.cpp
      72     42     605      2 GetKeyName@42-120@./Mount/Hotkeys.c
     162     42    1230      1 ExceptionHandlerThread@1657-1855@./Common/Dlgcode.c
     157     42    1130      0 LoadLanguageFile@104-313@./Common/Language.c
     249     41    1815      3 ProcessVolumeDeviceControlIrp@493-802@./Driver/Ntdriver.c
     190     41    1200     16 CreateVolumeHeaderInMemory@685-983@./Common/Volumes.c
      62     40     549     11 StringFormatter::StringFormatter@15-83@./Main/StringFormatter.cpp
     181     40    1177      1 GraphicUserInterface::RestoreVolumeHeaders@1112-1359@./Main/GraphicUserInterface.cpp
     140     39     693      0 CheckMountList@4445-4632@./Mount/Mount.c
     145     38    1182      1 IoThreadProc@298-479@./Driver/EncryptedIoQueue.c
     164     37     602      1 SwitchWizardToSysEncMode@749-947@./Format/Tcformat.c
     191     37    1353      4 TravelerDlgProc@2971-3216@./Mount/Mount.c
     139     37    1160      3 CoreLinux::MountVolumeNative@290-473@./Core/Unix/Linux/CoreLinux.cpp
     108     37     575      1 TextUserInterface::MountVolume@1061-1194@./Main/TextUserInterface.cpp
     218     36    1051      3 EncryptPartitionInPlaceBegin@302-635@./Format/InPlace.c
      92     36     521      1 GraphicUserInterface::MountVolume@647-755@./Main/GraphicUserInterface.cpp
     205     35    1448      1 VolumeCreationWizard::GetPage@75-335@./Main/Forms/VolumeCreationWizard.cpp
     111     34     622      4 MountFavoriteVolumes@7143-7282@./Mount/Mount.c
      47     34     350      0 InitOSVersionInfo@2223-2276@./Common/Dlgcode.c
     161     34    1219      1 PerformBenchmark@4261-4504@./Common/Dlgcode.c
     134     34    1141      1 DoAutoTestAlgorithms@1366-1563@./Common/Tests.c
     152     34     745      1 CoreUnix::MountVolume@393-575@./Core/Unix/CoreUnix.cpp
     128     33     877      4 LanguageDlgProc@317-480@./Common/Language.c
      93     33     630      4 BootEncryption::CreateBootLoaderInMemory@961-1082@./Common/BootEncryption.cpp
     141     32    1067      4 MountOptionsDlgProc@2386-2573@./Mount/Mount.c
     136     32     752      4 PerformanceSettingsDlgProc@8083-8254@./Mount/Mount.c
     134     32     755      6 OpenVolume@8457-8637@./Common/Dlgcode.c
     118     32     753      4 GetAvailableHostDevices@9333-9483@./Common/Dlgcode.c
     156     32     986      1 TextUserInterface::RestoreVolumeHeaders@1263-1476@./Main/TextUserInterface.cpp
     138     31     643      2 CheckRequirementsForNonSysInPlaceEnc@88-299@./Format/InPlace.c
     141     31     685      0 RepairMenu@842-1025@./Boot/Windows/BootMain.cpp
     145     31    1074      5 FormatFat@256-445@./Common/Fat.c
     187     31    1068      4 SecurityTokenKeyfileDlgProc@9056-9286@./Common/Dlgcode.c
```

```
     188      30     780       2 ExtractCommandLine@7772-8015@./Format/Tcformat.c
     149      30    1060       5 Process::Execute@26-201@./Platform/Unix/Process.cpp
      89      29     468       2 handleError@3842-3945@./Common/Dlgcode.c
     128      29     827       1 GraphicUserInterface::BackupVolumeHeaders@109-283@./Main/GraphicUserInterface.cpp
     153      28    1139       1 CoreService::StartElevated@339-524@./Core/Unix/CoreService.cpp
      93      28     623       0 PlatformTest::TestAll@234-347@./Platform/PlatformTest.cpp
     137      27     861       4 KeyfileGeneratorDlgProc@4862-5036@./Common/Dlgcode.c
      81      27     429       0 GetWindowsEdition@8223-8321@./Common/Dlgcode.c
      86      26     515       2 SaveFavoriteVolumes@650-761@./Mount/Favorites.cpp
     165      26    1249       4 MultiChoiceDialogProc@5497-5716@./Common/Dlgcode.c
      93      25     563       7 GetDrivePartitions@359-472@./Boot/Windows/BootDiskIo.cpp
     105      25     688       4 RandomPoolEnrichementDlgProc@4715-4844@./Common/Dlgcode.c
      67      25     421       1 MainFrame::OnHotkey@884-965@./Main/Forms/MainFrame.cpp
      85      25     432       0 MainFrame::OnTimer@1215-1317@./Main/Forms/MainFrame.cpp
      88      25    1061       1 aes_init@274-421@./Crypto/Aestab.c
      66      24     341       2 VerifySizeAndUpdate@1255-1334@./Format/Tcformat.c
     105      24     631       1 DecryptDrive@699-839@./Boot/Windows/BootMain.cpp
      98      24     593       5 DismountAll@3562-3685@./Mount/Mount.c
      76      24     454       2 main@26-127@./Main/Unix/Main.cpp
      73      24     415       6 TextUserInterface::ChangePassword@360-452@./Main/TextUserInterface.cpp
      50      23     261       1 UpdateNonSysInPlaceEncControls@1835-1896@./Format/Tcformat.c
     105      23     659       5 AnalyzeHiddenVolumeHost@8066-8213@./Format/Tcformat.c
     145      23    1078       3 MountDrive@220-404@./Driver/DriveFilter.c
      76      23     480       0 BootMenu@481-572@./Boot/Windows/BootMain.cpp
     115      23     601       4 SecurityTokenPreferencesDlgProc@8257-8401@./Mount/Mount.c
     133      23     785       2 KeyFilesApply@217-387@./Common/Keyfiles.c
     137      23     842       2 InitApp@2281-2472@./Common/Dlgcode.c
     103      23     595       4 TextInfoDialogBoxDlgProc@2717-2843@./Common/Dlgcode.c
     108      23     832       0 PlatformTest::SerializerTest@26-160@./Platform/PlatformTest.cpp
      61      22     308       3 PrintFreeSpace@2853-2919@./Format/Tcformat.c
      75      22     314       0 main@1035-1151@./Boot/Windows/BootMain.cpp
     103      22     679       0 Int13Filter@26-177@./Boot/Windows/IntFilter.cpp
      76      22     602       3 LoadFavoriteVolumes@501-597@./Mount/Favorites.cpp
     152      22     979       4 KeyFilesDlgProc@418-607@./Common/Keyfiles.c
      58      22     579       1 UpdateProgressBarProc@61-130@./Common/Progress.c
      60      22     382       1 MountVolume@65-133@./Core/Unix/CoreServiceProxy.h
     112      22     700       2 CoreService::ProcessRequests@84-225@./Core/Unix/CoreService.cpp
      75      22     415       4 File::Open@185-279@./Platform/Unix/File.cpp
      90      22     550       0 VolumeCreationWizard::OnVolumeCreatorFinished@397-512@./Main/Forms/VolumeCreationWizard.cpp
      78      22     376       3 UserInterface::DismountVolumes@144-233@./Main/UserInterface.cpp
      82      22     469       1 UserInterface::MountAllDeviceHostedVolumes@572-671@./Main/UserInterface.cpp
      77      21     552       1 FinalPreTransformPrompts@3148-3250@./Format/Tcformat.c
      46      21     324       3 LoadImageNotifyRoutine@1035-1091@./Driver/DriveFilter.c
      87      21     508       2 CopySystemPartitionToHiddenVolume@577-693@./Boot/Windows/BootMain.cpp
      72      21     282       2 ChangeSysEncPassword@3960-4050@./Mount/Mount.c
      60      21     301       2 HandleHotKey@7353-7429@./Mount/Mount.c
     100      21     504       4 BootLoaderPreferencesDlgProc@8410-8531@./Mount/Mount.c
      83      21     791       1 GetFatParams@27-132@./Common/Fat.c
     148      21    1227       4 BenchmarkDlgProc@4507-4712@./Common/Dlgcode.c
     110      21     829       1 VolumeCreator::CreateVolume@177-330@./Core/VolumeCreator.cpp
      83      21     791       1 GetFatParams@43-148@./Core/FatFormatter.cpp
      75      21     474       1 ChangePasswordDialog::OnOKButtonClick@75-165@./Main/Forms/ChangePasswordDialog.cpp
      92      21     601       1 TextUserInterface::BackupVolumeHeaders@233-358@./Main/TextUserInterface.cpp
      63      20     524       3 VolumeHeader::Deserialize@134-216@./Volume/VolumeHeader.cpp
      65      20     495       3 MoveClustersBeforeThresholdInDir@1588-1676@./Format/InPlace.c
      83      20     379       3 QueryFreeSpace@3038-3145@./Format/Tcformat.c
      77      20     374       1 DriverAttach@3423-3543@./Common/Dlgcode.c
      83      20     466       5 WriteRandomDataToReservedHeaderAreas@1088-1196@./Common/Volumes.c
      79      20     435       2 TextUserInterface::AskPassword@84-181@./Main/TextUserInterface.cpp
     110      20     598       0 GraphicUserInterface::OnInit@768-913@./Main/GraphicUserInterface.cpp
      55      19     477       1 dynamic@329-398@./Boot/Windows/Decompressor.c
      78      19     368       0 DriverUnload@3319-3420@./Common/Dlgcode.c
      60      19     372       1 EncryptionThreadPoolStart@218-307@./Common/EncryptionThreadPool.c
      82      19     531       2 CoreMacOSX::MountAuxVolumeImage@109-211@./Core/Unix/MacOSX/CoreMacOSX.cpp
      21      19     189       0 ChangePasswordDialog::OnPasswordPanelUpdate@167-194@./Main/Forms/ChangePasswordDialog.cpp
      48      19     267       1 MainFrame::OnClose@726-785@./Main/Forms/MainFrame.cpp
      99      18     586       1 DecoySystemWipeThreadProc@1692-1821@./Driver/DriveFilter.c
      80      18     435       1 SlowPoll@535-651@./Common/Random.c
      59      18     255       0 cleanup@233-308@./Common/Dlgcode.c
      64      18     452       0 BootEncryption::GetSystemDriveConfiguration@847-927@./Common/BootEncryption.cpp
      98      18     703       0 VolumeCreator::CreationThread@44-175@./Core/VolumeCreator.cpp
      60      17     235       6 OpenPartitionVolume@1177-1247@./Format/InPlace.c
      84      17     457       2 DumpFilterEntry@20-137@./Driver/DumpFilter.c
     103      17     509       2 MountDevice@2511-2641@./Driver/Ntdriver.c
      52      17     335       3 BroadcastDeviceChange@5884-5949@./Common/Dlgcode.c
      83      17     468       0 BootEncryption::GetPartitionForHiddenOS@409-518@./Common/BootEncryption.cpp
     103      17     580       3 BootEncryption::ChangePassword@2050-2193@./Common/BootEncryption.cpp
      69      17     508       1 EncryptionThreadProc@123-215@./Common/EncryptionThreadPool.c
      84      16     471       6 EncryptionThreadPool::DoWork@25-133@./Volume/EncryptionThreadPool.cpp
      62      16     383       3 UnmountDevice@2643-2727@./Driver/Ntdriver.c
      45      16     271       3 HiberDriverEntryFilter@957-1014@./Driver/DriveFilter.c
      55      16     259       2 GetPoolBuffer@34-103@./Driver/EncryptedIoQueue.c
      82      16     609       1 EncryptedIoQueueStart@868-979@./Driver/EncryptedIoQueue.c
      81      16     463       1 DisplayHotkeyList@165-266@./Mount/Hotkeys.c
      62      16     508       1 InitMainDialog@220-304@./Mount/Mount.c
      62      16     225       1 DecryptSystemDevice@4100-4170@./Mount/Mount.c
      71      16     294       1 CreateRescueDisk@4196-4277@./Mount/Mount.c
      53      16     416       3 AddMountedVolumeToFavorites@31-98@./Mount/Favorites.cpp
     105      16     779       1 InitDialog@1106-1239@./Common/Dlgcode.c
      38      16     207       0 BootEncryption::CheckRequirements@1862-1912@./Common/BootEncryption.cpp
      68      16     462       3 GetArgumentID@130-210@./Common/Cmdline.c
      48      16     222       3 Wipe35Gutmann@83-139@./Common/Wipe.c
      59      16     300       3 CoreMacOSX::DismountVolume@32-100@./Core/Unix/MacOSX/CoreMacOSX.cpp
      83      16     510       0 MainFrame::UpdateVolumeList@1455-1561@./Main/Forms/MainFrame.cpp
      48      16     362       1 UserInterface::ExceptionToMessage@298-365@./Main/UserInterface.cpp
      46      16     385       1 UserInterface::ExceptionToString@367-436@./Main/UserInterface.cpp
================================================================================
Total nloc  Avg.nloc  Avg CCN  Avg token  Fun Cnt  Warning cnt   Fun Rt   nloc Rt
```

```
-------------------------------------------------------------------------------
     79853        20     4.68     135.88     3272          170      0.05    0.43
```

Appendix B

Duplicate code (excerpt)

```
Boot/Windows/BootConsoleIo.cpp(50)
Boot/Windows/BootConsoleIo.cpp(35)
if (ScreenOutputDisabled)
return;
__asm
mov bx, 7
mov al, c

Boot/Windows/BootDiskIo.cpp(220)
Boot/Windows/BootDiskIo.cpp(143)
if (result == BiosResultEccCorrected)
result = BiosResultSuccess;
} while (result != BiosResultSuccess && --tryCount != 0);
if (!silent && result != BiosResultSuccess)

Boot/Windows/BootDiskIo.cpp(214)
Boot/Windows/BootDiskIo.cpp(136)
int 0x13
jnc ok
mov result, ah
ok:

Boot/Windows/BootDiskIo.cpp(200)
Boot/Windows/BootDiskIo.cpp(116)
BiosResult result;
byte tryCount = TC_MAX_BIOS_DISK_IO_RETRIES;
result = BiosResultSuccess;
__asm

Boot/Windows/BootMain.cpp(293)
Boot/Windows/BootConsoleIo.cpp(288)
__asm
push es
xor ax, ax
mov es, ax

Boot/Windows/Decompressor.c(142)
Boot/Windows/Decompressor.c(114)
local int decode(struct state *s, struct huffman *h)
    int len;
    int code;
    int first;
    int count;
    int index;

Boot/Windows/Decompressor.c(166)
Boot/Windows/Decompressor.c(127)
                return h->symbol[index + (code - first)];
            index += count;
            first += count;
            first <<= 1;
            code <<= 1;

Boot/Windows/IntFilter.cpp(525)
Boot/Windows/IntFilter.cpp(518)
popad
popf
leave
add sp, 2

Boot/Windows/IntFilter.cpp(616)
Boot/Windows/IntFilter.cpp(602)
mov ax, es:[si]
mov [di], ax
mov ax, es:[si + 2]
mov [di + 2], ax

Boot/Windows/IntFilter.cpp(154)
Boot/Windows/IntFilter.cpp(110)
IntRegisters.Flags &= ~TC_X86_CARRY_FLAG;
else
IntRegisters.Flags |= TC_X86_CARRY_FLAG;
passOriginalRequest = false;
break;

Boot/Windows/IntFilter.cpp(384)
Boot/Windows/IntFilter.cpp(318)
Print ("EAX:"); PrintHex (IntRegisters.EAX);
Print (" EBX:"); PrintHex (IntRegisters.EBX);
Print (" ECX:"); PrintHex (IntRegisters.ECX);
Print (" EDX:"); PrintHex (IntRegisters.EDX);
Print (" DI:"); PrintHex (IntRegisters.DI);

Boot/Windows/IntFilter.cpp(312)
Boot/Windows/IntFilter.cpp(46)
uint16 spdbg;
__asm mov spdbg, sp
PrintChar (' ');
PrintHex (spdbg);
PrintChar ('<'); PrintHex (TC_BOOT_LOADER_STACK_TOP);

Boot/Windows/Platform.cpp(49)
Boot/Windows/Platform.cpp(18)
__asm
jnc nocarry
mov carry, 1
nocarry:

Common/BaseCom.cpp(155)
Common/BaseCom.cpp(130)
catch (SystemException &)
return GetLastError();
catch (Exception &e)
e.Show (NULL);
return ERROR_EXCEPTION_IN_SERVICE;
catch (...)
return ERROR_EXCEPTION_IN_SERVICE;
return ERROR_SUCCESS;

Common/BaseCom.cpp(180)
Common/BaseCom.cpp(155)
catch (SystemException &)
return GetLastError();
catch (Exception &e)
e.Show (NULL);
return ERROR_EXCEPTION_IN_SERVICE;
catch (...)
return ERROR_EXCEPTION_IN_SERVICE;
return ERROR_SUCCESS;

Common/BaseCom.cpp(205)
Common/BaseCom.cpp(180)
catch (SystemException &)
return GetLastError();
catch (Exception &e)
e.Show (NULL);
return ERROR_EXCEPTION_IN_SERVICE;
catch (...)
return ERROR_EXCEPTION_IN_SERVICE;
return ERROR_SUCCESS;

Common/BaseCom.cpp(180)
Common/BaseCom.cpp(130)
catch (SystemException &)
return GetLastError();
catch (Exception &e)
e.Show (NULL);
return ERROR_EXCEPTION_IN_SERVICE;
catch (...)
return ERROR_EXCEPTION_IN_SERVICE;
return ERROR_SUCCESS;

Common/BaseCom.cpp(205)
Common/BaseCom.cpp(155)
catch (SystemException &)
return GetLastError();
catch (Exception &e)
e.Show (NULL);
return ERROR_EXCEPTION_IN_SERVICE;
catch (...)
return ERROR_EXCEPTION_IN_SERVICE;
return ERROR_SUCCESS;

Common/BaseCom.cpp(130)
Common/BaseCom.cpp(65)
catch (SystemException &)
```

```
return GetLastError();
catch (Exception &e)
e.Show (NULL);
return ERROR_EXCEPTION_IN_SERVICE;
catch (...)
return ERROR_EXCEPTION_IN_SERVICE;
return ERROR_SUCCESS;

Common/BaseCom.cpp(205)
Common/BaseCom.cpp(130)
catch (SystemException &)
return GetLastError();
catch (Exception &e)
e.Show (NULL);
return ERROR_EXCEPTION_IN_SERVICE;
catch (...)
return ERROR_EXCEPTION_IN_SERVICE;
return ERROR_SUCCESS;

Common/BaseCom.cpp(155)
Common/BaseCom.cpp(65)
catch (SystemException &)
return GetLastError();
catch (Exception &e)
e.Show (NULL);
return ERROR_EXCEPTION_IN_SERVICE;
catch (...)
return ERROR_EXCEPTION_IN_SERVICE;
return ERROR_SUCCESS;

Common/BaseCom.cpp(180)
Common/BaseCom.cpp(65)
catch (SystemException &)
return GetLastError();
catch (Exception &e)
e.Show (NULL);
return ERROR_EXCEPTION_IN_SERVICE;
catch (...)
return ERROR_EXCEPTION_IN_SERVICE;
return ERROR_SUCCESS;

Common/BaseCom.cpp(205)
Common/BaseCom.cpp(65)
catch (SystemException &)
return GetLastError();
catch (Exception &e)
e.Show (NULL);
return ERROR_EXCEPTION_IN_SERVICE;
catch (...)
return ERROR_EXCEPTION_IN_SERVICE;
return ERROR_SUCCESS;

Common/BootEncryption.cpp(2247)
Common/BootEncryption.cpp(2234)
BootEncryptionStatus encStatus = GetStatus();
if (encStatus.DriveMounted)
throw ParameterIncorrect (SRC_POS);
CheckRequirements ();

Common/BootEncryption.cpp(2360)
Common/BootEncryption.cpp(2344)
BootEncryptionStatus encStatus = GetStatus();
if (!encStatus.DeviceFilterActive || !encStatus.DriveMounted
|| encStatus.SetupInProgress)
throw ParameterIncorrect (SRC_POS);
BootEncryptionSetupRequest request;
ZeroMemory (&request, sizeof (request));

Common/BootEncryption.cpp(1151)
Common/BootEncryption.cpp(1129)
Device device (GetSystemDriveConfiguration().DevicePath);
byte mbr[TC_SECTOR_SIZE_BIOS];
device.SeekAt (0);
device.Read (mbr, sizeof (mbr));

Common/BootEncryption.cpp(1175)
Common/BootEncryption.cpp(1137)
device.SeekAt (0);
device.Write (mbr, sizeof (mbr));
byte mbrVerificationBuf[TC_SECTOR_SIZE_BIOS];
device.SeekAt (0);
device.Read (mbrVerificationBuf, sizeof (mbr));
if (memcmp (mbr, mbrVerificationBuf, sizeof (mbr)) != 0)
throw ErrorException ("ERROR_MBR_PROTECTED");

Common/BootEncryption.cpp(1323)
Common/BootEncryption.cpp(1265)
Device device (GetSystemDriveConfiguration().DevicePath);
byte mbr[TC_SECTOR_SIZE_BIOS];
device.SeekAt (0);
device.Read (mbr, sizeof (mbr));

Common/BootEncryption.cpp(1265)
Common/BootEncryption.cpp(1151)
Device device (GetSystemDriveConfiguration().DevicePath);
byte mbr[TC_SECTOR_SIZE_BIOS];
device.SeekAt (0);
device.Read (mbr, sizeof (mbr));

Common/BootEncryption.cpp(347)
Common/BootEncryption.cpp(238)
FILE_FLAG_RANDOM_ACCESS | FILE_FLAG_WRITE_THROUGH, NULL);
try
throw_sys_if (Handle == INVALID_HANDLE_VALUE);
catch (SystemException &)
if (GetLastError() == ERROR_ACCESS_DENIED && IsUacSupported())
Elevated = true;
else
throw;
FileOpen = true;
FilePointerPosition = 0;

[...]

Results:
  Lines of code: 48774
  Duplicate lines of code: 7091
  Total 1155 duplicate block(s) found.

  Time: 8.04078 seconds
```

Appendix C

Detail of the static analysis tools' warnings

Clang			
Kingdom	Warning type	N. of warnings	Security relevant
Input validation	malloc() size overflow	2	X
	Out of bound array access	4	X
API Abuse	Cast from non-struct type to struct type	55	
Code quality	Assigned value is garbage or undefined	1	
	Dead assignment	3	
	Dead initialisation	3	
	Dereference of null pointer	1	
	Uninitialized argument value	2	
	Use fixed address	1	

Cppcheck			
Kingdom	Warning type	N. of warnings	Security relevant
Input validation	Buffer access out of bounds	4	X
API Abuse	Memset with POD	2	
Code quality	C style cast	4	
	Redundant condition	1	
	Use Initialisation List	1	
	Uninitialised variable	33	
	Variable scope	80	
	Unassigned variable	4	
	Uninitialised member variable	26	
	Unused structure member	1	
	No copy constructor	1	
	Unread variable	4	
	Unused variable	3	
	Invalid printf argument type string	1	
	Invalid printf argument type int	2	
	Invalid scanf	1	
	Invalid scanf specific to libc	11	
	Wrong printf/scanf arguments	1	
	Clarify condition	2	
	Clarify calculation	3	
	Uninitialised data	4	
	Wrong copy of pointer	4	
	Use strcmp()	1	
	Redundant assignment	12	
	Bad use of c_str	2	
	Prefix operators for non primitive types	2	

Coverity Unix			
Kingdom	Warning type	N. of warnings	Security relevant
Input validation	Insecure data handling	2	X
	Integer handling issues	2	X
	Memory corruptions	1	X
API Abuse	API usage errors	1	
	Build system issues	2	
Time and state	Performance inefficiencies	1	
	Concurrent data access violations	1	
	Program hangs	3	
Code quality	Error handling issues	2	
	Null pointer dereferences	11	
	Parse warnings	4	
	Resource leaks	1	
	Uninitialised members	26	
	Various	1	

Coverity Windows			
Kingdom	Warning type	N. of warnings	Security relevant
Input validation	Insecure data handling	18	X
	Integer handling issues	1	X
	Memory corruptions	5	X
	Memory illegal accesses	7	X
	Various	1	X
API Abuse	API usage errors	1	
	Integer handling issues	3	
	Various	1	
Time and state	Program hangs	1	
	Security best practices	1	
Code quality	Code maintainability issues	1	
	Control flow issues	16	
	Error handling issues	19	
	Incorrect expression	6	
	Integer handling issues	1	
	Null pointer dereference	3	
	Performance inefficiencies	23	
	Parse warnings	3	
	Possible control flow issues	1	
	Resource leaks	13	
	Security best practices	33	
	Uninitialised variables	11	

Bibliography

[1] Anders Bakken. *rtags Webseite (GIT-Hub)*. URL: https://github.com/Andersbakken/rtags.

[2] Alex Balducci, Sean Devlin, and Tom Ritter. *Cryptographic Review*. Version 1.0. Open Crypto Audit Project, Mar. 2015. URL: https://opencryptoaudit.org/reports/TrueCrypt_Phase_II_NCC_OCAP_final.pdf.

[3] Milan Broz and Vashek Matyas. "The TrueCrypt On-Disk Format-An Independent View". In: *IEEE Security & Privacy* 12.3 (2014), pp. 74–77. DOI: 10.1109/MSP.2014.60. URL: http://dx.doi.org/10.1109/MSP.2014.60.

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224. URL: http://dl.acm.org/citation.cfm?id=1855741.1855756.

[5] Software Engineering Institute Carnegie Mellon University. *MSC06-CPP. Be aware of compiler optimization when dealing with sensitive data*. URL: https://www.securecoding.cert.org/confluence/display/cplusplus/MSC06-CPP.+Be+aware+of+compiler+optimization+when+dealing+with+sensitive+data.

[6] *Cscope (version 15.8a) Webseite (Sourceforge)*. URL: http://cscope.sourceforge.net/.

[7] Edsger W. Dijkstra. "Go To statement considered harmful". In: *Comm. ACM* 11.3 (1968). letter to the Editor, pp. 147–148.

[8] Niels Ferguson. *AES-CBC+ Elephant diffuser: A disk encryption algorithm for Windows Vista*. 2006.

[9] TrueCrypt Foundation. *TrueCrypt User's Guide. version 7.1a*. Feb. 2012.

[10] TrueCrypt Foundation. *TrueCrypt Web-Seite*. URL: http://truecrypt.sourceforge.net/.

[11] Geoffrey K. Gill and Chris F. Kemerer. "Cyclomatic Complexity Density and Software Maintenance Productivity". In: *IEEE Trans. Softw. Eng.* 17.12 (Dec. 1991), pp. 1284–1288. ISSN: 0098-5589. DOI: 10.1109/32.106988. URL: http://dx.doi.org/10.1109/32.106988.

[12] Alex Hornung. *tcplay Webseite (GIT-Hub)*. URL: https://github.com/bwalex/tc-play.

[13] "IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices". In: *IEEE Std 1619-2007* (2008), pp. c1–32. DOI: 10.1109/IEEESTD.2008.4493450.

[14] Andreas Junestam and Nicolas Guigo. *TrueCrypt Security Assessment*. Version 1.1. Open Crypto Audit Project, Mar. 2014. URL: https://opencryptoaudit.org/reports/iSec_Final_Open_Crypto_Audit_Project_TrueCrypt_Security_Assessment.pdf.

[15] Chris F. Kemerer. "Software complexity and software maintenance: A survey of empirical research". English. In: *Annals of Software Engineering* 1.1 (1995), pp. 1–22. ISSN: 1022-7091. DOI: 10.1007/BF02249043. URL: http://dx.doi.org/10.1007/BF02249043.

[16] Steve McConnell. *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press, 2004. ISBN: 0735619670, 9780735619678.

[17] MITRE. *CVE-2014-4115*. June 2014.

[18] MITRE. *CVE-2015-7358*. Sept. 2015.

[19] MITRE. *CVE-2015-7359*. Sept. 2015.

[20] Meiyappan Nagappan et al. "An empirical study of goto in C code." In: *PeerJ PrePrints* (2015). URL: https://dx.doi.org/10.7287/peerj.preprints.826v1.

[21] Katrina Tsipenyuk, Brian Chess, and Gary McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors". In: *IEEE Security & Privacy* 3.6 (2005), pp. 81–84. DOI: 10.1109/MSP.2005.159. URL: http://doi.ieeecomputersociety.org/10.1109/MSP.2005.159.

[22] Meltem Sönmez Turan et al. *Recommendation for Password-Based Key Derivation Part 1: Storage Applications*. Version Special Publication 800-132. NIST, 2010.

[23] Sven Türpe et al. "Attacking the BitLocker Boot Process". In: *Trusted Computing, 2nd International Conference, Trust 2009*. Ed. by Liqun Chen, Chris Mitchell, and Andrew Martin. Vol. 5471. LNCS. Oxford, UK, April 6-8, 2009. Berlin / Heidelberg: Springer, 2009, pp. 183–196. DOI: 10.1007/978-3-642-00587-9_12.

[24] Andrew Y. *Unofficial reference site of truecrypt.org*. 2014. URL: http://andryou.com/truecrypt/.