Forensic Methods and Techniques for the Detection of Deniable Encryption

ANGELA IRWIN, 45455249

SECURITY FORENSICS PROJECT, COSC 419

PGD COMPUTER SECURITY AND FORENSICS

UNIVERSITY OF CANTERBURY

20 OCTOBER 2008

Abstract

Deniable encryption allows an encrypted message to be decrypted to different sensible plaintexts, depending on the key or passphrase used or otherwise makes it impossible to prove the existence of the real message without the proper encryption key. This allows the sender to have plausible deniability if compelled to give up his or her encryption key. Deniable file systems allow for the creation of an encrypted volume on a hard disk and a second, hidden deniable file system created inside with access by a second password. Deniable file systems are different from encrypted file systems in that encrypted file systems are visible but indecipherable.

Deniable file systems were first introduced as a tool to assist human rights workers protect sensitive data in the field, however, they can be used to hide criminal and terrorist activity and espionage.

This paper reviews work already completed by scholars in the area of deniable decryption and deniable file systems and provides an overview of five of the most common deniable encryption tools currently available, namely, BestCrypt V8, FreeOTFE, TrueCrypt, PhoneBookFS and Rubberhose. The review examines the general features and techniques employed by each tool to deniably encrypt plaintext and investigates the methods employed by each to counter disk surface analysis, cryptanalysis attacks and identifies issues that may potentially impact on the products' ability to maintain plausible deniability. Deniable encryption vendors claim that, when their products are used, there is no way for an adversary to identify the existance of hidden volumes or data contained within them, thereby maintaining plausible deniability should a user be requested or forced to reveal their encryption keys. However, recent research by Czeskis et al, who reviewed the TrueCrypt file system, shows that the operating system itself, Microsoft Word and Google Desktop can compromise the deniability of a TrueCrypt file system.

FreeOTFE, one of the tools reviewed, is selected for in-depth disk surface analysis to investigate whether evidence of a hidden volume can be found. Disk surface analysis focuses on three main areas: examination the volume's critical data block, breakdown of the Master File Table (MFT) entries for the outer volume and the files contained inside the hidden volume to determine whether any useful information can be gained to provide clues to the presence of the hidden volume and searching for evidence of data leakage from the hidden volume into other areas of the hard disk.

The aim of this research is to establish whether or not evidence of a hidden volume can be found using forensic analysis tools. It is not the intention of this research to decrypt or view the data contained within the hidden volume.

Analysis shows that no useful information can be gained from the volume's critical data block (CDB) as the full CDB is encrypted. Also, the MFT entries provide no assistance in determining whether a hidden volume is present. However, there are many areas of data leakage from the hidden volume and the files contained within those volumes that occur during normal operation of the software that would result in suspicion of the presence of a hidden volume.

Contents Page

List of figures

List of tables

CHAPTER 1 INTRODUCTION

Information technology has advanced greatly in recent years and now pervades every corner of society. The proliferation and sophistication of technology has resulted in a change in the criminal landscape whereby criminals and terrorists use technology in the commissioning of their crimes and to communicate covertly.

Fears surrounding terrorism and organised crime have resulted in many countries introducing legislation that allows interception of communications, acquisition and disclosure of data relating to communications and acquisition of the means by which electronic data protected by encryption or passphrases may be decrypted or accessed [1]. For example, Section Three of the Regulation of Investigatory Powers Act (RIPA) came into force in the United Kingdom in October 2007, seven years after the original legislation passed through Parliament. The Act, which primarily intended to deal with terror suspects, allows police to demand encryption keys or be provided with a clear text transcript of encrypted text. Non-compliance with an order can result in up to 5 years imprisonment.

Steganography, the art of hiding messages within a cover file or document in such a way that others cannot discern its presence or contents, has been used by criminals, terrorists and child pornographers for a number of years to conceal their activities. However, advancements in steganography detection tools, forensic techniques and legislation, where laptop computers can be searched and seized at international borders, have rendered the use of steganography to hide evidence of unlawful activity inadequate.

The shift from reactive policing of incidents to proactive policing and management of risks has fuelled interest in deniable encryption tools and techniques.

Deniable encryption is encryption that allows its user to deny the fact that he encrypted a message he actually did encrypt. Traditional ciphertexts commit to a given plaintext and hence the encrypter cannot claim that he encrypted a different message. Deniable encryption, on the other hand, allows its user to point to a different (innocuous but plausible) plaintext and insist that this is what he encrypted. The holder of the ciphertext will not have the means to differentiate between the true plaintext and the bogus-claim plaintext. In other words, deniable encryption allows an encrypted message to be decrypted to different sensible plaintexts, depending on the key or passphrase provided or otherwise makes it impossible to prove the existence of the real message without the proper encryption key. This allows the sender to have plausible deniability if compelled to give up his or her encryption key.

Although there is escalating interest in deniable encryption in the media and an increase in the number of free and commercial deniable encryption products on the market, very little scholarly work has been completed in the area of deniable encryption and counter deniable encryption techniques.

Deniable encryption vendors claim that, when their products are used, there is no way for an adversary to identify the existance of hidden volumes or data contained within them, thereby maintaining plausible deniability should a user be requested or forced to reveal their encryption keys [3, 4, 5, 6, 7]. The aim of this research is to prove or disprove this claim by using WinHex, a forensic analysis tool, to analyse a FreeOTFE outer volume with a hidden volume created inside it to determine whether the hidden volume or any of its contents is detectable when disk surface analysis is performed on it.

The paper is organised as follows: Chapter 2 reviews the work already completed by scholars in the area of deniable decryption and deniable file systems. Chapter 3 provides an overview of the deniable encryption tools that are currently available. The review examines the general features, cryptographic algorithms, security features and techniques employed by each tool to deniably encrypt plaintext. Chapter 3 also looks at the methods employed by each tool to counter disk surface analysis and cryptanalysis attacks and discusses the issues that may potentially impact on the products' ability to maintain plausible deniability. Chapter 4 discusses the methodology used for testing the claim. Chapter 5 presents detailed results and discussion of the analysis and testing performed on the FreeOTFE outer and hidden volumes. Finally, Chapter 6 concludes the paper by discussing whether deniable plausibility can be maintained in FreeOTFE when analysed using forensic analysis tools such as WinHex.

CHAPTER 2 LITERATURE REVIEW

Despite the growing need for counter deniable encryption methods and techniques, to date scholars and researchers have overlooked the area of counter deniable encryption as a topic for investigation. To my knowledge only one paper has been published in the area of defeating encrypted and deniable file systems, therefore, this chapter of the report also examines papers that have been published on the use of deniable encryption for hiding data from adversaries.

Scholars have proposed many schemes for providing plausible deniability through the use of encrypted and steganographic file systems, however, these have shown varying degrees of success. This chapter of the report discusses the schemes proposed.

Canetti et al [2] published the first paper on the subject of deniable encryption in 1996. Until that point, researchers focussed on encryption for passive semantic security, which is necessary for protecting communications from passive eavesdropping. At that time [2] believed that commonly accepted cryptographic practices fell short of providing the necessary degree of protection in situations where a user was under duress or coercion to disclose his or her private keys. The aim of their research was to introduce the concept of deniable encryption where a user could generate fake random choices that make the cipher text look like an encryption of a different cleartext, therefore, keeping the real cleartext private.

The authors of [2] look at public-key encryption, where the sender and receiver have no prior shared secret information and shared-key encryption, where the sender and receiver share a random, secret key about which the adversary is assumed to no have prior information. They describe two public-key deniable encryption schemes. The first being a basic scheme, which offers a partial solution to the problem and acts as the building block in the construction of their main scheme.

Their main scheme, called the Parity Scheme, is where the probability of a successful attack vanishes linearly in the security parameter. Their schemes are sender-deniable and are based on the simple idea that the sender can pick an element in some domain either randomly or pseudo-randomly and that the receiver, having some secret information, can tell whether the element was chosen randomly or pseudo-randomly while a third party cannot tell the difference. With the basic scheme, the sender can fake his message in one direction only. To fake in both directions, [2] came up with two simple constructions of translucent sets, which use a trapdoor permutation and a third construction that relies on the lattice-based public-key cryptosystem.

They investigated a one-time pad shared-key deniable encryption scheme, which assumes that the sender and the receiver share a sufficiently long random string, and each message m is encrypted by bitwise XOR-ing it with the next unused $|m|$ bits of the key. The message m can be chosen as late as at time of attack. However, using the one-time pad is generally impractical, since the key has to be as long as all the communication between the parties. It is possible, however, to transform any given shared key encryption to a deniable one, without increasing the message length, by using a combination of public-key deniable scheme and private-key deniable scheme. This works by making the public-key deniable scheme more efficient by way for first sending (using public-key deniable scheme) a deniable shared key and then switching to a private-key deniable scheme.

Nathan Karsten's paper [9] published in 2006 discusses two deniable encryption schemes; TrueCrypt, a free, open source deniable encryption package and Deniability, a program written by the author based on a static, interleaved scheme.

The difference between the authors scheme and the TrueCrypt scheme is that the authors scheme does not allow a user to modify any enciphered data after it has been encrypted without extracting all of the hidden messages. This is compared to the TrueCrypt scheme, which allows at least one of the encoded messages to be modified without knowledge of all hidden messages.

Another important difference between the two schemes is that the TrueCrypt scheme creates hidden volumes and allocates a contiguous block for each hidden message, whereas the authors scheme divides each message into a collection of blocks of a known size, encrypts them, combines them together in a non-contiguous order and then stores them in an encapsulating file or device.

[9] believes his Deniability scheme to be very secure but acknowledges the need for a password salt and master password to be included and confesses that there is a small, but statistically unlikely, chance of

collisions when decrypting messages. These collisions may happen when decrypting a message a block from another message, which appears to belong to the message being decrypted, corrupts the current byte and all subsequent bytes causing the sequence numbers to be incorrect.

The most recent paper, published by Czeskis et al [10] in 2008, investigates the effectiveness of TrueCrypt version 5.1a in creating and maintaining a secure deniable file system. The paper highlights how Windows Vista operating system, Microsoft Word and Google Desktop compromise the deniability of TrueCrypt and go on to suggest some approaches to overcoming the challenges of modern operating systems in maintaining deniability.

The authors of the paper show that "deniability, even under a very weak model, is fundamentally challenging" and that the natural processes of the Windows operating system can "leak significant information outside the deniable volume". They identify three broad classes of information leakage vectors that affect the deniability of a TrueCrypt hidden volume. These are:

Leakage from the operating system via Windows Registry and shortcuts: Windows Registry reveals the fact that a person used TrueCrypt on their machine, and the locations where the TrueCrypt volumes were mounted but no other information about the container's file name, location, size or type which would directly compromise the deniability of the hidden volume. The shortcut, or .lnk files, which are created automatically by Windows, store a wealth of information about files, including the real file's name, length, date of creation, time of access and the volume serial number of the file system on which the real files are stored. The .lnk file can be used to determine whether a hidden volume is present.

Leakage from the primary application via Word Auto Saves: When a file is edited, many applications create a local copy of the file and record all of the changes made to the original. Although Word removes the backup file after the user closes the primary file, Word does not invoke a secure delete. The authors were able to repeatedly recover previously edited files that were stored on a hidden volume by running a free, easy to use data recovery tool. Copies of hidden files could be recovered after a reboot and, in the case of a premature termination of the software; auto-recovered files persisted on the local disk in clear view and could be recovered by Microsoft Word even though the TrueCrypt volume was not mounted.

Leakage from non-primary applications such as Google Desktop: Google Desktop allows previous states of Microsoft Word documents to be viewed. Google Desktop caches snapshots of documents and files and stores them for viewing at a later time, allowing an attacker to view the current state of the document and how it evolved over time. With Google Desktop installed and running, the authors were able to easily recover a number of snapshots of a hidden file by simply searching for a keyword, even after the volume was dismounted. They were able to repeat this irrespective of the mount point of the volume or the mount type. The authors suggest that Google Desktop indexing feature be paused whenever a TrueCrypt volume is being used.

Although the authors identify the aforementioned information leakage vectors, they recognise that there may be many other scenarios or applications that leak information that they have not identified. They go on to say that operating systems and their underlying hardware should assist in ensuring deniability as the use of operating systems such as HiStar, which ensures that information about a deniable file system are not leaked to another part of the system, may be "overkill to prevent most typical information leaks". They identify a more effective approach as installing a file system filter that disallows processes write access to a non-hidden volume, once that process reads information from a hidden volume.

Oler & Fray [12] investigated the use of deniable file systems as a means of hiding the existence of encrypted files. They explored the most important achievements in the area of steganographic file systems [13, 14, 15] but they consider each of the systems investigated to have serious drawbacks.

Firstly, performance in the Anderson et al scheme [13] is very poor because lots of files must be XOR-ed to ensure computational security and there is an assumption in their argument that an opponent has no prior knowledge of any part of the plain text, which they believe does not stand in real life. The second concept introduced by [13] is a file system that initially contains only random blocks and the data is stored in an encrypted form together with dummy blocks. The location of the real files is determined by the key, which is derived from the real file name. The use of encryption ensures that real files are indistinguishable from dummy blocks. One of the problems that arise from this scheme is that collision of blocks of the hidden files is possible. They rectify this by duplicating blocks and writing them to numerous locations before the encryption redundant data is added to it. Later, when the system tries to load such a block, all copies of the block are decrypted and redundant data is compared to determine whether the block is corrupted.

Inspired by the work of [13], RSA Laboratories, the authors of [14] proposed hidden files be placed in unused blocks of a normal (non-steganographic) partition and the location of the hidden files be determined by querying an encrypted external block allocation table. No attempt is made to hide the block allocation table because the presence of the driver would make it easy to establish its presence. Each block is encrypted in cipher block chaining (CBC) mode and its initialisation vector (IV) is stored in the block allocation table. A hard coded password for each block is derived from the password for a given security level and the block number. This hard coding makes it very difficult to change.

Zhou et al, the authors of [15] aimed to create a file system that was reliable and could efficiently manage large volumes of data and be easily adopted by the mass market. The authors believed the work of [13] and [14] were ineffective at achieving these goals. The main reasons being space utilisation, the excessive number of input/output operations required and lack of reliable protection from data loss.

The authors of [15] introduce the concept of “abandoned blocks” to counteract the problem of disclosing the existence of hidden files to an adversary. In previous systems, hidden files were not registered with the central directory, but the blocks occupied by them were marked as being used in the allocation bitmap. This prevented overwriting by other files but disclosed the existence of hidden files to the adversary. However, this approach too proved ineffective so the authors introduced the concept of “dummy hidden files”, these files do not contain any relevant data but are periodically updated to prevent the analysis of updates of unused blocks. Files are located by the address derived from the hash of the file name, access key and user Id. The hash is stored in the structure of the hidden file and is used to retrieve the file at a later date. Blocks for a new file are looked-up randomly and real data is not stored there. Although this system has increased performance and efficiency, a considerable drawback is its inability to overwrite or remove hidden files. This causes problems in multi-user environments where a malicious user fills up all free space, making it possible to estimate how much hidden data exists in the file system.

Oler & Fray [12] introduce the concept of a superblock, which they believe addresses the problems they identified in [13], [14] and [15]. The fundamental concepts of their approach is to use encryption and several independent hidden volumes on the same physical partition to allow users to place mildly incriminating data in one volume and highly sensitive data in another. Upon request, the user can give the passphrase for the first volume and deny the existence of the other. Each volume does not know that the others exist, unless they are open at the same time, the authors believe that this improves reliability and performance.

Lack of knowledge allows data to be overwritten. Writing to blocks used by closed volumes is a desirable situation as no suspicion of hidden files arises and it becomes possible to remove hidden data without the need to access the hidden volume. To compensate for possible data loss, all stored information is replicated. From the operating system’s point of view, each volume is treated as a separate file system and can have a separate mount point. The authors recognize that their work is still in its infancy and needs a lot more work to prove proof of concept in a real world environment.

Other authors who have researched steganographic filesystems for plausible deniability include:

McDonald and Kuhn [16] whose system uses an openly visible encrypted block-allocation table and allows users to completely fill the disk safely when all hidden levels are open. The only storage overhead comes from the adjustable replication of blocks. In addition, their scheme allows them to share a partition with a normal widely used file system, which simplifies installation and provides an additional degree of plausible deniability by making hidden files indistinguishable from unused blocks.

Pang et al [17] explore StegFS as a means of offering plausible deniability to owners of protected files. Unlike previous steganographic schemes, they believe their construction satisfies the prerequisites of a practical file system in ensuring integrity of the files and maintaining efficient space utilisation. Their experiment results confirm that StegFS achieves order of magnitude improvements in performance and/or space utilisation over existing schemes.

CHAPTER 3 SURVEY OF DENIABLE ENCRYPTION TOOLS

Chapter 3 provides an overview of the deniable encryption tools that are currently available. The review examines the general features, cryptographic algorithms, security features and techniques employed by each tool to deniably encrypt plaintext. Chapter 3 also looks at the methods employed by each tool to counter disk surface analysis and cryptanalysis attacks and discusses the issues that may potentially impact on the products' ability to maintain plausible deniability.

Table 1 below details the 7 deniable encryption tools identified during Internet research. All of the tools identified with the exception of StegFS and Off-the-Record Messaging, have been included in the review.

No.	Name	Platform	Availability	Notes
1	BestCrypt	MS Windows	Commercial	
2	FreeOTFE	MS Windows, PocketPC, PDA	Free open source	
3	TrueCrypt	Windows, Mac & Linux	Free open source	
4	PhoneBookFS	Linux	Free open source	No longer maintained
5	Rubberhose	Linux	Free open source	No longer maintained
6	StegFS	Linux	Free open source	Contains file corrupting bug
7	Off-the-Record Messaging	Instant Messaging	Free open source	

Table 1: Deniable encryption products

StegFS was not reviewed as it is reported to contain a file-corrupting bug and Off-the-Record Messaging was not reviewed as it is used for deniably encrypting instant messages. Although there may be other less well-known deniable encryption tools available, research was only carried out on well-established ones.

3.1 BestCrypt version 8

BestCrypt [3] is a commercial data encryption system that provides plausible deniability through the use of hidden containers. BestCrypt can be used in both Windows and Linux environments.

BestCrypt supports numerous cryptographic hash and cipher algorithms. Hash algorithms supported are AES, Blowfish, CAST, GOST 28147-89, RC-6, Serpent and Twofish. Ciphers supported are GOST, SHA-1 and SHA-256. The modular design of BestCrypt allows third party encryption software or hardware to be inserted as security extensions into the BestCrypt software.

BestCrypt version 8 utilises tweakable narrow-block encryption mode, also known as LRW encryption mode, which is designed for applications working at disk sector level and is more secure than modes such as Cipher Block Chaining (CBC). LRW addresses threats such as copy-and-paste and dictionary attacks.

BestCrypt works by creating a container file on the hard disk and mounting the container file to a virtual drive. All files stored in the virtual drive are stored in the mounted container in encrypted form. Multiple containers can be created on the hard disk, with each container having its own passphrase, which is specified by the user when the container is created. This same passphrase is used when the virtual drive linked to the container is opened.

The virtual drive is used to access the encrypted data and files stored in containers. To access the data, the user must mount the appropriate container to the selected virtual drive and open the virtual drive using the container's passphrase. The virtual drive must be closed after use to prevent unauthorised users from gaining access. If a computer is closed down incorrectly or it loses power, all virtual drives are closed automatically and the keys generated disappear, the virtual drive letter is removed from the list of available disks and access becomes impossible. When a virtual drive is closed, data cannot be read or stored on it.

BestCrypt uses transparent encryption, which means that every read operation on the virtual drive causes decryption of the data, and every write operation causes encryption of data to be written. This ensures that data is always stored in a safe, encrypted form and appears in its natural form to the applications that process the data.

BestCrypt drivers monitor all read and write requests and performs encryption and decryption of the transferring data on the fly, see Figure 1 below:

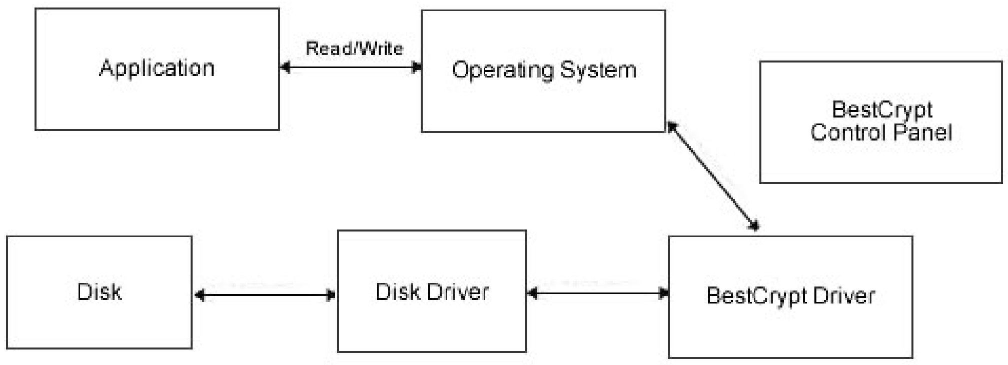


Figure 1: BestCrypt monitoring data

The BestCrypt driver does not process all input and output requests, instead, it creates and supports its own virtual drives and only processes input and output operations for these drives.

BestCrypt provides plausible deniability by creating hidden containers inside original containers. A BestCrypt original container file consists of the three parts shown in Figure 2 below. The first 512 bytes of the container holds the data required to verify the integrity of the file, the second part of the container is the Key Data Block; this stores the array of encryption keys and the final part of the container holds the encrypted data.

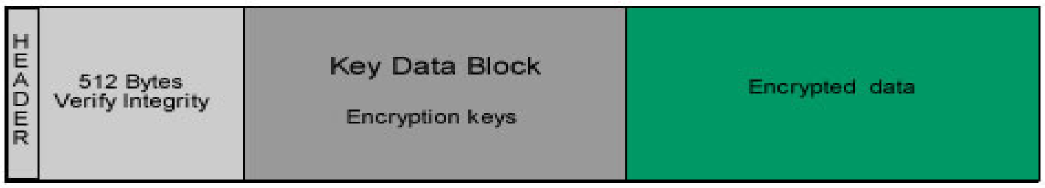


Figure 2: BestCrypt original container

When mounting the original container, BestCrypt verifies the integrity of the container. It then calculates a hash according to the passphrase entered by the user and uses this hash for decrypting the encryption key from the Key Data Block. BestCrypt then uses the key from the Key Data Block to provide transparent encryption of the data in the third part of the container.

BestCrypt can encrypt and isolate a header from its container and store it on a removable device such as a USB drive. Isolating the header increases the security of the container, as the container cannot be mounted if the header, which contains the encryption keys, is not yielded at the time of mounting.

If a hidden part is created inside the container, BestCrypt creates a new encryption key for the hidden part and stores it in the Key Data Block of the original container (See Figure 3 below). The position where the key for the hidden part is stored appears to be marked as unused, this makes it impossible to determine whether any key for a hidden part exists or not. Spare disk space within the container is itself encrypted as random data, so replacing some random data with a new randomly generated key does not compromise the hidden part. The hidden part is stored inside part 3 of the original container without its own Key Data Block, so that it is impossible to define the borders of the hidden part inside the original container.

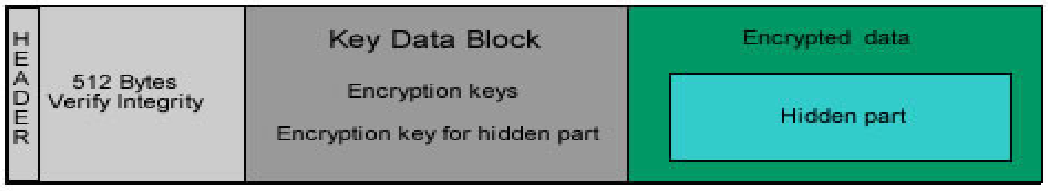


Figure 3: BestCrypt original container holding hidden data

The procedure for mounting a container with the hidden part inside is as follows:

- BestCrypt tries using the passphrase for mounting the original container first, as if there is no hidden part inside it.
- If this passphrase is inappropriate for mounting the original container, BestCrypt checks for the existence of a hidden part and uses the hash value generated from the passphrase to extract the encryption key for the hidden part.
- If the passphrase is appropriate for opening the hidden part, BestCrypt will mount this part and report to the user that the hidden part was found. This message informs the user which object was mounted - the original container or the hidden part.

Once the hidden container has been created within the original container, no additional data can be written to the original container. This is due to the BestCrypt original container having no knowledge of the hidden parts' presence – this is a feature designed to make the original container appear to be the only container and therefore provide protection from analytical tools. If data is written to the original container the hidden data may be destroyed.

If a hidden part is created, it means that the data stored inside the original container has no meaning and exists only for one reason - to disguise the information stored in the hidden part, therefore, mounting of the original container should be avoided. In fact, the passphrase for the original container should be used as an "Alarm" passphrase, which means that it should only be used if forced to do so by a challenger. The alarm passphrase can be used to mount the original container and write some data into it to destroy the hidden part.

BestCrypt states that only one additional passphrase can be used for the hidden part. For example, if passphrase 'A' is used for the hidden part and a new passphrase 'B' is added for the hidden part, the hidden part can now be mounted with either passphrase 'A' or passphrase 'B'. If, however, a new passphrase 'C' is added to the hidden part, BestCrypt is not aware that some "unused" slot of its header may correspond to passphrase 'B,' and BestCrypt will simply overwrite the slot with encrypted data for the new passphrase 'C'. The hidden part will be mountable with 'A' and 'C' passphrases, but passphrase 'B' will become invalid for the hidden part.

Although it is possible to format the original container, as well as a hidden part inside it, using any available file system (FAT, FAT32, NTFS), it is recommended that only FAT or FAT32 file systems be used for the original container.

NTFS is not recommended, for the following reason: NTFS places its tables not only at the beginning of the drive (as FAT and FAT32 do), but also in the middle sectors of disk and last sectors. Hence, when data is written to the hidden part, the possibility of corrupting NTFS file system tables in the original container is high. However, since the FAT and FAT32 file systems store their tables at the beginning of the original container, the tables will not be damaged when data is written to the hidden part. NTFS can be used for formatting the hidden part inside containers.

3.1.1 Issues that may affect plausible deniability of BestCrypt

It is possible to restore deleted files from a hard disk if the user has not used a wiping utility to erase information from the physical disk sectors. BestCrypt provides a 'Move with Source Wiping' command to wipe deleted files, free space and file slack on specified disks but the user must invoke this. It may be possible to recover data between wipe commands.

When files are transferred between conventional files and BestCrypt logical disks, the operating system does not erase the files' contents from the source disk; it will only delete references to the files in the file allocation table. Contents of the deleted files continue to be stored on the disk and can be easily restored using a forensics tool.

Windows can put a whole document or segments of it into the swap file in an unencrypted form, even if the original document is encrypted. Encryption keys, passphrases, and other sensitive information can also be swapped to the hard drive. Even if all of the security measures of the latest Windows versions are used, simply investigating the swap file in DOS mode may allow a challenger to extract a lot of information from the file. BestCrypt allows encryption of the swap file contents but the user must invoke it.

3.2 FreeOTFE

FreeOTFE [4] is a free of charge, on the fly transparent disk encryption program that can be used on Microsoft Windows 2000/XP/Vista platforms, Windows Mobile 2003/2005 and Windows Mobile 6 Personal Digital Assistants (PDAs) and mobile telephones.

FreeOTFE allows users to create one or more virtual disks or volumes on their computer, with anything written to them automatically, and securely, being encrypted before being stored on the computers' hard drive. Encrypted volumes can be file or partition based.

FreeOTFE supports many cryptographic hash and cipher algorithms. Hash algorithms supported include MD2, MD4, MD5, SHA-1, SHA-512, RIPEMD-160, Tiger and Whirlpool. Ciphers supported include AES 128, 192 & 256, Blowfish, CAST5 & 6, DES, 3DES, MARS, RC-6, Serpent and Twofish. The modular design of FreeOTFE allows third party drivers to be created to incorporate new hash/cipher algorithms.

Before the user's passphrase is used to encrypt or decrypt the critical data block (CDB), a Passphrase-Based Key Derivation Function 2 (PBKDF2) is applied to it to increase security. PBKDF2 applies one of the cryptographic hashes and one of the ciphers to the entered passphrase, along with a salt value of up to 512 bits and repeats the process many times (depenant upon the number of key iterations chosen by the user – default value is 2048) to produce a derived key. The derived key is then used as the cryptographic key in subsequent operations.

Adding the salt to the passphrase reduces the ability to use a preset dictionary to attack the passphrase. Assume a user's secret key is stolen and he or she is known to use one of 200,000 English words as his or her passphrase. The system uses a default 256-bit salt. Because of this salt, the attacker's pre-calculated hashes are of no value. He or she must calculate the hash of each word with each of 2^256 (1.15792089 x 10 ^{77}) possible salts appended until a match is found. The total number of possible inputs can be obtained by multiplying the number of words in the dictionary with the number of possible salts:

$$2^{256} \times 200,000 = 2.315841785 \times 10^{82}$$

To complete a brute-force attack, the attacker must now compute 2.315841785 x 10^{82} hashes, instead of only 200,000. Even though the passphrase itself is known to be simple, the secret salt makes breaking the passphrase virtually impossible. If the salt changes, the entire attack dictionary needs to be rebuilt.

FreeOTFE uses an encrypted master key system to secure volumes. Every volume has its own master encryption key, which is generated when the volume is created. This master key is used to carry out the actual encryption and decryption process used to secure data stored within the volume. A volume's master encryption key is, in turn, encrypted with the user's passphrase, which has been processed using PBKDF2. As a result, only the encrypted master encryption key needs to be decrypted and re-encrypted and not the entire volume, making changes to a volume/keyfile's passphrase an extremely quick operation compared to a complete volume re-encryption.

Each time a volume is mounted that does not use the default 256-bit salt length, the user must specify the correct salt length. If the user utilises the keyfile function, the keyfile's salt length must be specified.

A keyfile is a small file of about 512 bytes, which the user can generate for each volume created. Keyfiles are useful as they allow the critical information used to mount a volume to be stored separately from the volume to which they relate. For example, a volume may be stored on a computer, but the information required to access it can be stored on a cd-rom or a USB drive and locked in a physically secure location. Keyfiles are encrypted based on a user-supplied keyfile passphrase; this passphrase must be supplied before the keyfile can be used.

Keyfiles may also be used to provide multiple users with the ability to mount and use the same volume, with each using a passphrase of their own choice - this is because keyfiles are specific to the volume they are created for, not to the user who holds them. Although a keyfile for one volume may be able to successfully mount another volume, the virtual drive shown will appear to be unformatted and the files within the volume will remain securely encrypted and unreadable.

Normally a volume's Critical Data Block (CDB) will be stored in the first 512 bytes of the volume. This increases the size of the volume by the size of the CDB, which can make FreeOTFE volumes more distinctive and more obvious that a volume file is in fact a volume file. For example, when creating a file based volume, a 2GB volume will be 2,147,484,160 bytes in length - made up of a 2,147,483,648 byte (2GB) encrypted disk image, plus a 512 byte embedded CDB. However, to get around this weakness, the developers have made it possible to create a volume without the embedded CDB. The CDB is stored in a separate file as a standard keyfile that can be stored in a different location to the volume.

FreeOTFE provides users the ability to create hidden volumes that can be stored inside other host volumes. An offset is used to select the number of bytes from the start of the host volume to determine where the hidden volume should start. The offset must be in multiples of 512 and should be large enough to ensure that system areas of the host volume (FAT or NTFS) or files already written to the host are not overwritten.

More than one hidden volume can be stored within the same host volume by using different offsets. FreeOTFE does not store offset value information so it is important that the same offset value be entered into the passphrase entry dialog every time the hidden volume is mounted or parts of the hidden volume may be overwritten and its data destroyed.

Plausible deniability is made possible through the use of these hidden volumes. Plausible deniability in a FreeOTFE system is largely based on the theory that "you can claim that your volume files are not encrypted data; you don't know what they are - you can't be expected to know every operation that your operating system carries out!" [2].

A claim of plausible deniability is only possible with systems that do not embed any kind of signature into their encrypted data (typically an unencrypted critical data area). FreeOTFE claims to have no such embedding of signatures.

3.2.1 Issues that may affect the plausible deniability of FreeOTFE

The host volume file must be initialised by having random data written to it in order for deniable encryption to be successful. This is required because, to generate a file large enough, the host volume file might have been created by simply writing 0x00's to the hard disk drive. Any hidden volume stored within such a host volume file may cause suspicion as to whether a hidden volume exists, as the hidden volume will appear as a large amount of high-entropy data, stuck in the middle of the volume file; interrupting the neat pattern of 0x00's.

The randomness used for this process cannot simply be pseudorandom data, given the size of a typical volume file, as pseudorandom data can potentially be identified as such, and become predictable. In this case, the hidden volume will not appear as high-entropy data stuck in the middle of a series of 0x00 bytes, but as high-entropy data interrupting the pattern formed by the pseudorandom data.

FreeOTFE recommends that users mount the host volume and overwrite all of the free space with a single pass of pseudorandom data. The data written to the mounted volume will be encrypted as it is written to the host volume file. But, if the user is forced to hand over the key to the outer host volume, analysis could be applied to the mounted plaintext version of the host volume, which may be visible in the pattern of pseudorandom data.

The solution suggested by FreeOTFE is to encrypt the pseudorandom data before it is used to overwrite the mounted volume's free space; thereby making it impossible to differentiate between any encrypted pseudorandom number generated data and the encrypted hidden volume, even with the key to the outer host volume.

If the FreeOTFE drive is not dismounted after use, the virtual drive and the sensitive information contained within it will be stored in the computers' memory.

3.3 TrueCrypt

Like FreeOTFE, TrueCrypt [5] is a software system for establishing and maintaining an on the fly encrypted volume. TrueCrypt never saves any decrypted data to disk; it stores it temporarily in RAM. When the volume is mounted, data stored in the volume is still encrypted. When Windows is restarted or the computer is turned off, the volume is dismounted and files stored within it become inaccessible and encrypted. If the power supply is suddenly interrupted, without proper system shut down, files stored in the volume become inaccessible and encrypted. To make the files accessible again, the volume must be remounted and the correct passphrase and/or keyfile must be provided.

As with FreeOTFE, TrueCrypt creates a volume that can reside inside a file, also called container, inside a partition or inside a drive. The minimum size for a TrueCrypt volume is 275 KB for a FAT volume and 2829 KB for an NTFS volume.

To create a volume, the user must select cryptographic hash and cipher algorithms, choose a passphrase and generate random data by moving the mouse around the screen for up to 30 seconds. This random data is then used to format the volume and generate the encryption keys.

Hash and cipher algorithms supported by TrueCrypt are as follows: Hash algorithms - SHA-512, RIPEMD-160 and Whirlpool. Cipher algorithms - AES 256, Serpent, Twofish, AES-Twofish, AES-Twofish-Serpent, Serpent-AES, Serpent-Twofish-AES and Twofish-Serpent.

TrueCrypt supports two types of plausible deniability: hidden volumes (see sub-section 3.3.1) and a hidden operating system (see sub-section 3.3.3).

3.3.1 TrueCrypt hidden volumes

When creating a hidden volume, it can be challenging for all but the most experienced and accomplished user to set the size of the hidden volume in a way that the hidden volume does not overwrite data on the outer volume. Therefore, a volume creation wizard has been provided which automatically scans the cluster bitmap of the outer volume to determine the maximum possible size of the hidden volume. The wizard scans the cluster bitmap to determine if there is an area of uninterrupted free space whose end is aligned with the end of the outer volume. The area of uninterrupted free space limits the size of the hidden volume. On Linux and Mac OS X platforms, the wizard does not scan the cluster bitmap; instead the driver detects any data written to the outer volume and uses their position as described above.

When saving data to the outer volume, it is possible to overwrite data on the hidden volume. TrueCrypt provides the option of protecting the hidden volume from overwriting by decrypting the hidden volume header in RAM and retrieving information about the size of the hidden volume. The outer volume is then mounted and any attempt to save data to the area of the hidden volume is declined until the outer volume is dismounted. As soon as the volume is dismounted the hidden volume protection is lost and there is no way of determining whether or not the volume ever used hidden volume protection. Mounting an outer volume with hidden volume protection automatically enabled would seriously hinder the hidden volumes plausible deniability, as its existence within the outer volume would be visible. Hidden volume protection is therefore automatically disabled after a mount attempt is completed, regardless of whether it is successful or not.

A hidden volume can be mounted the same way as a standard volume; the volume mounted is determined by which passphrase is entered. For example, when the passphrase for the outer volume is entered, the outer volume is mounted; when the passphrase for the hidden volume is entered, the hidden volume is mounted.

TrueCrypt first attempts to decrypt the standard volume header using the passphrase entered by the user. If this passphrase fails, the area of the volume where the hidden volume header can be stored (see Figure 5 below) is opened in RAM and attempts are made to decrypt it. If the header is successfully decrypted, the information about the size of the hidden volume is retrieved from the decrypted header, which is still in RAM, and the hidden volume is mounted. Hidden volume headers cannot be identified as such, as they appear to consist entirely of random data.

The principle of the TrueCrypt hidden volume is that one TrueCrypt volume is created within the free space of another TrueCrypt volume (see Figures 4 and 5 below). Even when the outer volume is mounted, it is impossible to prove that a hidden volume resides within it; this is because free space on any TrueCrypt volume is filled with random data when the volume is created and no part of the dismounted hidden volume can be distinguished from this random data. In fact, TrueCrypt claims that it is impossible to identify a TrueCrypt volume as being such since, until decrypted, a TrueCrypt volume appears to consist of nothing more than random data, containing no signature of any kind, thus making it impossible to prove that a file, a partition or a device is a TrueCrypt volume or that it has been encrypted. However, this does not hold true if the Quick Format and Dynamic options are used when the volume is created.

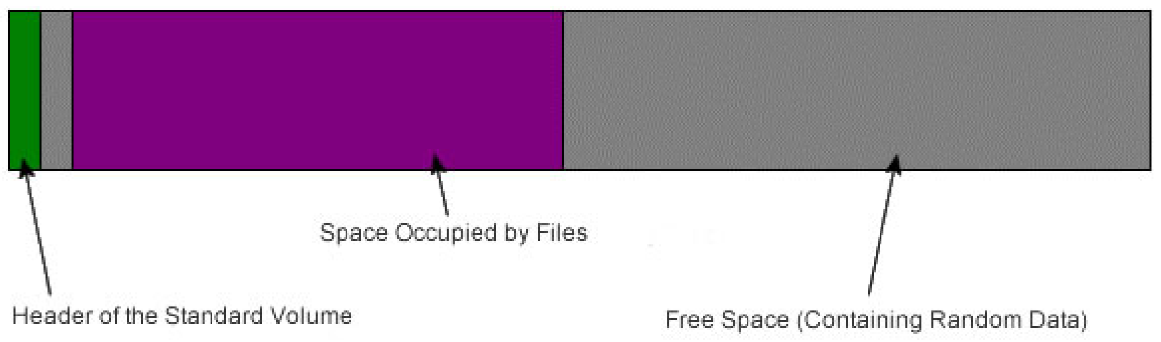

Space Occupied by Files
Header of the Standard Volume
Free Space (Containing Random Data)

Figure 4: A standard TrueCrypt volume

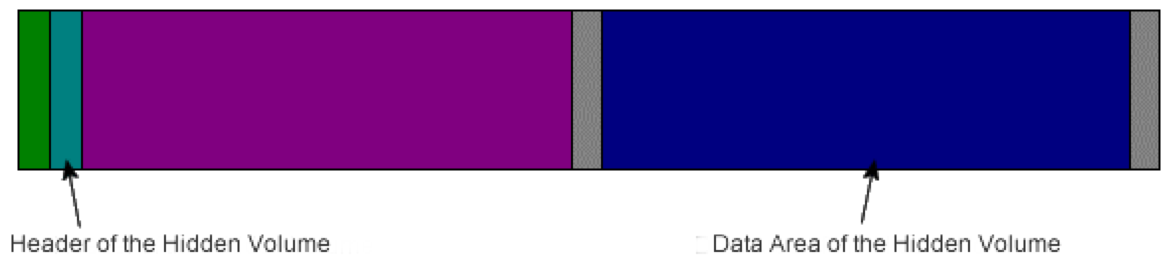


Figure 5: A standard TrueCrypt volume after a hidden volume has been created within it

In order to achieve and maintain plausible deniability, TrueCrypt does not modify the file system or change information about free space within the outer volume in any way. However, the optional .tc file extension, which is officially associated with TrueCrypt and must not be added to the volume if plausible deniability is to be maintained, may inadvertently be added as a file extension by an inexperienced user, therefore making it possible to identify volumes and hidden volumes.

When formatting a hard disk partition as a TrueCrypt volume, the partition table, which includes the partition type, is never modified and no TrueCrypt signature or ID is written to the partition table. However, it may be possible to identify activity on a file-hosted volume or a keyfile because TrueCrypt preserves the timestamp of when the container or keyfile was last accessed or last modified, unless the user disables this behaviour in the preferences section. This is another area where the efforts to achieve and maintain plausible deniability may fall down. Details of other well-published weaknesses in maintaining plausible deniability in TrueCrypt hidden volumes are detailed in Section 4.3.2 below.

3.3.2 Issues that may affect plausible deniability of hidden volumes in TrueCrypt

If a challenger has access to a dismounted TrueCrypt volume at several points over time, he or she may be able to determine which sectors of the volume are changing due to normal operations such as creating and copying new files to the hidden volume, modifying, deleting, renaming or moving files from the hidden volume. These everyday operations will visibly alter the cipher text contained in the sectors of the hidden volume, therefore, indicating the presence of the hidden volume.

The issue described above may also occur when the file system, in which a file-hosted TrueCrypt container is stored, has been defragmented and a copy of the TrueCrypt container, or some of its fragments, remain in the free space of the defragmented file system on the host volume and also stored in a journaling file system such as that provided on NTFS.

To counteract these weaknesses, TrueCrypt recommends that the user securely erase free space on the host volume after defragmenting or avoid defragmenting file systems in which TrueCrypt volumes are stored. Also, they recommend that users store containers in a non-journaling file system such as FAT32.

Another important issue that can affect plausible deniability is the wear levelling technique used on many USB devices. Wear levelling is a technique used to help prolong the life of USB flash drives and works by distributing data evenly across each memory block of the entire drive. This process decreases the total wear on the drive, thereby increasing the lifetime of the drive [3]. Wear levelling can leave a copy or a fragment of the TrueCrypt volume on the flash drive.

3.3.3 TrueCrypt hidden operating system

A hidden operating system is installed inside a hidden TrueCrypt volume. In order to boot a system encrypted by TrueCrypt, an unencrypted copy of the TrueCrypt Boot Loader must be stored on the system drive or on a TrueCrypt Rescue Disk. The presence of the TrueCrypt Boot Loader can indicate that there is a system encrypted by TrueCrypt on the computer. Therefore, a second encrypted operating system called a decoy operating system must be created to provide a plausible explanation for the presence of the TrueCrypt Boot Loader.

As Figure 6 below shows, the decoy operating system is not installed in the hidden volume; therefore, it should not contain any sensitive files or data. As the existence of the decoy is not designed to be secret, the decoy operating system passphrase can be revealed to a challenger demanding the pre-boot authentication passphrase.

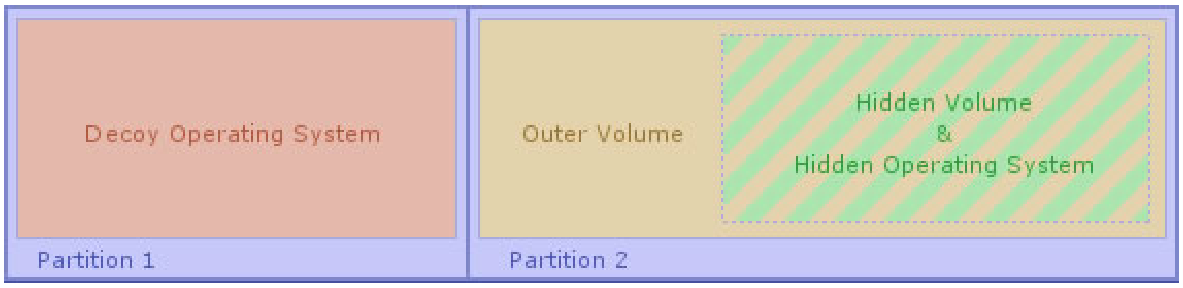


Figure 6: Example layout of system drive containing a hidden operating system

There are two pre-boot authentication passphrases: one for the hidden system and one for the decoy system. To start the hidden operating system, the hidden system passphrase should be entered into the TrueCrypt Boot Loader screen. Alternatively, to start the decoy system, the passphrase for the decoy operating system should be entered into the TrueCrypt Boot Loader screen.

When the pre-boot authentication passphrase is entered, the TrueCrypt Boot Loader first attempts to decrypt the last 512 bytes of the first logical track of the system drive, as this is where the encrypted master key data for non-hidden encrypted system partitions are stored. If authentication fails, the TrueCrypt Boot Loader automatically tries to decrypt the area of the first partition behind the boot partition where the encrypted header of a possible hidden volume might be stored. As mentioned previously, TrueCrypt never knows if there is a hidden volume in advance.

If the header is decrypted successfully, the information about the size of the hidden volume is retrieved from the decrypted header, which is still stored in RAM, and the hidden volume is mounted, determined by the size of its offset.

When running, the hidden operating system appears to be installed on the same partition as the decoy system. However, in reality, it is installed within the partition behind it in the hidden volume. All read and write operations are transparently redirected from the system partition to the hidden volume. Neither the operating system nor applications running on the system will know that data written to and read from the system partition are actually written to and read from the hidden volume. Data is encrypted and decrypted on the fly, with a different encryption key from the one that is used for the decoy operating system.

A TrueCrypt system makes use of a third passphrase. This is not a pre-boot authentication passphrase, but a standard TrueCrypt volume passphrase, which can be disclosed to a challenger if the user is forced to reveal the passphrase for the encrypted partition where the hidden volume containing the hidden operating system resides. This allows the hidden volumes' existence to remain undisclosed.

When the hidden operating system is created, the user must create a partition for it on the system drive. It must be the first partition behind the system partition and it must be at least 5% larger than the system partition. However, if the outer volume is formatted as NTFS, the partition for the hidden operating system must be at least 110% larger than the system partition. This is due to the NTFS file system storing internal data exactly in the middle of the volume and, therefore, the hidden volume, which is to contain a clone of the system partition, only being able to reside in the second half of the partition.

After some sensitive-looking files have been copied to the outer volume, the cluster bitmap of the volume is scanned to determine the size of uninterrupted area of free space whose end is aligned with the end of the outer volume. The maximum possible size of the hidden volume is determined by the size of the system partition and verification will be required to ensure that it is greater than the size of the system partition, thus ensuring that no data stored on the outer volume will be overwritten by data written to the area of the hidden volume.

The process of copying the system is performed in the pre-boot environment before Windows starts and, depending on the size of the system partition and performance of the computer, may take several hours or even several days to complete.

When a hidden operating system is running, TrueCrypt ensures that all local unencrypted filesystems and non-hidden TrueCrypt volumes are read-only. Data can be written to filesystems within hidden TrueCrypt volumes during this time. The reason for this is that it enables the creation of a secure environment for the mounting of hidden TrueCrypt volumes and it prevents the analysis and comparing of filesystem journals, file timestamps, application logs and error logs.

3.3.4 Issues that may affect plausible deniability of the hidden operating system in TrueCrypt

If the decoy operating system were not used frequently enough, i.e. for all non-sensitive data, the absence of regular activity would indicate the presence of a hidden operating system on the computer.

To prevent Windows saving parts of the program and data files that do not fit into memory to the hard drive, paging files must be disabled on the system until the process of creating a hidden operating system is finalised, otherwise, plausible deniability of the hidden operating system will be compromised. The TrueCrypt installer disables paging files by default when TrueCrypt is installed or updated. If paging is enabled, the user cannot continue creating the hidden system. However, if during the process of creating the hidden operating system, the user hibernates the system, the contents of the systems memory will be written to the hibernation storage file residing on the systems drive, thereby compromising the plausible deniability of the hidden operating system.

Most operating systems, including Windows, can be configured to write debugging information and contents of the system memory to memory dump files when an error occurs. These memory dump files may contain sensitive data. TrueCrypt cannot prevent cached passphrases, encryption keys, and the contents of sensitive files opened in RAM from being saved unencrypted to memory dump files. When a file stored on a TrueCrypt volume is opened in a text editor, for example, the content of the file is stored unencrypted in RAM until the computer is switched off. Also when a TrueCrypt volume is mounted, its unencrypted master key is stored in RAM.

To avoid this, users must disable memory dump file generation on their computers whilst working with sensitive data and whilst mounting a TrueCrypt volume.

TrueCrypt is disk encryption software; it does not encrypt data in RAM. Most programs do not clear the buffers in which they store unencrypted portions of files they load from the TrueCrypt volume, allowing unencrypted data to remain in memory until the computer is switched off.

If a user opens a file that is stored in a TrueCrypt volume using a text editor and then he forces the TrueCrypt volume to dismount, the file will remain unencrypted in the RAM allocated to the text editor. This also applies to forced auto-dismount.

Inherently, unencrypted master keys are stored in RAM that are erased when the computer is cleanly restarted or shut down and the TrueCrypt volumes automatically dismounted. However, when a computer loses its power supply suddenly, is reset, hibernates or the system crashes, the keys and other sensitive data cannot be erased from RAM.

The volume header contains the master key with which a volume is encrypted. If a challenger is allowed to make a copy of the volume before the volume passphrase and/or keyfile(s) are changed, the challenger may be able to use a copy of the old header to mount the volume using a compromised passphrase gained through keylogging software.

If a challenger knows the passphrase or has the keyfiles and has access to the volume, it is possible to retrieve and keep its master key. If this happens, the volume can be decrypted even after the passphrase has been changed; this is due to master keys never changing, even when the volume passphrase or keyfiles are changed. To recover from this, the user must create a new TrueCrypt volume and move all files from the old volume to the new one. The same situation may occur when defragmenting has been performed.

Windows stores various data in the registry file, which TrueCrypt cannot securely and reliably erase. After examining the registry file, a challenger may be able to tell that TrueCrypt was run on the system and that a TrueCrypt volume was mounted. However, the challenger cannot determine what the location/filename/size/type the volume was and which drive letters have been used for TrueCrypt volume(s). To prevent this, the registry file must be encrypted, by encrypting the system partition or drive.

When the file system in which a TrueCrypt container is stored is defragmented, a copy of the TrueCrypt container can remain in the free space on the host volume. This has security implications in that if the volume passphrases or keyfile(s) are changed, a challenger may find the old header of the TrueCrypt volume and use it to mount the old volume using an old, compromised passphrase. This can be avoided by using a partition/device-hosted TrueCrypt volume instead of a file-hosted one, erasing free space on the host volume and refraining from defragmenting systems in which TrueCrypt volumes are stored.

Finally, when a file-hosted TrueCrypt container is stored in a journaling file system, a copy of the TrueCrypt container may remain in the free space on the host volume. This has the same security implications as discussed in defragmenting above, with the addition that some journaling file systems also internally record file access times and other potentially sensitive information. To prevent security issues related to journaling file systems, a partition/device-hosted TrueCrypt volume can be used instead of file-hosted one or the container can be stored in a non-journaling file system such as FAT32.

3.4 PhoneBookFS

PhoneBookFS [6] is a filesystem for Linux operating systems that offers encryption with plausible deniability features. Hash and cipher algorithms supported by PhoneBook are SHA-1 and 256-bit Blowfish respectively. Inspired by Rubberhose, PhoneBook aims to be easy to install and use and is aimed at intermediate to advanced Linux users.

PhoneBook uses encrypted volumes with multiple layers of encryption and 'chaffing' to make it difficult or impossible for a challenger to determine a users full compliance with decryption orders. See Figure 7 below.

For each logical file or directory that can be seen from the mount point, two physical files are written to the datastore: an inode file, which contains the properties, and the real data file.

The name of the inode file is deterministically hashed from a combination of the layer name and passphrase plus a randomly generated 31-bit inode number. The passphrase and initialisation vector (IV) for the inode file are a deterministic hash of layername+layerpassphrase+inodenum. The root directory inode number for each is hardwired as zero.

The name of the physical data file, passphrase and IV are all randomly selected. These details are stored in cleartext within the inode file and protected by the inode file’s encryption.

The layermap file is another file type that is stored in the physical datastore. The name of the layermap file, its passphrase and IV are a deterministic hash of permutations of layermapname+layermappassphrase. The layermap names and passphrases effectively serve as 'skeleton keys', which unlock the names, passphrases and IVs of all the constituent layers and files.

When the ls (list directory contents) command is entered on the command line, a list of files is displayed; these files have garbled names and garbled content. Any challenger analysing the filesystem and the files contained within it should not be able to make sense of it. With PhoneBook, the more layers that are created and chaffing applied, the more difficult it becomes to determine how many layermaps, layers or files are contained in the filesystem, this is because it becomes impossible to determine what is chaff and what is not.

The number of layers contained in the filesystem contributes greatly to its security; if there are only a few layers the system can become vulnerable to dictionary attacks and passphrase guessing attacks. However, the greater the number of layers created, the greater the number of passphrases that the user is required to remember. This contributes to weak passphrase choices and insecure methods of remembering them.

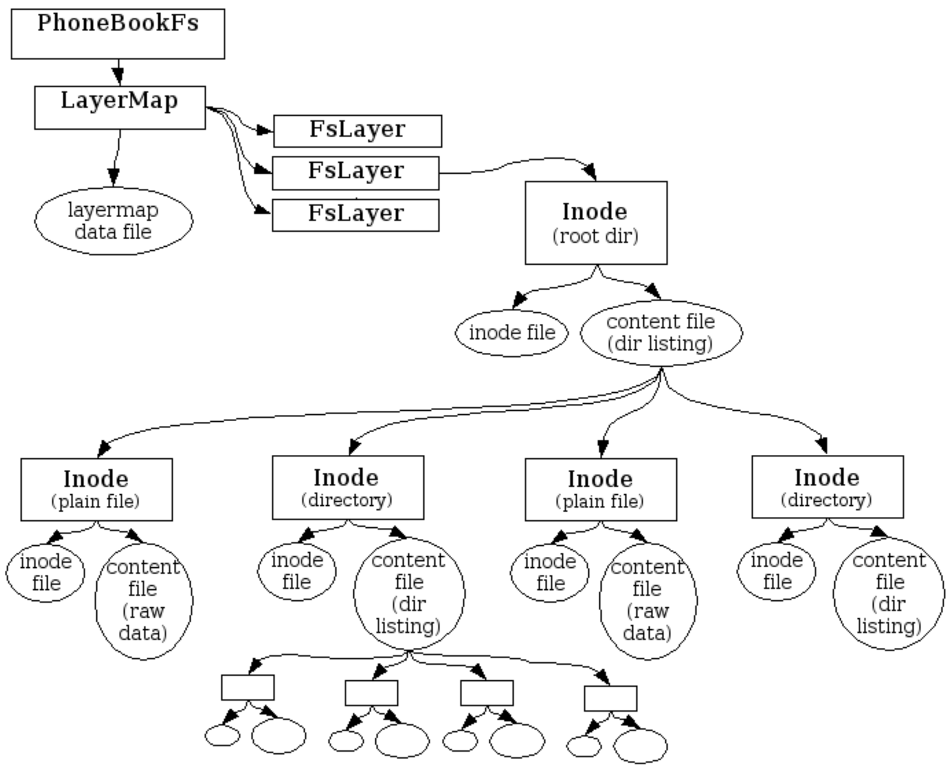


Figure 7: PhoneBook filesystem file structure

To counteract this, PhoneBook introduced layermaps. The basic concept behind layermaps is that a user can create a named layermap with its own passphrase and add any combination of layers to it. The layermap can then be opened using a single name and passphrase and all of the layers associated with it will be automatically activated in the right order. A major advantage of using layermaps, from a forensic point of view, is that if the passphrase for the layermap is obtained it will yield all of the data on all of the layers that are contained within it.

The filesystem in PhoneBook is hierarchical. For example, if both layer1 and layer2 have a file called /somedir/somefile.jpg and the user tries to open /mnt/pbfs/somedir/somefile.jpg, he will get the version of somefile.jpg from layer1. Any files written to the filesystem will also be written to the top-most layer, in this case, layer1. However, to take full and advantage of the layered cryptographic features of PhoneBook, the user will, at times, need to control exactly which layers he reads from and writes to. To this end, mounted PhoneBook filesystems have a 'magic directory' called __layers. Each active layer appears by name as a subdirectory of __layers.

As mentioned earlier, PhoneBook supports plausible deniability by adding chaff to the physical datastore, thereby making it difficult, if not impossible, to determine the data that has intentionally been hidden. This is compounded by all of the files in the datastore having hashed or random names, encrypted or random contents, and random dates. PhoneBook recommends that between 10% and 400% of the total amount of real data or 100% or 500% of the size of the largest layer be added as chaff.

3.4.1 Issues that may affect plausible deniability in PhoneBook

The pbfs utility in Linux allows passphrases to be entered on the command line. If this is done, passphrases are automatically copied into the shell command history files.

It is possible for PhoneBook internal data structures to be swapped out to disk using some versions of Linux, thereby revealing layer details and passphrases.

A PhoneBook’s filesystem mount point can be exported over a local network but this will lead to layer/layermap names and passphrases being transported in plaintext.

3.5 Rubberhose

Rubberhose [7] is a freely available disk encryption system that allows the hiding of encrypted data. Released in 1997, Rubberhose was the first successful deniable cryptography program in the world. Rubberhose currently runs on Linux and NetBSD platforms but there are plans to port it to Windows NT.

Cipher algorithms supported by Rubberhose are DES, 3DES, IDEA, RC-6, Blowfish, Twofish and CAST. In addition to the built-in ciphers, Rubberhose supports all of the symmetric algorithms from the latest release of OpenSSL. The modular design of Rubberhose makes it easy to add new algorithms.

When Rubberhose is run for the first time the program writes random characters to the entire drive. Rubberhose refers to the drive as an "extent". The random noise generated in this initialisation is indecipherable from the encrypted data that will be stored on the disk.

When Rubberhose creates partitions, known as 'aspects', it does not operate like a normal disk partitioning program, taking large blocks for each partition, instead, it breaks up the pieces of the encrypted portion into very small pieces and randomly scatters them across the entire drive so that the bits of data cannot be tracked and re-assembled.

For example, if we have a hard drive of 1GB and we wish to fill the first Rubberhose-encrypted aspect with 400MB of data and the second aspect with 200MB of data, Rubberhose does not know that we intend to divide the aspects in this way; it assumes that each of the aspects will be 1GB in size and fills the whole drive. Rubberhose fills the drive on a first come first serve basis and it continues to provide more space to any one aspects until the overall disk capacity has been reached.

When the 400MB encrypted aspect is decrypted it looks as though it is 1GB in size with 600MB of free space. This is how Rubberhose hides the existence of data in the remaining aspect of the disk. When the second 200MB encrypted aspect is created, Rubberhose writes these bits randomly over the whole 1GB drive, taking care not to overwrite the bits assigned to the first 400MB encrypted aspect. Each Rubberhose aspect has its own passphrase and must be decrypted separately.

A Rubberhose aspect is simply a view of the extent from a different perspective, like viewing the same object from different angles. If the data is seized, a challenger will be unable to determine how many aspects are on the drive or how much real data is on any one aspect.

Rubberhose relies on internal maps to locate where the actual bits of data are stored amid the random characters. Each aspect has its own corresponding map and that aspect's map can only be decrypted when the correct passphrase has been provided. Rubberhose aspects know nothing about other aspects on the extent, including their size, maps or existence. This prevents one aspect divulging the presence of another. The only thing an aspect needs to know is when to avoid overwriting other aspects. When a passphrase is entered, Rubberhose will only decrypt the tables that map the bits for that one single aspect; all the other mapping tables for other Rubberhose aspects stay securely encrypted.

Every time a new aspect is created, Rubberhose generates a very long and randomly generated secret internal master passphrase. This internal master passphrase is then encrypted with the passphrase provided by the user for the new aspect. The internal master passphrase stays the same for the lifetime of the aspect, but the passphrase for the aspect can change any number of times. The use of the internal master passphrase makes Rubberhose highly resistant to dictionary attacks.

When a new passphrase is created, Rubberhose loops for a period of time re-encrypting the first output into the second, and the second into the third. Each subsequent encryption output is fed back into the next round of encryption. This makes an attack based on guessing the passphrase extremely difficult, as an attacker must test each guess through the entire loop. The only way that an attacker can obtain the original master passphrase is to reverse the process, feeding the output of the decryption into the input of the next decryption -- through about 175,000 rounds, depending on the speed of the symmetric cipher algorithm selected.

The salted internal master aspect key unlocks everything in that aspect, including the map of where the bits of data are located across the drive and the ability to actually decrypt that data. Since this key could be a point of vulnerability, Rubberhose has designed large walls around the master key. In fact, it could be said that the master key doesn't touch anything else in the program directly; there is always an obscuring barrier protecting it.

Rubberhose ensures that if a challenger manages to decrypt one of the many small blocks of data in an aspect, it will not help the challenger to decrypt the remaining blocks and therefore piece together all the data from an entire aspect.

Using strong pseudo-random number generators, Rubberhose creates a lattice generator for each aspect. This lattice, which can only be decrypted by using the internal master key for that aspect, is used for calculating unique encryption and decryption keys for each block of data assigned to a single aspect. Specifically, the lattice uses a mathematical algorithm to transmute a block number into a key for that block. Rubberhose puts these blocks inside its own larger blocks.

A Rubberhose-encrypted drive has two types of block: an operating system block (typically 512-8192 bytes) and a Rubberhose 'surface-block', which are randomly scattered all over the drive. The size of the surface blocks can be configured to suit the needs of the user when Rubberhose is initialised on the hard drive. Setting Rubberhose to create many smaller surface blocks is more secure than choosing larger sized blocks, but these will take longer to generate.

In theory a challenger can examine the magnetic properties of the ferrite coating on a disk surface in order to determine how frequently a program has read or written to a particular section of the drive. This permits the challenger to guess if a geographic area on the disk is blank (full of random noise) or contains hidden data. If the challenger can decrypt, for example, aspect 1, but not any other aspect, he can overlay a map of frequently used drive sections on a map of aspect 1's data map showing unused and used sections. If he sees an unused section has been accessed for reading or writing very frequently, he can guess that there is more likelihood that there is hidden material stored there from another aspect. However, to thwart disk surface analysis based on the intensity of block use and contiguous block prediction, Rubberhose conducts frequent block-swapping by automatically and invisibly shuffling surface blocks of data around to new locations on the drive.

Rubberhose uses a technique called 'whitening code' to reduce the effectiveness of cryptanalysis. The whitener is made up of random bits which, when merged with decrypted data from the data blocks, changes the data in the data blocks slightly by flipping a corresponding bit in the data block. For example, any bit in the whitener that is a 1, flips a corresponding bit in the decrypted data block. All 0 bits are "held" or "not flipped". The result is that some bits in the decrypted data block are flipped and some are not. This prevents known plaintext attacks because the challenger has no plaintext to work with that is directly translatable into the encrypted material.

3.5.1 Issues that may affect plausible deniability in Rubberhose

The whitener is vulnerable in the sense that if it is broken for one block, it can be broken for all blocks. However, there are still other lines of defence, such as separate keys for every block, which limits the success of cryptanalysis.

CHAPTER 4 METHODOLOGY

This chapter looks at the methods and principles applied and tools used during the analysis phase in order to achieve the results obtained in Chapter 5.

4.1 Methods and principles

One of the deniable encryption products reviewed in Chapter 3 was selected for in-depth disk surface analysis to investigate whether or not evidence of a hidden volume could be found. FreeOTFE [4] was chosen because it operates on the MS Windows platform, the most common operating system for laptop and personal computers and also because it is a free and open source product, which increases its attractiveness to potential users. In addition, FreeOTFE was chosen because, unlike TrueCrypt [5], no similar analysis has been carried out.

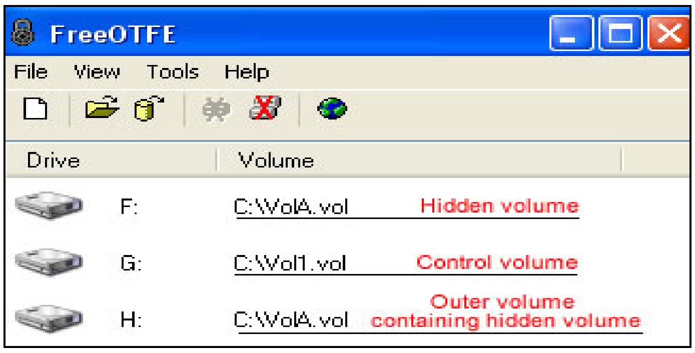
Two FreeOTFE volumes were created on the C drive. Both volumes were created using the SHA-512 hash algorithm and AES 256-bit XTS cipher. Microsoft CryptoAPI random number generator was used in the creation of both volumes to salt the password, create the volume's master key and create random padding data. FreeOTFE encrypts the pseudorandom data before it is used to overwrite the mounted volume's free space; thereby making it impossible to differentiate between any encrypted pseudorandom number generated data and the encrypted hidden volume, even with the key to the outer host volume.

Once the volumes were successfully created, they were mounted, formatted and the free space overwritten/shredded with pseudorandom data. Each of the mounted volumes were automatically allocated the following drive letters:

Volume 1 (control volume)
The first volume, named 'Vol1', was created as a control volume. This volume is 60MB in size and contains a small Microsoft Word document called doc1CV.doc. This volume is used to gain an understanding of the layout and setup of a standard volume. When Volume 1 was mounted, it was given a drive address of G, see Figure 8 below.

Volume 2 (outer volume)
The second volume, named 'VolA', was created as the outer/host volume inside which the hidden volume would be created. This outer volume is 200MB in size and contains a small Microsoft Word document called doc1OV.doc. When Volume 2 was mounted, it was given a drive address of H, see Figure 8 below.

Volume 2 (hidden volume)
The hidden volume was created inside 'VolA' and was created as 60MB in size with an offset of 102400. This means that the hidden part of the volume starts at byte 102400 of the outer/host volume. A small Microsoft Word document called doc2HV.doc was placed inside the hidden volume. When the hidden volume was mounted, by entering the passphrase and offset value, it was given a drive address of F, see Figure 8 below.

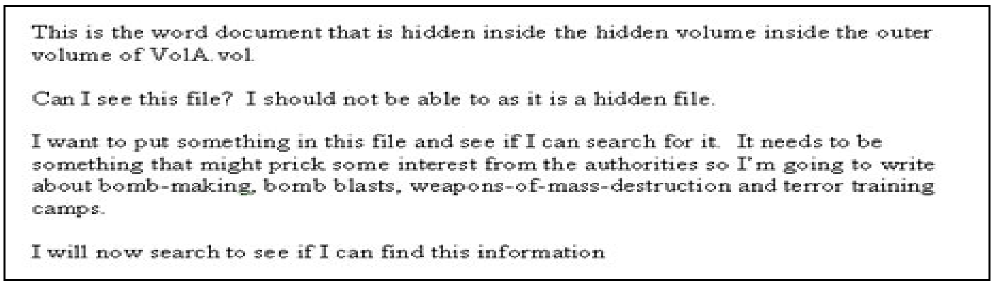


Drive	Volume
F:	C:\VolA.vol Hidden volume
G:	C:\Vol1.vol Control volume
H:	C:\VolA.vol Outer volume containing hidden volume

Figure 8: FreeOTFE mounted volumes

As mentioned above, a Microsoft Word document was added to each of the volumes in order to determine whether contents of those documents would be leaked into other areas of the operating system or environment under normal operating conditions, thereby affecting the product's ability to maintain plausible deniability.

Each of the documents created contain suspicious content that may prick the attention or interest of authorities that exercise a search and seizure order on a user's laptop or personal computer. Figure 9 below shows the contents of the doc2HV.doc document that was placed inside the hidden volume.


This is the word document that is hidden inside the hidden volume inside the outer volume of VolA.vol.

Can I see this file? I should not be able to as it is a hidden file.

I want to put something in this file and see if I can search for it. It needs to be something that might prick some interest from the authorities so I'm going to write about bomb-making, bomb blasts, weapons-of-mass-destruction and terror training camps.

I will now search to see if I can find this information

Figure 9: Contents of doc2HV.doc

Once the documents were placed inside their respective volume, all volumes were dismounted. This is in compliance with FreeOTFE instructions that claim "if drives are not dismounted after use, the virtual drive and the sensitive information contained within it will be stored in the computers' memory".

The aim of this research is to establish whether or not evidence of a hidden volume can be found using forensic analysis tools. It is not the intention of this research to decrypt or view the data contained within the hidden volume.

4.2 Disk analysis tool used

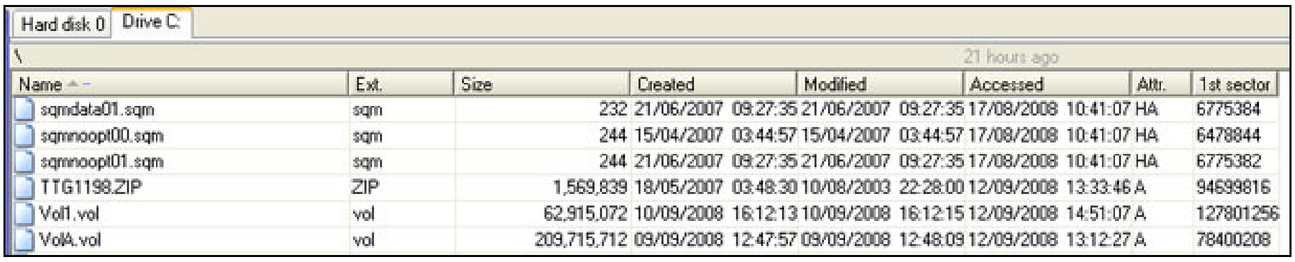
The evaluation version of WinHex [11], made by X-Ways Software Technology AG of Germany, was the disk analysis tool used to perform disk surface analysis and gather evidence during the project. WinHex was used due to its compatibility with the Windows platform, its forensic features and its easy to use interface. Features include: case management, log and report feature, report tables, volume snapshots, directory browser, internal viewer, logical and simultaneous search, indexing, hash database and evidence file containers.

Unfortunately, many of the advanced forensics features of WinHex were not available to me during this project as the evaluation version was being used; this made disk analysis very time consuming and cumbersome.

CHAPTER 5 RESULTS AND DISCUSSION

This chapter of the report presents the results and findings obtained during the process of disk surface analysis. Disk surface analysis focused on three main areas: examination the volume's critical data block, see Section 5.1, breakdown of the Master File Table (MFT) entries for the outer volume and the files contained inside the hidden volume, see Section 5.2, and searching for evidence of data leakage from the hidden volume into other areas of the hard disk, see Section 5.3.

As can be seen from Figure 10 below, WinHex shows FreeOTFE Volume 1 (Vol1.vol) and Volume 2 (VolA.vol) but does not show the hidden volume created inside Volume 2.



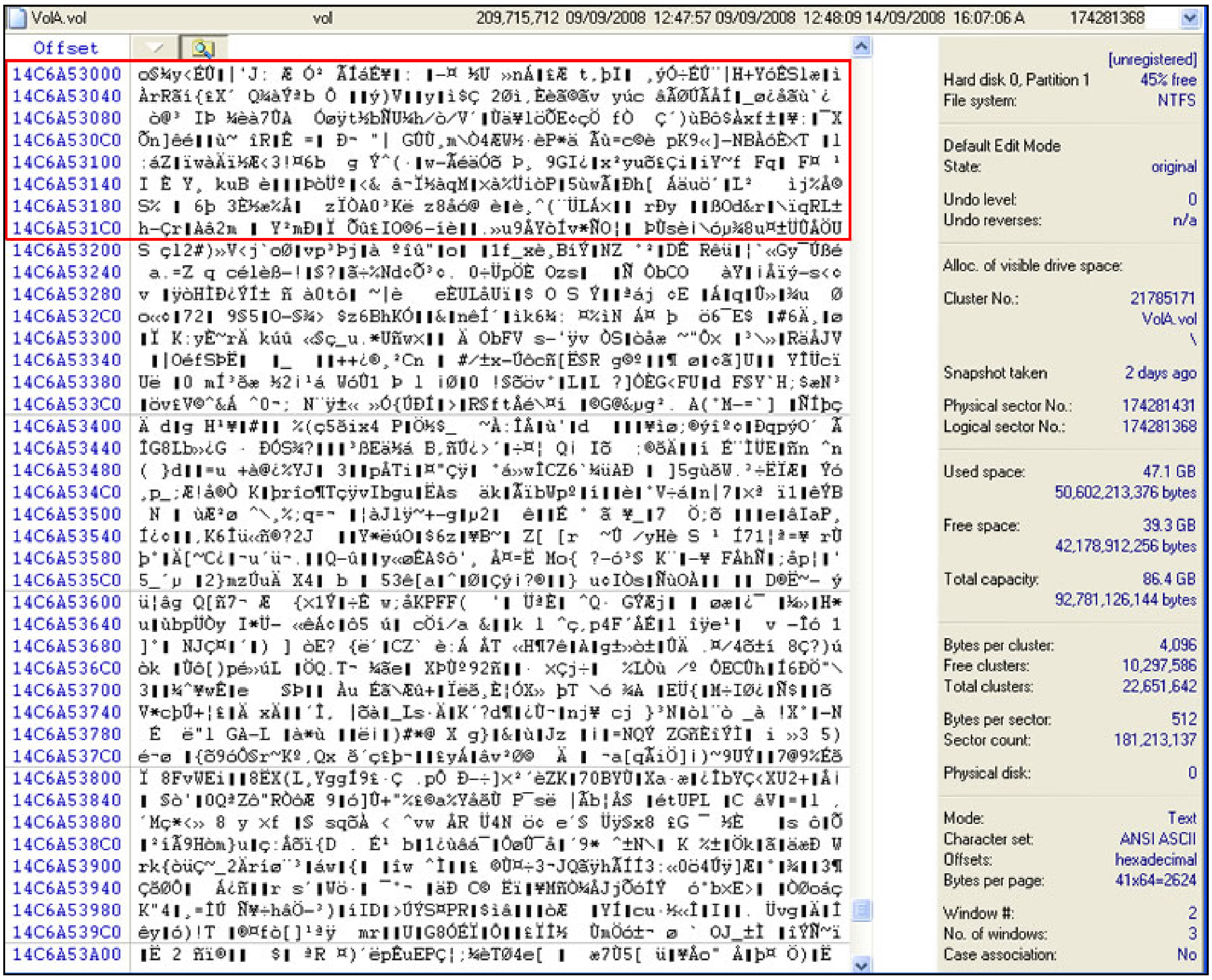
Hard disk 0 | Drive C:

\ 21 hours ago

Name	Ext.	Size	Created	Modified	Accessed	Attr.	1st sector
sqmdata01.sqm	sqm	232	21/06/2007 09:27:35	21/06/2007 09:27:35	17/08/2008 10:41:07	HA	6775384
sqmnoopt00.sqm	sqm	244	15/04/2007 03:44:57	15/04/2007 03:44:57	17/08/2008 10:41:07	HA	6478844
sqmnoopt01.sqm	sqm	244	21/06/2007 09:27:35	21/06/2007 09:27:35	17/08/2008 10:41:07	HA	6775382
TTG1198.ZIP	ZIP	1,569,839	18/05/2007 03:48:30	10/08/2003 22:28:00	12/09/2008 13:33:46	A	94699816
Vol1.vol	vol	62,915,072	10/09/2008 16:12:13	10/09/2008 16:12:15	12/09/2008 14:51:07	A	127801256
VolA.vol	vol	209,715,712	09/09/2008 12:47:57	09/09/2008 12:48:09	12/09/2008 13:12:27	A	78400208

Figure 10: FreeOTFE volumes in Drive C

Figure 11 shows that VolA.vol is within Hard Disk 0, Partition 1, which is the C Drive and has a cluster number of 21785171, physical sector number of 174281431 and logical sector number of 174281368 and demonstrates that the full volume has been overwritten with pseudorandom data.



VolA.vol vol 209,715,712 09/09/2008 12:47:57 09/09/2008 12:48:09 14/09/2008 16:07:06 A 174281368

```
Offset
14C6A53000
14C6A53040
14C6A53080
14C6A530C0
14C6A53100
14C6A53140
14C6A53180
14C6A531C0
14C6A53200
14C6A53240
14C6A53280
14C6A532C0
14C6A53300
14C6A53340
14C6A53380
14C6A533C0
14C6A53400
14C6A53440
14C6A53480
14C6A534C0
14C6A53500
14C6A53540
14C6A53580
14C6A535C0
14C6A53600
14C6A53640
14C6A53680
14C6A536C0
14C6A53700
14C6A53740
14C6A53780
14C6A537C0
14C6A53800
14C6A53840
14C6A53880
14C6A538C0
14C6A53900
14C6A53940
14C6A53980
14C6A539C0
14C6A53A00
```

	[unregistered]
Hard disk 0, Partition 1	45% free
File system:	NTFS
Default Edit Mode	
State:	original
Undo level:	0
Undo reverses:	n/a
Alloc. of visible drive space:	
Cluster No.:	21785171
	VolA.vol
	\
Snapshot taken	2 days ago
Physical sector No.:	174281431
Logical sector No.:	174281368
Used space:	47.1 GB
	50,602,213,376 bytes
Free space:	39.3 GB
	42,178,912,256 bytes
Total capacity:	86.4 GB
	92,781,126,144 bytes
Bytes per cluster:	4,096
Free clusters:	10,297,586
Total clusters:	22,651,642
Bytes per sector:	512
Sector count:	181,213,137
Physical disk:	0
Mode:	Text
Character set:	ANSI ASCII
Offsets:	hexadecimal
Bytes per page:	41x64=2624
Window #:	2
No. of windows:	3
Case association:	No

Figure 11: VolA.vol location information and critical data block

5.1 Examination of the volume's critical data block

Each volume's critical data block (CDB) is contained within the first 512 bytes of the volume, illustrated by the red box in Figure 11 above. The critical data block contains the password salt, CBD format ID, volume flags, encrypted partition image, master key length, master key, requested drive letter, volume initialisation vector length, sector initialisation vector method and random padding, see Figure 12. The CDB could provide us with valuable information about its associated volume.

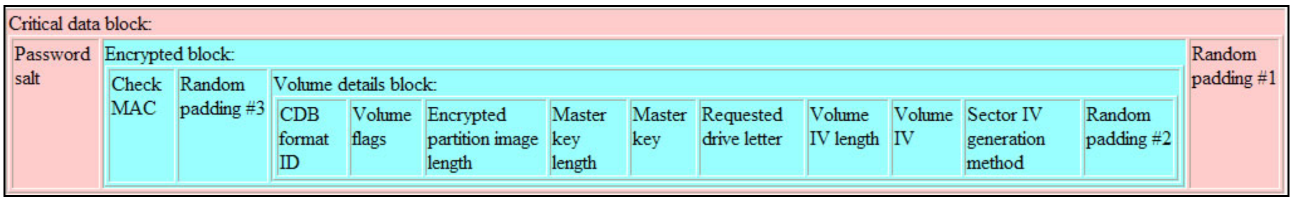


Figure 12: Layout and contents of a critical data block

CDB format ID is 8 bits in length and is a version ID, which identifies the layout of the remainder of the volume details block. When this layout format is used, this will always be set to 3. Later volume file layouts may have different items in this section, or the layout might change; in which case a different version ID will be used here.

Volume flags are 32 bits in length and is a bitmap flagging various items:

Bit 0 is unused
Bit 1 is Sector ID zero is at the start of the file
    0 = Sector ID zero is at the start of the encrypted data
    1 = Sector ID zero is at the start of the host volume file/partition
Bit 2 (unused)
Bit 3 (unused)
Bit 4 is the volume timestamps normal operation
    0 = on dismount, volume file timestamps will be reset to the values they were when mounted
    1 = on dismount, volume file timestamps will be left as-is (i.e. will indicate the date/time the volume was last written to)
    Note: This bit gets ignored by the GUI, which will operate it according to the user options set at the time the volume is mounted

Encrypted partition image length is 64 bits and stores the length of the encrypted partition image in bytes

Master key length is 32 bits – this will be set to the length of the master key in bits.

Master key is variable and is the cipher key size in bits. This is set to the random data generated when the volume was created; and is the en/decryption key used to encrypt the encrypted partition image.

Requested drive letter is 8 bits in size and is the drive letter the volume should be normally mounted as. This is set to 0x00 if there is no particular drive letter the volume should be mounted as (i.e. mount using the first available driver letter).

Volume IV length is 32 bits in length. This is set to the length of the Volume IV in bits. If the cipher's block size is >=0, this will be set to the cipher's block size. Otherwise, this is set to 0.

Volume IV if cipher block size is > 0 then cipher block size is set to the random data generated when the volume was created. When each sector of the encrypted partition is encrypted/decrypted, this value is XORd with any (hashed or unhashed) sector ID before being used as the sector IV. This guarantees that every sector within the encrypted partition has a non-predictable IV.
If cipher block size is <= 0, then this is zero.

Sector IV generation method is 8 bits in size. This is set to indicate the method of generating sector IVs. If a volume IV is present, then it will be XORd with the IV generated using this method, before it is used for the encrypted/decryption. The IV generated will be right-padded/truncated to the cipher's block size.

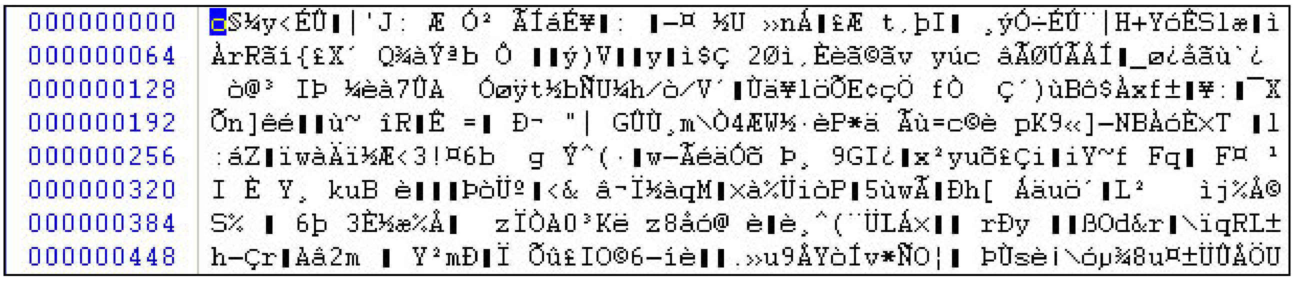
    If the cipher's block size is <= 0, then this must be set to 0
    0 = No sector IVs (Null sector IV)

1 = Sector IV is the 32 bit sector ID (LSB first)
2 = Sector IV is the 64 bit sector ID (LSB first)
3 = Hash of the 32 bit sector ID (sector ID is LSB first)
4 = Hash of the 64 bit sector ID (sector ID is LSB first)
5 = ESSIV

The "Volume Flags" item is used to determine the location of sector zero (start of encrypted data, or start of host/file partition).

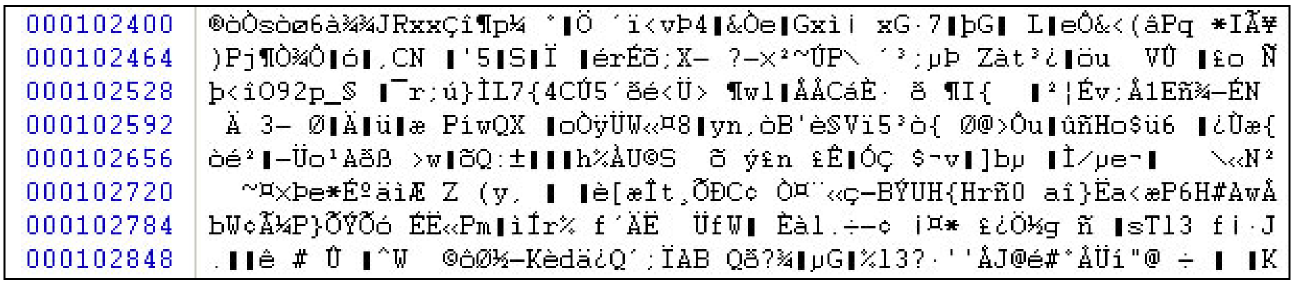
Random padding #2 – Random padding data. Required to pad out the encrypted block to a multiple of bs, and to increase the size of this block to a maximum length that can fit within the critical data block.

Each volume has a CDB, therefore, I interrogated the first 512 bytes of the outer volume and hidden volume to see if I could garner any information or identify a pattern conforming to that mentioned above. Figure 13 shows the contents of the CDB for the outer volume of VolA.vol and Figure 14 shows the contents of the CDB for the hidden volume.



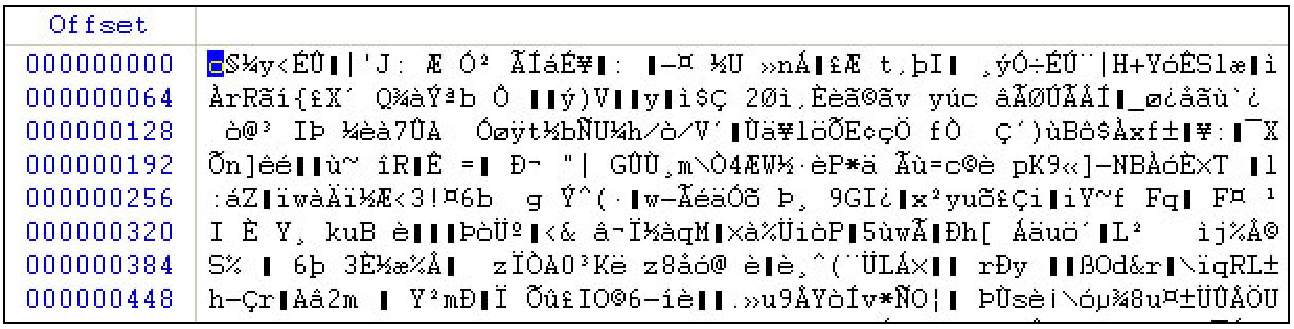
```
000000000  cS¼y<ÉÛ▮|'J: Æ Ó² ÃÌáÉ¥▮: ▮–¤ ½U »nÁ▮£Æ t,þI▮ ¸ýÓ÷ÉÛ¨|H+YóÊSlæ▮i
000000064  ÅrRãi{£X´ Q¾àÝªb Ô ▮▮ý)V▮▮y▮i$ç 2Øi,Èèã©ãv yúc âÃØÛÃÂÍ▮_ø¿âãù`¿
000000128   ò@³ IÞ ¼èà7ÛÀ  ÓøÿtkbÑU¼h/ò/V´▮Üä¥lòÕE¢çÖ fÒ  Ç´)ùBô$Àxf±▮¥:▮¯X
000000192  Õn]êé▮▮ù~ îR▮Ê =▮ Ð¬ "| GÛÜ¸m\Ò4ÆW½·èP*ä Ãù=c©è pK9«]–NBÀóÊ×T ▮l
000000256  :áZ▮iwàÄi½Æ<3!¤6b  g Ý^(·▮w–Ãéäóõ Þ¸ 9GI¿▮x²yuõ£Çi▮iY~f Fq▮ F¤ ¹
000000320  I È Y¸ kuB è▮▮▮ÞòÜº▮<& â¬Ï¼aqM▮xà%ÜiòP▮5ùwÃ▮Ðh[ Áäuö´▮L²   ij%Å©
000000384  S% ▮ 6þ 3È½æ%Å▮  zÏÒA0³Kë z8åó@ è▮è¸^(¨ÜLÁ×▮▮ rÐy ▮▮ßOd&r▮\iqRL±
000000448  h–Çr▮Aâ2m ▮ Y²mÐ▮Ï Õû£IO®6–ié▮▮.»u9ÅYòÍv*ÑO¦▮ ÞÛsèi\óµ¾8u¤±ÜÛÀÖU
```

Figure 13: CDB for outer volume of VolA.vol



```
000102400  ®òÒsòø6à¾¾JRxxÇí¶p¼ ˆ▮Ö ´ï<vÞ4▮&Òe▮Gxìi xG·7▮þG▮ L▮eÔ&<(âPq *IÃ¥
000102464  )Pj¶Ò¾Ô▮ó▮,CN ▮'5▮S▮Ï ▮érÉõ;X– ?–×²~ÚP\ ´³;µÞ Zàt³¿▮öu  VÛ ▮£o Ñ
000102528  þ<í092p_S ▮¯r;ú}ÌL7{4CÛ5´ðé<Ü> ¶wl▮ÂÁCáÈ· ð ¶I{  ▮²|Év;Á1Eñ¾–ÉN
000102592   À 3– Ø▮Ä▮ü▮æ PiwQX ▮oÒÿÜW«¤8▮yn,òB'èSVi5³ò{ Ø@>Ôu▮ûñHo$ü6 ▮¿Ûæ{
000102656  òé²▮–Üo¹AõB >w▮õQ:±▮▮▮h%ÀU®S  õ ý£n £Ê▮ÓÇ $¬v▮]bµ ▮Ì/µe¬▮    \«N²
000102720    ~¤×Þe*ɺäiÆ Z (y, ▮ ▮è[æÎt¸ÕÐC¢ Ò¤¨«ç–BÝUH{Hrñ0 aî}Ëa<æP6H#AwÀ
000102784  bW¢Ã¼P}ÕÝÕó ÉË«Pm▮iÍr% f´ÀÈ  ÜfW▮ Èàl.÷–¢ i¤* £¿Ö½g ñ ▮sT13 fi·J
000102848  .▮▮ê # Û ▮^W  ®óØ¾–Kèdä¿Q´;ÏAB Qõ?¾▮µG▮%13?·''ÅJ@é#˚ÅÜi"@ ÷ ▮ ▮K
```

Figure 14: CDB for hidden volume of VolA.vol



```
  Offset
000000000  cS¼y<ÉÛ▮|'J: Æ Ó² ÃÌáÉ¥▮: ▮–¤ ½U »nÁ▮£Æ t,þI▮ ¸ýÓ÷ÉÛ¨|H+YóÊSlæ▮i
000000064  ÅrRãi{£X´ Q¾àÝªb Ô ▮▮ý)V▮▮y▮i$ç 2Øi,Èèã©ãv yúc âÃØÛÃÂÍ▮_ø¿âãù`¿
000000128   ò@³ IÞ ¼èà7ÛÀ  ÓøÿtkbÑU¼h/ò/V´▮Üä¥lòÕE¢çÖ fÒ  Ç´)ùBô$Àxf±▮¥:▮¯X
000000192  Õn]êé▮▮ù~ îR▮Ê =▮ Ð¬ "| GÛÜ¸m\Ò4ÆW½·èP*ä Ãù=c©è pK9«]–NBÀóÊ×T ▮l
000000256  :áZ▮iwàÄi½Æ<3!¤6b  g Ý^(·▮w–Ãéäóõ Þ¸ 9GI¿▮x²yuõ£Çi▮iY~f Fq▮ F¤ ¹
000000320  I È Y¸ kuB è▮▮▮ÞòÜº▮<& â¬Ï¼aqM▮xà%ÜiòP▮5ùwÃ▮Ðh[ Áäuö´▮L²   ij%Å©
000000384  S% ▮ 6þ 3È½æ%Å▮  zÏÒA0³Kë z8åó@ è▮è¸^(¨ÜLÁ×▮▮ rÐy ▮▮ßOd&r▮\iqRL±
000000448  h–Çr▮Aâ2m ▮ Y²mÐ▮Ï Õû£IO®6–ié▮▮.»u9ÅYòÍv*ÑO¦▮ ÞÛsèi\óµ¾8u¤±ÜÛÀÖU
```

Figure 15: CDB for outer volume of VolA.vol once mounted

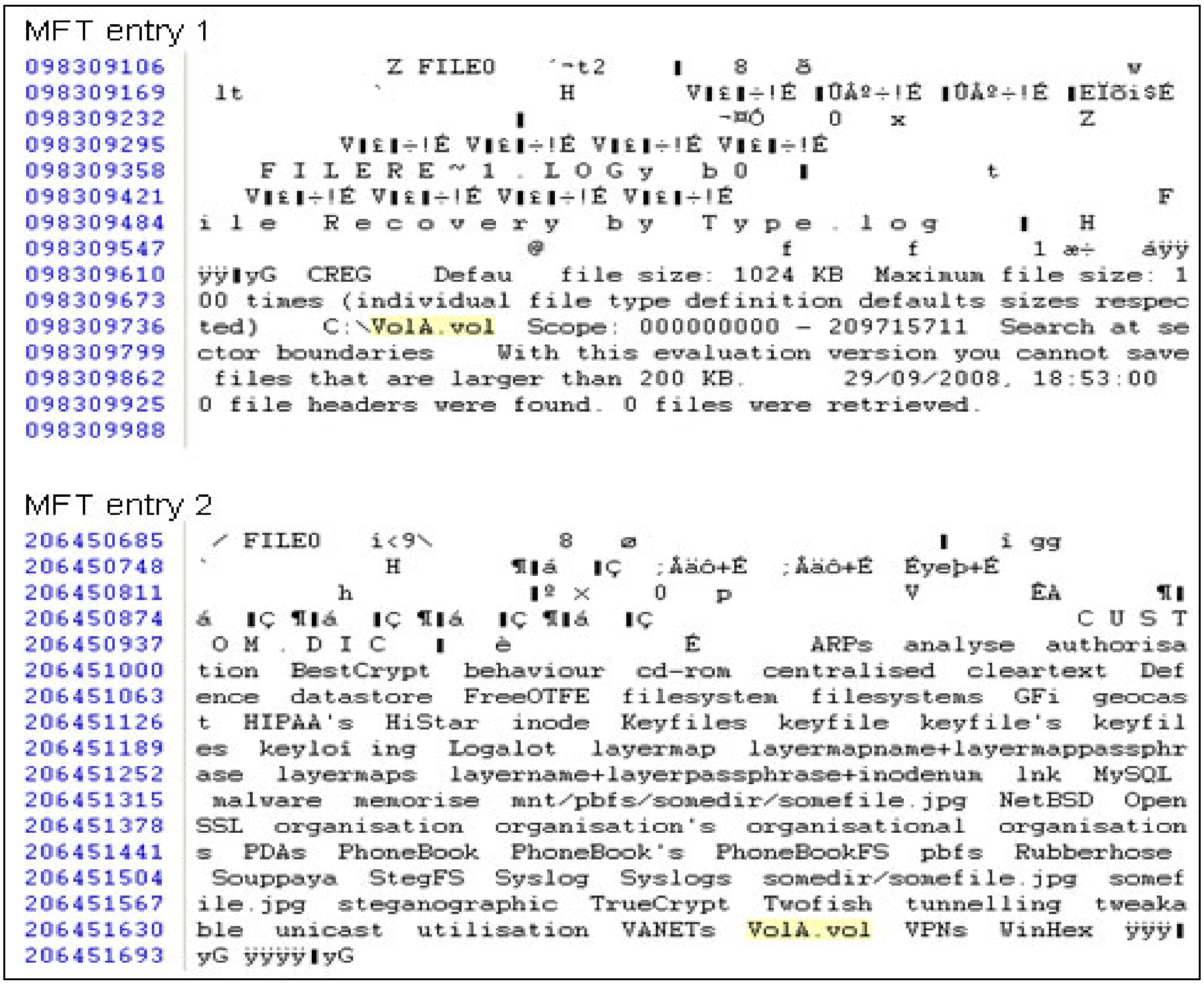
Unfortunately, the CDB offers no assistance in identifying the presence of a hidden volume as no consistent pattern could be identified in the CDBs examined. This is due to the CDB being encrypted, even when the outer volume of VolA.vol is mounted.

5.2 Examination of the Master File Table (MFT)

The $MFT folder on Hard Disk 0, Partition 1 was searched in order to find MFT entries for VolA.vol (the outer volume that contains the hidden volume) and doc2HV.doc files (the file that is hidden inside the hidden volume).

Two MFT entries were found during the search for VolA.vol, see Figure 16 below. However, none of these were MFT entries for VolA.vol, instead, they referred to a recovered log file, “FileRecoverybyType.log” and a custom dictionary, both of which contained the word VolA.vol.

No other MFT entries could be found for VolA.vol. I can only conclude from this that FreeOTFE is designed not to add volume information to the master file table.



MFT entry 1

```
098309106              Z FILE0    ´¬t2     ▮    8    ŏ                     v
098309169  1t            `            H        V▮£▮÷!É ▮ÛÅº÷!É ▮ÛÅº÷!É ▮EÏõi$É
098309232                              ▮             ¬¤Ó     0    x           Z
098309295          V▮£▮÷!É V▮£▮÷!É V▮£▮÷!É V▮£▮÷!É
098309358     F I L E R E ~ 1 . L O G y   b 0    ▮            t
098309421     V▮£▮÷!É V▮£▮÷!É V▮£▮÷!É V▮£▮÷!É                               F
098309484  i l e   R e c o v e r y   b y   T y p e . l o g      ▮   H
098309547                            @            f         f        1 æ÷    áÿÿ
098309610  ÿÿ▮yG  CREG    Defau   file size: 1024 KB  Maximum file size: 1
098309673  00 times (individual file type definition defaults sizes respec
098309736  ted)    C:\VolA.vol  Scope: 000000000 - 209715711  Search at se
098309799  ctor boundaries    With this evaluation version you cannot save
098309862   files that are larger than 200 KB.      29/09/2008, 18:53:00
098309925  0 file headers were found. 0 files were retrieved.
098309988
```

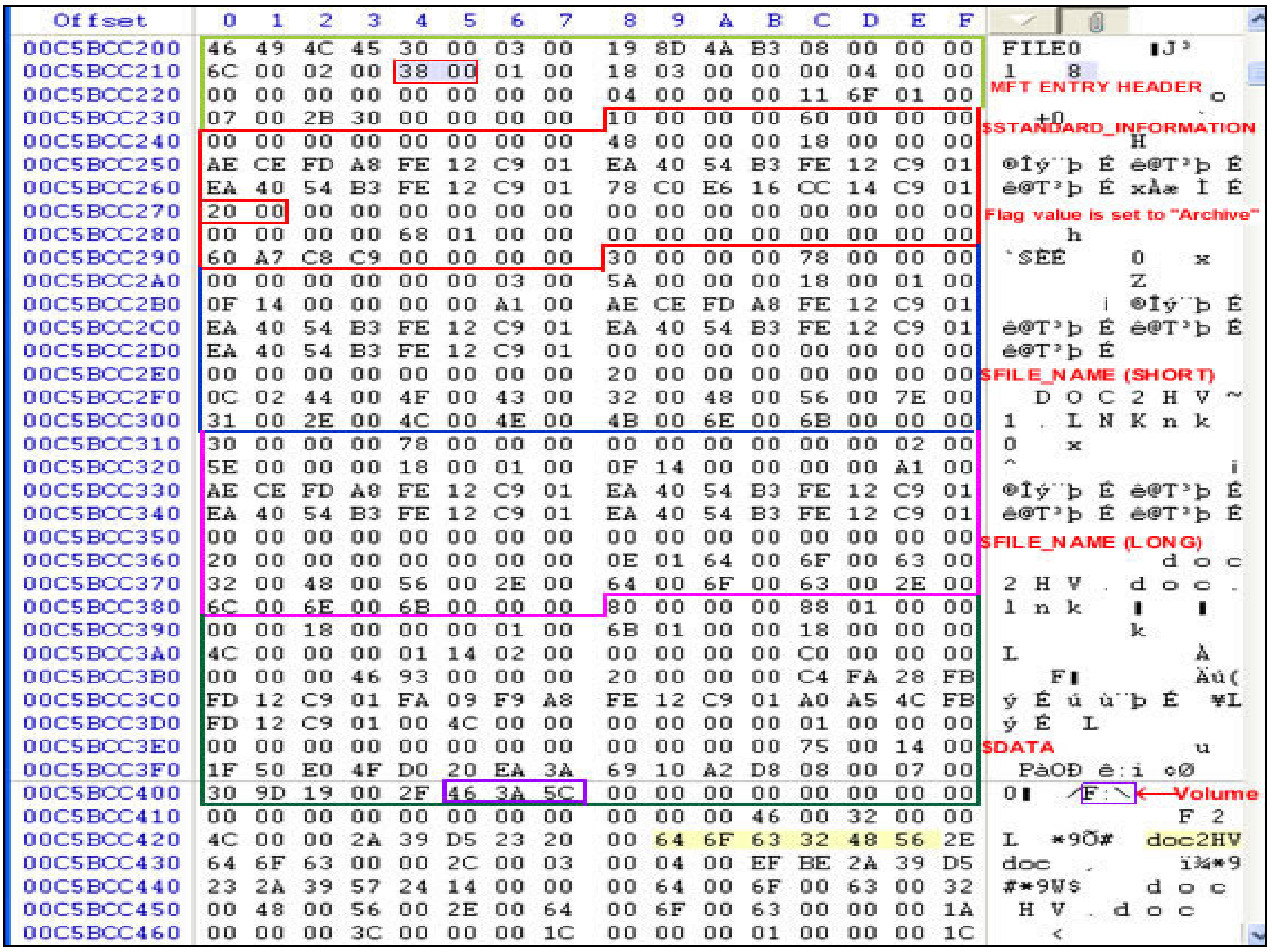
MFT entry 2

```
206450685   ⁄ FILE0    i<9\          8    ⌀                         ▮    i gg
206450748  `          H         ¶▮á   ▮Ç  ;Åäô+Ê  ;Åäô+Ê  Êyeþ+Ê
206450811            h              ▮º ×     0    p             V         ÊA        ¶▮
206450874  á  ▮Ç ¶▮á   ▮Ç ¶▮á   ▮Ç ¶▮á   ▮Ç                                 C U S T
206450937   O M . D I C   ▮    è              É         ARPs  analyse  authorisa
206451000  tion  BestCrypt  behaviour  cd-rom  centralised  cleartext  Def
206451063  ence  datastore  FreeOTFE  filesystem  filesystems  GFi  geocas
206451126  t  HIPAA's  HiStar  inode  Keyfiles  keyfile  keyfile's  keyfil
206451189  es  keyloi ing  Logalot  layermap  layermapname+layermappassphr
206451252  ase  layermaps  layername+layerpassphrase+inodenum  lnk  MySQL
206451315   malware  memorise  mnt/pbfs/somedir/somefile.jpg  NetBSD  Open
206451378  SSL  organisation  organisation's  organisational  organisation
206451441  s  PDAs  PhoneBook  PhoneBook's  PhoneBookFS  pbfs  Rubberhose
206451504   Souppaya  StegFS  Syslog  Syslogs  somedir/somefile.jpg  somef
206451567  ile.jpg  steganographic  TrueCrypt  Twofish  tunnelling  tweaka
206451630  ble  unicast  utilisation  VANETs  VolA.vol  VPNs  WinHex  ÿÿÿ▮
206451693  yG ÿÿÿÿ▮yG
```

Figure 16: MFT entries for VolA.vol

The search for doc2HV.doc returned 6 MFT entries. These were examined in detail to investigate whether they could provide evidence of the hidden volume in which they were contained. Figure 17 below shows one of the MFT entries found for doc2HV.doc. This entry will be deconstructed and the results discussed later in this section.

Before deconstructing the doc2HV.doc MFT entry, it is necessary to introduce the basic concepts of NTFS and MFT data structures.



```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00C5BCC200  46 49 4C 45 30 00 03 00  19 8D 4A B3 08 00 00 00   FILE0    ▮J³
00C5BCC210  6C 00 02 00 38 00 01 00  18 03 00 00 00 04 00 00   l   8
00C5BCC220  00 00 00 00 00 00 00 00  04 00 00 00 11 6F 01 00               o
00C5BCC230  07 00 2B 30 00 00 00 00  10 00 00 00 60 00 00 00     +0
00C5BCC240  00 00 00 00 00 00 00 00  48 00 00 00 18 00 00 00           H
00C5BCC250  AE CE FD A8 FE 12 C9 01  EA 40 54 B3 FE 12 C9 01   ®Îý¨þ É ê@T³þ É
00C5BCC260  EA 40 54 B3 FE 12 C9 01  78 C0 E6 16 CC 14 C9 01   ê@T³þ É xÀæ Ì É
00C5BCC270  20 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00C5BCC280  00 00 00 00 68 01 00 00  00 00 00 00 00 00 00 00       h
00C5BCC290  60 A7 C8 C9 00 00 00 00  30 00 00 00 78 00 00 00   `§ÈÉ    0   x
00C5BCC2A0  00 00 00 00 00 00 03 00  5A 00 00 00 18 00 01 00           Z
00C5BCC2B0  0F 14 00 00 00 00 A1 00  AE CE FD A8 FE 12 C9 01         ¡ ®Îý¨þ É
00C5BCC2C0  EA 40 54 B3 FE 12 C9 01  EA 40 54 B3 FE 12 C9 01   ê@T³þ É ê@T³þ É
00C5BCC2D0  EA 40 54 B3 FE 12 C9 01  00 00 00 00 00 00 00 00   ê@T³þ É
00C5BCC2E0  00 00 00 00 00 00 00 00  20 00 00 00 00 00 00 00
00C5BCC2F0  0C 02 44 00 4F 00 43 00  32 00 48 00 56 00 7E 00     D O C 2 H V ~
00C5BCC300  31 00 2E 00 4C 00 4E 00  4B 00 6E 00 6B 00 00 00   1 . L N K n k
00C5BCC310  30 00 00 00 78 00 00 00  00 00 00 00 00 00 02 00   0   x
00C5BCC320  5E 00 00 00 18 00 01 00  0F 14 00 00 00 00 A1 00   ^             ¡
00C5BCC330  AE CE FD A8 FE 12 C9 01  EA 40 54 B3 FE 12 C9 01   ®Îý¨þ É ê@T³þ É
00C5BCC340  EA 40 54 B3 FE 12 C9 01  EA 40 54 B3 FE 12 C9 01   ê@T³þ É ê@T³þ É
00C5BCC350  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00C5BCC360  20 00 00 00 00 00 00 00  0E 01 64 00 6F 00 63 00             d o c
00C5BCC370  32 00 48 00 56 00 2E 00  64 00 6F 00 63 00 2E 00   2 H V . d o c .
00C5BCC380  6C 00 6E 00 6B 00 00 00  80 00 00 00 88 01 00 00   l n k   ▮   ▮
00C5BCC390  00 00 18 00 00 00 01 00  6B 01 00 00 18 00 00 00           k
00C5BCC3A0  4C 00 00 00 01 14 02 00  00 00 00 00 C0 00 00 00   L           À
00C5BCC3B0  00 00 00 46 93 00 00 00  20 00 00 00 C4 FA 28 FB      F▮       Äú(
00C5BCC3C0  FD 12 C9 01 FA 09 F9 A8  FE 12 C9 01 A0 A5 4C FB   ý É ú ù¨þ É  ¥L
00C5BCC3D0  FD 12 C9 01 00 4C 00 00  00 00 00 00 01 00 00 00   ý É  L
00C5BCC3E0  00 00 00 00 00 00 00 00  00 00 00 00 75 00 14 00               u
00C5BCC3F0  1F 50 E0 4F D0 20 EA 3A  69 10 A2 D8 08 00 07 00    PàOÐ ê:i ¢Ø
00C5BCC400  30 9D 19 00 2F 46 3A 5C  00 00 00 00 00 00 00 00   0▮  /F:\
00C5BCC410  00 00 00 00 00 00 00 00  00 00 00 46 00 32 00 00              F 2
00C5BCC420  4C 00 00 2A 39 D5 23 20  00 64 6F 63 32 48 56 2E   L  *9Õ#  doc2HV.
00C5BCC430  64 6F 63 00 00 2C 00 03  00 04 00 EF BE 2A 39 D5   doc  ,     ï¾*9Õ
00C5BCC440  23 2A 39 57 24 14 00 00  00 64 00 6F 00 63 00 32   #*9W$    d o c 2
00C5BCC450  00 48 00 56 00 2E 00 64  00 6F 00 63 00 00 00 1A    H V . d o c
00C5BCC460  00 00 00 3C 00 00 00 1C  00 00 00 01 00 00 00 1C      <
```

Figure 17: MFT entry for doc2HV.doc

The Master File Table (MFT) is at the heart of NTFS and has an entry for every file and directory. MFT entries are a fixed size and contain only a few fields. To date, the entries have been 1,024 bytes in size, but the size is defined in the boot sector. Each MFT entry uses fixup values, so the on-disk version of the data structure has the last two bytes of each sector replaced by a fixup value.

Fixup values are used for increased reliability and are incorporated into data structures that are over one sector in length. The fixup values in large data structures are replaced with a signature value when the data structure is written to disk. The signature is later used to verify the integrity of the data by verifying that all sectors have the same signature. The MFT file structure is shown in Figure 18 below.

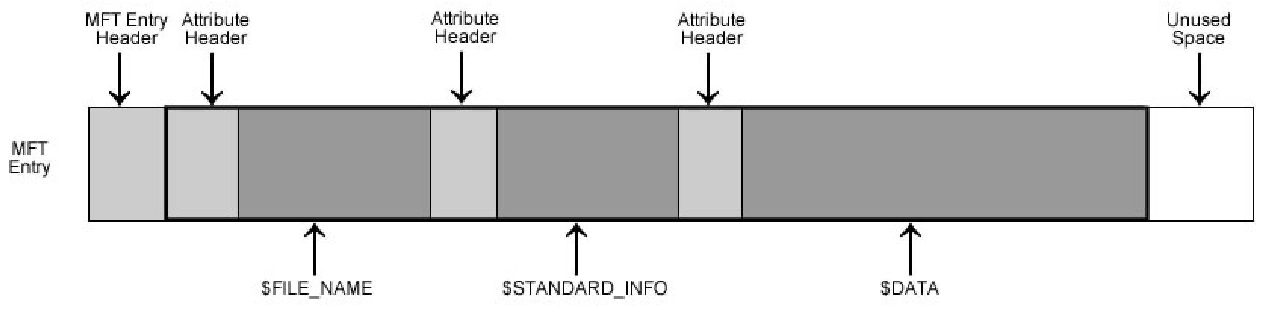


Figure 18: MFT file structure

The data structure for a basic MFT entry is shown in Table 2. Notes have been provided in the description field wherever possible.

Byte Range	Description	Essential
0 – 3	**Signature ("FILE")** The standard signature value is "FILE", but some entries may have "BAAD" if chkdsk found an error in it.	No
4 – 5	**Offset to fixup array** The next two fields are fixup values and the array is typically stored after byte 42. The offset values are relative to the start of the entry.	Yes
6 – 7	**Number of entries in the fixup array**	Yes
8 – 15	**$LogFile Sequence Number (LSN)** The LSN is used for the file system log. It records when metadata is updated to the file system so that a corrupt file system can be fixed more quickly.	No
16 – 17	**Sequence value** The sequence value is incremented when the entry is either allocated or unallocated, which is determined by the operating system.	No
18 – 19	**Link count** The link count shows how many directories have entries for the MFT entry. If hard links are created for a file, this number is incremented by one for each link.	No
20 – 21	**Offset to the first attribute** The first attribute for the file is found using the offset value, which is relative to the start of the entry. All other attributes follow the first one, and these are found by advancing ahead using the size field in the attribute header. The end of file marker 0xffffffff exists after the last attribute. If a file needs more than one MFT entry, the additional entries will have the file reference of the base entry in their MFT entry.	Yes
22 – 23	**In-use and directory flags** The flags field has only two values. The 0x01 bit is set when the entry is in use and 0x02 is set when the entry is for a directory.	Yes
24 – 27	**Used size of MFT entry**	Yes
28 – 31	**Allocated size of MFT entry** The allocated size is how much space the record takes up on disk. This should be a multiple of the cluster size and should probably be equal to the size of an MFT file record.	Yes
32 – 39	**File reference to base record** This is zero for base MFT records. When it is not zero it is a MFT reference pointing to the base MFT record to which the record belongs. The base record contains the information about the extension record. This information is stored in an ATTRIBUTE_LIST attribute	Yes
40 – 41	**Next attribute ID** The attribute ID that will be assigned to the next attribute added to the MFT Record. This is incremented each time it is used. Every time the MFT record is reused this ID is set to zero.	No
42 – 1023	**Attributes and fixup values**	Yes

Table 2: Data structure for basic MFT entry

Every MFT entry contains a $STANDARD_INFORMATION, $FILE_NAME, and $DATA attribute. Each attribute contains a header that occupies the first 16 bytes of the attribute. The data structure of the first 16 bytes of an attribute is shown in Table 3. Again, notes have been provided in the description field wherever possible.

Byte Range	Description	Essential
0 – 3	**Attribute type identifier**	**Yes**
4 – 7	**Length of attribute**	**Yes**
8 - 8	**Non-resident flag** When set to 1 the attribute is non-resident. The flag's value identifies if the attribute is compressed (0x0001), encrypted (0x40000) or sparse (0x8000).	**Yes**
9 – 9	**Length of name**	**Yes**
10 – 11	**Offset to name** This is relative to the start of the attribute.	**Yes**
12 – 13	**Flags**	**Yes**
14 – 15	**Attribute identifier** This is the number that is unique to this attribute for this MFT entry.	**Yes**

Table 3: Data structure for the first 16 bytes of an attribute

We will now use the information provided above to interpret the MFT entry for doc2HV.doc shown in Figure 17 above. Table 4 shows the data extracted from the MFT entry for this file.

Byte Range	Description
0 – 3	**Signature ("FILE")** Bytes 0 to 3 show the standard signature value for doc2HV is "FILE".
4 – 5	**Offset to fixup array** Bytes 4 and 5 show that the fixup array for doc2HV is located 48 bytes (0x0030) into the MFT entry.
6 – 7	**Number of entries in the fixup array** Bytes 6 and 7 show that the fixup array has 3 values.
8 – 15	**$LogFile Sequence Number (LSN)** Bytes 8 to 15 show the metadata that has been recorded for doc2HV.
16 – 17	**Sequence value** Bytes 16 and 17 show that the sequence value for the doc2HV MFT entry is 108, which means that the entry has been used 108 times.
18 – 19	**Link count** Bytes 18 and 19 show that the link count is 2, so we know that it has two names.
20 – 21	**Offset to the first attribute** Bytes 20 and 21 show that the first attribute is located at byte offset 56 (0x0038).
22 – 23	**In-use and directory flags** Bytes 22 and 23 show that this entry is in use (0x0001)
24 – 27	**Used size of MFT entry**
28 – 31	**Allocated size of MFT entry**
32 – 39	**File reference to base record** The base entry values in bytes 32 to 39 are 0, which shows that this is a base entry.
40 – 41	**Next attribute ID** Bytes 40 and 41 show that the next attribute ID to be assigned is 4. Therefore, we should expect to see attributes with IDs 1 to 3.
42 – 1023	**Attributes and fixup values** The fixup array starts at byte 48 (see offset to fixup array above). The first two bytes show the signature value, which is 0x0007. The next two values are the original values that should be used to replace the signature value.

Table 4: Data structure for basic MFT entry

The first attribute is the $STANDARD_INFORMATION attribute, which is always resident and contains the basic metadata for the file or directory. This attribute starts at byte 56 and the attribute type is shown in the first four bytes as 16 (0x10). Bytes 4 to 7 show that it has a size of 96 bytes (0x60). Byte 8 shows that this is a resident attribute (0x00), and byte 9 shows that it does not have a name (0x00). The flags and id values are set to 0 in bytes 12 to 13 and 14 to 15. Bytes 16 to 19 show that the attribute is 72 bytes (0x48) long, and bytes 20 and 21 show that it starts 24 bytes (0x18) from the start of the attribute. The 24 bytes offset and 72 bytes attribute length equal a total of 96 bytes, which is the reported length of the attribute.

The data extracted from the $STANDARD_INFORMATION attribute for doc2HV.doc is shown in Table 5:

Byte Range	Description
0 – 7	**Creation time** 01 C9 12 FE A8 FD CE AE [1]
8 – 15	**File altered time** 01 C9 12 FE B3 54 40 EA [2]
16 – 23	**MFT altered time** 01 C9 12 FE B3 54 40 EA [3]
24 – 31	**File accessed time** 01 C9 14 CC 16 E6 C0 78 [4]
32 – 35	**Flags** The flag value is set to 0x00000020, which means that the file permissions is set to archived.
36 – 39	**Maximum number of versions** 0x00000000 means that file versions are not being used.
40 – 43	**Version number** 0x00000000 means that file versions are not being used.
44 – 47	**Class ID** Class ID is set to 0.
48 – 51	**Owner ID** Owner ID is set to 0.
52 – 55	**Security ID** Security ID is set to 01 68.
56 – 63	**Quota Charged** Quota charged is set to 0.
64 - 71	**Update Sequence Number (USN)** The update sequence number is set to 00 00 00 00 C9 C8 A7 60 which means that change journaling is enabled.

Table 5: $STANDARD_INFORMATION attribute data for doc2HV.doc

Unfortunately, the $STANDARD_INFORMATION attribute provides little information to help us identify the hidden volume.

The second attribute, $FILE_NAME attribute, is used to store the file's name and parent directory information and it is used in a directory index. When it is used in an MFT entry, it does not contain any essential information, but when it is used in a directory index it does. The $FILE_NAME attribute contains the fields shown in Table 6. The data extracted from the $FILE_NAME attribute for doc2HV.doc is also shown in Table 6.

Byte Range	Description
0 – 7	**File reference of parent directory** The parent directory is 5135 and the sequence number is 161, which is the entry for the root directory.
8 – 15	**File creation time** 01 C9 12 FE A8 FD CE AE
16 – 23	**File modification time** 01 C9 12 FE B3 54 40 EA

[1 - 4] Unfortunately, I was unable to convert the date to UTC time

24 – 31	**MFT modification time** 01 C9 12 FE B3 54 40 EA
32 – 39	**File access time** 01 C9 12 FE B3 54 40 EA
40 - 47	**Allocated size of file** Allocated size of file is set to 0. Since this value is set to zero, it means that this attribute is not used in a directory index. If it had been used in a directory index, the value for this attribute would be accurate.
48 - 55	**Real size of file** Real size of file is set to 0. Since this value is set to zero, it means that this attribute is not used in a directory index. If it had been used in a directory index, the value for this attribute would be accurate.
56 - 59	**Flags** The flag value is set to 0x0020, which means that the file permissions is set to archived.
60 - 63	**Reparse value** Reparse value is set to 0, this is because it is not essential.
64 – 64	**Length of name** Length of name is 12 letters long
65 – 65	**Namespace** Byte 65 shows it is in name space 2, which is DOS compliant.
66+	**Name** The name is DOC2HV~1.LNKnk

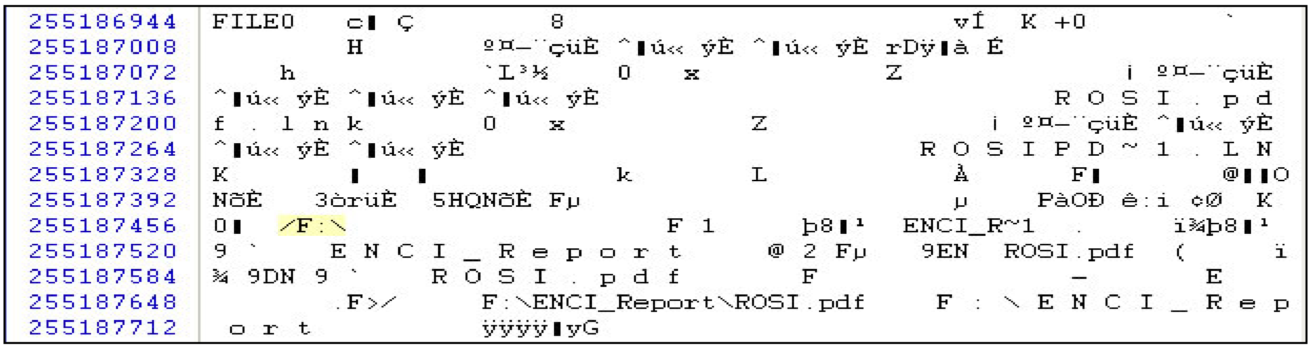
Table 6: $FILE_NAME attribute data for doc2HV.doc

As the namespace is DOS compliant, Windows requires that two $FILE_NAME attributes exist, a $FILE_NAME (SHORT) and a $FILE_NAME (LONG). $FILE_NAME (SHORT) has already been discussed above but, due to the $FILE_NAME attribute for this entry not being used in a directory index and the usefulness of this attribute being limited, I will not discuss the FILE_NAME (LONG) attribute any further other than to say that the long file name for this entry is doc2HV.doc.lnk.

The $DATA attribute is the simplest to understand because it has no native structure. After the header, there is only raw content that corresponds to the contents of the file. It has a type identifier of 128 and has no minimum or maximum size. For most files, this is the last attribute in the MFT entry. The only useful information that can be obtained from the $DATA attribute is a reference to the F drive, which is the drive letter assigned to the hidden volume when it was mounted. It was observed that all six of the doc2HV.doc files returned in the search, identified drive F as being the location of the document we know to be hidden.

Unfortunately, the MFT entries for doc2HV.doc have been unable to provide sufficient evidence of the hidden volume, other than associating it with the F drive. We know that the F drive is our hidden volume but in cases where there is zero knowledge of an offending file or document, having access to the drive letter would be insufficient proof of a hidden volume. This is because the Drive F could plausibly be removable storage, such as a flash drive, or a removable hard drive.

To confirm this belief, a search was performed on the $MFT folder to find entries associated with the /F:\ to determine whether I could find evidence of the F Drive being associated with files other than those stored in the hidden volume. Although many of the entries found referred to the doc2HV.doc document, there were also MFT entries that referred to documents I know to have been saved to removable storage in the past, an example of one of these documents is shown in Figure 19 below.



```
255186944  FILE0    c▮ Ç         8                          vÍ   K +0         `
255187008          H         º¤–¨çüÈ ^▮ú« ýÈ ^▮ú« ýÈ rDý▮à É
255187072     h              `L³¾    0    x               Z                ¡ º¤–¨çüÈ
255187136  ^▮ú« ýÈ ^▮ú« ýÈ ^▮ú« ýÈ                                  R O S I . p d
255187200  f . l n k         0    x            Z               ¡ º¤–¨çüÈ ^▮ú« ýÈ
255187264  ^▮ú« ýÈ ^▮ú« ýÈ                                  R O S I P D ~ 1 . L N
255187328  K        ▮    ▮           k        L            À        F▮        @▮▮O
255187392  NõÈ    3òrüÈ   5HQNõÈ Fµ                          µ     PàOÐ ê:i ¢Ø   K
255187456  0▮   /F:\                          F 1      þ8▮¹  ENCI_R~1   .     ï¾þ8▮¹
255187520  9 `      E N C I _ R e p o r t      @ 2 Fµ   9EN  ROSI.pdf   (       ï
255187584  ¾ 9DN 9 `     R O S I . p d f        F                  –         E
255187648           .F>/      F:\ENCI_Report\ROSI.pdf     F : \ E N C I _ R e p
255187712   o r t             ÿÿÿÿ▮yG
```

Figure 19: Example of document saved in MFT

5.3 Evidence of data leakage from the hidden volume

Ambient data can reside in non-traditional computer storage areas and formats such as file slack, the Windows swap or page file and unallocated file space, therefore, the hard drive was investigated to determine whether any data had leaked from the hidden volume or any of the documents contained within it, thereby, compromising a user's plausible deniability.

Another Word document was added to the hidden volume and a new volume snapshot was taken, see Figure 20 below. To add the new document to the hidden volume, the volume was mounted and then dismounted immediately to reduce the risk of data leakage. This caused changes to the drive addresses. The new drive addresses for the outer volume and hidden volume are also shown in Figure 20 below.

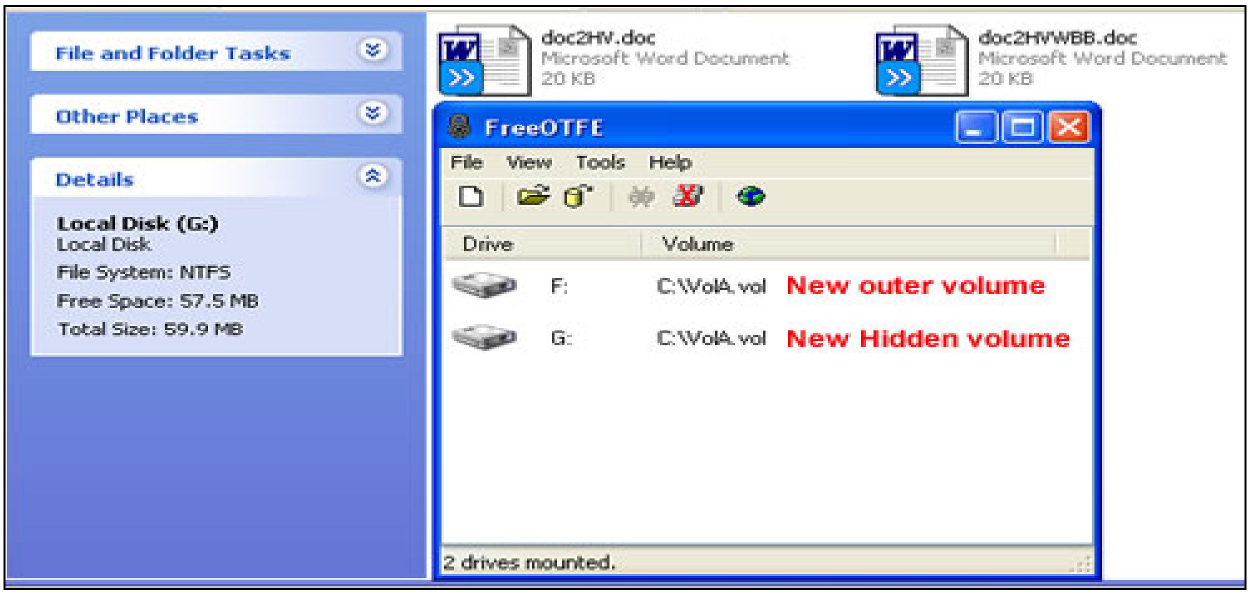
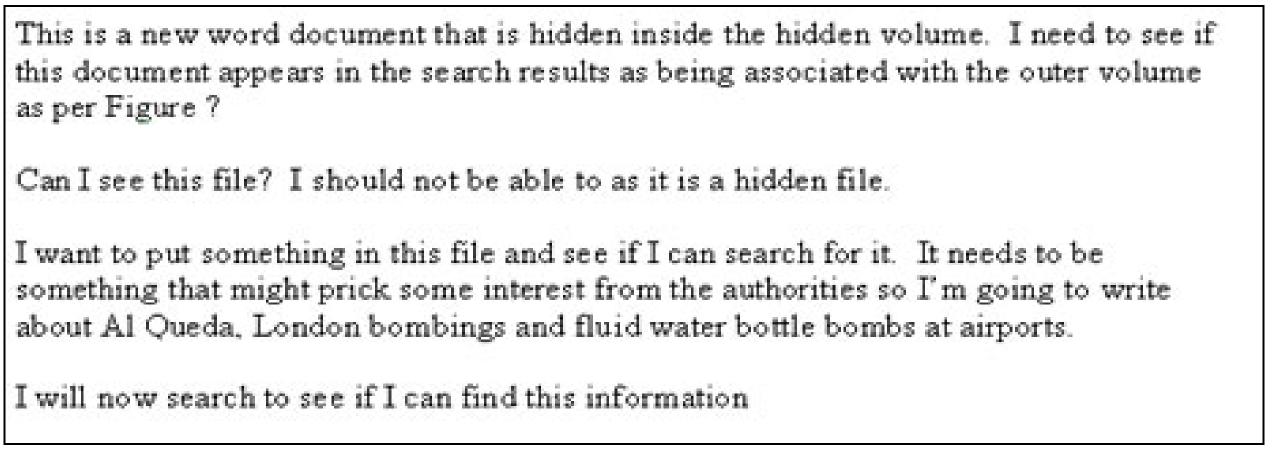


Figure 20: Files contained in hidden volume

Since the purpose of a hidden volume is to conceal evidence of potentially damaging documents, files and criminal activity, the new document, named doc2HVWBB.doc, was created to include content that would spark the interest of a forensic investigator or authorities, see Figure 21.



This is a new word document that is hidden inside the hidden volume. I need to see if this document appears in the search results as being associated with the outer volume as per Figure ?

Can I see this file? I should not be able to as it is a hidden file.

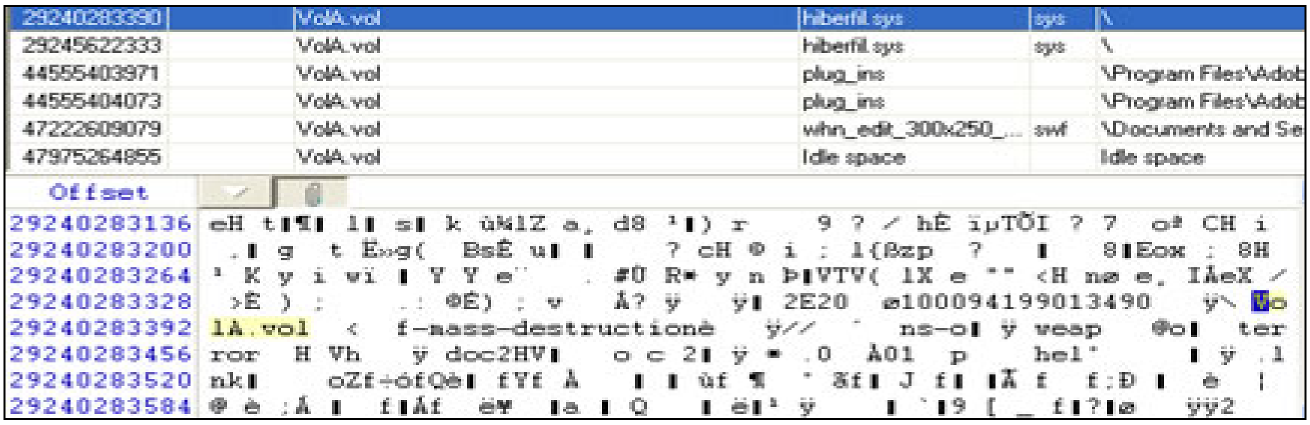
I want to put something in this file and see if I can search for it. It needs to be something that might prick some interest from the authorities so I'm going to write about Al Queda, London bombings and fluid water bottle bombs at airports.

I will now search to see if I can find this information

Figure 21: Contents of Hidden file doc2HVWBB.doc

On analysis of Hard Disk 0, Partition 1, it was discovered that data had leaked into a number of places on the hard drive, namely: c:\hiberfil.sys, $MFT, free space, ntuser.dat and System Volume Information folder.

<u>C:\hiberfil.sys</u>
Instead of shutting down and restarting the computer, Windows takes a snapshot of everything running on the system, copies it to the hard drive and then turns off most the of hardware. Hibernation takes everything in memory and writes it to the hard drive as the hiberfil.sys file. As can be seen in Figure 22, reference to the doc2HV.doc document from VolA.vol has been saved in the hiberfil.sys file.



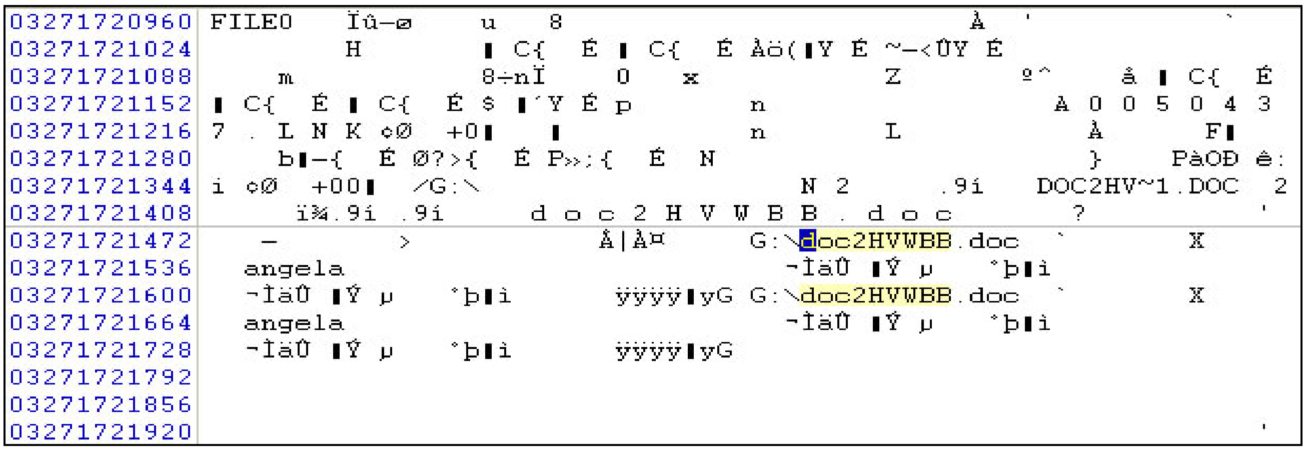
```
29240283390    VolA.vol    hiberfil.sys               sys   \
29245622333    VolA.vol    hiberfil.sys               sys   \
44555403971    VolA.vol    plug_ins                         \Program Files\Adob
44555404073    VolA.vol    plug_ins                         \Program Files\Adob
47222609079    VolA.vol    whn_edit_300x250_...       swf   \Documents and Se
47975264855    VolA.vol    Idle space                       Idle space
```

```
Offset
29240283136 eH t▮¶▮ l▮ s▮ k ù¾lZ a, d8 ¹▮) r    9 ? ⁄ hÈ ïµTÕI ? 7  o² CH i
29240283200  .▮ g  t È»g(  BsÈ u▮ ▮     ? cH ® i ; l{ßzp  ?   ▮   8▮Eox ; 8H
29240283264 ¹ K y i wï ▮ Ÿ Ÿ e¨     . #Ü R▪ y n Þ▮VTV( lX e "" <H nø e, IÅeX ⁄
29240283328  >È ) ;     . : ®È) ; v   Å? ÿ    ÿ▮ 2E20  ø10009419901349O   ÿ\ Vo
29240283392 lA.vol  <  f-mass-destructionè   ÿ⁄⁄  ˆ   ns-o▮ ÿ weap   ®o▮  ter
29240283456 ror  H Vh   ÿ doc2HV▮   o c 2▮ ÿ ▪ .0  À01  p   hel"     ▮ ÿ .1
29240283520 nk▮    oZf÷ófQè▮ fYf À    ▮ ▮ ùf ¶   " ¾f▮ J f▮ ▮Ã f  f;Ð ▮  è  ¦
29240283584 @ è ;Å ▮   f▮Åf  ë¥  ▮a ▮ Q   ▮ ë▮¹ ÿ    ▮ `▮9 [ _ f▮?▮ø   ÿÿ2
```

Figure 22: Hiberfil.sys

$MFT

As can be seen from Figure 23 below, an entry for the hidden document doc2HVWBB.doc has been saved in the master file table. The MFT entry shows the document name, the drive letter and the author of the document.

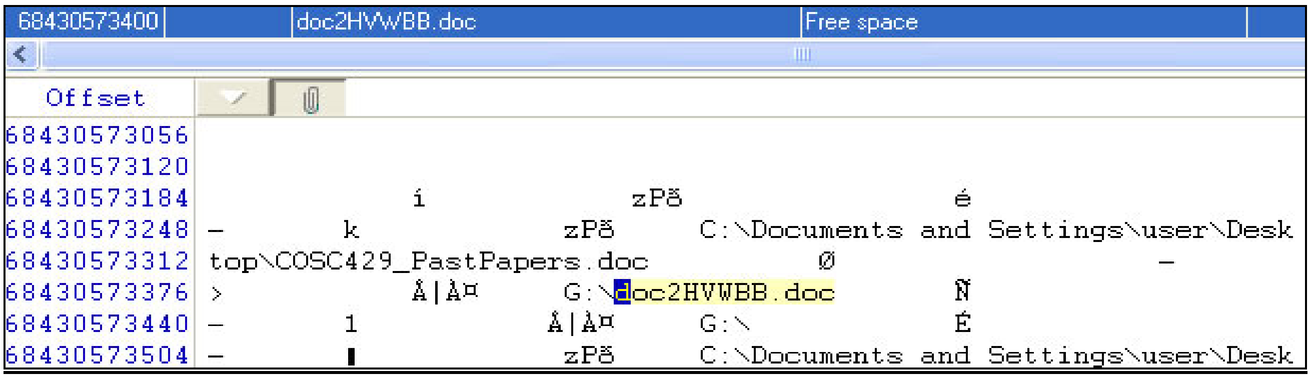


```
03271720960 FILE0   Ïû—ø    u   8                          À  '        `
03271721024         H       ▮ C{  É ▮ C{  É Àö(▮Y É ~—<ÛY É
03271721088   m             8÷nÏ    0   x          Z        º^    â ▮ C{  É
03271721152 ▮ C{  É ▮ C{  É $ ▮´Y É p       n                   A 0 0 5 0 4 3
03271721216 7 . L N K ¢Ø  +0▮   ▮           n       L           À      F▮
03271721280     b▮—{  É Ø?>{  É P»;{  É  N                       }    PàOÐ ê:
03271721344 i ¢Ø  +00▮  ⁄G:\                      N 2     .9i   DOC2HV~1.DOC  2
03271721408       ï¾.9i .9i     d o c 2 H V W B B . d o c       ?           '
03271721472   —       >            Å|À¤     G:\doc2HVWBB.doc  `          X
03271721536   angela                           ¬ÌäÛ ▮Ý µ   ˆþ▮i
03271721600   ¬ÌäÛ ▮Ý µ   ˆþ▮i       ÿÿÿÿ▮yG G:\doc2HVWBB.doc  `          X
03271721664   angela                           ¬ÌäÛ ▮Ý µ   ˆþ▮i
03271721728   ¬ÌäÛ ▮Ý µ   ˆþ▮i       ÿÿÿÿ▮yG
03271721792
03271721856
03271721920                                                                 '
```

Figure 23: MFT entry for doc2HVWBB.doc

Free space

Free space, clusters marked by the file system as not in use, contain evidence of the hidden document, doc2HVWBB.doc and hidden volume that was previously mounted as Drive G, see Figure 24 below.



```
68430573400    doc2HVWBB.doc    Free space
```

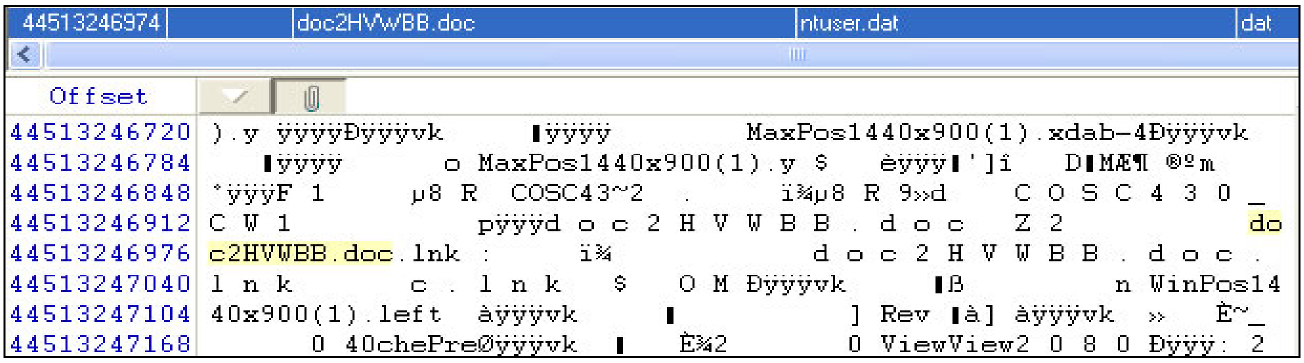
```
Offset
68430573056
68430573120
68430573184            i           zPõ               é
68430573248 —      k          zPõ     C:\Documents and Settings\user\Desk
68430573312 top\COSC429_PastPapers.doc         Ø              —
68430573376 >           Å|À¤      G:\doc2HVWBB.doc          Ñ
68430573440 —      1          Å|À¤     G:\               É
68430573504 —      ▮          zPõ     C:\Documents and Settings\user\Desk
```

Figure 24: Reference to hidden document and hidden volume contained in free space

ntuser.dat

Within the root of the profile, a file named ntuser.dat contains the user's personalised settings for the majority of software installed on the computer, including Windows itself. When the user logs-on, ntuser.dat becomes merged with the computer's registry. A link reference to the doc2HVWBB.doc document has been stored in the ntuser.dat file, see Figure 25 below. The shortcut, or .lnk files, which are created automatically by

Windows, store a wealth of information about files, including the real file's name, length, date of creation, time of access and the volume serial number of the file system on which the real files are stored. The .lnk file can be used to determine whether a hidden volume is present.



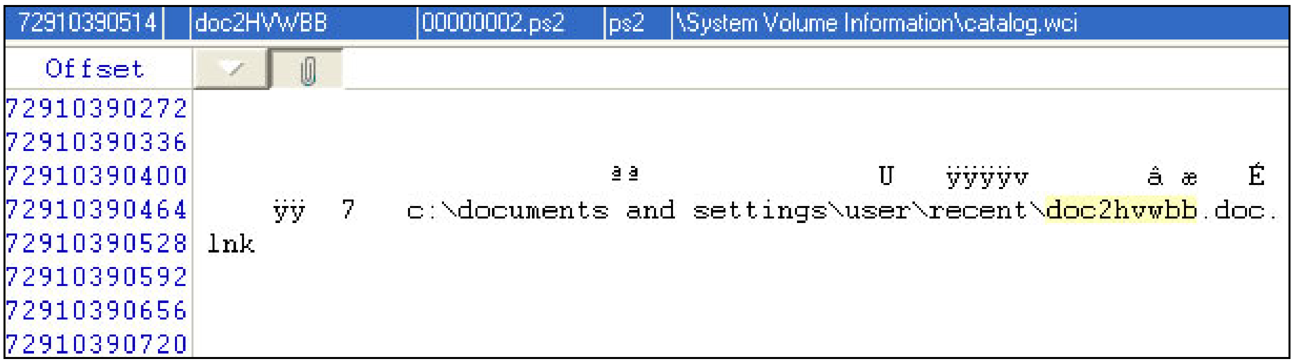
```
44513246974 |            | doc2HVWBB.doc            | ntuser.dat              | dat

Offset
44513246720 ).y ÿÿÿÿÐÿÿÿvk      ▮ÿÿÿÿ        MaxPos1440x900(1).xdab-4Ðÿÿÿvk
44513246784    ▮ÿÿÿÿ       o MaxPos1440x900(1).y $    èÿÿÿ▮']î    D▮MÆ¶ ®ºm
44513246848 ˚ÿÿÿF 1      µ8 R  COSC43~2  .        ï¾µ8 R 9»d    C O S C 4 3 0 _
44513246912 C W 1              pÿÿÿd o c 2 H V W B B . d o c  Z 2              do
44513246976 c2HVWBB.doc.lnk :        ï¾                d o c 2 H V W B B . d o c .
44513247040 l n k        c . l n k   $   O M Ðÿÿÿvk      ▮ß           n WinPos14
44513247104 40x900(1).left  àÿÿÿvk     ▮               ] Rev ▮à] àÿÿÿvk  »   È~_
44513247168       0 40chePreØÿÿÿvk  ▮    È¾2          0 ViewView2 0 8 0 Ðÿÿÿ: 2
```

Figure 25: Reference to doc2HVWBB.doc.lnk in ntuser.dat

System Volume Information

The System Volume Information folder is a part of System Restore, the tool that allows users to set points in time to roll back their computer. The System Volume Information folder is where XP stores these points and associated information that makes them accessible.

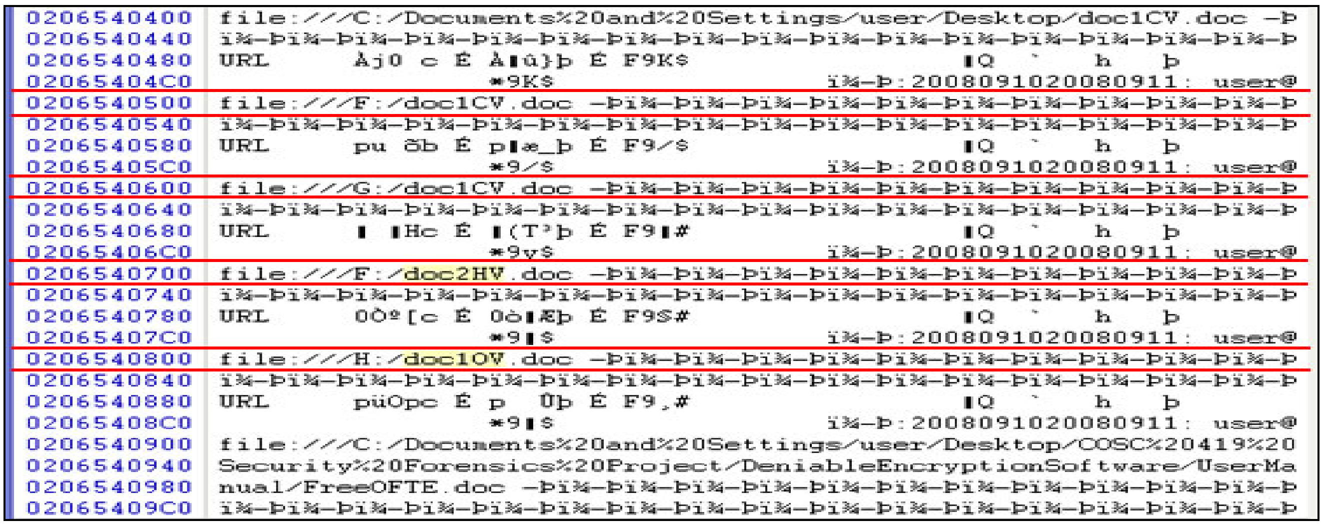
The system stored a copy of the doc2hvwwb.doc inside the c:\documents and settings\user\recent folder and created a link to this folder inside the System Volume Information folder, see Figure 26 below.



```
72910390514 | doc2HVWBB | 00000002.ps2 | ps2 | \System Volume Information\catalog.wci

Offset
72910390272
72910390336
72910390400                         ª ª              U   ÿÿÿÿv        â æ   É
72910390464    ÿÿ  7   c:\documents and settings\user\recent\doc2hvwbb.doc.
72910390528 lnk
72910390592
72910390656
72910390720
```

Figure 26: Reference to hidden document in System Volume Information folder

Figure 27 below shows evidence of the volumes that have been mounted on the system. We know that the F drive is the mounted hidden drive and doc2HV.doc is the document contained inside it.



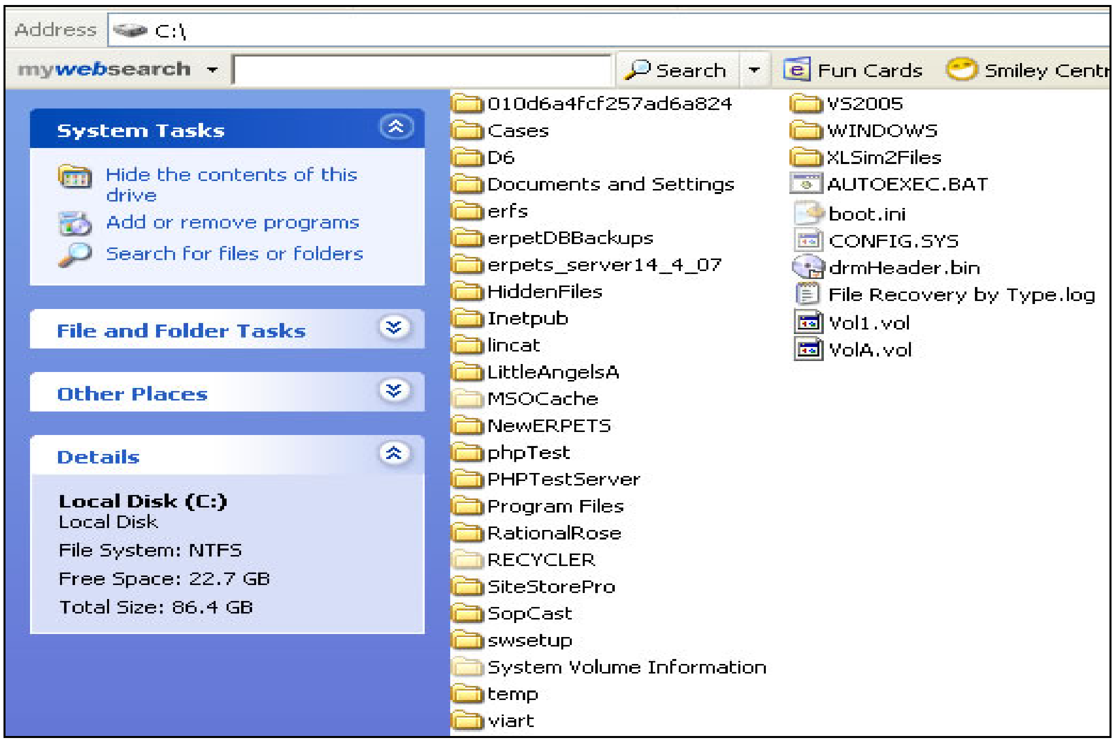
```
0206540400 file:///C:/Documents%20and%20Settings/user/Desktop/doc1CV.doc -Þ
0206540440 ï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þ
0206540480 URL     Àj0 c É À▮û}þ É F9K$                    ▮Q  ˘   h   þ
02065404C0                   *9K$                ï¾-Þ:2008091020080911: user@
0206540500 file:///F:/doc1CV.doc -Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þ
0206540540 ï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þ
0206540580 URL     pu õb É p▮æ_þ É F9/$                    ▮Q  ˘   h   þ
02065405C0                   *9/$                ï¾-Þ:2008091020080911: user@
0206540600 file:///G:/doc1CV.doc -Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þ
0206540640 ï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þ
0206540680 URL     ▮ ▮Hc É ▮(T³þ É F9▮#                    ▮Q  ˘   h   þ
02065406C0                   *9v$                ï¾-Þ:2008091020080911: user@
0206540700 file:///F:/doc2HV.doc -Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þ
0206540740 ï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þ
0206540780 URL     0Òº[c É 0ò▮Æþ É F9S#                    ▮Q  ˘   h   þ
02065407C0                   *9▮$                ï¾-Þ:2008091020080911: user@
0206540800 file:///H:/doc1OV.doc -Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þ
0206540840 ï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þ
0206540880 URL     püOpc É p  Ûþ É F9¸#                    ▮Q  ˘   h   þ
02065408C0                   *9▮$                ï¾-Þ:2008091020080911: user@
0206540900 file:///C:/Documents%20and%20Settings/user/Desktop/COSC%20419%20
0206540940 Security%20Forensics%20Project/DeniableEncryptionSoftware/UserMa
0206540980 nual/FreeOFTE.doc -Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þ
02065409C0 ï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þï¾-Þ
```

Figure 27: Evidence of the drives that have been mounted

The following scenario demonstrates how evidence of a hidden volume may be obtained from a suspect's laptop.

Scenario

A laptop is searched and seized at the airport. Using Windows Explorer, the investigator can see that the laptop contains two volumes, Vol1.vol and VolA.vol, see Figure 28 below:



Address C:\

mywebsearch Search Fun Cards Smiley Centr

System Tasks
Hide the contents of this drive
Add or remove programs
Search for files or folders

File and Folder Tasks

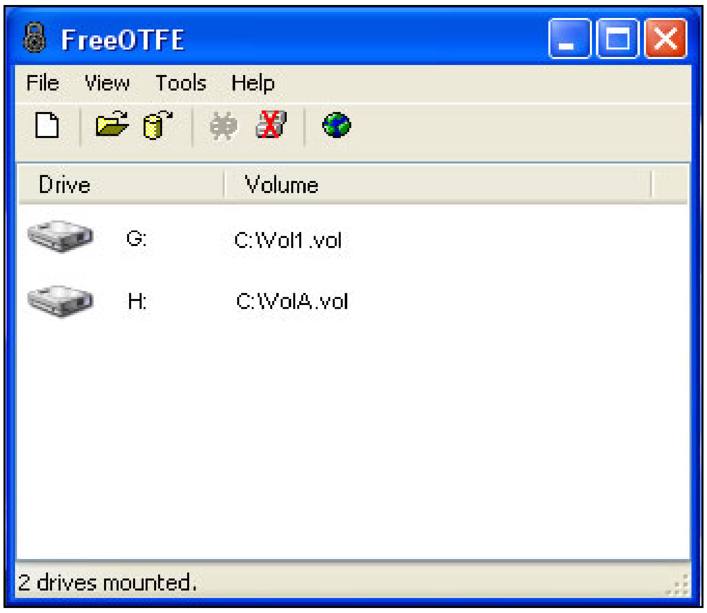
Other Places

Details
Local Disk (C:)
Local Disk
File System: NTFS
Free Space: 22.7 GB
Total Size: 86.4 GB

010d6a4fcf257ad6a824
Cases
D6
Documents and Settings
erfs
erpetDBBackups
erpets_server14_4_07
HiddenFiles
Inetpub
lincat
LittleAngelsA
MSOCache
NewERPETS
phpTest
PHPTestServer
Program Files
RationalRose
RECYCLER
SiteStorePro
SopCast
swsetup
System Volume Information
temp
viart
VS2005
WINDOWS
XLSim2Files
AUTOEXEC.BAT
boot.ini
CONFIG.SYS
drmHeader.bin
File Recovery by Type.log
Vol1.vol
VolA.vol

Figure 28: Folders and files visible in Windows Explorer

The investigator asks the laptop owner to mount both volumes in order to inspect them. Figure 29 shows the drive letters automatically assigned to the volumes when they were mounted.



FreeOTFE

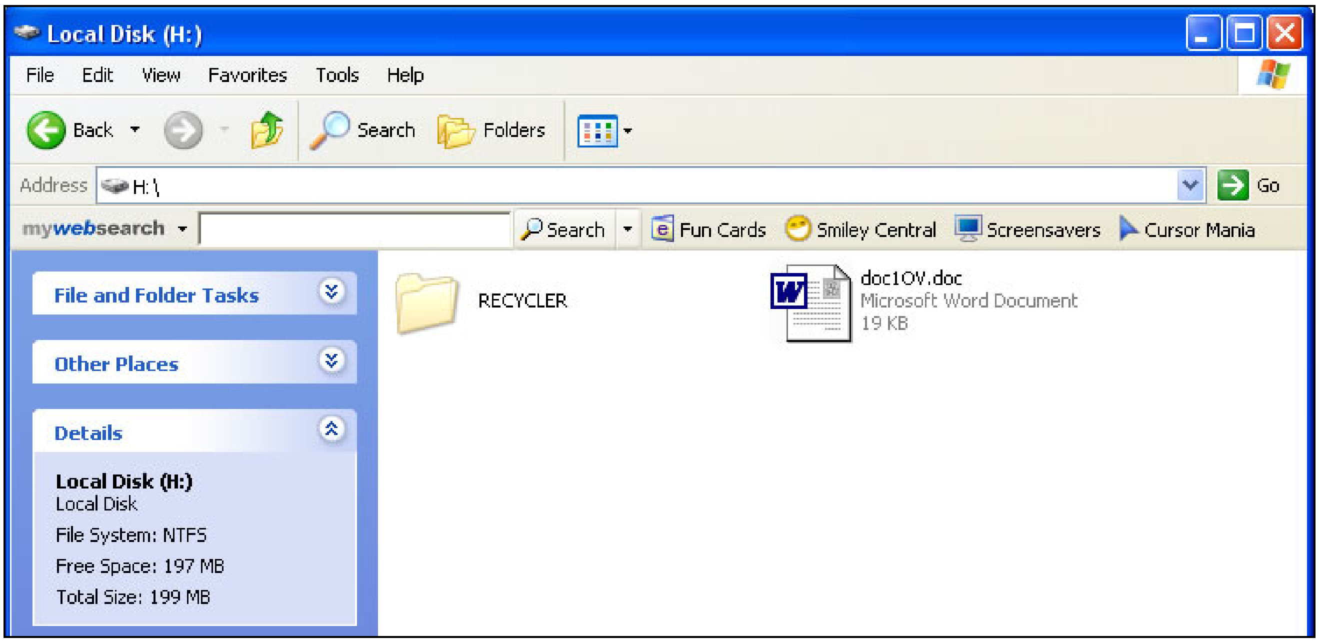
File View Tools Help

Drive	Volume
G:	C:\Vol1.vol
H:	C:\VolA.vol

2 drives mounted.

Figure 29: Drive letters assigned to mounted volumes

When the investigator inspects the H Drive using Windows Explorer, he can see that it contains one document, doc1OV.doc, and an empty RECYCLER folder, see Figure 30 below.


Local Disk (H:)
File Edit View Favorites Tools Help
Back Search Folders
Address H:\
Go
mywebsearch Search Fun Cards Smiley Central Screensavers Cursor Mania
File and Folder Tasks
Other Places
Details

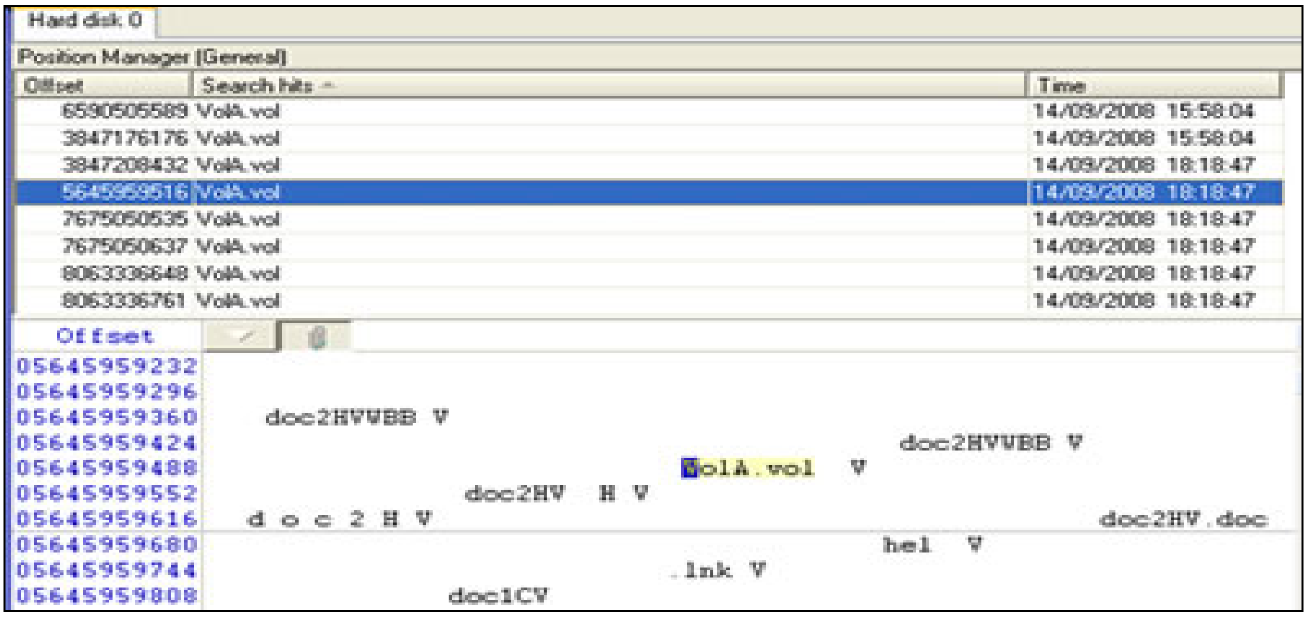
RECYCLER

doc1OV.doc
Microsoft Word Document
19 KB

Local Disk (H:)
Local Disk
File System: NTFS
Free Space: 197 MB
Total Size: 199 MB

Figure 30: Contents of the H Drive

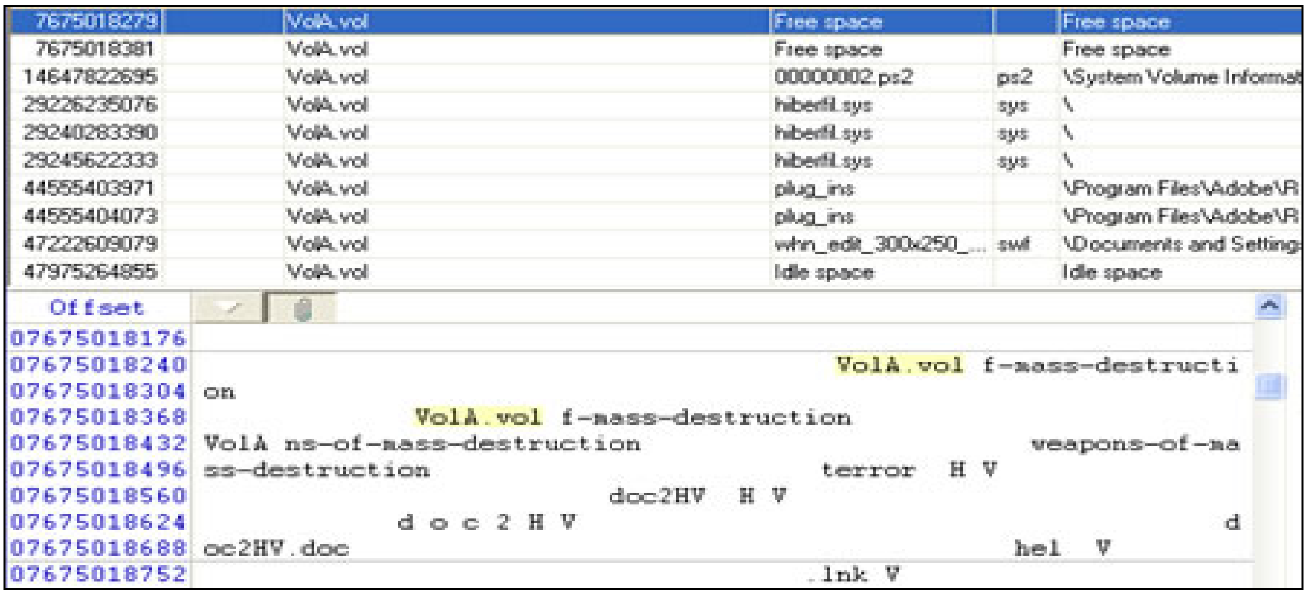
The doc1OV.doc document is found to contain nothing of interest. Using a forensic tool, the investigator performs a search for VolA.vol on the laptop's hard drive. As can be seen from Figures 31 and 32 below, VolA.vol contains documents that are not visible in Windows Explorer, namely doc2HV.doc and doc2HWWBB.doc. From the search results returned, the investigator find suspicious content in document doc2HV.doc, see Figure 32 below. This content raises his suspicious sufficiently enough to prompt him to do more in-depth analysis of the file system.



Offset	Search hits	Time
6590505589	VolA.vol	14/09/2008 15:58:04
3847176176	VolA.vol	14/09/2008 15:58:04
3847208432	VolA.vol	14/09/2008 18:18:47
5645959516	VolA.vol	14/09/2008 18:18:47
7675050535	VolA.vol	14/09/2008 18:18:47
7675050637	VolA.vol	14/09/2008 18:18:47
8063336648	VolA.vol	14/09/2008 18:18:47
8063336761	VolA.vol	14/09/2008 18:18:47

```
Offset
05645959232
05645959296
05645959360   doc2HVWBB V
05645959424                                      doc2HVWBB V
05645959488                         VolA.vol   V
05645959552             doc2HV   H V
05645959616   d o c 2 H V                                       doc2HV.doc
05645959680                                     hel    V
05645959744                         .lnk V
05645959808             doc1CV
```

Figure 31: Documents contained in VolA.vol not visible through Windows Explorer



7675018279		VolA.vol	Free space		Free space
7675018381		VolA.vol	Free space		Free space
14647822695		VolA.vol	00000002.ps2	ps2	\System Volume Informat
29226235076		VolA.vol	hiberfil.sys	sys	\
29240283390		VolA.vol	hiberfil.sys	sys	\
29245622333		VolA.vol	hiberfil.sys	sys	\
44555403971		VolA.vol	plug_ins		\Program Files\Adobe\R
44555404073		VolA.vol	plug_ins		\Program Files\Adobe\R
47222609079		VolA.vol	whn_edit_300x250_...	swf	\Documents and Setting
47975264855		VolA.vol	Idle space		Idle space

```
Offset
07675018176
07675018240                                            VolA.vol f-mass-destructi
07675018304 on
07675018368             VolA.vol f-mass-destruction
07675018432 VolA ns-of-mass-destruction                            weapons-of-ma
07675018496 ss-destruction                           terror   H V
07675018560                         doc2HV   H V
07675018624               d o c 2 H V                                        d
07675018688 oc2HV.doc                                             hel   V
07675018752                                          .lnk  V
```

Figure 32: Contents of documents contained in VolA.vol not visible in Windows Explorer

Further analysis would be to check the MFT entries for both documents to determine whether they are recent documents, through their file creation, file modification, file access and MFT modification times. The flags could also be checked to see if the documents are in use or deleted.

I believe the information obtained by the investigator during his analysis of the hard drive would be sufficient to raise suspicion that there is a hidden volume contained inside VolA.vol. Under these circumstances the user would be forced to disclose the passphrase or en/decryption keys for the hidden volume or face up to five years imprisonment.

CHAPTER 6 CONCLUSIONS

In conclusion, disk surface analysis on a hard disk that contains a FreeOTFE hidden volume has shown that no useful information can be obtained from the critical data block or master file table to establish the presence of the hidden volume.

However, leakage of data from the hidden volume and its enclosed files and documents, into various areas of the Windows operating system is sufficient to raise suspicion that a hidden volume is present. Therefore, the claim made by FreeOTFE that, when their product is used, there is no way for an adversary to identify the existance of hidden volumes or data contained within them, thereby maintaining plausible deniability should a user be requested or forced to reveal their encryption keys, has been disproved.

Some vendors claim that improper use of their products may result in loss of plausible deniability but FreeOTFE was installed and used in accordance with the instructions provided in the FreeOTFE user manual and evidence could still be obtained of the potential presence of a hidden volume.

In the United States of America, the aim of plausible deniability is to raise reasonable doubt as to “what” volume files really are and leaving it up the prosecution to prove, beyond a reasonable doubt, that they store encrypted data. However, in the UK, things are slightly different with the introduction of the Regulation of Investigatory Powers Act (RIPA), where a user is required to prove that data found is not a FreeOTFE volume or any other form of encrypted data, which may be very difficult or impossible for a user to do when they use FreeOTFE.

REFERENCES

[1] Regulation of Investigatory Powers Act 2000, 2000 CHAPTER 23 [www document] http://www.opsi.gov.uk/acts/acts2000/ukpga_20000023_en_1 (accessed: 27 July 2008)

[2] Canetti, R; Dwork, C; Naor, M & and Ostrovsky, R (1996), Deniable Encryption, Lecture Notes in Computer Science, Volume 1294, pp 90–104

[3] BestCrypt for Windows [www document] http://www.jetico.com/bcrypt8.htm (accessed: 24 July 2008)

[4] FreeOTFE [www document] http://www.freeotfe.org/docs/description.htm (accessed: 26 July 2008)

[5] TrueCrypt [www document] http://www.truecrypt.org/ (accessed: 22 July 2008)

[6] PhoneBookFS online manual [www document] http://www.freenet.org.nz/phonebook/manual.html (accessed: 22 July 2008)

[7] Rubberhose [www document] http://iq.org/~proff/rubberhose.org/ (accessed: 19 July 2008)

[8] Corsair (2007) USB Flash Wear-Leveling and Life Span [www document] http://www.corsairmemory.com/_faq/FAQ_flash_drive_wear_leveling.pdf (accessed: 1 August 2008)

[9] Karstens, N L (2006) Deniable Encryption [www document] http://www.karstens.us/DeniableEncryption.pdf (accessed: 18 July 2008), pp 1 - 10

[10] Czeski, A; St Hilaire, D J; Koscher, K; Gribble, S D; Kohno, T & Schneier, B (2008) Defeating Encrypted and Deniable File Systems: TrueCrypt v.5.1a and the Case of the Tattling OS and Applications [www document] http://en.wikipedia.org/wiki/Steganographic_file_system (accessed: 8 September 2008)

[11] WinHex: Computer Forensics & Data Recovery Software, Hex Editor & Disk Editor [www document] http://www.x-ways.net/winhex/ (accessed: 2 October 2008)

[12] Oler, B & El Fray, I (2007) Deniable File System--Application of Deniable Storage to Protection of Private Keys, Computer Information Systems and Industrial Management Applications, 2007, CISIM '07, 6th International Conference on 28 - 30 June 2007, pp 225 - 229

[13] Anderson, R; Needham, R; & Shamir A (1998) The Steganographic File System, In Information Hiding, 2nd International Workshop, Portland, Oregon, USA, Spring 1998

[14] RSA Laboratories: PKCS #5 v2.0: Password-Based Cryptography Standard, 25 March 1999

[15] Zhou, X; Pang, H & Tan, K (2004) Hiding Data Access in Steganographic File System in Data Engineering, 2004 Proceedings, 20th International Conference on 30 March - 2 April 2004, pp 572 – 583

[16] McDonald, A & Kuhn, M (1999) StegFS: A Steganographic File System for Linux, university of Cambridge [www document] http://www.cl.cam.ac.uk/~mgk25/ih99-stegfs.pdf (accessed: 13 September 2008)

[17] Pang, H; Tan, K & Zhou, X (2003) StegFS: a steganographic file system, Data Engineering, 2003. Proceedings, 19th International Conference on 5 - 8 March 2003, pp 657 – 667