



ASSIGNMENT OF MASTER'S THESIS

Title: Security Analysis of BestCrypt
Student: Bc. Jakub Soušek
Supervisor: Ing. Josef Kokeš
Study Programme: Informatics
Study Branch: Computer Security
Department: Department of Computer Systems
Validity: Until the end of summer semester 2016/17

Instructions

- 1) Research current publicly available analyses of disk encryption software.
- 2) Download, install, and describe software BestCrypt by Jetico Inc. (BC).
- 3) Explore and evaluate the security implications inherent in BC's user interface.
- 4) Study the BestCrypt Development Kit provided by the developer, describe its requirements, installation, structure, and functionality.
- 5) Using available information, duplicate the decryption procedure of an encrypted container using a well-known open source crypto library.
 - Describe the container's structure.
 - Describe and implement the password key derivation.
 - Decrypt the container key.
 - Attempt to decrypt selected sectors from the encrypted volume.
- 6) Evaluate the impact of your findings on the developer's security claims.

References

Will be provided by the supervisor.

L.S.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 2, 2016

CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS



Master's thesis

Security analysis of BestCrypt

Bc. Jakub Souček

Supervisor: Ing. Josef Kokeš

3rd May 2016

Acknowledgements

I would like to thank my supervisor, Ing. Josef Kokeš, for his constant help with the thesis and for providing me with this topic. I would also like to thank the Jetico development team for their help and communication.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 3rd May 2016

.....

Czech Technical University in Prague

Faculty of Information Technology

© 2016 Jakub Souček. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Souček, Jakub. *Security analysis of BestCrypt*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

Práce se zabývá bezpečnostní analýzou programu BestCrypt, který umožňuje šifrování souborů a složek na disku. Zaměřuje se na proces generování klíče z hesla, operace šifrování a dešifrování a bezpečnost programu jako takového.

Klíčová slova klíč, bezpečnostní analýza, BestCrypt, kontejner, BestCrypt Development Kit, šifrování, dešifrování

Abstract

The thesis provides a security analysis of BestCrypt. BestCrypt is a software that allows users to encrypt their files and folders. The analysis focuses on the key generation process, the cryptographic processes and the overall security of the application.

Keywords key, security analysis, BestCrypt, container, BestCrypt Development Kit, encryption, decryption

Contents

Introduction	1
1 State-of-the-art	3
1.1 TrueCrypt with an attack on keyfile algorithm	3
1.2 TrueCrypt Security Assessment	5
1.3 TrueCrypt Cryptographic Review	6
1.4 Security Analysis of TrueCrypt	6
1.5 BitLocker Security Analysis	7
1.6 Summary and further direction	8
2 BestCrypt	9
2.1 Data manipulation process	10
2.2 The encryption modes	11
2.3 Algorithm drivers and keys	13
2.4 Functionalities	14
2.5 Installation	16
2.6 Usage	17
3 BestCrypt Development Kit	19
3.1 Installation	19
3.2 BestCrypt container structure	21
3.3 BDK and container files	22
3.4 Functionalities	23
3.5 BDK sample illustration	24
4 Key generation process	25
4.1 Password retrieval	26
4.2 Password key generation	26
4.3 Master key generation	28
4.4 Correctness verification	28

4.5	Key store	29
4.6	I/O commands	29
5	BestCrypt security	31
5.1	Default encryption parameters	31
5.2	Password strength	33
5.3	Password manipulation	34
5.4	Container Re-encryption	35
5.5	Hidden container creation	36
5.6	Random number generation	37
5.7	Network connection	40
5.8	Security conclusion	40
6	Cryptographic processes re-implementation	41
6.1	Covered areas	41
6.2	Used tools	42
6.3	Program setup	42
6.4	How the program works	43
6.5	Key generation	43
6.6	Container data decryption	44
6.7	Verification steps	45
6.8	Necessary modifications	46
6.9	The implementation results	48
7	Impact evaluation	51
7.1	Data leaks prevention	51
7.2	Transparent file encryption	51
7.3	Strong encryption algorithms	52
7.4	Improved security in new key generator	52
7.5	Hidden part security	52
	Conclusion	53
	Bibliography	55
	A Acronyms	59
	B Contents of enclosed CD	61

List of Figures

2.1	BestCrypt realisation of the transparent encryption [1].	10
2.2	CBC mode encryption scheme [2].	11
2.3	CBC mode decryption scheme [3].	12
2.4	XTS mode encryption scheme [4].	13
2.5	BestCrypt architecture [5].	13
2.6	Key map view.	15
2.7	BestCrypt Control Panel.	17
2.8	BestCrypt Container Properties window.	17
3.1	Creating version 8 container.	20
4.1	String D.	27
4.2	String S.	27
4.3	String P.	27
5.1	BestCrypt Plug-in Manager.	33
5.2	Warning when changing or removing a password.	34
5.3	Warning when removing the last password.	35
5.4	Re-encryption settings.	36
5.5	Hidden part discovery attack result.	39
6.1	Correctly decrypted key block.	44
6.2	Correctly decrypted Master Boot Record.	45

List of Tables

2.1	Supported encryption algorithms and their encryption modes. . . .	10
2.2	Algorithm parameters from Windows Registry Database.	16
3.1	Container file structure scheme.	21
6.1	Key block IV forms.	47

Introduction

Almost anyone has sometimes encountered the problem of having some important files on their computer and wanting to be sure that these files are accessible only by them even if someone stole the computer. There are many solutions to this problem and one of them is a software called BestCrypt developed by Jetico Inc.

BestCrypt is the focus of this thesis. It is introduced along with its features and the ideas behind it. All the important information about the principle of achieving the security of the confidential files is explained in detail, but the main idea is the same as in most of the similar software – using cryptography.

The main problem with such software is that the user who decides to use it needs to trust that the developers created a reliable and secure software and that he is really the only one who can access the data he chose to protect. A common concern is the question whether the software does not contain some sort of back door to allow some powerful third party access when requested, for any reason.

Therefore, companies who want their data security software to be trustworthy need to provide some means for enabling anyone to convince themselves that the product is secure. Security analysis can then be performed by individual users or, more often, by professionals. What do the analyses look for and how do they prove or disprove the credibility of the product they examine is explained as well.

Some companies decide to make the source codes publicly available. Anyone can then look at them, look up the part he is interested in and see how it really works. Jetico chose a different way to enable its users to validate the security of BestCrypt by themselves: they provided the BestCrypt Development Kit (BDK).

BDK is a library which contains all the encryption, decryption and key generation processes. It describes data structures defining the representation of data it works with and functions to execute the important cryptographic operations.

The main aim of this thesis is to perform a security analysis of BestCrypt. The analysis examines three main parts of the application. It analyses the key generation process and its security, verifies the correctness of the supported encryption algorithms and examines the overall security of the software.

The whole process of securing data confidentiality is deeply analysed and reimplemented using a well-known cryptographic tool to demonstrate that the used algorithms work the same way as they should and yield the same results.

Besides cryptography, BestCrypt is analysed partially from the user's point of view. The focus is on areas like the resistance against options that could lead to a weak protection, the password strength and management or the random number generation. The BDK is used whenever possible.

The analysis re-implements the encryption algorithms to prove their correctness. Any implementation specific issues and things that differ from the expected behaviour are mentioned and explained. In the end, Jetico's security claims are compared with the analysis results and a final decision is made.

State-of-the-art

This chapter focuses on the current situation of security analyses of file encryption software. It lists several important security analyses while focusing on their aim, approach and findings. This information helps to understand possible ways of analysing such software and their advantages and differences.

Analysing the security of this kind of software is of great importance. Any company that creates such software should provide some way to enable anyone to verify for themselves that they are satisfied with the product. The easiest way to do so is to make the source code public, yet not every company can or wants to do so. If the source code is not made available, some other way to provide enough ground to allow a security analysis should be taken.

Several analyses of a similar software TrueCrypt [6] are the best source. TrueCrypt serves the same purpose as BestCrypt. It helps users to secure their files. The authors of TrueCrypt remained anonymous and that was one of the reasons that led to the creation of Open Crypto Audit Project (OCAP) [7], a project that decided to fully analyse the software and evaluate its security. TrueCrypt developers agreed to make their source codes publicly available.

After some time, there appeared a warning on the official TrueCrypt website declaring that "Using TrueCrypt is not secure as it may contain unfixed security issues". This raised even more effort to further analyse the software as mentioned in the article by Matthew Green, one of the OCAP leaders [8].

Some possibly unknown terms are used in this chapter. All of them are explained in chapter 2. For a basic understanding of how TrueCrypt works, the article by Vlastimil Klíma [9] could be helpful. It does not go into detail, but it provides a solid overview of the software.

1.1 TrueCrypt with an attack on keyfile algorithm

One of the first TrueCrypt analyses is the one focusing on the Linux version of TrueCrypt. The same organization performed private analyses of previous versions and published the one focusing on version 7.0a [10].

Its focus is wide, ranging from general code inspection to encryption algorithms examination. The authors started by a detailed description on how to build the program under Linux and Windows operating systems. Since they analysed previous versions, they were able to use their own findings and compare their data with the version 7.0a.

Since TrueCrypt source codes are publicly available, the first part of the analysis consists of fully examining the source codes statically. Since the authors had previous version's source codes as well, the main focus was on the parts that differed from the older versions.

A program for analysing the headers of TrueCrypt containers was created as a part of the analysis and published with its source code. Its usage is described in the analysis report. The authors created test containers using all available algorithms at least once. They tested hidden containers as well. No backdoors or mistakes were discovered in the container encryption or its header format.

Besides some minor issues, an increased amount of iterations when processing the password was suggested. No weakness was found in the implementation of the encryption modes either while the main focus was the XTS mode.

The hidden container functionality was tested as well and found completely secure. The random number generator was found secure when used properly. The flaw the authors mention is that while the randomness originates from mouse movement, the user is not able to see the random generation progress and can end the process too early. They also suggest disabling the feature to allow the user to see the actual generated bytes since an attacker with access to the monitor could intercept it.

There is an anomaly connected with random data filling mentioned. On Linux computers, a specific area is filled with zero bytes while on Windows it contains random data. The authors proved that this is not a backdoor through the source code analysis.

A major flaw was found in the keyfile algorithm. The authors discovered a way to modify the keyfiles so that they do not affect the password at all. When a weak password is used (or none at all), this can lead to a successful attack. However, no container not using keyfiles is affected by this flaw in any way.

A fundamental problem was found in the relation between the binaries available from the TrueCrypt website and the source codes that were published. The issue was that the hash of the binaries compiled from the source codes did not match with the one of the provided binaries. However, this issue was not further analysed.

This security analysis represents a very complex approach. It analyzes the whole software as it is based mainly on the source codes and covers topics like the compilation process and the license distribution. It presents possible issues while not forgetting the strong secured parts it verified and classified as

safe. Along with the main issue it offers a detailed description of the keyfile attack and its importance and impact.

1.2 TrueCrypt Security Assessment

This is the first analysis performed by the Open Crypto Audit Project. The report [11] offered a great summary of what the audit had been examining and what it found.

Unlike the previous analysis, this one focuses strictly on the source code itself. Its goal is to find and examine any weaknesses that could lead to attacks like elevation of privilege or information disclosure. The parts of the code the analysis focuses on are the bootloader and Windows kernel driver. Some parts of the code are intentionally left out. The examined version is TrueCrypt 7.1a. The authors used public automated tools, manual test techniques and performed a source code review.

Overall, 11 possible vulnerabilities were found, four of them rated as medium severity, four as low severity and three as informational. The majority of these issues were connected with data exposure and data validation. The analysis also defines what exactly each of the severity categories means and explains the categories the issues were divided into based on the area they are associated with.

Based on the report, the source code did not meet the expected standards for secure code. The authors mention problems like lack of comments, usage of deprecated functions or inconsistent variable types. They also advise against using checksums in place of hash functions or HMAC.

Aside from listing the findings, the authors also mention their recommendations to improve TrueCrypt. These are mainly updating the build environment and improving the source code quality.

This analysis goes into more detail when describing the vulnerabilities. After introducing them, it explains each one in detail, suggests how an exploit could happen and offers recommendation on how to solve the issue. In some cases it even highlights the problematic parts of the code.

The last part of the analysis focuses on the source code quality. It mentions issues like signed and unsigned variables mismatch or inconsistent integer variable types and provides highlighted source code parts where such issues can be seen. The unnecessarily complicated and long functions are mentioned too.

This approach provides significant results. It is important to know whether the code itself is vulnerable to attacks of any kind. It consists of static analysis methods since the source code is examined and the application does not need to run. There are many great tools to perform static analysis that can be used to help the process. As seen in this case, it is not necessary to examine

the whole software. Analysing only a few important or possibly weak parts is sufficient if it provides reasonable results.

1.3 TrueCrypt Cryptographic Review

This analysis [12] is the second one that the Open Crypto Audit Project performed. Its structure is very similar to the first one.

It focuses strictly on the cryptographic processes. The authors analysed the source code statically again to discover any vulnerabilities that could lead to a possible attack to break or bypass the cryptographic operations.

Overall, 4 issues were found, two of high severity and one of low. The remaining issue's severity was undetermined. Since the first analysis did not find any vulnerabilities rated higher than medium, this one discovered two important issues. Similar to the first analysis, for each issue an exploit scenario is offered while providing specific tips on how to solve the issue.

Similar to the analysis 1.1, issues with the keyfile algorithm as well as the improper data integrity checks using checksums are mentioned in this report. The recommendations the authors suggested are simplifying the application logic, additional error handling and improvements to the code quality.

One high-severity-rated issue is connected to the random number generator. The analysis discovered an improper call to one of the Windows API functions that could in some cases lead to a failure that would not be noted by TrueCrypt. The impact would be weaker random numbers and possibly an option for a brute force attack. The other major vulnerability was found in some of the AES algorithm implementations that were found vulnerable to the cache-timing attacks [13].

Compared to the analysis 1.1, it is safe to say that both reports indicate the same results. Since both analyses were done on different versions and the latter was performed 4 years after the first, these issues apparently persisted in the software.

To summarize this approach, it is a very specifically focused one. Similar to the analysis 1.2, it takes only a part of the application to examine. Cryptography is without a doubt one of the essential parts of such software and it deserves an analysis dedicated only to it. As proved in this analysis, it can provide significant results.

1.4 Security Analysis of TrueCrypt

Probably the most comprehensive analysis [14] is the one performed by the Fraunhofer Institute for Secure Information Technology. The main reason for this analysis was the warning that appeared on the TrueCrypt official website as mentioned earlier. Since there is other software that originates

from TrueCrypt, such as Trusted Disk, any errors found in TrueCrypt could be there as well.

The focus is the software as a whole. Part of the analysis is based on the results of the Open Crypto Audit Project. Besides that it tests the cryptographic processes, the code quality, documentation and architecture. It uses the previous analyses reports, the source code review and automated static analysis tools.

First of all, it defines the differences between versions 7.0a and 7.1a (the versions that the previously mentioned analyses were based on). The code differs in two areas. Outdated encryption algorithms were removed as a command line option and choosing the files for the the keyfile algorithm ignores hidden files.

The authors provide their own notes to the findings of both the Open Crypto Audit Project analyses. They confirm the presence of all the findings and agree with the previous classifications and results.

The cryptographic processes are tested using an open source cryptographic library *libgcrypt* [15]. The goal behind it is to ensure that the algorithms work correctly (meaning the output data from TrueCrypt encryption match the one from *libgcrypt*). Detailed examination is dedicated to the random number generator and the key derivation process.

It provides additional information besides the analysis itself. Some sections are dedicated to the explanation of related attacks and their techniques. The automated tools for static code analysis are explained as well.

The analysis covers the whole application including the hidden containers, the boot code and the TrueCrypt driver. The weak code spots analysis similar to the one performed in 1.2 is included as well. In addition to what the previous analyses reported, this analysis mentions unnecessarily complicated application logic. It provides a list of too complicated functions and even identifies code duplicities.

This approach is the most robust one. Every aspect of the application is examined and evaluated. It is not always possible to perform such complex analysis due to lack of resources, yet it often provides all the results that are expected from a security analysis.

1.5 BitLocker Security Analysis

The TrueCrypt website currently offers only a tutorial on how to migrate from TrueCrypt to BitLocker. BitLocker is a Microsoft Windows feature for disk encryption included in some versions of the operating system. Fraunhofer Institute performed a security analysis of BitLocker too, focusing on attacking its boot process [16].

The authors focus on the boot process. Rather than implementing an attack or examining source code, they examine some possibly vulnerable scen-

arios and describe how an attack could be executed in theory. Overall, six possible ways of gaining unauthorized access to the confidential data via manipulating the boot process are mentioned.

Besides a brief description of BitLocker, the analysis offers the description of the attacks it proposes. Each attack scenario is described and its requirements and impact are discussed. A special section is devoted to discussing the vulnerability causes and contribution factors in general.

In the end, this analysis does not present a specific issue but rather provides a basis for further examination. Issues like the attack implementation difficulty or how to prevent them are left for further analysis. The authors themselves claim they plan to continue in their research related to BitLocker.

This approach did not provide security proof, but rather a basis another analyses can build on. Overall, presenting possible attacks can be very useful if the scenarios would be examined more deeply.

1.6 Summary and further direction

No security analysis of BestCrypt itself was mentioned. The reason for it is that none has been published yet. Only discussions and articles were published, but none went into some details. There were speculations about issues like parts of the password being stored inside the container in plain text form, but they were all confirmed false [17].

Based on the mentioned analyses, several different approaches were presented and explained. The security analysis does not need to cover every aspect and can focus only on an important part. It can point out weaknesses or design attacks against its target to prove a vulnerability.

Every analysis needs to provide a conclusion where it summarizes its findings. The importance and risk of these findings should be evaluated and, in the best case, solutions offered.

The problem with BestCrypt analysis is that the source code is not public. However, that issue is partially solved by the BDK that was designed to allow analysing the security. This analysis is based on it and analyses three main areas – the key generation, the encryption correctness and the overall security from the user’s point of view.

BestCrypt

BestCrypt is a software designed to provide additional security for any important files on disk. Using cryptographic techniques it offers protection for any sensitive information such as private correspondence, secret documents or any other data that the user decides to keep confidential for any reason.

The main principle is the following: the user is allowed to create a container, a file representing an encrypted disk image. This file can be mounted to a virtual drive that will be managed by BestCrypt. When done, it can be used as a standard drive. New files can be added and existing files can be edited or removed. The container file contains data in encrypted form even while it is mounted, so the data are always secured. Upon dismounting, any remaining changes are written to the container file and the container becomes inaccessible again.

Creating a container requires one to create a password associated with it. Knowing the correct password is the necessary step to access container data, otherwise the container cannot be mounted or accessed. There is no way to recover a forgotten password, such as a safety question or any other similar mechanism. Once the password is forgotten, the data is not accessible in any way.

One container can have more than one password associated with it. That allows multiple users to access the same data without knowing each other's password. Passwords can be managed (added, removed or changed) at any time after the container creation.

BestCrypt allows the user to choose the parameters of the container encryption mechanism. This includes the encryption algorithm, the encryption mode, the key generator and the hash algorithm. Three possible encryption schemes are available: password based encryption, public key encryption and the secret-sharing scheme. The password based encryption is the only considered scheme further on, since it is the most common used one.

Table 2.1 shows the default available algorithms and modes. It is not a complete list because more can be added using the BestCrypt Plug-in Man-

2. BESTCRYPT

Enc. Mode	Encryption Algorithm						
	3DES	Blowfish	GOST	RC6	AES	Serpent	Twofish
CBC	✓	✓	✓	✓	✓	✓	✓
LRW	✓	✓	✓	✓	✓	✓	✓
XTS	✗	✗	✗	✓	✓	✓	✓

Table 2.1: Supported encryption algorithms and their encryption modes.

ager. It is even possible to develop a new algorithm and use it for the containers encryption.

2.1 Data manipulation process

BestCrypt uses the approach known as *transparent encryption* [18]. It means, that when reading from the virtual drive, only the requested data are decrypted in memory and when writing, the data to be written are encrypted and stored into the container file. This method leads to the data being stored in the container in the encrypted form at all times.

In practice, it is done in the following way. BestCrypt Driver intercepts any I/O operation directed towards a device managed by BestCrypt and performs the necessary encryption or decryption in addition. Therefore, when reading, the driver reads the requested data from the container file, decrypts it and passes it out as the result. Similarly, when writing, the driver first encrypts the data and then writes it into the container file.

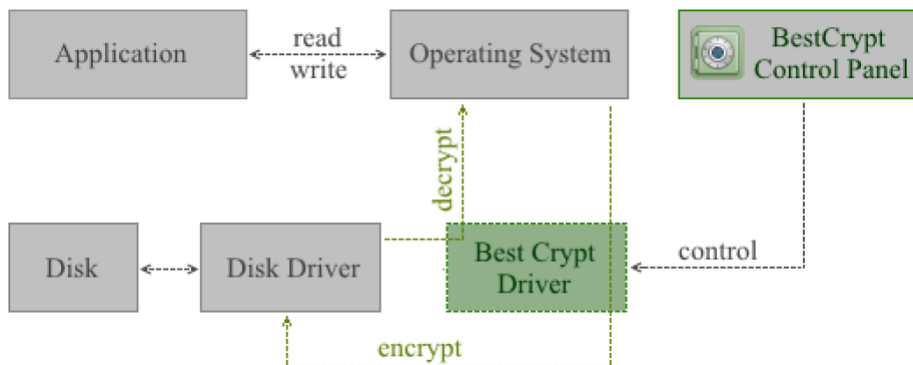


Figure 2.1: BestCrypt realisation of the transparent encryption [1].

Another result of this is that any application can use the container data in the most simple way. When the password is verified and the container is mounted, there is no more authentication required. Since the BestCrypt Driver intercepts all the I/O commands directed towards the mounted con-

tainer, it does not matter which application requests the access. Therefore, all applications have the access to the data until the container is dismantled again using the BestCrypt Control Panel.

2.2 The encryption modes

Three encryption modes, often known as block cipher operation modes, are supported, though not every algorithm is available for every mode. The relation between ciphers and the supported modes for them can be seen in table 2.1. All of the operation modes are briefly described in this section.

2.2.1 CBC

The first and probably the best known supported operation mode is the Cipher block chaining mode [19]. Images 2.2 and 2.3 show the encryption and decryption schemes.

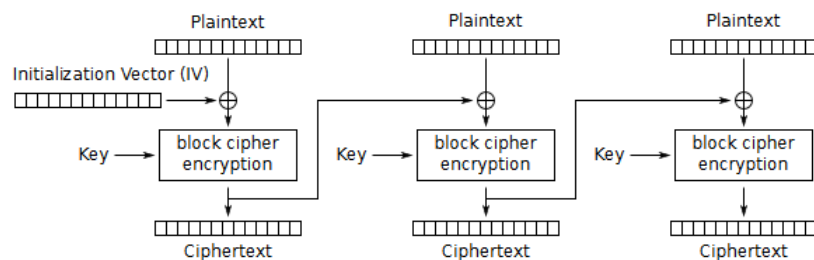


Figure 2.2: CBC mode encryption scheme [2].

The disadvantage of this encryption scheme is that it allows only sequential processing. Every block can be encrypted only with the knowledge of the previous encrypted block. Therefore, encrypting larger amount of data can take significantly more time than using the other modes. In addition, if any bit of the plaintext (or IV) is corrupted, it does not only affects the current ciphertext block, but all the following blocks too, because the corrupted ciphertext block is being propagated further.

On the other hand, decryption can be done in parallel, because the correct plaintext block can be recovered from two consecutive ciphertext blocks. While decrypting, the corrupted ciphertext block affects only the current and the following block. Therefore, with the correct key and one incorrect ciphertext block, only two plaintext blocks will be decrypted incorrectly.

Because of the fact that a correct block of plaintext can be obtained from two consecutive ciphertext blocks, the mode itself is often modified by other mechanisms, such as propagating the IV or processed blocks forward to af-

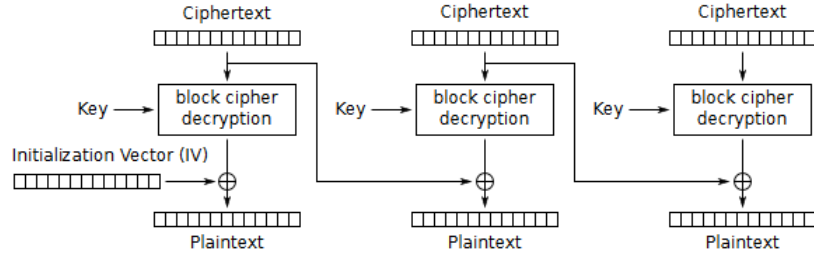


Figure 2.3: CBC mode decryption scheme [3].

fect a wider area of data. The most common modifications are PCBC and CBCC [20].

2.2.2 LRW

The Liskov, Rivest, Wagner mode [21] is an example of a *tweakable* cipher. It uses two keys: encryption key K and an additional key (*tweak*) F . The tweak serves in the place of an IV in the CBC mode. K is the key the block cipher uses and F is a key of the length of a block. The encryption of a block P with index I into block C is defined as follows

$$X = F \otimes I$$

$$C = E_K(P \oplus X) \oplus X$$

All the operations are done in $GF(2^k)$, where k is the length of the block. From the encryption rules, it is obvious that no two blocks are mutually dependent. The encryption of one block requires only the computation of the correct value X based on the tweak. The decryption rules would be similar, replacing the encryption for decryption and reversing the process.

2.2.3 XTS

The XTS mode [22] represents another tweakable cipher. It is a mode designed specifically for disk encryption purposes. It differs from the previous modes in using two keys, both for encryption. One key is used to encrypt the sector number, the second one is used for the remaining encryption operations. Mathematically, the encryption rules for encrypting block j with keys K_1 and K_2 of sector i are the following

$$X = E_{K_2}(i) \otimes \alpha^j$$

$$C = E_{K_1}(P \oplus X) \oplus X$$

The element α is the primitive element in $GF(2^{128})$. Again, each block can be encrypted independently, except for the last two where ciphertext stealing takes place. The size of key K_2 is the same as the size of a block, K_1 is the key for the block cipher.

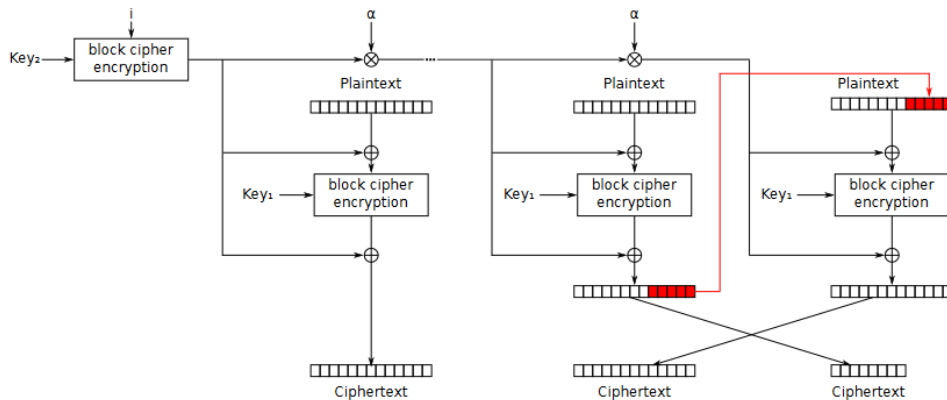


Figure 2.4: XTS mode encryption scheme [4].

The principle does not differ much from the LRW mode. The difference is in the usage of the tweak (second key). LRW uses it instead of the primitive element α while XTS uses it to encrypt the sector number.

2.3 Algorithm drivers and keys

Each encryption algorithm is represented by a driver. The reason for it is that drivers are located in kernel mode, so they are not easily accessible from user mode. Therefore, each driver offers the implementation of one specific algorithm and each functionality is requested via communication with that driver.

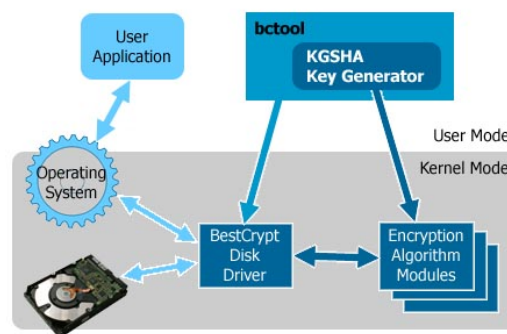


Figure 2.5: BestCrypt architecture [5].

It was said in the beginning that a password is necessary for creating a container. Similarly, a cryptographic key is required to decrypt container

data. This key is derived from the password and other parameters upon the container creation. With the knowledge of the key, one can decrypt the whole container even without knowing the password. The method of how the key is generated is described in chapter 4 in detail.

To protect the generated key, BestCrypt uses an approach that guarantees the key can not be easily recovered from memory: it introduces the *key handles*. A key handle is a unique identifier for a cryptographic key. Since a key is associated with a specific algorithm, the keys are stored inside the driver of the algorithm they belong to. Upon their creation, a handle is associated with them. When the key should be used for encryption or decryption, only the key handle is passed to the algorithm driver. The driver then takes the correct key from a memory storage and uses it for the necessary operations.

The content of the drivers is not be described in the thesis, because it would require to reverse engineer them. The focus is rather their functionality as a black box.

2.4 Functionalities

The functionalities listed below represent the basic possibilities BestCrypt offers for a container. All of them can only be performed while the container is not mounted.

The mentioned functionalities are all connected with cryptography very tightly and are mentioned because the analysis examines them. BestCrypt offers more advanced functionalities like the key files mechanism or some special settings of the software itself. All these functionalities are described in the BestCrypt documentation [1], but since they are not important for the purpose of the analysis, they are not explained here.

2.4.1 Hidden containers

A hidden container can be understood as a container within another container. When the user is asked for password when mounting such container, he has two choices. If he enters the password for the original (outer) container, the container behaves just as expected and offers the content of the outer container. When the hidden container password is entered, BestCrypt recognizes it and mounts the hidden container. The data stored inside the hidden container is different from the data in the outer one. The outer container does not show in any way a presence of a hidden container nor that the container would contain some other key.

When a user would be somehow forced to give up the container password, he can give the password to the outer container. It will look like everything is in order and the intruder will not be able to get to the hidden container data or even find out if there is any hidden container.

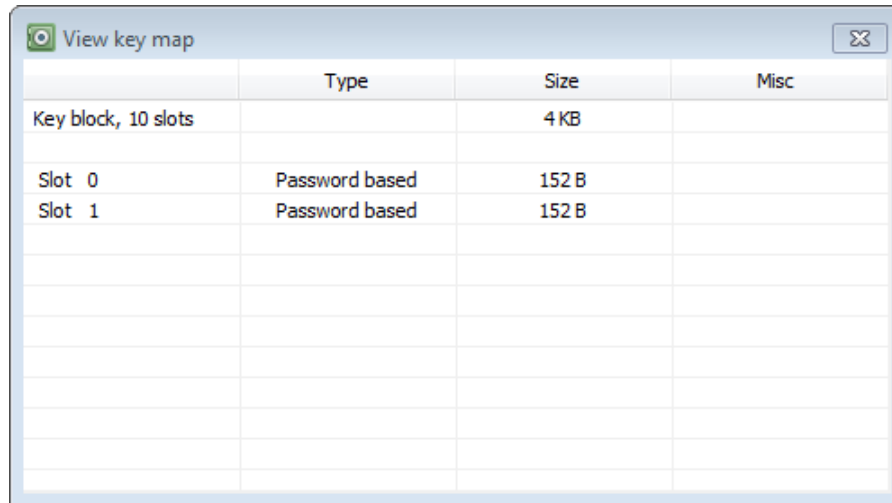
The limitation is that once a hidden container is created, no data should be written into the original container. No one, not even BestCrypt, will have any knowledge of whether there is a hidden container, therefore its data cannot be protected. When writing into the original container, the data could overwrite the hidden data and damage them or destroy the whole container.

2.4.2 Re-encryption

BestCrypt allows the user to re-encrypt the whole container. Besides being able to change the algorithm and encryption mode, the user needs to choose a new password. The data will be kept unharmed, but probably needless to say that any hidden containers will be lost, since there is no way to protect them. For larger containers, this process may take quite some time, because the whole container needs to be encrypted all over again.

2.4.3 Password management

BestCrypt offers several ways to manage passwords. First of all, the list of information about all keys associated with the original container can be displayed. This information consists of their number, type and the key block size.



The screenshot shows a window titled "View key map" with a close button in the top right corner. The window contains a table with the following data:

	Type	Size	Misc
Key block, 10 slots		4 KB	
Slot 0	Password based	152 B	
Slot 1	Password based	152 B	

Figure 2.6: Key map view.

A password can be changed as well as removed, assuming the user has a password to access the container. New passwords can be added as well due the possibility of having multiple user accesses to the container data as mentioned earlier.

2.4.4 Header encryption

Another feature related to cryptography is the possibility to encrypt the container header. The container files (as is described in section 3.2) have a part of its data decrypted by default. This part is called the header and BestCrypt uses it to identify decryption parameters and to find container files on disk.

The header gives away some important information and enables anyone to easily distinguish a container file from any other file. This functionality encrypts the header too, resulting in a container file consisting of what looks like random data.

2.5 Installation

There is nothing special about the installation process itself. Nevertheless, two things should be mentioned as they may not be expected

1. The drivers are placed into the Windows system directory
2. Drivers information is written into the Windows Registry Database

The algorithm drivers and the BestCrypt driver will require a system restart as well to be loaded into memory.

display name	algorithm ID	key length	block size
IDEA	32	128	8
DES	64	64	8
CAST	224	128	8
BLOWFISH	128	256	8
BLOWFISH-128	128	128	8
SERPENT	232	256	16
RC6	230	256	16
GOST	170	256	8
Triple DES	208	192	8
BLOWFISH-448	176	448	8
TWOFISH	160	256	16
AES	240	256	16

Table 2.2: Algorithm parameters from Windows Registry Database.

`LOCAL_MACHINE/SOFTWARE/Jetico/BestCrypt/Algorithms` is the important registry key where BestCrypt stores information for algorithm drivers. The information consists of the service name (used when communicating with the driver), the display name, algorithm ID, key length and block size. Table 2.2 shows the most important information useful to further usage. Note that

on systems with active UAC the registry key is virtualized and therefore the location will be different.

2.6 Usage

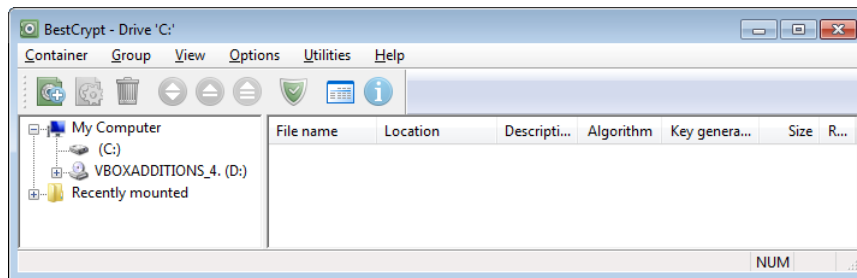


Figure 2.7: BestCrypt Control Panel.

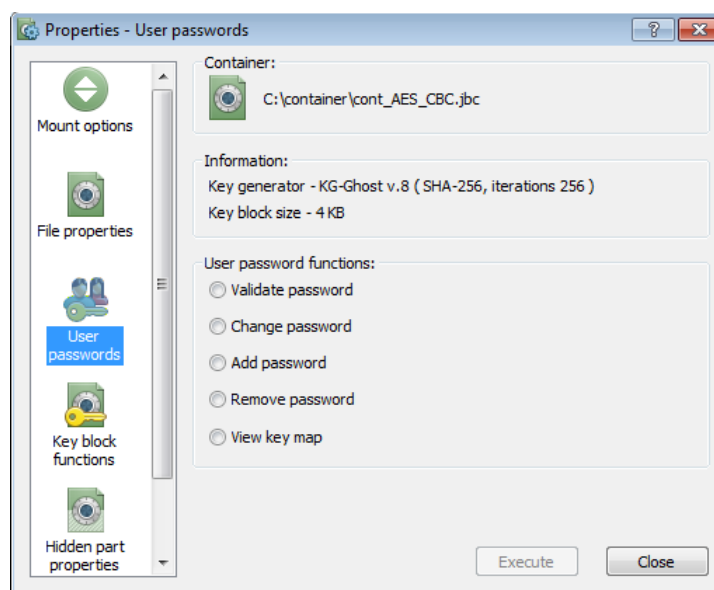


Figure 2.8: BestCrypt Container Properties window.

This section briefly explains how to use BestCrypt. A BestCrypt Control Panel is the window that appears after running BestCrypt. Everything can be done via this panel. To create a container, use *Container*→*New*. In the *Advanced options* part of the first window to appear, the encryption algorithm and operation mode can be changed. Upon clicking *Create*, a password is requested. Again, under the *Advanced options* parameters related to

2. BESTCRYPT

the container protection (type, salt, hash algorithm) can be modified. Finally, BestCrypt generates a random seed by random key pressing or mouse movement.

The right part of the BestCrypt panel will show available containers. If the container to mount is not there, BestCrypt offers to let the user browse the file system and find it or let BestCrypt find available containers in a specified path. To mount a container, right click it and choose *Mount* or double click it.

Once a container is mounted, not much can be done besides manipulating and using its data. However, once it was dismounted, the user can right click it and choose *Properties*. That is an option that hides all the main additional functionalities like password managing, hidden part creation or re-encryption. For notes on how to use them please refer to the BestCrypt Container Encryption guide [1].

BestCrypt Development Kit

The BestCrypt Development Kit (BDK) [23] is a library Jetico released to enable anyone to verify the safety and correctness of BestCrypt. It contains the source codes of the key generation, the encryption and decryption processes and other useful information. Additionally, it contains important definitions of structures that are used to represent the container file data correctly.

That said, it is probably clear why the BDK plays a major part in understanding how BestCrypt containers work and how to use the container files. It is the main source of information about how the container is processed and it describes the details of transforming a password into a key. Therefore, this section is devoted to a deeper explanation of how it works and how to use it. Unfortunately, the BDK lacks some detailed reference or documentation. The help file associated with it uses links that are no longer active so one has to more or less rely on the source code comments that in some cases do not tell much.

It is important to mention right at the beginning that the BDK was released when BestCrypt version 8 was active. Therefore, containers created by a newer version of BestCrypt will not be acknowledged as correct. Keeping this issue in mind, Jetico offers an option to create version 8 compatible containers. When choosing a container password, the *Advanced Settings* allow the user to do so, as shown on picture 3.1. Version 8 compatible containers are also cross-platform compatible, unlike the newer containers, which makes them still useful for a variety of users; by no means can they be considered obsolete.

3.1 Installation

The zip archive containing the BDK can be downloaded from the official BDK website [23]. The archive contains two main folders. The first one, `doc`, contains the help file, but as mentioned earlier the links are no longer active, so it does not offer much. The second folder, `sources`, contains five folders of

3. BESTCRYPT DEVELOPMENT KIT

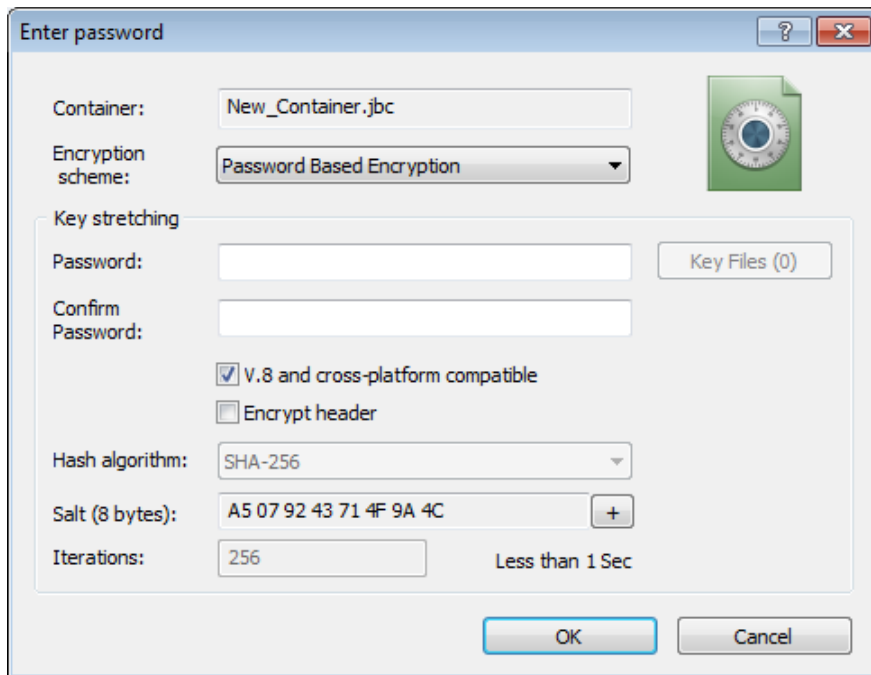


Figure 3.1: Creating version 8 container.

source codes divided according to the area of usage. The description of what each folder represents can be found in the `!readme.txt` file.

For building the library on Windows, the best way is to use the Microsoft Visual Studio solution `sources/kgghost.sln`. Simply build the solution after loading it. Administrator privileges are required, otherwise it fails with insufficient write privileges.

When the build finishes, the result can be found at `C:/Program Files (x86)/Jetico/BestCrypt`. Overall four files are created:

1. `KGghost.ilc`
2. `KGghost.dll`
3. `KGghost.exp`
4. `KGghost.lib`

The most important are the `KGghost.dll`, which is the dynamic library that provides the BDK functionalities, and the `KGghost.lib`, which is the import library for the DLL.

In some versions of Visual Studio the error *"cannot open include file `afxres.h`"* can appear. In that case the easiest solution is to add Microsoft

foundation classes. This can be done by going to *Control Panel*→*Programs and Features*→*Microsoft Visual Studio Professional*→*Change*→*Modify* where the *Microsoft Foundation Classes* has to be added.

The BDK requires BestCrypt to be installed on the computer as well. BDK functions use the algorithm drivers and these need to be provided by BestCrypt. Besides the usage of the drivers, no other relation to BestCrypt has to be explicitly specified.

3.2 BestCrypt container structure

part	offset	field name	content
DATA_BLOCK	0	header	Basic container info
	704	reserved	Reserved area
	1024	pool	CBC mode pool
	1536	key block area	Key blocks
Container data	4096	sector 1	Master Boot Record
		...	
		sector n	Last sector

Table 3.1: Container file structure scheme.

As mentioned earlier, BestCrypt creates the so called containers. Using the BDK, the overall structure of a container can be determined.

The table 3.1 shows the overall structure of the container file. The second part contains simply disk sectors one after the other in encrypted form starting with the Master Boot Record. Because the content of this part depends entirely on the data inside the container, nothing else can be said about it without decrypting it.

The first part, the `DATA_BLOCK`, contains all the necessary information about the container and its key(s). This structure is defined in the BDK as follows.

```
struct DATA_BLOCK
{
    encBlock<dbHeader>    header;
    BYTE reserved_1K[1024 - sizeof(encBlock<dbHeader>)];
    BYTE pool[512];
    BYTE keyTab[10][256];
}
```

The first part is the `dbHeader`. By default, the header is decrypted, but BestCrypt offers a way to keep the header encrypted too. It contains mainly the encryption algorithm identifier, the encryption mode identifier and the hash algorithm identifier. The `dbContInfo` determines the offset of the beginning of the second part and its length within the container file.

3. BESTCRYPT DEVELOPMENT KIT

Since the BDk works with version 8, the only acceptable hash algorithm is SHA-256. The algorithms identifiers were listed in table 2.2 and the encryption mode identifiers are defined in the BDk.

```
struct dbHeader
{
    BYTE   busiedFlag [3];
    char   signature [8];
    BYTE   containerID [4];
    char   busiedName [28];
    char   volumeLabel [11];
    DWORD  keyGenId;
    DWORD  version;
    wchar_t description [descriptionLength];
    dbContInfo   contInfo;

    DWORD  contAlgId;      // Encryption algorithm
    DWORD  encModeId;     // Encryption mode
    DWORD  hashAlgId;     // Hash algorithm

    dbKeyId   keyMap [DB_maxKeyQ];    // Key block information
}
```

The `keyMap` holds the used keys parameters. For each key it stores its size, type and a parameter. The type determines the encryption scheme. In this analysis, the password based encryption is the only acceptable scheme.

The pool of the `DATA_BLOCK` is very important when using the CBC encryption mode. It is used to modify initialization vectors and also plays a part in sector decryption. Section 6.8.3 describes its usage in detail.

The third important field of the `DATA_BLOCK` is the `keyTab`, which will be referred to as the key block area. It holds the encrypted key blocks for the keys specified by `keyMap` in the header. Each key block holds a key information structure and is explained in sections 4.2 and 4.3.

3.3 BDk and container files

One important thing needs to be mentioned here. The BDk does not change the container file in any way. In some cases it modifies the `DATA_BLOCK`, but only in memory. That is why the `DATA_BLOCK` is required as a parameter to many functions, as will be seen in the following section. When a function requests a container file name as well, it is because it will display a dialogue window and use the name to display what container it is dealing with.

An example can be seen when changing the container password. The BDk asks the user to enter current password, new password and changes it successfully by modifying the key block in the `DATA_BLOCK`. But without actually rewriting the container file with the newly created `DATA_BLOCK`, the change will not be promoted into the container file itself and the old password will be the correct one.

3.4 Functionalities

This section concentrates on what can be achieved using the exported functions. The BDK offers 18 exported functions. Not all of them are necessary for the purpose of the analysis and some have been left there for compatibility reasons.

The `CreateKeyHandle_V2` function serves many purposes. Using different flags, it can do the following:

- Create new `DATA_BLOCK`
- Load and verify a key to a container
- View content of the key block
- Add, change or remove container password

This is the most important function for the purpose of this analysis since the encryption or decryption is done inside the algorithm drivers that are not available.

When loading and verifying a key, the whole key generation process is shown. One can see the key generation, the verification steps to ensure the key is correct, the final key check and the key creation. All these information are most useful and this function is used to describe the process in detail in chapter 4.

Creating a new `DATA_BLOCK` may be useful for better understanding of each field of its structure. When creating one, a new password needs to be specified, since it is in fact creating a new container. The cryptographic parameters are not requested by a dialogue, as they are when creating a container using `BestCrypt`, but rather retrieved from the parameters passed to the function.

Viewing the key block content means viewing how many keys of which types are associated with the container. Manipulating passwords can be useful to observe changes in the key block. It corresponds to the same functionality offered by `BestCrypt` in container properties described earlier.

The `DataBlockAllocate` and `DataBlockFree` functions can be used to allocate and destroy the `DATA_BLOCK` structure. These are simple functions to allocate and free the space for a `DATA_BLOCK`.

The `FreeKeyHandle` function releases the previously created key. It is done via sending the `IOCTL_PRIVATE_FREE_KEY_HANDLE` command to the algorithm driver. As mentioned earlier, `BestCrypt` does not keep keys stored in user space, but rather in the kernel space of the algorithm driver. Calling the `CreateKeyHandle_V2` returns the key handle; when destroying the key, the handle is passed to this function and the key is released and no longer available.

The `UpdateHeader` function encrypts or decrypts the header part of the `DATA_BLOCK`. This corresponds to the `BestCrypt`'s feature to encrypt the container header, so even the container information will not be visible. However,

BDK requires the user to know the encryption parameters or try all the combinations manually in contrast to BestCrypt where this is done automatically.

The `CreateHiddenPartEx` is related to the hidden container functionality. It corresponds to creating new hidden part inside the container. The process is exactly the same as when doing this in BestCrypt. A seed is generated, new key is created from it and stored inside the `DATA_BLOCK`. No container file modification is done, therefore the hidden part will not really exist. It is important to note here that the `contInfo` in the parameters structure has to have the length set to 4096. The reason for this requirement is explained in chapter 4.

3.5 BDK sample illustration

To use the BDK in any application, just link the library to it. If the application should use the BDK structures as well, change the include directory to the root folder of the source codes.

A small program was developed to demonstrate the basic BDK usage. It offers the basic operations like key verification, password managing, header encryption or `DATA_BLOCK` creation.

One has to provide a container file created by BestCrypt as an argument. The program will not modify the file in any way, but will use it to load the `DATA_BLOCK`. The program simplifies the understanding of each specific parameter by deriving it from the container file and user input. The container file is not required for new `DATA_BLOCK` creation.

Key generation process

The key generation process is a process of creating a master decryption key from a password. Two main terms need to be understood regarding this process. The first term is the master key. That is the key stored in encrypted form inside a key block. It is used for the whole container data encryption and it can be recovered by decrypting the key block using the password key. The password key is a byte sequence generated from the password and the salt.

The BDK offers the `CreateKeyHandle_V2` function that can be used to retrieve a key handle. The process includes the key generation so it is a great source of information for understanding how it works in detail.

```

BOOL CreateKeyHandle_V2(
    HWND hWnd,
    const wchar_t* text,           //container file name
    const wchar_t* caption,       //dialog window caption
    DWORD contAlgId,
    DWORD createFlag,
    BYTE **pDataBlock,
    DWORD *pDBSize,
    DWORD *pKeyHandle,
    DWORD *pErrCode,
    DWORD *pFlags,
    sCKHParams_MP_AI *pParams     //the container parameters
);

```

First, let's examine the function's parameters. The `hWnd` is a handle to the parent window of the password dialogue that will appear. This is meaningless to the analysis and can be set to `NULL`. The `text` and `caption` represent strings that will appear in the dialogue window and have no greater meaning as well. The `contAlgId` defines the encryption algorithm. The `createFlag` determines what the function will do (see the BDK functionalities in section 3.4). The `pDataBlock` and `pDBSize` hold the `DATA_BLOCK` and its size. The `pKeyHandle` will contain the key handle if the function succeeds.

It may be noticed that in the function declaration here, a different parameter type of the parameters than in the BDK is used. The reason for it is that the `sCKHParams_MP_AI` is actually used, being a subclass of the original that was used in the previous versions. It is important that the parameters are filled with correct values, since the function relies on them.

When the `CFLAG_VERIFY_AND_LOAD_KEY` flag is used, the function does the following:

1. Ask for the password to the container
2. Generate the password key
3. Decrypt the master key data using the password key
4. Verify that the process succeeded
5. Store the key inside the algorithm driver
6. Return the handle to the created key

Amongst the steps mentioned above, data integrity is checked multiple times. These steps are noted as well.

4.1 Password retrieval

There is nothing special about retrieving a password. The important part is the form the password must be transformed to afterwards. The password must be a big-endian widechar string terminated with a zero double-byte. That means, if the password is *"abc"*, the retrieved password will take the form of byte array `[0x00, 0x61, 0x00, 0x62, 0x00, 0x63, 0x00, 0x00]`. The ending zero is regarded as a part of the password.

4.2 Password key generation

Password key is the first step. It is generated from the password and salt, but before going further it is necessary to clarify how the key block area works, so that we know where to find the necessary data and how to deal with it. When describing the `DATA_BLOCK`, it was explained that the header contains key blocks information. A password based key block has the following structure:

```
struct PBE_KeyBlock
{
    BYTE salt[8];
    encBlock< sKeyInfo > keyInfo;
};
```

The key block area consists of key blocks and random data. How do we tell which one is correct when there can be more of them? The function goes through all the key blocks and tries them all until all were tried or the decrypted data were correct. Why examine all the key blocks and not just the ones defined by the header? Because hidden containers keys are stored in key blocks too, but they are not mentioned in the header.

The length of the password key is equal to the length of the key that the chosen encryption algorithm requires with regard to the encryption mode.

```

0x00709420 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
0x00709430 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
0x00709440 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....
0x00709450 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 .....

```

Figure 4.1: String D.

```

0x007094A0 2e a2 15 6e 4b 53 c3 4e 2e a2 15 6e 4b 53 c3 4e .φ.nKSÄN.φ.nKSÄN
0x007094B0 2e a2 15 6e 4b 53 c3 4e 2e a2 15 6e 4b 53 c3 4e .φ.nKSÄN.φ.nKSÄN
0x007094C0 2e a2 15 6e 4b 53 c3 4e 2e a2 15 6e 4b 53 c3 4e .φ.nKSÄN.φ.nKSÄN
0x007094D0 2e a2 15 6e 4b 53 c3 4e 2e a2 15 6e 4b 53 c3 4e .φ.nKSÄN.φ.nKSÄN

```

Figure 4.2: String S.

```

0x00709520 00 70 00 61 00 73 00 73 00 00 70 00 61 00 73 ...p.a.s.s...p.a.s
0x00709530 00 73 00 00 00 70 00 61 00 73 00 73 00 00 70 ...s...p.a.s.s...p
0x00709540 00 61 00 73 00 73 00 00 70 00 61 00 73 00 73 ...a.s.s...p.a.s.s
0x00709550 00 00 00 70 00 61 00 73 00 73 00 00 70 00 61 ...p.a.s.s...p.a

```

Figure 4.3: String P.

That concerns the LRW and XTS modes, because they use two keys. Both keys need to be retrieved, so the password key will have the length of a sum of both key lengths.

Let's see how the key is derived. The salt is taken from the beginning of the password based key block. The diversifier string D is created. It is a 64 bytes long string of values 0x01. The salt string S is a 64 bytes long string of the salt bytes repeating until the required length is reached. Similarly, the password string P is also a 64 bytes long string, but consists of copies of the password. It is important to notice the format of the password described earlier and the fact that the ending zero double-byte is considered a part of the password. For more understanding refer to the images 4.1, 4.2 and 4.3.

The S string is concatenated with the P string into the I string. The following operation is then performed.

```

int c = (pwdKeySize + digest_size - 1) / digest_size;
Buffer A[c * digest_size];
Buffer B[v];

for( i = 0; i < c; i++ )
{
    BYTE *Ai = A + i * digest_size //current position
    Ai = H(H(H(...D|I))), 256 times

    for ( i = 0; i < v; i++)
        B[i] = Ai[i % digest_size];

    for ( j = 0; j < I.bufferSize; j += v )
    {
        BYTE *Ij = I.buffer + j;
        Ij = ( Ij + B + 1 ) mod 2^v
    }
}
memcpy(passwordKey, A.buffer, pwdKeySize)

```

This pseudocode is in a signified form, but the result is the same as should be. Buffer A holds the password key at current state, buffer B is used to modify the I string after every iteration. The main step is the first one where strings D and I are joined and their hash is calculated 256 times. This value is then stored as the current `digest_size` bytes of the buffer. Based on that, buffer B is created and the whole string I is modified by it for the next iteration.

If you want to look up the original password key generation process in the BDK, it is a function `P12_producePseudoRandomBytes` in the `pkcs12_funcs.cpp` file.

4.3 Master key generation

Using the password key, master key can be obtained. To do that, the key information needs to be decrypted. It can be found as the `encBlock<sKeyInfo>` part of the `PBE_KeyBlock`. For the decryption process, the initialization vector IV is required. The IV is located in the `m_tail` field of the `encBlock<sKeyInfo>`.

```
template < class TT > class encBlock
{
    TT          m_body;
    BYTE        m_reserved[ RoundTail( sizeof( TT ), DB.MaxEncBlockSize ) ];
    EBTail      m_tail;
}

struct EBTail
{
    BYTE  m_digest[32];
    BYTE  m_iv[16];
}
```

The IV is required for the decryption. It is stored in the `m_tail.m_iv` field in plaintext form. When decrypting the `encBlock<sKeyInfo>` structure, the IV is left out and the rest of the data is decrypted. After decryption, `m_body` contains the decrypted `sKeyInfo` that holds the key information. Besides the fields already understood, the master key itself can be found in the `key` field. That is the first time the master key appears in memory.

```
struct sKeyInfo
{
    BYTE  size;
    BYTE  hashAlgId;
    DWORD keyAlgId;
    DWORD encModeId;
    dbContInfo contInfo;
    BYTE  key [64];
}
```

Why is the container info here as well when it was already present in the `dbHeader`? Because the key could belong to a hidden part. If so, the container information would differ and hold the information where the hidden part starts¹.

4.4 Correctness verification

After this process, its correctness is verified. This is done by calculating the hash value of the `m_body` field of the key block and checking the value against the `m_tail.m_digest` field. Note that the latter was decrypted as well in the previous step. The used hash algorithm is the one specified in the `DATA_BLOCK` header. If the values match, the result is considered correct.

The BDK shows that a further verification using a verification block is used. That part is left out of the analysis, but can be examined using the library.

¹The hidden part key information will have the container length always set to 4096. That is how it is determined whether it is a key for hidden part in the BDK.

4.5 Key store

As mentioned earlier, BestCrypt uses key handles instead of plain text keys because of higher security. When the master key is successfully generated and classified as correct, the key bytes are sent to the algorithm driver and the key handle is created. The driver itself has the key stored and associated with the correct handle. The `IOCTL_PRIVATE_CREATE_KEY_HANDLE` command is used to create the key.

The only time the key exists in user mode memory is from its decryption by the algorithm driver until its correctness is verified and it is actually created. This approach prevents the necessity of creating an unverified and possibly invalid key and having to destroy it right after. The downside of it is that the key actually appears in memory, which could be avoided. The key info could be decrypted by the algorithm driver and the key stored immediately so only the handle would be returned and not the actual key bytes.

4.6 I/O commands

```

BOOL DeviceIoControl(
    HANDLE         hDevice ,
    DWORD         dwIoControlCode ,
    LPVOID        lpInBuffer ,
    DWORD         nInBufferSize ,
    LPVOID        lpOutBuffer ,
    DWORD         nOutBufferSize ,
    LPDWORD       lpBytesReturned ,
    LPOVERLAPPED lpOverlapped
);

```

Now when the whole process is known, let's see how the I/O commands are used. The `DeviceIoControl` function takes the control code, input buffer and output buffer. Let's examine how it is used during the two commands sent during the key generation – decryption command and create key command.

The `IOCTL_PRIVATE_ENCRYPT_DECRYPT_V8` is used for the encryption and decryption. In BDK, the `Alg_Decrypt` function in the `alg.cpp` file executes the command along with correct data passing. The input buffer is used to tell the driver the encryption parameters. It contains mainly the requested operation id, the key handle, the IV and the operation mode. The output buffer contains the actual data to encrypt or decrypt.

The `IOCTL_PRIVATE_CREATE_KEY_HANDLE` is used for the key creation. In BDK, this operation is done by the `Alg_GetKeyHandler` function in the `alg.cpp` file. It is a bit different in this case. Both input and output buffers contain the same information. The data that are passed consists mainly of the key and key length.

BestCrypt security

Now that the functionalities and their usage and limitations and container structure have been explained, let's look at how BestCrypt provides the security it claims to offer. The security is evaluated based on the user interface and the actions are compared to the corresponding BDK source code whenever it is possible. All the analysed parts are related to cryptography.

Specifically, the covered topics are the password strength, the default cryptographic parameters, security against making the container data accidentally inaccessible, the password management and network communication. Random number generation is examined as well with regard to the BDK.

It is also important to mention that by the time the thesis was finished, the BestCrypt Control Panel version 9.02.9 and BestCrypt driver version 4.54 were the most recent software versions. The current version can be found in *Help* → *About BestCrypt*.

5.1 Default encryption parameters

The first possible weakness could lie in the default encryption settings. An inexperienced user who wants to create a container is not likely to change the advanced settings. Therefore, the security depends on the default settings and these should provide the ideal level of security. By default, BestCrypt offers the AES encryption in the XTS operation mode.

The default key generation parameters are hash algorithm Whirlpool-512 with 16384 iterations. Salt is 32 bytes long and is created randomly. These parameters are valid for the newest BestCrypt version. Since the thesis focuses on version 8, it is important to observe how this settings change when creating the version 8 compatible container: the only acceptable hash algorithm is SHA-256 and the only possible iterations count is 256 then.

The default encryption parameters change when the user changes them and clicks *Create*. After that, the software remembers the changed parameters

and offers them as default ones for future attempts. For ciphers that do not support the XTS mode, LRW is the default choice.

According to NIST password key derivation recommendations [24] from 2010, the password key length should be at least 112 bits long, salt should be at least 128 bits long and the iterations count should be at least 1000. Let's observe how well are these conditions met.

Since all the encryption algorithms use keys at least 128 bits long (except DES, which is not available unless added manually), the first requirement is met, because the password key is of the same length as the encryption key (or double the size for the XTS mode). Note that the password key is the concern here and not the master key, because master key is stored encrypted while the password key is generated.

The salt is 64 bits long which does not meet the required minimum. However, the too short salt in version 8 does not represent a significant problem. The general purpose of the salt is that two users with the same password do not have the same password key (or password hash). This aspect is important in applications that expect a large amount of users to log in, but not in BestCrypt where a container never holds two matching passwords.

The iteration count represents a more important problem. The version 8 iteration count is 256, which is even less than the NIST document recommends as a lower bound. It was an issue in one of the TrueCrypt analyses mentioned in chapter 1 where the authors considered 1000 iterations insufficient and recommended 100000 iterations. 256 iterations do not represent enough security against current brute force attacks and the count should be increased.

To illustrate why this is a problem, let's look at SHA-256 benchmark [25]. The data are to be represented as number of bytes that can be processed per second given the block size. The data to process at the beginning are at most $64 \times 3 = 192$ bytes long (algorithm with 32 bytes long key in XTS mode). If the speed should be for example 70 MB/s, the following calculation would apply:

$$P = \frac{70 \times 2^{20}}{2 \times (192 + 255 \times 32)} \approx 4394 [pwdKeys/sec]$$

This calculation corresponds to the algorithm presented in section 4.2. Note that this applies for situations where only one average computer is used to the brute force computation. If the password should be 8 characters long and consist only of the small characters of the alphabet, it would take at most D days to crack it by such computer based on the following computation (key block decryption is not taken into account):

$$D = \frac{26^8}{3600 \times 24} \approx 550 [days]$$

Additional algorithms can be added using the *BestCrypt Plug-in Manager*. Simply select an algorithm and check the *Use for new containers* checkbox. The changes become visible after restarting the Control Panel. The security of such algorithms is left for the user to consider.

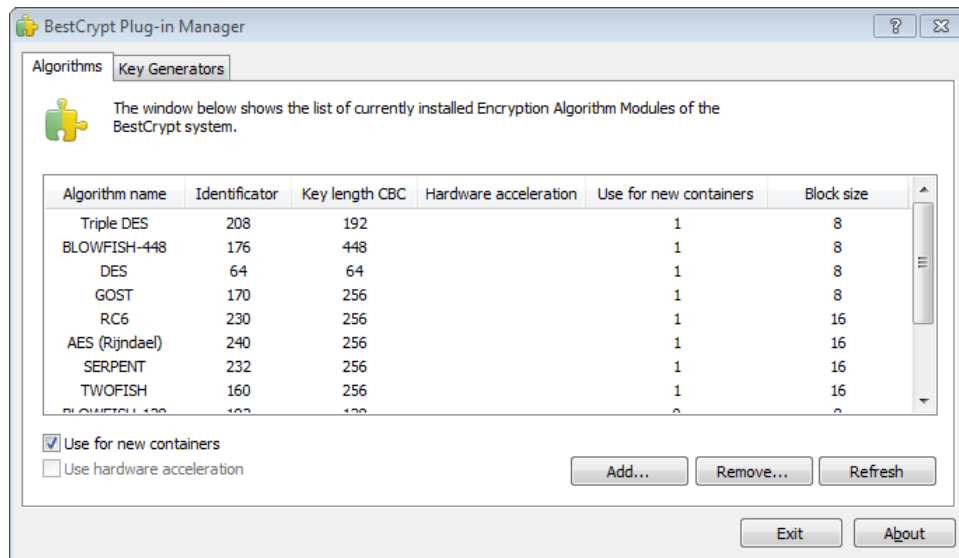


Figure 5.1: BestCrypt Plug-in Manager.

5.2 Password strength

Typical requirements [26] for password strength are that it has to contain at least one character from at least three of the following categories:

- Upper case letters
- Lower case letters
- A digit
- Non-alphanumeric characters

Additional requirements about the password length can be made as well. Usually the minimum password length is considered about 7 or 8 characters. Other well known requirements like the password age or password history do not apply to this kind of software. There is no password history kept nor any mechanism to determine the password age. These criteria apply to different type of applications.

BestCrypt requires the password to be at least 8 characters long and 256 characters long at most. This meets the typical requirements for password length. No shorter or longer password is accepted.

There are no requirements about the password form nor any indication about the password strength. It is left for the user to decide how strong

a password he wants to apply to the container. A password strength indication could help inexperienced users to create strong password, but it is just a possible recommendation.

5.3 Password manipulation

This section focuses on how the password manipulation processes are done. It examines how the access is verified and also observes how actions which would remove all access to the container are handled.

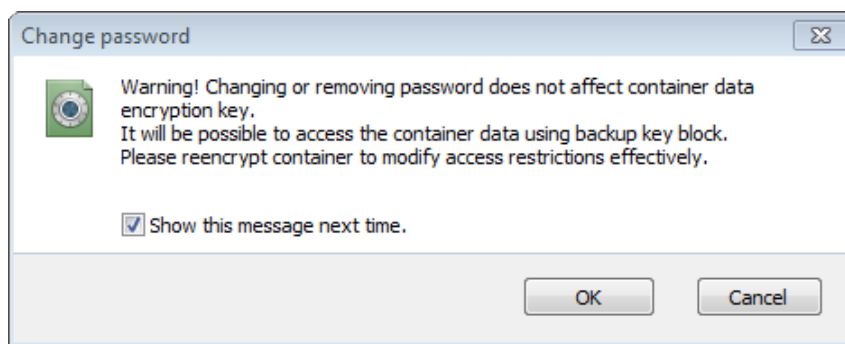


Figure 5.2: Warning when changing or removing a password.

The first tested scenario is changing the container password. BestCrypt warns user that changing the password does not affect the encrypted container data. That means the container will not be re-encrypted. This is important, because key block backups can be created and the container will still be accessible using the backups (and therefore the old password). The warning also prompts the user to re-encrypt the container manually to prevent any unwanted access. This is a correct approach since it informs an inexperienced user about the risk.

The same warning is displayed when the user tries to remove a password. The reasons for it are the same. Nevertheless, what if the user decides to remove the last password? That would mean that the container would become accessible only using a key block backup if one was created. This scenario is addressed as well and a corresponding warning displayed too.

BestCrypt does not allow two matching passwords associated with one container. The reason for it is quite simple. The container data do not differ when different passwords are used with the exception of hidden containers. So BestCrypt checks whether the password the user wants to add is already in use (by using it to decrypt all the key blocks and check if the decrypted data correspond to a correctly decrypted key information) and if it is, it prompts the user to enter a different one.

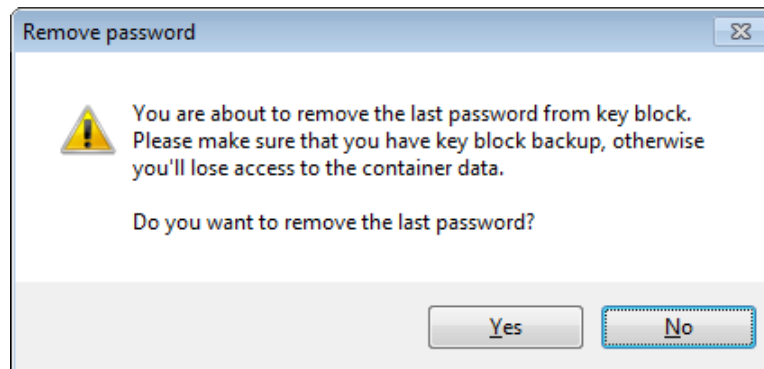


Figure 5.3: Warning when removing the last password.

5.3.1 User Interface interaction

All these actions are performed from the *Properties* menu of the container the operations are to be performed on. It was already said that this can be done only while the container is dismounted. When the user opens this menu and performs any of the password operations, he is asked to enter a container password first. This allows BestCrypt to verify the access right.

BestCrypt drops the authentication confirmation when the user closes the *Properties* menu. When opened again, the password is requested once again. The second discovered case when the authentication is lost is when a successful password change is completed, regardless of the number of passwords present in the container.

This is a correct approach since it seems that BestCrypt does not store the authentication confirmation in memory which could lead to some possible issues regarding faking the authentication process.

5.4 Container Re-encryption

While BDK provides a flag that could be used to execute the re-encryption process, it does not do anything. On the other hand, since BDK does not change the container files in any way, this is a logical behaviour.

Re-encryption can be executed from the *Properties* menu. The user is presented with four possible changes (see image 5.4). That seems like a good approach since the user can see the current settings and choose to change only the ones he decides to.

The only issue is based on the fact that not every algorithm is supported in every operation mode. Imagine the situation where user creates a container with AES-XTS and then chooses to re-encrypt. He goes to the *Properties* option and chooses to change the AES algorithm to TripleDES. XTS mode is

not supported for TripleDES algorithm, yet the settings allow such configuration. Upon clicking *Reencrypt*, an error reporting *”Undefined Key Generator Error”* appears. More thorough encryption settings validation could be performed here.

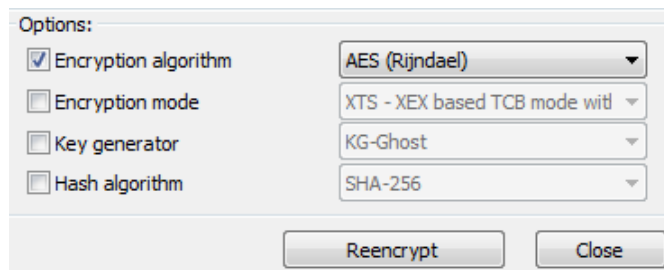


Figure 5.4: Re-encryption settings.

When re-encrypting a version 8 compatible container it is important to check the compatibility creation checkbox when entering the new password, otherwise the container will be changed to a current version and will lose its compatibility.

The re-encryption removes all the previous passwords and creates a new one. That is the only way because BestCrypt does not store the passwords in plain form which would be necessary to re-encrypt the corresponding key blocks as well.

5.5 Hidden container creation

To create a hidden container, one has to have the administrative privileges. It is required for the first part of the process where BestCrypt obtains a map of the container’s free space.

The BDK can be used to observe where the hidden keys are stored. The idea is that passwords to the outer part are stored one after another from the beginning and the hidden part keys are stored at the end of the key block area. When the password is being checked, the key blocks are traversed from the first to the last one.

With that in mind, if a hidden container should be created using the same password used for the original one, the hidden container would become inaccessible. The password would be recognized as a password to the original part before it would even come to checking it against the hidden part one.

To prevent the problem, BestCrypt checks all the present key blocks and verifies that the password is not used already. If it is used, a warning is displayed, informing the user and requiring him to choose a different password.

This is the same approach as when the user wants to add a password that is already present in the container.

The same behaviour can be observed in the BDK. There is no difference to what BestCrypt does. No flaw considering the hidden part creation was found, but there is an issue connected with the hidden part existence. It is explained in section 5.6.2 in detail.

5.6 Random number generation

Random values are used in several scenarios. They are required when creating a new container to fill the empty key blocks, to fill the key blocks of removed passwords or to generate the encryption key upon container creation. It is important to ensure that the randomness is good enough to make any kind of attack based on it impossible.

5.6.1 Encryption key generation

First of all, let's see how a master key is generated when a new container is being created. The first step is to generate a random seed. The generation reacts to the user's mouse movement and keyboard input and based on it, random data are created. The user is allowed to see the generation progress. The BDK contains this process as well. The functions `getRandomDlgProc` and `progressSubclassProc` present in the `GenerateSeed.cpp` file prove that the generation really relies on user activity, which is a good source of randomness. A total of 128 random bytes is generated.

The master key is generated in the same way that a password key is, with three differences: no salt is used, the seed is used in place of a password and only one iteration is done. The result is the key used for the encryption.

Jetico confirmed that this is the approach in the current version as well, with only two differences: the seed is longer (1 kB) and user is allowed to stop the process. This can be confirmed in the user interface - the process takes longer than when using the BDK and the *Stop* button stops the generation.

One of the analyses from section 1 mentioned a problem in TrueCrypt when the user was not able to see the random generation process and could create a container with poor randomness. In BestCrypt, the generation process is visible, but stopping it could be harmful. The seed is initialized to zeros and then filled from the beginning as BDK shows. When the process ends (or is stopped), the remaining bytes remain zero. This can lead to insufficient randomness and a possible attack, because the seed (used as a password) would be very short. Combining this with only one hashing operation could lead to obtaining the master key without even knowing the password. Anyway, it is important to stress out that this does not affect the keys that were generated using enough randomness generation at all.

This matter was brought to Jetico and they proposed a solution for the next release. The user will be able to stop the generation only after at least four times the key length bytes were generated. If the process is stopped, the rest of the seed will not remain zero, but will be filled using the same function that generates the password key. The source for the function will be the previously generated byte sequence. This solves the problem sufficiently.

5.6.2 Empty data initialization

When creating a `DATA_BLOCK` the `ShredData` function from the `Kblock.cpp` file is used. It is used as well when removing a password to wipe the corresponding key block.

```
void ShredData(BYTE* data, int dataLength)
{
    srand( (DWORD)time(NULL) ^ (DWORD)clock() );
    for(int i = 0; i < dataLength; i++){
        data[i] = (BYTE)rand();
    }
}
```

That is not the correct way to initialize data. There are many sources [27] claiming that the `rand()` function does not meet the requirements for a secure pseudo-random number generator. The period is 2^{32} at most, which is not sufficient compared for example to the Mersenne Twister algorithm² [28] that has the period of 2^{19937} .

The performed analysis discovered that this usage allows an attack. The attack does not lead to getting unauthorized access to the container data, but it allows to determine whether there is a hidden part inside the container or not. That violates the principal idea of a hidden part, because it should not be possible to detect its existence in any way³. Let's see how the attack works.

The key is not marked in the key block header, but it is stored encrypted in the key block area. The idea how to keep it safe is that an encrypted key is not recognizable from random data of the empty key blocks. However, using `rand()` allows an attacker to distinguish an empty key block from a stored encrypted key.

The `rand()` implementations differ, yet all of them allow 2^{32} possible seeds. Using `srand()` the seed can be set. The attack looks like follows

```
for (seed = 0; seed < UINT_MAX; seed++){           //try all the seeds
    srand(seed)                                   //set the seed
    for(i = 0; i < sizeof(keyBlock); i++){        //and check the key block
        if(keyBlock[i] != (BYTE)rand())
            break;
    }
}
```

²However, Mersenne Twister is not a cryptographically secure pseudo-random generator either.

³This is called *Plausible deniability*

```
Analyzing key block 0 (file offset 0x600)
Key block can not be generated using rand().

Analyzing key block 1 (file offset 0x700)
Key block can be generated using rand() with seed 740751

Analyzing key block 2 (file offset 0x800)
Key block can be generated using rand() with seed 2433679

Analyzing key block 3 (file offset 0x900)
Key block can be generated using rand() with seed 4388751

Analyzing key block 4 (file offset 0xa00)
Key block can be generated using rand() with seed 6605967

Analyzing key block 5 (file offset 0xb00)
Key block can be generated using rand() with seed 9085327

Analyzing key block 6 (file offset 0xc00)
Key block can be generated using rand() with seed 11826831

Analyzing key block 7 (file offset 0xd00)
Key block can be generated using rand() with seed 14830479

Analyzing key block 8 (file offset 0xe00)
Key block can be generated using rand() with seed 1319055

Analyzing key block 9 (file offset 0xf00)
Key block can not be generated using rand().

Container file analysis summary:
=====
Key block 0 contains a key.
Key block 1 does not contain a key.
Key block 2 does not contain a key.
Key block 3 does not contain a key.
Key block 4 does not contain a key.
Key block 5 does not contain a key.
Key block 6 does not contain a key.
Key block 7 does not contain a key.
Key block 8 does not contain a key.
Key block 9 contains a key.
```

Figure 5.5: Result of the attack on the `ShredData` function. The container contains one normal and one hidden key and the existence of a hidden part key is discovered.

If for any seed the key block matches the pseudo-random byte sequence, it almost certainly does not contain a key. Each number of the `rand()` sequence depends only on the previous one (seed in the case of the first number). Therefore, for every sequence there is a seed that can generate it. This is the reason why this attack works. The only problem could be with the `rand()` settings, but trying all does not increase the computation complexity.

This issue was brought to Jetico as well and they changed the empty blocks filling in the same way as the seed generation – using the same function that generates the password key. The source for the function will be a combination of system time and clock, which is similar to the current seed used for `srand()`.

5.7 Network connection

Observing BestCrypt behaviour regarding any network communication is important for assurance that container information is not being sent anywhere. This analysis observes network communication during container creation, its password manipulation, changing its name or location and deleting the container. Wireshark [29] was used as a tool to monitor the network traffic.

It should be mentioned first that BestCrypt works perfectly fine without any active internet connection. While this provides some assurance, it is not possible to conclude that no network communication would be done if a connection were available.

However, based on performed scenarios and the network traffic observations it is safe to say that BestCrypt does not communicate anyhow over the network. All the scenarios were tested while the traffic was being monitored and no communication was discovered.

5.8 Security conclusion

Let's summarize what this part of the analysis has achieved. BestCrypt security was tested. The focus was on the parts of the application related to cryptography. The analysis was done based on the user interface and the BDK was used whenever possible.

The analysis found no issues regarding the password manipulation. The password strength requirements are low, yet this does not lead directly to any issue. It could be helpful to add some password strength notification in some future release.

More serious problem was found regarding the hidden parts. It was discovered that currently the existence of a hidden part can be proved or disproved due to insecure empty key blocks filling. That stands against the main idea of it. A specific guide on how to reveal such information was presented, implemented and tested. The issue was consulted with Jetico and the authors acknowledged it and prepared a solution for the next release. The solution was presented as well and evaluated as sufficient.

A possible problem was found regarding the premature stopping of the seed generation process. The seed is initialized to zeros, so stopping the process too early could result in insufficient randomness and lead to attack on the encryption key itself. This problem was brought to the authors as well and in this case a solution was offered too. This solution was presented as well.

The analysis found no backdoor or suspicious behaviour in the areas it tested. No network communication was discovered during the container's manipulation and modification.

As a side effect, it was verified everywhere where it was possible that the BDK corresponds to what BestCrypt does.

Cryptographic processes re-implementation

To verify the cryptographic correctness, the key generation and data decryption have been implemented without using the BestCrypt algorithm drivers or any BDK functions. The process helped to really understand all the ideas behind the BestCrypt principles. All the basics of the created program and how it works are presented in this chapter as well as all the modifications that are required for correct behaviour but may not be obvious.

In chapter 1, some of the analyses used the source code to analyse the cryptographic processes and did not implement their own version of them. The reason why this is not a possible approach with BestCrypt is that the BDK contains source codes for the important parts of the application, but the whole software can not be built using it. Therefore the analysis can not just rely on it without verifying that the code corresponds to what BestCrypt and the algorithm drivers do somehow.

Re-implementing the encryption and hashing is the way to do so. The first step is to verify that the independently implemented key generation process generates the same key as the BDK. If this step is done, the second step is to try to decrypt the actual container data. Actual container data decryption is not documented in the BDK and relies on different verification mechanisms.

6.1 Covered areas

Two main areas need to be implemented independently to enable the cryptographic processes to work correctly – decryption and hash functions. Only the SHA-256 hash function is necessary for previously mentioned reasons. As for the encryption algorithms, all of them are well known and well documented so many cryptographic tools offer them for use.

All the three supported encryption modes are well known as well. The

problem is that since most of the tools focus on network security and the XTS and LRW modes are used mainly for disk encryption and almost never for network communication, almost no tool offers them. Since it would leave out two of the three supported modes, only the LRW mode was left out since it is older than XTS and in most cases XTS replaced it.

Due to issues not directly related to the analysis, the GOST algorithm is left out. All the remaining algorithms were tested.

6.2 Used tools

OpenSSL [30] is probably the best known cryptographic tool. However, it offers only AES-XTS encryption and several algorithms with the CBC mode, but using it would not cover even a half of the algorithm and operation mode combinations that BestCrypt offers.

Crypto++ [31] is used as the reference cryptographic library instead. None of the other well known cryptographic libraries provides all the required algorithms. *Crypto++* is widely used [32] and very easy to use in any application.

TrueCrypt source codes [33] are used for the XTS mode implementation. While this approach may seem to be improper, it causes no issue. While the TrueCrypt implementation of the operation mode is used, encryption and decryption is done using *Crypto++* and the XTS code only controls the whole encryption process. To ensure its correctness, AES-XTS from *OpenSSL* was used to verify that both implementations yield the same results.

The BDK is used only for the structures it offers that are necessary to represent the `DATA_BLOCK` and the key block. The structures could be re-implemented too, but it would not make any difference.

6.3 Program setup

A few things may be required to modify in the *Project Properties* to make the program work. First of all, the BDK must be added to the include path in order to use its structured. However, it does not need to be linked since none of its functions is used.

Next, *Crypto++* needs to be linked. It is compiled as a static library and it needs to be added by setting path to `cryptlib.lib` in *Linker* → *Input* → *Additional Dependencies*.

All the required files are included in the project. It is just necessary to verify that the paths are correctly set before using the program.

6.4 How the program works

Now that the preliminaries are completed lets see how to use the program. It requires two parameters – a container file and a correct password for it. The container file is not changed in any way. It is used only to read the `DATA_BLOCK` from and use it to generate the key.

Using it, the program tries to generate the key and decrypt the container data. The result is not a container that can be mounted using some appropriate tool, but a decision whether the data is correct or not.

The third parameter is an optional one. If provided, the program assumes that BestCrypt is installed and the container is mounted as the device specified by the parameter. The program then opens the container as a device and reads its data. Since this operation is intercepted by the BestCrypt driver, the received data are already decrypted by the corresponding algorithm driver. These data are used for further verification of correctness by comparing it with the independently decrypted data.

The program accepts all containers encrypted by any of the algorithms and encryption modes mentioned. The algorithm parameters are the same as the BDK or BestCrypt define them. Hidden container keys are recognized as well and `DATA_BLOCK` headers can be decrypted. Only the containers created to be version 8 compatible are supported.

6.5 Key generation

The first necessary thing to implement is the key generation. The whole process corresponds to the one used in the BDK. The password the user enters is transformed to the correct form and the password key is derived from it. The password key generation can be found in the `PasswordKeyGeneration.cpp` file as the `generateKeyFromPassword` function.

When generating a password key, the only thing that has to differ is the hashing. The SHA-256 hashing is implemented in `CryptoFuncs.cpp` file as the `hashSHA256` function. The password key length has to be calculated with regard to the algorithm (based on the algorithm key length) and the operation mode (XTS mode requires key size doubled).

The master key decryption is a bit simplified compared to the BDK version, because not all of the verifications and complex operations are necessary. In fact, only decrypting the `sKeyInfo` part of the key block is required. The hash digest is then verified to ensure the master key correctness.

Containers with multiple passwords are supported as well. As the BDK does, the implementation also iterates over all the key blocks and tries to decrypt them with the password key. The program detects hidden part keys too, but does not decrypt its sectors since they are not of the same form

0x002FF754	5a 80 a0 00 00 00 04 00 00 bc	00 10 00 00 00 00	Z€
0x002FF764	00 00 00 f0 9f 00 00 00 00 00	4e 12 20 5e 3c b3	...öÿ....N. ^<
0x002FF774	c0 70 3d 16 d3 d4 71 10 9c 96	db ab 60 4c b1 37	Àp=.óÔq.æ-Ù«`L±7
0x002FF784	4d 0e 45 b7 db 34 02 b2 3f f9	a0 ee 90 59 36 cd	M.E·Ú4...?ù î.Y6í
0x002FF794	90 4f d9 f0 d7 26 c3 d1 c7 36	ad 58 0d ea 70 40	.OÙðx&ÃÑÇ6.X.êp@
0x002FF7A4	2b b9 ad 08 f6 fd f8 17 fd 75	cc cc cc cc cc cc	+...öÿø.ÿuïïïïï

Figure 6.1: Correctly decrypted sKeyInfo data. The fields are marked red and the master key content green.

as in original containers. The master key generation can be found in the `MasterKeyGeneration.cpp` file as the `retrieveMasterKey` function.

To determine whether the key belongs to the normal part or the hidden one, one has to look at the container offset and length present in the decrypted key information. If the length is 4096, it is a key to a hidden part and the offset tells where it starts. The data at this offset do not represent a Master Boot Record, but an allocation map containing the locations of the hidden part sectors.

6.6 Container data decryption

With the master key, the actual container data can be decrypted. The data are to be interpreted as 512 bytes long disk sectors starting with the Master Boot Record. Each sector has been encrypted separately using the algorithm and encryption mode specified by the header. That way it is possible to use the transparent encryption approach mentioned earlier, because it is easy to decrypt or encrypt only a part of the data and leaving the rest untouched.

The program does not go on with decrypting the whole container. It is sufficient to get the information from the Master Boot Record, the Partition Boot Record(s) and check whether they are valid either by guessing by the look of the data, or by comparing it to the correct data if `BestCrypt` is available as well. Sectors from the beginning, middle and end of the partitions are compared with the real ones if the third parameter is supplied to further verify it matches.

This process is not described in the documentation or anywhere else. Communication with Jetico was the only source of information on how to decrypt the sector data. In some cases some modifications are necessary to enable a proper decryption. All of them are mentioned later on. The sector decryption code can be found in the `SectorDecryption.cpp` file as the `decryptSector` function.

Let's see how to represent the data. This is not a `BestCrypt` specific approach, but it is important to know how to retrieve the interesting sectors. When decrypted manually, the Master Boot Record is in the form where

```

0x002FF54C 33 c0 8e d0 bc 00 7c fb 50 07 50 1f fc be 1b 7c 3ĂžĐ.,|ŮP.P.Ů..|
0x002FF55C bf 1b 06 50 57 b9 e5 01 f3 a4 cb bd be 07 b1 04 ĸ..PW.ă.óHĚ...t.
0x002FF56C 38 6e 00 7c 09 75 13 83 c5 10 e2 f4 cd 18 8b f5 8n.|.u.fA.ăđí..đ
0x002FF57C 83 c6 10 49 74 19 38 2c 74 f6 a0 b5 07 b4 07 8b fĕ.It.8,tđ μ.´..
0x002FF58C f0 ac 3c 00 74 fc bb 07 00 b4 0e cd 10 eb f2 88 đ-<.tŮ»..´.Ī.eđ^
0x002FF59C 4e 10 e8 46 00 73 2a fe 46 10 80 7e 04 0b 74 0b N.ĕF.s*þF.ĕŮ..t.
0x002FF5AC 80 7e 04 0c 74 05 a0 b6 07 75 d2 80 46 02 06 83 ĕŮ..t. ħ.uđEF..f
0x002FF5BC 46 08 06 83 56 0a 00 e8 21 00 73 05 a0 b6 07 eb F..fv..ĕ!.s. ħ.ĕ
0x002FF5CC bc 81 3e fe 7d 55 aa 74 0b 80 7e 10 00 74 c8 a0 ..>þ}U=t.ĕŮ..tĚ
0x002FF5DC b7 07 eb a9 8b fc 1e 57 8b f5 cb bf 05 00 8a 56 .e@.Ů.W.đĚĸ..ŠV
0x002FF5EC 00 b4 08 cd 13 72 23 8a c1 24 3f 98 8a de 8a fc .´.Ī.r#ŠĀš?~ŠþŠŮ
0x002FF5FC 43 f7 e3 8b d1 86 d6 b1 06 d2 ee 42 f7 e2 39 56 C+ă.Ń.đ±.đİb+ă9V
0x002FF60C 0a 77 23 72 05 39 46 08 73 1c b8 01 02 bb 00 7c .w#r.9F.s...|.
0x002FF61C 8b 4e 02 8b 56 00 cd 13 73 51 4f 74 4e 32 e4 8a .N..V.Ī.sQ0tN2ăŠ
0x002FF62C 56 00 cd 13 eb e4 8a 56 00 60 bb aa 55 b4 41 cd V.Ī.ĕăŠV.´»U'ĂĪ
0x002FF63C 13 72 36 81 fb 55 aa 75 30 f6 c1 01 74 2b 61 60 .r6.đU=đ06Ā,t+a`
0x002FF64C 6a 00 6a 00 ff 76 0a ff 76 08 6a 00 68 00 7c 6a j.j.ŷv.ŷv.j.h.|j
0x002FF65C 01 6a 10 b4 42 8b f4 cd 13 61 61 73 0e 4f 74 0b .j.´B.đĪ.aas.Ot.
0x002FF66C 32 e4 8a 56 00 cd 13 eb d6 61 f9 c3 49 6e 76 61 2ăŠV.Ī.ĕđăŮĪInva
0x002FF67C 6c 69 64 20 70 61 72 74 69 74 69 6f 6e 20 74 61 lid partition ta
0x002FF68C 62 6c 65 00 45 72 72 6f 72 20 6c 6f 61 64 69 6e ble.Error loadin
0x002FF69C 67 20 6f 70 65 72 61 74 69 6e 67 20 73 79 73 74 g operating syst
0x002FF6AC 65 6d 00 4d 69 73 73 69 6e 67 20 6f 70 65 72 61 em.Missing opera
0x002FF6BC 74 69 6e 67 20 73 79 73 74 65 6d 00 00 00 00 00 ting system....
0x002FF6CC 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x002FF6DC 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x002FF6EC 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x002FF6FC 00 00 00 00 00 2c 44 63 30 50 c4 fe 00 00 00 01 .....Dc0PĀþ....
0x002FF70C 01 00 07 fe ff ff 00 08 00 00 f8 47 00 00 00 00 ...þŷŷ....đG...
0x002FF71C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x002FF72C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x002FF73C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa .....Uă

```

Figure 6.2: Correctly decrypted Master Boot Record. The partition entries are marked red, the signature green. The boot code is unmarked.

the boot code has not been executed yet. The important part are the four partition entries. Each entry contains mainly the starting sector number and total count of sectors of that partition.

The first sector of the partition is the Partition Boot Sector. Besides the signature to check it, the file system type is found here. All the mentioned structures can be found in the `Defs.h` file.

6.7 Verification steps

There are a few verification steps to decide, whether the data we work with are correct or whether an error occurred or the password is invalid.

The first step of the verification lies in ensuring the master key is correct. This can be done after the key block decryption. Since it cannot be simply compared to the correct value, the same verification as in the BDK is used. The hash of the decrypted data is computed and compared to the requested value contained in the key block.

The Master Boot Record is checked by verifying the correct signature and by verifying the partition list holds plausible information. There is no other

way to be sure the data are correct without analysing the boot code.

The Partition Boot Sector(s) are checked by verifying the signature as well. Besides that, the file system type is verified. The only two possibilities are NTFS and FAT. BestCrypt offers to format the container after its creation and lets the user choose the file system type.

When BestCrypt is available and the container mounted, the further verification is done by comparing the blocks in memory. Since the first sector read from the container as a device will be the Partition Boot Sector, the Master Boot Record can not be checked the same way. Besides that, sectors from the beginning, middle and end of the container are compared as well to further ensure correctness.

6.8 Necessary modifications

Now that the principle of the implementation was introduced, let's see what steps are necessary to take that are not obvious, but are essential for the program to work correctly. None of the modifications mean that the BestCrypt algorithms would differ from its official definition. Some of them are mechanisms applied on top of the decryption and some require to alter the IV.

6.8.1 Correct data form

Not all the algorithm drivers expect data in the same form. Passing the correct data in a wrong form to the driver will result in incorrect decryption.

All of the tested algorithm drivers request little-endian data form, except for Blowfish-448. The Blowfish-448 algorithm driver requires big-endian data, so before passing it to the driver, it is necessary to swap the data to big endian (if not swapped already) and similarly swap it back after decryption. According to Jetico, the CAST and IDEA algorithm drivers expect the same data form.

6.8.2 The IV for key block decryption

The initial vector in the key block is 16 bytes long, though it is used in different ways based on the used algorithm or the encryption mode. Note that this is the IV for the key block decryption, not the IV for sector decryption.

First of all, in any case only the first 8 bytes are used. Since the IV needs to be the same size as the block the cipher uses, it often needs to be prolonged. When using the XTS mode, it is simple. The first 8 bytes are used as the first part and the rest are zeros. The IV represents the 'sector number' in the meaning of the operation mode even though it is no sector.

This is different if the CBC mode is used. The first 8 bytes are used in the same way, but are copied as the second 8 bytes as well.

The Blowfish-448 algorithm requires big-endian data, so the IV has to be in the correct form. Table 6.1 should clarify this.

type	IV			
original	00 11 22 33	44 55 66 77	88 99 AA BB	CC DD EE FF
XTS	00 11 22 33	44 55 66 77	00 00 00 00	00 00 00 00
CBC	00 11 22 33	44 55 66 77	00 11 22 33	44 55 66 77
BFISH-448	33 22 11 00	77 66 55 44	33 22 11 00	77 66 55 44

Table 6.1: Key block IV forms.

The different approaches to the IV form may be confusing, but hopefully the table helps to understand how to set the IV correctly. Since some algorithms use the 8 bytes long blocks, only the first 8 bytes are used. This is the only necessary modification regarding the key block decryption IV specifically.

6.8.3 The pool usage

There is the field `pool` of the `DATA_BLOCK` that has not been discussed yet. It is used when decrypting a sector using the CBC mode regardless of the algorithm and when decrypting a container header. It has two usages.

The first usage is simple. After the decryption is done, the result is modified by the `pool` using XOR. Since the pool is 512 bytes long, there is no confusion with this.

The second usage is to modify the IV. When decrypting a sector in XTS, the sector number is used as the IV. In CBC, it is the same, but the IV is further modified then. An offset in the pool is obtained and the IV is modified by a part of the pool. The following code shows how it is done.

```
void transformIV(BYTE *iv, BYTE *pool){
    unsigned int offset = (iv[0] & 0x3F) * 8;
    for(i = 0; i < 8; i++){
        iv[i] = iv[i] ⊕ pool[offset + i];
    }
    memcpy(iv + 8, iv, 8);
}
```

As the code shows, six bits of the sector number determine the offset in the `pool` and the IV is then modified by the following 8 bytes. After that the second 8 bytes are the copy of the first 8. No further modifications need to be done in the CBC mode implementation. The usage for the header decryption is the same.

The whole pool is used and an overflow is not possible, because
 $0x3F \times 8 = 63 \times 8 = 504 = 512 - 8$

6.8.4 XTS modification

A slight modification has to be done to the XTS mode taken from TrueCrypt source code. Besides having to retype some variables because of the lack of TrueCrypt definitions, there is one more important issue. The original code decrypts data blocks 256 bytes long. After each 256 bytes it increases the sector number and decrypts the following 256 bytes. Since the sectors are 512 bytes long, the result would be wrong. It was verified that the first 256 bytes were being decrypted correctly while the second 256 bytes were not.

The modification lies in increasing the block count for one iteration. When doubling it (modifying from 16 to 32), everything works exactly as meant to. How the resulting code looks can be seen in the `decryptXTS` function in the `CryptoFuncs.cpp` file.

This is only a TrueCrypt's implementation specific feature. It does not mean that the BestCrypt's implementation of the XTS mode is incorrect.

6.9 The implementation results

The previous sections explain how the created program works and what modifications are necessary. This section summarizes what the program was used for and what were the gained results.

Containers using all the possible combinations of algorithms and encryption modes were created using BestCrypt. All of them were tested with the program except the ones using LRW operation mode and the ones that use the GOST algorithm. All the test containers were version 8 compatible.

The program was able to successfully retrieve a master encryption key for every tested container. In all cases it was able to decrypt the data part of the container and conclude that the data were correctly decrypted. All of the containers were mounted and then their data were read. Even in this case the data matched.

Based on the previously obtained results it is safe to conclude that all the drivers representing the tested algorithms work correctly and yield the expected results. The SHA-256 hash algorithm was used for password generation and therefore the BestCrypt version of it works properly in all tested cases. With regard to the necessary modifications that were provided by Jetico, it is also safe to conclude that both XTS and CBC operation mode work correctly in all tested cases.

Special features were tested as well. A container containing multiple passwords was successfully decrypted using all of them (all of them resulted in the same decryption key). A hidden key was correctly identified in the key block of a container with a hidden part. The header of a container with an encrypted header was successfully decrypted using the correct password and encryption parameters.

Besides that, in all the possible cases the program results were compared with the ones provided by the BDK. It is safe to say that in these cases the BDK shows the same behaviour that BestCrypt has.

To conclude it, the analysis found nothing suspicious about the algorithms, encryption modes and hashing processes that would result in any strange behaviour. The algorithms work exactly as they are required to by their specifications.

Impact evaluation

This chapter summarizes all three parts of the analysis and compares the results to what Jetico claims. The analysis focused on key generation, security evaluation using the user interface and the BDK and cryptographic processes re-implementation and verification. Relevant security claims are taken into account, commented on and compared with the analysis.

7.1 Data leaks prevention

The authors claim that there is no way to access the data without having the correct password or key. That part is true considering that without having the password, the key can not be generated and without having the key the container data can not be decrypted.

The authors do not mention how hard it would be to gain this information using some sort of attack. The analysis concluded that the key generation process itself is secure, but the iteration count 256 is too low and should definitely be increased to prevent possible brute force attacks. It concerns the version 8 compatible containers only, but since they are used for cross-compatibility as well, they should be secured as well. The possible solution with keeping the compatibility with older versions could be to separate the version 8 and cross-platform compatibility.

7.2 Transparent file encryption

According to Jetico, the container data should be accessible transparently by any application after verifying the access. This was confirmed during the re-implementation phase of the analysis. The container was opened as a device after being mounted by BestCrypt and the data read from the container were already correctly decrypted, yet the container file remained unchanged, so the data inside remained protected.

The way how it is done in theory is described in section 2.1.

7.3 Strong encryption algorithms

BestCrypt should use strong and verified encryption algorithms with the largest possible key size and utilize the XTS encryption mode when possible. This claim was verified by the implementation part of the analysis. Since BestCrypt utilizes encryption via algorithm drivers, the algorithms were independently implemented and tested.

In all cases the results matched and the analysis concluded that the algorithms work properly as their specifications require. Minor modifications were necessary to enable correct decryption in all cases, but none required to change the algorithm logic. These modifications can not be found anywhere in the documentation nor the BDK. Jetico was the only source of how they work and when they are used.

7.4 Improved security in new key generator

The authors claim that the iterations count was increased. The truth of this was verified using the version 9 to create containers. The higher iteration count solves the problem that was discovered in version 8 where the iteration count is just 256. This count is too low and enables a brute force type of attack.

The random seed was increased from 128 B to 4 kB. The ability to stop the process was added as well. Possible protection weakness based on stopping the generation too early was discovered. The authors acknowledged it and proposed a solution.

7.5 Hidden part security

The BestCrypt documentation states that "The potential intruder cannot prove whether an additional (hidden) container exists or not: the information stored in the hidden container is regarded as random data." [1]. This claim was shown to be incorrect.

The analysis found out that the empty key block filling is not done properly and enables an attack to discover the hidden key(s). The attack was presented, implemented and tested. The results proved that this is an issue.

The problem was brought to the authors and they acknowledged it and prepared a solution for future release immediately. The solution was found sufficient.

Conclusion

The thesis focused on BestCrypt Container Encryption by Jetico Inc., a software designed for files encryption. That is achieved via creating containers protected by passwords, that are used as virtual drives that files can be stored in. The thesis presents a security analysis of this software to determine whether it is safe to use or not.

The whole analysis was done with Jetico's approval. Many issues were communicated with them throughout the the analysis and they were very responsive regarding any problem or help request.

Different analyses of a similar software TrueCrypt were presented. Each analysis represented a different approach that a security analysis can take. The approach that this analysis took was determined from a combination of the presented ones.

Unlike in the case of TrueCrypt, Jetico did not make the source code publicly available. Instead, they created a library, the BestCrypt Development Kit, to enable anyone to do their own security examination. Both BestCrypt and BestCrypt Development Kit were described in detail along with necessary information about their usage, functionalities and features. In the case of the BestCrypt Development Kit, a sample program to show how to use it correctly was created.

After the initial necessary software introductions, the first important part of the analysis was done. The process of how a cryptographic key is generated from the password was examined and explained in detail. This process is a necessity for any further actions. The process consists of two parts. In the first part the so called password key is derived from the password and in the second one the corresponding data are decrypted using the password key to retrieve the master key, that is used for the actual container data encryption and decryption.

The second part of the analysis focused on BestCrypt itself from the user's point of view. Since the whole application can not be independently built, the analysis chose a different approach. It tested all the actions (password chan-

ging, adding or removing, container creation and deletion, container header encryption etc.) using the user interface and used the BestCrypt Development Kit when it was possible to compare what BestCrypt does with what the library shows it should do.

In this part of the analysis, two flaws were discovered. The first one is connected with the hidden part functionality. It was discovered that an unsafe approach was used to fill the empty key blocks upon the container or key block initialization. This enables an attack that is able to determine whether there is or is not a hidden part in the container. Since this stands against the idea of hidden parts, it presents a serious problem.

The second problem was discovered in the random seed generation. The process is based on user's mouse movement and keyboard clicks, which is a correct way to gain randomness. However, the process can be stopped. Stopping the process too early can result in an insufficient randomness and could lead to a possible attack.

Both these problems were brought to the authors. They acknowledged them both and prepared a solution for them. These solutions were mentioned as well and considered sufficient by the analysis.

In the last part of the analysis, ensuring the proper behaviour of the used algorithms and encryption modes was the focus. The key generation process and the container data decryption were implemented independently of BestCrypt using *Crypto++*, an independent cryptographic tool to do so. BestCrypt realizes encryption and decryption by algorithm drivers, so it was important to verify that the algorithms work as they are supposed to.

This part of the analysis tested two of the three operation modes and all but one algorithms. Using created sample containers it verified that all the algorithms work properly. The encryption modes are used correctly as well and the hashing is done right too.

The analysis did not discover any sort of a back door nor any way that could lead to container data leakage. Besides the mentioned issues regarding the random number generation, no problematic parts were identified.

The data are protected by strong algorithms that work properly and no way how to bypass them was found. No backdoors were found either. The master key generation process is secure, although the iteration count for the version 8 containers definitely should be increased. The issues with the seed generation and the data initialization were promised to be fixed in the next release. To conclude it, BestCrypt can be considered secure once the mentioned issues are fixed.

Bibliography

- [1] Jetico Inc. BestCrypt Container Encryption User Manual. [cit. 21.4.2016]. Available from: https://www.jetico.com/web_help/PDF/BCCE.pdf
- [2] Wikipedia. Image of Cipher Block Chaining (CBC) mode encryption. [cit. 21.4.2016]. Available from: https://upload.wikimedia.org/wikipedia/commons/8/80/CBC_encryption.svg
- [3] Wikipedia. Image of Cipher Block Chaining (CBC) mode decryption. [cit. 21.4.2016]. Available from: https://upload.wikimedia.org/wikipedia/commons/2/2a/CBC_decryption.svg
- [4] Wikipedia. Image of XEX with tweak and ciphertext stealing (XTS) mode encryption. [cit. 21.4.2016]. Available from: https://upload.wikimedia.org/wikipedia/commons/b/b5/XTS_mode_encryption.svg
- [5] Jetico. BestCrypt Help. [cit. 21.4.2016]. Available from: <http://www.jetico.com/linux/bcrypt-help/index.htm>
- [6] TrueCrypt official website. May 2014, [cit. 21.4.2016]. Available from: <http://truecrypt.sourceforge.net/>
- [7] Green, M.; Hoffman, M.; White, K. Open Crypto Audit Project. [cit. 21.4.2016]. Available from: <https://opencryptoaudit.org/>
- [8] Green, M. Another update on the Truecrypt audit. February 2015, [cit. 21.4.2016]. Available from: <http://blog.cryptographyengineering.com/2015/02/another-update-on-truecrypt-audit.html>
- [9] Klíma, V.; Rosa, T. Šifrování USB flash disků zdarma. August 2007, [cit. 21.4.2016]. Available from: http://crypto-world.info/klima/2007/ST_2007_08_09_09.pdf

BIBLIOGRAPHY

- [10] Ubuntu Privacy Remix Team. Security Analysis of TrueCrypt 7.0a with an Attack on the Keyfile Algorithm. Technical report, August 2011, [cit. 21.4.2016]. Available from: https://www.privacy-cd.org/downloads/truecrypt_7.0a-analysis-en.pdf
- [11] Junestam, A.; Guigo, N. Open Crypto Audit Project, TrueCrypt Security Assessment. Technical report, iSECpartners, February 2014, [cit. 21.4.2016]. Available from: https://opencryptoaudit.org/reports/iSec_Final_Open_Crypto_Audit_Project_TrueCrypt_Security_Assessment.pdf
- [12] Balducci, A.; Devlin, S.; Ritter, T. Open Crypto Audit Project, TrueCrypt Cryptographic Review. Technical report, nccgroup, March 2015, [cit. 21.4.2016]. Available from: https://opencryptoaudit.org/reports/TrueCrypt_Phase_II_NCC_OCAP_final.pdf
- [13] Bernstein, D. Cache-timing attacks on AES. Technical report, The University of Illinois at Chicago, [cit. 21.4.2016]. Available from: <http://cr.ypt.to/antiforgery/cachetiming-20050414.pdf>
- [14] Baluda, M.; Fuchs, A.; Holzinger, P.; et al. Security Analysis of TrueCrypt. Technical report, Fraunhofer Institute for Secure Information Technology, November 2015, [cit. 21.4.2016]. Available from: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/Truecrypt/Truecrypt.pdf?__blob=publicationFile&v=2
- [15] Libgcrypt. August 2015, [cit. 21.4.2016]. Available from: <https://www.gnu.org/software/libgcrypt/>
- [16] Türpe, S.; Poller, A.; Steffan, J.; et al. Attacking the BitLocker Boot Process. Technical report, Fraunhofer Institute for Secure Information Technology, [cit. 21.4.2016]. Available from: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.149.5116&rep=rep1&type=pdf>
- [17] Jetico, Inc. Discussion about BestCrypt security. 1998, [cit. 21.4.2016]. Available from: <https://groups.google.com/forum/#!topic/sci.crypt/mgIDq3gWmqY>
- [18] Microsoft. Transparent Data Encryption (TDE). 2016, [cit. 21.4.2016]. Available from: <https://technet.microsoft.com/en-us/library/bb934049%28v=sql.110%29.aspx>
- [19] Fruhwirth, C. New Methods in Hard Disk Encryption. Technical report, July 2005, [cit. 21.4.2016]. Available from: <http://clemens.endorphin.org/nmihde/nmihde-A4-os.pdf>
- [20] Schneier, B. *Applied Cryptography, Second Edition: Protocols, Algorithms and Source Code in C*. 1996, ISBN 0471128457, [cit. 21.4.2016].

-
- [21] Liskov, M.; Rivest, R. L.; Wagner, D. Tweakable Block Ciphers. Technical report, Massachusetts Institute of Technology, [cit. 21.4.2016]. Available from: <http://people.csail.mit.edu/rivest/LiskovRivestWagner-TweakableBlockCiphers.pdf>
- [22] Jain, R. Block Cipher Operation. Technical report, Washington University in Saint Louis, 2011, [cit. 21.4.2016]. Available from: http://www.cse.wustl.edu/~jain/cse571-11/ftp/l_06bco.pdf
- [23] Jetico. BestCrypt Development Kit. [cit. 21.4.2016]. Available from: <http://www.jetico.com/support/bestcrypt-development-kit>
- [24] Turan, M. S.; Barker, E.; Burr, W.; et al. Recommendation for Password-Based Key Derivation. Technical report, NIST Special Publication 800-132, December 2010, [cit. 21.4.2016]. Available from: <http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf>
- [25] Harold, T. G. MD5 vs SHA-1 vs SHA-256 performance. 2015, [cit. 21.4.2016]. Available from: <http://tgharold.blogspot.cz/2015/05/md5-vs-sha-1-vs-sha-256-performance.html>
- [26] CERN Computer Security Team. Password Recommendations. 2016, [cit. 21.4.2016]. Available from: <https://security.web.cern.ch/security/recommendations/en/passwords.shtml>
- [27] Eternally Confuzzled. Rand usage. [cit. 21.4.2016]. Available from: http://www.eternallyconfuzzled.com/arts/jsw_art_rand.aspx
- [28] Matsumoto, M.; Nishimura, T. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. Technical report, Keio University, [cit. 21.4.2016]. Available from: <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.pdf>
- [29] Combs, G. Wireshark. [cit. 21.4.2016]. Available from: <https://www.wireshark.org/>
- [30] OpenSSL. [cit. 21.4.2016]. Available from: <https://www.openssl.org/>
- [31] Dai, W. Crypto++ Library 5.6.3. August 2015, [cit. 21.4.2016]. Available from: <https://www.cryptopp.com/>
- [32] Dai, W. Crypto++ usage in products. [cit. 21.4.2016]. Available from: https://www.cryptopp.com/wiki/Related_Links
- [33] TrueCrypt source codes. [cit. 21.4.2016]. Available from: <https://sourceforge.net/projects/truecrypt/files/TrueCrypt/Other/>

Acronyms

AES Advanced Encryption Standard

BDK BestCrypt Development Kit

CBC Cipher block chaining encryption mode

DES Data Encryption Standard

LRW Liskov, Rivest, Wagner encryption mode

XTS XEX with tweak and ciphertext stealing encryption mode

Contents of enclosed CD

readme.txt.....	CD contents and usage
programs.....	the programs used for the analysis
├─ BestCryptAnalysis	the main analysis program files
├─ BestCrypt_RandAttack.....	the hidden part attack program files
├─ BDK_usage	the program illustrating the BDK usage files
libs	the tools used for the analysis
├─ BDK.....	the BDK compilation result
├─┬─ source	the BDK source code
├─ CryptoPP.....	the Crypto++ compilation result
├─┬─ source	the Crypto++ source code
containers.....	test container files
├─ containers_info.txt	test containers parameters
src	source codes
├─ thesis	L ^A T _E X source codes of the thesis
text	the thesis text
├─ thesis.pdf.....	the thesis text in PDF format