Infiltrate the Vault: Security Analysis and Decryption of Lion Full Disk Encryption

Omar Choudary
University of Cambridge
omar.choudary@cl.cam.ac.uk

Felix Gröbert [*]
felix@groebert.org

Joachim Metz [*]
joachim.metz@gmail.com

Abstract

With the launch of Mac OS X 10.7 (Lion), Apple has introduced a volume encryption mechanism known as FileVault 2. Apple only disclosed marketing aspects of the closed-source software, e.g. its use of the AES-XTS tweakable encryption, but a publicly available security evaluation and detailed description was unavailable until now.

We have performed an extensive analysis of FileVault 2 and we have been able to find all the algorithms and parameters needed to successfully read an encrypted volume. This allows us to perform forensic investigations on encrypted volumes using our own tools.

In this paper we present the architecture of FileVault 2, giving details of the key derivation, encryption process and metadata structures needed to perform the volume decryption. Besides the analysis of the system, we have also built a library that can mount a volume encrypted with FileVault 2. As a contribution to the research and forensic communities we have made this library open source.

Additionally, we present an informal security evaluation of the system and comment on some of the design and implementation features. Among others we analyze the random number generator used to create the recovery password. We have also analyzed the entropy of each 512-byte block in the encrypted volume and discovered that part of the user data was left unencrypted [1].

[*]The opinions expressed in this paper are mine alone and do not reflect the opinions of my employer or affiliates of my employer unless otherwise explicitly stated.

[1]This was reported to Apple in 2011 and FileVault Disk Encryption has been altered accordingly. See Apple CVE-2011-3212.

1 Introduction

Since the launch of Mac OS X 10.7, also known as Lion, Apple includes a volume encryption software named FileVault 2 [8] in their operating system. While the previous version of FileVault (introduced with Mac OS X 10.3) only encrypted the home folder, FileVault 2 can encrypt the entire volume containing the operating system (this is commonly referred to as full disk encryption). This has two major implications: first, there is now a new functional layer between the encrypted volume and the original file system (typically a version of HFS Plus). This new functional layer is actually a full volume manager which Apple called CoreStorage [10] Although this full volume manager could be used for more than volume encryption (e.g. mirroring, snapshots or online storage migration) we don' t currently know of any other applications. Therefore in the rest of this paper we will use the term CoreStorage for the combination of the encrypted volume and the functional layer that links this volume to the actual HFS Plus filesystem.

The second implication is that the boot process is slightly modified since the user password or other token must be retrieved before being able to decrypt the data.

Apple' s volume encryption is equivalent to solutions such as PGP Whole Disk Encryption [12], TrueCrypt [13], Sophos SafeGuard [20], Credant [21], WinMagic SecureDoc [22], or Check Point FDE [23]. All these solutions can be used on corporate laptops, which generally contain sensitive data that must be protected at all times (even when the computer is turned off).

To the best of our knowledge, Apple has not released any documentation or source code on FileVault 2, which obstructs security experts and consumers to assess the security of the system. Also, the missing documentation and interfaces effectively limit the development of any third-party tools that can help in data recovery or forensic investigations.

In this paper we present the results of our analysis of FileVault 2: we were able to find most of its algorithms and parameters to the extent that we are able to read and mount a CoreStorage volume on a Linux machine. In the following sections we describe in detail the key derivation mechanisms and the encryption process. There remains unknown information in the volume header and metadata, that we believe is used for verification routines or other functionality of CoreStorage which does not seem to affect the read of the contents of the encrypted volume.

Based on these findings, we have developed a cross-platform library that reads and mounts CoreStorage volumes. Such a library can be used to analyse the contents of a particular file (or block) from an encrypted volume, without having to use Mac OS or even without having physical access to the Apple computer in question (e.g. by booting from a Linux live CD and connecting to the machine via the network).

Our work provides the security and forensic community with an open source library that can be used to analyse CoreStorage volumes, having the user password or recovery password. It is also possible to recover the data from a CoreStorage volume using a private key but our library does not support this yet. However, this merely involves updating the key derivation process, which we fully describe in this paper.

As part of our analysis we have also done an (informal) assessment of the security of the encryption mechanisms, including the random number generator mechanism used to create the recovery password. We found that some data is unnecessary protected (however weakly) while some other basic mechanisms such as code mangling are not used at all. We have also performed an entropy test on each 512-byte block of the encrypted volume and we found that part of the user data was left unencrypted.

1.1 Goals and motivations

Our primary goal has been to determine how FileVault 2 operates. As we will detail in the next section, this task involves finding how and where the encrypted volume master key is stored, how this key can be obtained from the user password or another token, what other data (the metadata) is available and how is used, and finally how the disk encryption and decryption are performed.

As part of this process we created an open source tool that can read and decrypt a CoreStorage volume.

Our motivation is twofold. First of all we needed a tool for digital forensic investigations. When a computer is suspected of malware or some malicious access we need to obtain certain files from the disk. If the computer is at a distant location it might not be feasible or convenient to send the computer or disk (for a Mac Book Air is not even trivial to remove the disk) in order to analyse it at the forensic laboratory. Even if we had physical access to a computer, we would not trust the operating system to extract the necessary files since malware could tamper with the OS. With FileVault 2 enabled, we cannot read the disk contents unless the Mac OS is running. Although it is possible to access the disk of a Mac computer with another Mac computer using a FireWire connection, many times we use forensic tools on another operating system than Mac OS and we cannot rely on remote locations having spare Mac computers for this task. Having cross-platform software to read encrypted volumes solves these problems. We can simply boot the Mac computer from a Linux live CD and use the software either locally or remotely to access the necessary files.

The second part of the motivation has been our desire to have a security assessment of the system. Since FileVault 2 can be used on corporate machines we needed to make sure that using this system instead of other known encryption solutions will not introduce vulnerabilities.

The next paragraphs provide a brief background on full disk encryption after which the details of FileVault 2 are discussed.

2 Background

Commonly a hard disk is logically organised in multiple sections, often referred to as either partitions or volumes. These volumes can be used for various purposes, and they are often structured according to a file system format (e.g. NTFS, FAT, HFS, etc.). It is possible to have a single disk with 3 volumes, where the first volume is formatted with NTFS and contains a Windows operating system, the second volume is formatted with EXT3 and contains an installation of a Linux distribution, while the third volume is formatted with FAT and only contains data (no operating system). The reader is encouraged to check the excellent book by Brian Carrier [28] for more details on the topic.

Volume encryption is a mechanism used to encrypt the contents of an entire volume. This is sometimes referred to as `full disk encryption`, which is misleading, since a physical disk can actually contain multiple volumes, each encrypted independently. The term `full disk encryption` is incorrectly used in the case of FileVault 2 and other systems such as BitLocker, that in fact perform volume encryption. Nonetheless, due to its popularity, we will continue to use the term `full disk encryption` in the remaining of this paper.

Now, here is one the main problems: since we are encrypting the entire volume, and this volume might be the volume containing the operating system, there is no unencrypted code left to actually boot the system. Therefore the minimum code required to decrypt the operating

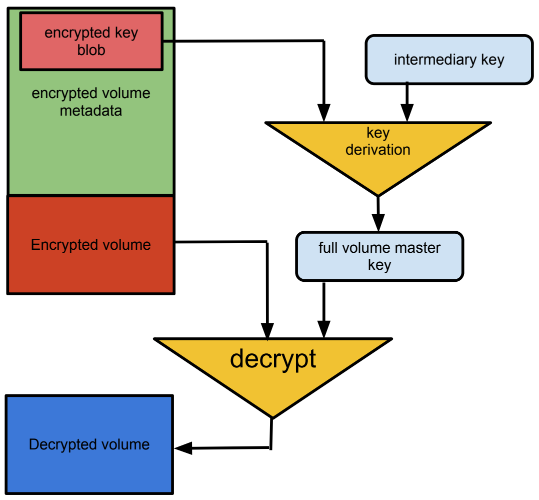


Figure 1: General framework for key derivation used in full disk encryption. The metadata containing the encrypted version of the volume master key is generally isolated from the actual encrypted volume, although it could exist in the same logical volume.

system (or an initial subset of it, enough to initialise the file system and read the OS) must reside somewhere else. This is a problem that is tackled differently by the many implementations of full disk encryption. In the next sections we describe how FileVault 2 deals with this problem.

Another very important matter is the key derivation. The volume is generally encrypted using some algorithm that relies on AES or other symmetric cipher (asymmetric cryptography would impact the read/write performance too much). Therefore, there must be a key that can unlock this encrypted volume.

A typical AES key is 128 or 256 bits, too long for users to remember or type every time they boot the machine: 128 bits can be represented with a minimum of 22 characters using base64 although base32 or hexadecimal are more user-friendly, each requiring 26 or 32 characters respectively. As a result, most full disk encryption schemes actually store this key (known as the `volume master key`) in an encrypted blob (usually by encrypting or hashing the key along with some additional information several times). This blob can only be decrypted using an intermediary key that is derived from the user password or another user-trusted input, e.g. a private key stored on a USB stick, a smart card, or a trusted platform module (TPM) linked to the encrypted volume (see Ferguson' s paper [1] for a detailed discussion on how to use these methods). This process is depicted in figure 1.

As you can observe from figure 1, the blob containing the encrypted volume master key is stored in a metadata section. This metadata is generally stored in the same disk as the encrypted volume, but on a different location. In the next sections we will explain exactly how this is done in FileVault 2.

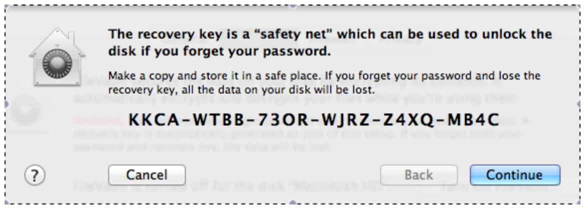
The third important aspect of full disk encryption, perhaps among most important from a security perspective, is the encryption operation itself. It can be tempting to think that simply using AES in a standard mode of operation such as CBC might work. However there are some issues to be taken into consideration.

File systems generally work with disk allocations (known as blocks) of 512 bytes or multiples of this size (e.g. 4096 bytes). This is mainly because traditional disks (those using magnetized platters) have their hardware and software drivers optimized for accessing blocks of 512 bytes (these are referred to as sectors). Therefore, in order to minimize the access time to a disk (which is probably the slowest operation done by a computer program), we should work with blocks multiple of this size. This requirement might hold even with new technologies such as solid state drive (SSD), since they use the same interface and logical block addressing (LBA) as older disks.

As a consequence of the 512 byte access restriction, full disk encryption is generally performed also on data chunks multiple of 512 bytes. If we decide to apply AES in CBC mode to each chunk we need to deal with a few issues. Using the same initial value (IV) for each encrypted block would lead to identical encrypted blocks for the same data, therefore allowing an attacker to watermark the disk and to determine the existence of certain data (i.e. breaking CPA security). We can improve this by using the block number, eventually encrypted under a key, as the IV. Although this approach can work (and is actually used in several full disk encryption implementations such as Linux Unified Key Setup [2]), it has the disadvantage that flipping one bit in the encrypted block will result in one bit flipped in the unencrypted version of the block, along with 16 corrupted bytes. For more details on these problems the reader in encouraged to see the work of Clemens [2] and Ferguson [1].

Bitlocker, the full disk encryption mechanism implemented in Windows since the version of Vista, deals with the flipping problem by adding a mechanism called `Elephant Diffuser`. FileVault 2 instead uses a tweakable encryption scheme known as AES-XTS. We detail this method in the next sections but for the moment we mention that it has several advantages over AES in CBC mode with a block-derived IV, and solves most of the problems discussed earlier.

The last important problem we should mention refers to the storage of the volume master key during system operation and sleep modes. As we mentioned earlier,



The recovery key is a “safety net" which can be used to unlock the disk if you forget your password.

Make a copy and store it in a safe place. If you forget your password and lose the recovery key, all the data on your disk will be lost.

KKCA-WTBB-73OR-WJRZ-Z4XQ-MB4C

Figure 2: the recovery password shown by Mac OS when FileVault 2 is enabled.

during the boot process the volume master key is derived from the user password or another token. Once this key is derived, the OS stores it in memory in order to read and write blocks efficiently without having to derive this key on every disk access. Halderman et al. [3] explain how an attacker with temporary access to a running system can scan the memory in order to retrieve the volume master key and therefore decrypt the contents of the disk. As far as we know this problem is still largely unsolved although computer manufacturers may use proprietary methods (such as on-board tamper resistant memories) to mitigate such attacks.

Having discussed the basic concepts used in full disk encryption, we move to the core of this paper, presenting the details of FileVault 2. A summary of the system is given in section 3.6.

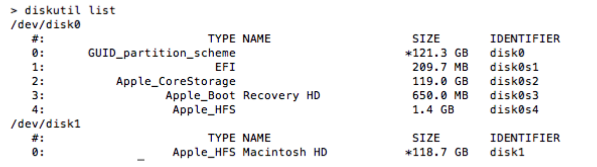
3 FileVault 2 architecture

3.1 Enabling FileVault 2

In Mac OS X Lion, after FileVault 2 is enabled, a series of events take place. First of all the user is presented with a 24 character recovery password (see figure 2). This password can be used to access the encrypted volume even if the user password is lost. We comment later on the security of this recovery method.

Next, the file system in the main volume is converted from the native HFSPlus type to CoreStorage (encrypted). During this operation, the user can still use the system at will and the `ConversionStatus` field in the EncryptedRoot.plist file (details below) contains the string `Converting`. After the encryption process is complete the string changes to `Complete`.

At the moment we do not know how the Mac OS keeps track of the encrypted blocks during the conversion process, so our tool cannot correctly mount volumes that are in a `Converting` state. This information could exist only somewhere in the last blocks of the volume, since all the data before the encrypted volume is zeroed (details in section 4). Comparing data from a partially encrypted volume with one fully encrypted we see



```
> diskutil list
/dev/disk0
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:      GUID_partition_scheme                        *121.3 GB   disk0
   1:                        EFI                         209.7 MB   disk0s1
   2:          Apple_CoreStorage                         119.0 GB   disk0s2
   3:                 Apple_Boot Recovery HD             650.0 MB   disk0s3
   4:                  Apple_HFS                         1.4 GB     disk0s4
/dev/disk1
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:                  Apple_HFS Macintosh HD           *118.7 GB   disk1
```

Figure 3: output of **diskutil list** run on a Mac Book Air with FileVault 2 enabled.

that the encrypted metadata (see details later) is identical, while the Disk Label metadata only differs in some bytes. While these bytes might be related to the encryption status, we believe the information, if available, is actually found near the encrypted metadata, in one of the encrypted chunks that follow the encrypted volume (see figure 9).

In addition to the encryption itself, a new volume generally called `Recovery HD` appears alongside the main `Macintosh HD` existing volume. This new partition contains the encrypted volume master key, as described below.

3.2 The new volume, boot process and EncryptedRoot.plist file

Running the command `diskutil list` on a Mac OS installation with FileVault 2 enabled will show an output similar to figure 3.

In figure 3 you can see the encrypted volume (disk0s2), the new volume (disk0s3), the original unmodified EFI volume (disk0s1), and also the unlocked (unencrypted) version of the main volume (disk1). There is also an additional partition (disk0s4) which we created just for testing purposes.

The new `Recovery HD` volume contains a series of new files, including new EFI boot code to deal with the encrypted volume. The EFI (Extensible Firmware Interface), now known as UEFI (Unified EFI) [24], is a recent standard meant to replace the traditional BIOS system to boot a computer. It contains all the necessary POST routines to check the hardware and the necessary code to locate the volume that has an operating system and start booting from there. Around 2006 Apple switched to the Intel architecture and decided to use EFI instead of the traditional BIOS [25]. When FileVault 2 is enabled, the existing EFI code (generally divided between a separate non-volatile memory location on the main board and a volume containing a `FAT` file system named `EFI`) is supplemented with code available in the new `Recovery HD` volume. This new code allows the system to display a UI where the user can type its password in order to unlock the encrypted volume key and load the actual OS.

Among all the files available in the new volume the most important for FileVault 2 operation is the `EncryptedRoot.plist.wipekey` file [2], which contains all the information needed to extract the volume master key from the user' s password or a recovery token.

The `EncryptedRoot.plist.wipekey` file is encrypted using AES-XTS (details later) with an all-zeros tweak key, but the encryption key is easily available in the header (first block) of the CoreStorage volume (see table 2 in Appendix). Among other data, the header block contains also the size of the entire volume (including metadata), another UUID which is used as a key to decrypt part of the metadata (see details later), and a CRC32 checksum. The actual polynomial that is used for the CRC32 calculation is currently unknown to us. However the open source code contains the precomputed computed CRC table found during the analysis. The CRC32 checksum is used to validate the values in the FileVault 2 metadata structures.

Once decrypted, the file `EncryptedRoot.plist` has an XML structure with the following important entries:

- PassphraseWrappedKEKStruct(1)
  - 284 byte structure for recovery password.
- PassphraseWrappedKEKStruct(2)
  - 284 byte structure for user password.
- KEKWrappedVolumeKeyStruct(1)
  - unused.
- KEKWrappedVolumeKeyStruct(2)
  - contains wrapped volume master key.

We mention that if multiple users are registered on the same machine then the `EncryptedRoot.plist` file will have a separate PassphraseWrappedKEK structure for each user.

In the following we present the key derivation mechanisms.

3.3 Key derivation

The volume master key is derived from the user password, a private key token or the recovery password. Since the volume master key is the same but the initial input is different, FileVault 2 uses an intermediary key that is decrypted under different inputs: password or recovery token. This intermediary key, which decrypts the volume master key, is called the volume key Key-Encryption-Key (KEK). The overall key derivation process is depicted in figure 4.

[2] found in `com.apple.boot.`*X*`/System/Library/Caches/com.apple.corestorage/` (where *X* changes randomly between *R*, *S* and *P*)

In this section we describe the key derivation process in the case of a volume master key derived from the user password. The first step is to derive the intermediary volume key KEK from the given password; this in turn is accomplished in two stages. We first need to use PBKDF2 to derive a key from the user password and then we need to decrypt the AES-wrapped version of the volume key KEK with this key.

PBKDF2 [15] is an algorithm used to derive a standard cryptographic key (of any length) from a user password (i.e. a written language text). The main objective of this algorithm is to make it difficult for an attacker to brute-force all possible values of the user password (which could be short or easy to guess). This is done by the use of two counter-measures: a salt and a large number of iterations. A random salt is required to avoid rainbow tables; such tables can be used to create precomputed results of a cryptographic operation on various inputs such as dictionary keywords. The number of iterations specifies how many times to apply a pseudo-random function such as HMAC-SHA1 [14] on the user password (and derived results) before creating the final key; this step effectively slows down any attacker trying to brute-force the user password (see section 4 for a discussion on this matter).

The salt value for PBKDF2 is found in the PassphraseWrappedKEK structure (see table 3 in Appendix) related to the user. However the number of PBKDF2 iterations is not defined there. In our analysis we initially found that FileVault 2 derived the number of iterations based on the amount of time required to compute a multiple of 1000 PBKDF2 computations. We have later seen there is a default of **41000** iterations if the precomputed value is lower, and this appears to be always the case on current hardware. Probably the static value will be used most of the time, since a variable value might lead to incompatibility between devices (now you can access the encrypted volume of an Apple computer from another device via FireWire) and even blocked access to your machine if it performs slower or faster in time. For completeness we mention that the pseudo-random function used by FileVault 2 with PBKDF2 is HMAC-SHA256.

Once we have derived the PBKDF2 key we can use this key to unwrap the intermediary KEK using the AES Wrap algorithm [16]. This algorithm is used to wrap (encrypt) a cryptographic key under another key. In the case of FileVault 2, the PBKDF2 key is used to decrypt the intermediary volume key KEK, which in turn is used to decrypt the volume master key (also encrypted using the AES Wrap algorithm). The volume master key is wrapped under the intermediary KEK to support its main goal: only authorised users should be able to decrypt the disk contents (hence to access this key), while the in-

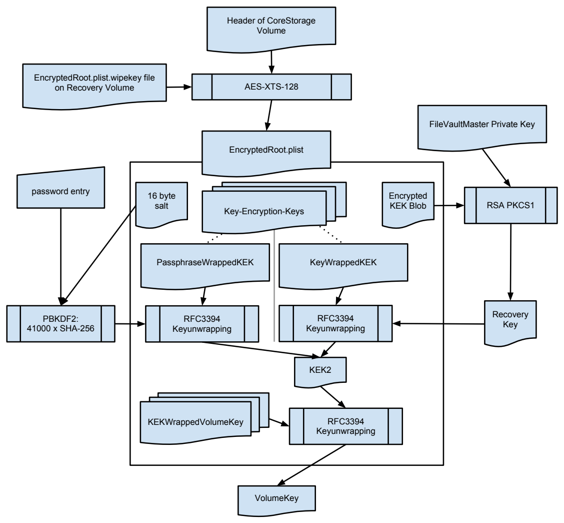


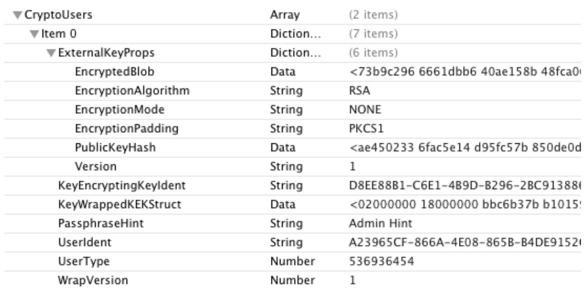
Figure 4: FileVault 2 key derivation process.

termediary KEK is again wrapped (instead of deriving it directly from the user password) under different keys to support multiple inputs: password, private key or recovery password. It could have been possible to simply use different intermediary keys, which represented the volume master key encrypted under different inputs, but (we can only speculate here) Apple probably decided to use this design in order to have more flexibility.

The AES Wrap algorithm has several properties which make it useful to wrap (encrypt) a key under another key. Firstly, we can concatenate other data to the key in question before encryption. Secondly, even if no other data is added, the algorithm uses a somewhat large number of iterations of the cryptographic operation (AES-CBC) to remove any correlation between input and output but also to slow down any potential brute-force attack. Thirdly, the algorithm uses a standard initial value (generally eight bytes with the value **0xA6**) which can be used to recognize if the unwrapping operation has been successful.

After deriving the volume key KEK using the AES Wrap algorithm (applying the unwrap operation), we can use this key to unwrap the volume master key from the KEKWrappedVolumeKey structure (see table 4 in Appendix).

In both unwrapping operations (to obtain the intermediary KEK and the volume master key), the wrapped data has 24 bytes but the unwrapped key has only 16 bytes. That is because the first 8 bytes of the unwrapped data actually contain the initial value of **0xA6** (a different value indicates a wrong decryption key), leaving only 16 bytes for the actual unwrapped key.



▼CryptoUsers	Array	(2 items)
▼Item 0	Diction...	(7 items)
▼ExternalKeyProps	Diction...	(6 items)
EncryptedBlob	Data	<73b9c296 6661dbb6 40ae158b 48fca0
EncryptionAlgorithm	String	RSA
EncryptionMode	String	NONE
EncryptionPadding	String	PKCS1
PublicKeyHash	Data	<ae450233 6fac5e14 d95fc57b 850de0d
Version	String	1
KeyEncryptingKeyIdent	String	D8EE88B1-C6E1-4B9D-B296-2BC91388
KeyWrappedKEKStruct	Data	<02000000 18000000 bbc6b37b b1015
PassphraseHint	String	Admin Hint
UserIdent	String	A23965CF-866A-4E08-865B-B4DE9152
UserType	Number	536936454
WrapVersion	Number	1

Figure 5: Example of an `EncryptedRoot.plist` when a private key is used.

The key derivation process can be summarised as:

```
p = get_user_password()
salt = get_salt_from_PassphraseWrappedKEK()
iterations = 41000
pk = pbkdf2(p, salt, iterations, HMAC-SHA256)
kek_wrapped = get_kek_from_PassphraseWrappedKEK()
kek = aes_unwrap(kek_wrapped, pk)
vmk_wrapped = get_vmk_from_KEKWrappedVolumeKey()
vmk = aes_unwrap(vmk_wrapped, kek)
```

The recovery password shown in figure 2 can be used exactly as the user password, with the mention that one must include the dashes as well.

If instead we use a private key from a recovery certificate, we need to get the intermediary KEK from the KeyWrappedKEK structure (instead of the PassphraseWrappedKEK, as depicted in figure 4). In order to unwrap the KeyWrappedKEK one needs to extract the recovery key which is used as input to the AES unwrapping algorithm. The recovery key is saved as an encrypted blob in the `EncryptedRoot.plist`, protected using RSA and PKCS#1 padding. The encrypted blob is added along the ExternalKeyProps to the `EncryptedRoot.plist` when a certificate is used for recovery (see figure 5 for an example). The recovery key can be extracted from the encrypted blob using a private key which is stored in a FileVaultMaster certificate, installed generally under the path `/Library/Keychains/FileVaultMaster.cer`.

Next we describe the encryption algorithm, AES-XTS, and present the last missing piece: the tweak key.

3.4 AES-XTS

AES-XTS [18] is the encryption mechanism used by FileVault 2 to encrypt a volume. The overall architecture of AES-XTS is shown in figure 6. AES-XTS is a type of tweakable encryption, using AES [19] as the block cipher. The XTS construction for tweakable encryption is based on Rogaway's XEX [7]. It allows each block on

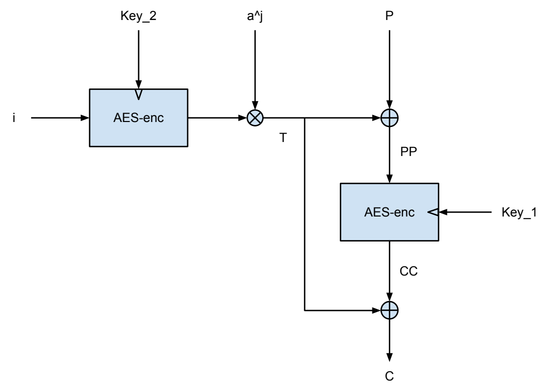


Figure 6: AES-XTS encryption.

the volume to be encrypted differently, even if the plaintext is the same, based on a tweak value.

The AES-XTS encryption operation is performed per block (of arbitrary size, although multiples of 128-bit are generally used). For each data block to be encrypted the algorithm expects two keys named key_1 and key_2 (also known as the tweak key), which can be either 128 or 256 bits long, and a 128-bit tweak value i which is usually derived from the block offset. The input data block is partitioned by the algorithm into 128-bit units (the last unit can have less than 128 bits). Each of these data units is assigned a sequential number j starting from 0, and is encrypted as follows: first the tweak value is converted to little endian and encrypted under key_2 using AES in ECB mode (this only needs to be done once per data block), then each byte of the encrypted tweak value is left shifted by j bits (this is the group multiplication shown in the diagram) with a possible addition of a carry and a special value of 135 on the first byte. The shifted result T (the encrypted tweak) is xor-ed into the corresponding input data unit P and the output is then encrypted under key_1 using AES-ECB. The result is then xor-ed with the same encrypted tweak yielding the final encrypted data unit. The complete encrypted data block is simply the concatenation of all the encrypted data units. Decryption is done similarly, replacing the second AES encryption by decryption.

AES-XTS has several advantages over alternatives such as AES in CBC: there is no requirement for an initialization vector (the tweak key can be derived from the block number); each block is encrypted differently (since the tweak value will be different); and unlike AES-CBC, AES-XTS prevents an attacker from changing one specific bit in a data unit by xor-ing each AES input with a different shifted version of the encrypted tweak.

FileVault 2 uses AES-XTS in several places, always with 128-bit keys, as shown in table 1. It is used to encrypt the EncryptedRoot.plist file, where key_1 is available on the main volume header and key_2 is 128 bits of zero (i.e. 16 zero bytes); the tweak value is 0 and the whole file is treated as one large block. AES-XTS is also used to encrypt part of the metadata (details below), where key_1 is the same key used to encrypt the `EncryptedRoot.plist` file but key_2 is another value, known as Physical Volume UUID, also found on the volume header. Finally, perhaps most importantly, AES-XTS is used to encrypt the main volume. In the previous section we have shown how to derive the volume master key, which is used as key_1 with AES-XTS. However we have not said anything about key_2 (the tweak key). Finding this tweak key was probably one of the most difficult parts in our search. The tweak key turned out to be derived from the volume master key and another value, known as the `Logical Volume Family UUID`, which is found in the encrypted metadata.

The AES-XTS encryption might be used in other places as well, since we found unknown key values during live debugging for unknown data. We have indications that these keys are used to encrypt paged data or other memory contents.

Next we present the metadata structures used in FileVault 2, how to derive the volume tweak key and how to use these data to decrypt the volume.

3.5 Metadata structures

The structures needed to decrypt the main volume, along with a header and other additional information, are stored inside the CoreStorage volume listed in figure 3. The layout of these structures within the volume is shown in figure 8.

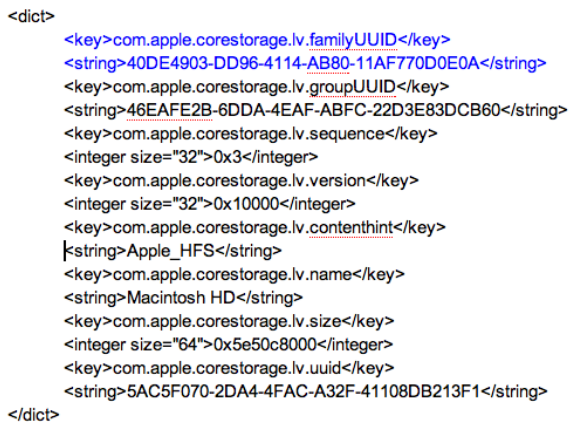
There are two essential metadata structures needed to decrypt the main volume: the `Disk Label` metadata and the `encrypted metadata`. The `Disk Label` metadata block offset can be found from the CoreStorage volume header (64-bit little endian value at byte offset 104, see table 2 in Appendix). This offset can be multiplied with the 32-bit block size value at byte offset 96 in the volume header to obtain the byte offset within the volume. The block size is generally 4096 (0x1000), which is also the standard block size for the HFSPlus file system. The size of the `Disk Label` metadata is given in the volume header, but in our experiments all the data of interest was available within the first 8704 bytes (i.e. seventeen 512-byte blocks).

We show the most important fields of the `Disk Label` metadata in table 5 in Appendix. The `Disk Label` metadata provides the offset of the encrypted metadata using the formula:

$$offset = disk\_label[disk\_label[220] + 32]$$

Table 1: Parameters for AES-XTS when used with different parts of FileVault 2.

data	key_1	key_2	**tweak start**	**block size**
`EncryptedRoot.plist`	offset 176 in CS header	0	0	entire file
Encrypted metadata	offset 176 in CS header	PV UUID	0	8192 bytes
Main volume	volume master key	volume tweak key	0	512 bytes



```
<dict>
        <key>com.apple.corestorage.lv.familyUUID</key>
        <string>40DE4903-DD96-4114-AB80-11AF770D0E0A</string>
        <key>com.apple.corestorage.lv.groupUUID</key>
        <string>46EAFE2B-6DDA-4EAF-ABFC-22D3E83DCB60</string>
        <key>com.apple.corestorage.lv.sequence</key>
        <integer size="32">0x3</integer>
        <key>com.apple.corestorage.lv.version</key>
        <integer size="32">0x10000</integer>
        <key>com.apple.corestorage.lv.contenthint</key>
        <string>Apple_HFS</string>
        <key>com.apple.corestorage.lv.name</key>
        <string>Macintosh HD</string>
        <key>com.apple.corestorage.lv.size</key>
        <integer size="64">0x5e50c8000</integer>
        <key>com.apple.corestorage.lv.uuid</key>
        <string>5AC5F070-2DA4-4FAC-A32F-41108DB213F1</string>
</dict>
```

Figure 7: Example of data in the first xml metadata structure.

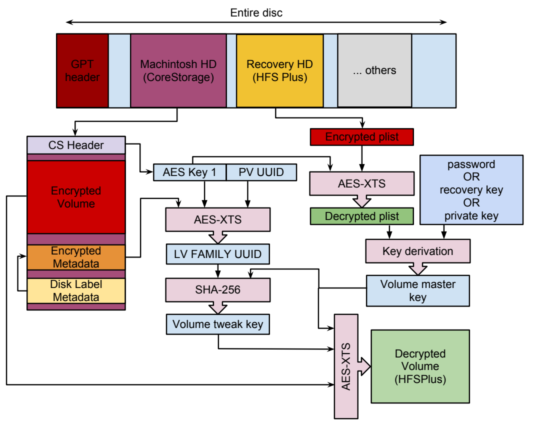


Figure 8: Diagram of FileVault 2 architecture.

The `encrypted metadata`, as our given name implies, is encrypted. We found that we can decrypt its contents using AES-XTS with key_1 and key_2 from the CoreStorage volume header (see table 2 in Appendix), starting with a tweak value of 0 and using an 8192-byte block size.

The contents of the encrypted metadata structure are shown in table 6 in Appendix. The essential details are: the encrypted volume size (which equals the size of the decrypted volume – this is smaller than the size of the entire CoreStorage volume), the encrypted volume offset, and the offset of the first xml metadata which contains the `Logical Volume Family UUID` (under the xml key `com.apple.corestorage.lv.familyUUID`).

The first, second and third xml metadata structures contain various UUIDs and other information related to the encrypted volume. However for decryption purposes we only need the `lv.familyUUID` (i.e. the `Logical Volume Family UUID`) which is available in both the first and third xml metadata structures. An example of the first xml metadata is shown in figure 7.

Using the Logical Volume Family UUID we can derive the full disk encryption tweak key by applying SHA-256 on the concatenation between the volume master key and this UUID, and then retaining only the first 16 bytes:

$$key_2 = MSB_{16}(SHA_{256}(volume\_master\_key|lv.familyUUID))$$

3.6 Full disk encryption and decryption

Now that we have presented the building blocks of FileVault 2 we can describe the entire volume decryption process, as depicted in figure 8. Firstly we need to decrypt the `EncryptedRoot.plist` file using the key from the volume header. Then, using the user' s password or a recovery token we can extract the volume master key. We can now derive the full disk encryption tweak key from the encrypted metadata and the volume master key. At this point we can use the volume master key as key_1 and the full disk encryption tweak key as key_2 with AES-XTS to decrypt the main volume, with a tweak value starting from 0 and a block size of 512 bytes.

4 Security analysis: problems and solutions

In this section we present the results of our informal security analysis of FileVault 2.

4.1 Random number generator and recovery password

Naturally, the security of the complete FileVault 2 system relies on the quality of the encryption keys. While some keys are derived from the user provided password, as explained earlier, other keys are randomly generated, for example the recovery password, key encryption keys and volume keys. Using the recovery password, it is possible to mount and decrypt the complete CoreStorage volume.

FileVault 2 features two recovery tokens: an RSA key and a recovery password. The asymmetric key recovery for corporate deployments is done using the same mechanism as in the first FileVault implementation: a FileVaultMaster certificate is installed in the system' s keychain and the public key of the certificate is used to encrypt the intermediary KEK key, which is then added to the `EncryptedRoot.plist` file.

In this section we will take a closer look at the second recovery mechanism, the recovery password, targeted at consumers. When activating FileVault 2, the System Preferences application displays a randomly generated 120 bit password, encoded with base32, to the end user and advises them to securely store the password for recovery (see figure 2). The recovery password is read from `/dev/random` (through libcsfde and SecCreateRecoveryPassword() in Security.framework).

Therefore, the security of the FileVault 2 system can be reduced to the security of the pseudo random number generator (PRNG) used in Mac OS X Lion for /dev/random. Mac OS relies on Counterpane' s implementation of the Yarrow PRNG [6] with modifications by Apple available as open source [9]. The Yarrow PRNG design has been obsoleted by Fortuna [11], written by the original authors.

To evaluate the strength of Apple' s implementation of Yarrow, we evaluated the seeding of the PRNG. Because the state of the PRNG is kept between reboots, we assume a scenario in which the end user activates FileVault 2 right after the first boot after operating system installation. This is a worst-case scenario, in which the PRNG has only been seeded with the least amount of entropy. During boot-time the PRNG is seeded with 8, 20 and 332 bytes. After boot-time, the PRNG is periodically seeded with 332 bytes every 10 minutes. If an attacker could guess the content of the seed, he would be able to recreate the PRNG' s state and predict its output, and hence determine the recovery password. The sources for the seeding are as follows:

- 8 bytes boot seed: this seed is deterministic because it is the value of the current `microtime()` during boot.

- 20 bytes boot seed: this is read from the `SystemEntropyCache` file which contains the previous state of the PRNG before reboot. The file is written by `EntropyManager` every 6 hours and during shutdown. It contains 20 byte output from `/dev/random`. This seed is deterministic in our above scenario because the system has booted the first time.

- 328 byte boot and periodic seed: this seed is triggered by `securityd` and the seed' s contents are collected in the kernel' s `kdbg_getentropy()` function. It is the core seed for the PRNG. The data contains 41 samples of `mach_absolute_time()`, which returns an 8 byte nano-precision time offset for different kernel threads. During 1000 reboots, we sampled the entropy seed of the PRNG. Our estimate is that the total seeding entropy is 40 samples of 8 bits of `mach_absolute_time`. This would result in 320 bits of total entropy because the nano-precision timestamps are only unpredictable in the lower bits and higher byte values have clearly reoccurring patterns. Thus, this represents a search space which is suboptimal, i.e. not every input to the seed is unpredictable and the amount of entropy input is less than what other operating systems seed [4, 5]. However, the search space is large enough that it is not brute-force-able.

For highly security-critical scenarios, the PRNG should be reseeded by manually writing entropy to `/dev/random` before activation of FileVault 2.

4.2 Plaintext bits in encrypted volume

Once we found most of the details about FileVault 2 operation we computed the entropy of each 512-byte block of the CoreStorage volume to verify our assumptions and also to make sure we were not missing any data. A bitmap of the volume is shown in figure 9, where blue corresponds to plain text, red to encrypted data and white to data with zero entropy (i.e. all bytes have the same value, such as all bytes 0x00 or 0xFF).

As you can see from the bitmap in figure 9, there is a large block of zero data (with the exception of the first header block) at the beginning of the disk, then a large portion of encrypted data corresponding to the encrypted volume, and at the end there is a mix of plain text, encrypted and zero data corresponding to the metadata, encrypted metadata and related structures, and the backup header (last block).

When we first looked at this data using the previous version of OS X (10.7.1), we discovered two interesting facts: firstly, there was a mix of plain text, zero and encrypted data at the beginning of the disk. Part of this data looked like unencrypted HFSPlus file system structures. We could observe what seemed to be valid HFSPlus headers, allocation and extents files. This led us to think for a moment that FileVault 2 might be a file based encryption mechanism rather than volume based. We could observe an allocation file unencrypted,

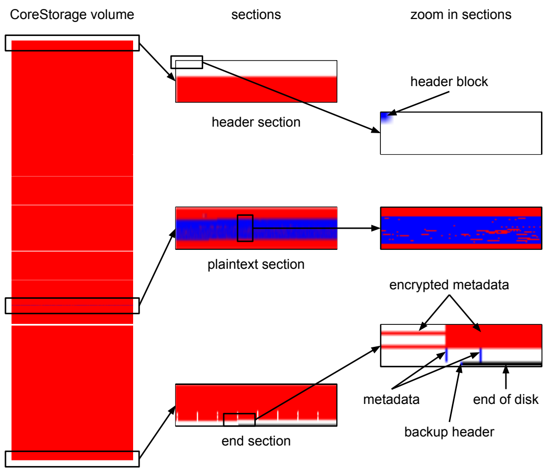


Figure 9: Entropy bitmap of the CoreStorage volume created by Mac OS X 10.7.2. Each pixel corresponds to a 512-byte block. Blue corresponds to plain text (low entropy), white to zero or other constant data (zero entropy), and red to encrypted data (very high entropy).

but the extents file was located inside an encrypted area. The plain text allocation file was still of interest and we thought that it may be used to track which blocks are encrypted (since during initial encryption one can still use the OS, which has to access both encrypted and unencrypted blocks). However this data turned out to be just a reminder of the original unencrypted volume. We discussed this matter with Judson Powers, Computer Forensics Director at ATC-NY, and he later reported this issue to Apple, who has now fixed this issue in the 10.7.2 update [26]. After the fix, all the data at the beginning of the volume is zeroed (see figure 9), which invalidates our assumption about the allocation file.

The second interesting fact about the volume data was a significant portion of plaintext (around 250 MB) in the middle of the encrypted volume. The plaintext blob contains code, dictionaries, journal metadata, error messages, debug messages and some user data.

Our best guess is that this is data from the base OS installation which has been encrypted elsewhere, but has not been wiped from the disk. We determined that long-used clear volumes could contain personal data in that part of the encrypted volume after activation of FileVault 2. We think this data should be analysed in more detail, and Apple should scrutinize its functionality, since any plaintext can be easily targeted by an attacker. We have advised Apple about this issue [3].

[3]Apple Product Security was contacted February 9th 2012, ticket id 191364581.

4.3 Possible attacks on the user password

In section 3.3 we mentioned that PBKDF2 is used to slow down an attacker that is trying to brute-force the user password. Let us now analyse the concrete case of FileVault 2. As Raeburn explains [17], for a given number of iterations N and a known salt, a 2GHz machine can perform approximately $2^{17}/N$ PBKDF2 iterations per second. Therefore, this machine would require about $N \cdot 2^{32-17} = N \cdot 2^{15}$ seconds to test all possible inputs in a large password set containing 2^{32} possible words. In the case of FileVault 2 we know the salt (from the `EncryptedRoot.plist file`) and we also know $N = 41000$ (between 2^{15} and 2^{16}). With $N = 41000$ the machine can do about 4 attempts per second and in order to try all the 2^{32} values the machine needs around 2^{30} seconds (or about 34 years). This situation is well above most security requirements.

However, let us look into a more attacker-favorable case, which might be common enough to be considered. Let us assume the password is a weak 6 character common word thus limiting the search to about 2^{16} possibilities. This effectively reduces the search to only $N \cdot 2^{16-17} = N/2 = 20500$ seconds, or 5.6 hours (even less with a better machine). This should be taken in consideration when trusting the user data is completely secure simply because FileVault 2 is enabled.

The details provided in this paper enable an organization to verify that FileVault 2 users have secure passwords, without requiring them to disclose their password. The site administrator could use a tool that tries to brute-force the user password against a defined set as described above. If the password is revealed then the administrator could request the user to choose a better password and then perform the check again. This step will ensure that users do not employ known weak passwords with FileVault 2.

The problem described above is inherent when a user password protects the underlying encryption. We are not aware of how we could improve this system without increasing the number of PBDKF2 iterations (which would increase the login delay) or requiring additional login tokens.

In contrast, Bitlocker does not have the same problem since it relies on an external TPM module to unlock the volume master key. The assumption here is that the user password is protected by the OS, which in turn can only be unlocked by the TPM. Therefore, you cannot get an image of the disk data and try to brute-force the key derivation because you do not have the TPM key (typically 256 bits).

4.4 Extracting keys from memory

As we point out in section 2, Halderman et al. [3] have shown that is possible to extract encryption keys from memory under many circumstances. FileVault 2 does not protect against extracting the keys from memory while the system is running: we have been able to retrieve all the necessary keys from memory using the standard GNU debugger (gdb).

Compared to Bitlocker in TPM mode, FileVault 2 is more resistant to key extraction attacks when the computer is turned off. That is because the volume master key is never loaded in memory unless the user provides the correct authentication token. On the other hand, Bitlocker loads the volume master key from the TPM without the need of the user password. For completeness we mention that Bitlocker can also be used with a recovery key instead of the TPM but this is not very common.

Maximillian Dornseif has shown that it is possible to also extract keys from memory using Firewire in DMA mode [27]. This enables any attacker with physical access to a running system to easily extract the memory contents, bypassing the OS and CPU since the transfer takes place via DMA. Apple has now blocked this feature with the OS X 10.7.2 update [26].

4.5 Obfuscation features

While analysing FileVault 2 we were a bit confused by some design decisions. We are not sure of the real advantage introduced by encrypting the `EncryptedRoot.plist` file. This file contains the keys in an encrypted blob (therefore security measures have already been taken), but the key to decrypt the file is available as plain text in the header of the CoreStorage volume (so any attacker can do that; for a dictionary attack, as detailed earlier, an attacker only needs to decrypt this file once). We believe that the main intend of encrypting the `EncryptedRoot.plist` file is obfuscation of the FileVault 2 system' s working, an approach contrary to Kerckhoff' s principle.

5 Conclusions

In this paper we have presented in detail the architecture of FileVault 2, the full disk encryption mechanism deployed by Apple with Mac OS X 10.7 (Lion), based on a broad analysis of the system components.

Our work allows any forensic investigator to use arbitrary tools to decrypt any data from a FileVault 2 encrypted volume, when the user password or a recovery token of the system are known. Further more, we have implemented an open source library and tooling to analyze and mount volumes encrypted with FileVault 2.

We have also made an informal security analysis of the system and found, among others, that the entropy of the recovery password can be improved and that part of the user data is available in the clear.

6 Acknowledgements

We are very thankful to Darren Bilby for his support in this work. We also thank Germano Caronni, Michael Cohen, Jan Monsch and Frank Stajano for many useful comments and discussions.

Omar Choudary is a recipient of the Google Europe Fellowship in Mobile Security, and this research is supported in part by this Google Fellowship. The opinions expressed in this paper do not represent the views of Google unless otherwise explicitly stated.

7 Availability

Our library to mount FileVault 2 encrypted volumes is available on Google Code:

`http://code.google.com/p/libfvde/`

These web pages also contain documents with full specs of the metadata used by FileVault 2.

References

[1] Niels Fergusson, "AES-CBC + Elephant difusser: A disk encryption algorithm for Windows Vista", *Microsoft Corp*, 2006

[2] Clemens Fruhwirth, "New methods in hard disk encryption", *Institute for Computer Languages, Theory and Logic*, 2005

[3] Halderman, J. Alex and Schoen, Seth D. and Heninger, Nadia and Clarkson, William and Paul, William and Calandrino, Joseph A. and Feldman, Ariel J. and Appelbaum, Jacob and Felten, Edward W., "Lest we remember: cold boot attacks on encryption keys", in *Proceedings of the 17th conference on Security symposium*, USENIX Association, pp 45–60, 2008

[4] Zvi Gutterman, Benny Pinkas, Tzachy Reinman, "Analysis of the Linux Random Number Generator", in *IEEE Symposium on Security and Privacy*, 2006.

[5] Leo Dorrendorf, Zvi Gutterman, Benny Pinkas, "Cryptanalysis of the random number generator of the Windows operating system", in *ACM Trans. Inf. Syst. Secur.*, 2009.

[6] Kelsey, John and Schneier, Bruce and Ferguson, Niels, "Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator", *Selected Areas in Cryptography*, 2000, Springer LNCS vol. 1758, pp 13–33

[7] P. Rogaway. "Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC", in *Advances in Cryptology*, ASIACRYPT 2004, Springer LNCS vol. 3329.

[8] Apple, "FileVault 2 features", at `http://www.apple.com/macosx/whats-new/features.html#filevault2`

[9] "Apple implementation of Yarrow", at `http://opensource.apple.com/source/xnu/xnu-1699.24.8/bsd/dev/random/`

[10] "Mac OS X Lion Adds CoreStorage, a Volume Manager " `http://blog.fosketts.net/2011/08/04/mac-osx-lion-corestorage-volume-manager/`

[11] Niels Ferguson and Bruce Schneier, *Practical Cryptography*, Wiley 2003.

[12] "PGP Whole Disk Encryption", at `http://www.symantec.com/business/whole-disk-encryption`

[13] "True Crypt open source project", at `http://www.truecrypt.org`

[14] "HMAC: Keyed-Hashing for Message Authentication", RFC 2104

[15] "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898

[16] "Advanced Encryption Standard (AES) Key Wrap Algorithm", RFC 3394

[17] "Advanced Encryption Standard (AES) Encryption for Kerberos 5", RFC 3962

[18] "The XTS-AES Tweakable Block Cipher", IEEE Std 1619-2007

[19] "Advanced Encryption Standard", FIPS 197

[20] "Sophos SafeGuard", at `http://www.sophos.com/de-de/products/encryption/safeguard-disk-encryption-for-mac.aspx`

[21] "Credant", at `http://www.credant.com/products/cmg-enterprise-edition/cmg-enterprise-edition-for-mac.html`

[22] "WinMagic SecureDoc", at `http://www.winmagic.com/products/full-disk-encryption-for-mac`

[23] "Check Point FDE", at `http://www.checkpoint.com/products/full-disk-encryption/index.html`

[24] "Unified Extensible Firmware Interface", at `http://www.uefi.org`

[25] "A Brief History of Apple and EFI", at `http://refit.sourceforge.net/info/apple_efi.html`

[26] "Apple OS X 10.7.2 security update", at `http://support.apple.com/kb/HT5002`

[27] Maximillian Dornseif, "0wned by an iPod", in *PacSec*, 2004

[28] Brian Carrier, "File System Forensic Analysis", *Addison-Wesley 2010*

Appendix

Table 2: CoreStorage volume header structure. All integer values are in little endian. Missing fields are either unknown or not essential and removed for brevity.

byte offset	length (bytes)	data
0	8	Checksum CRC32 for bytes 8..511, aligned to 64 bits with CRC32 at first bytes
8	40	Version number and other verification values
48	8	Disk sector size in bytes (generally 512).
64	8	Number of bytes in entire volume (including header and metadata sections)
88	2	Version String (0x43 0x53 = `CS`)
96	4	block size in bytes (generally 0x1000 = 4096)
100	4	disklabel metadata block size
104	8	block offset for disklabel metadata
168	4	key length (seen 0x10)
172	4	crypto algorithm version (seen 0x02)
176	16	AES-XTS key_1 used for the EncryptedRoot.plist file and the encrypted metadata
304	16	PhysicalVolume UUID. AES-XTS key_2 (tweak key) used for the encrypted metadata
320	16	LogicalVolumeGroup UUID

Table 3: PassphraseWrappedKEK structure, total length is 284 bytes. Integers in little endian.

byte offset	length (bytes)	data
0	8	Uncertain. May be tag and len of next field.
8	16	PBKDF2 salt.
24	8	Uncertain. May be tag and len of next field.
32	24	AES-wrapped volume KEK (used to unwrap the volume key).
56	228	Unknown.

Table 4: KEKWrappedVolumeKey structure, total length is 256 bytes. Integers in little endian.

byte offset	**length (bytes)**	**data**
0	8	Uncertain. May be tag and len of next field.
8	24	AES-wrapped volume key.
32	224	Unknown.

Table 5: Disk Label metadata structure. Integers represented in little endian. Missing fields are unknown or removed for brevity.

byte offset	**length (bytes)**	**data**
0	8	CRC32 padded to 64-bit.
168	1	Block size in bytes as a power of 2 (e.g. 0x0c means $2^{12} = 4096$).
220	4	Offset in this header where to look for the offset of the encrypted metadata. Usually 8192 (0x2000).
240	8	CoreStorage Physical volume size in units as defined in offset 168.
variable	8	Block offset of encrypted metadata: $position = disk\_label[220] + 32$.

Table 6: structure of the encrypted metadata structure. Integer values are in little endian. Omitted fields are unknown or removed for brevity.

byte offset	**length (bytes)**	**data**
248	8	block offset of data concerning the encrypted volume (start and size). Block size is 4096 (0x1000)
280	8	block offset of first xml metadata
variable	8	encrypted volume size in blocks $position = encrypted\_metadata[248] * 4096 + 72$
variable	8	block offset of beginning of the encrypted volume $position = encrypted\_metadata[248] * 4096 + 80$
variable	4	byte offset within current block of first xml metadata $position = encrypted\_metadata[280] * 4096 + 128$
variable	4	size of first xml metadata
variable	from above	first xml metadata $position = encrypted\_metadata[280] * 4096 + byte\_offset(first\_xml\_metadata)$
variable	variable	second xml metadata. Not essential. Third xml metadata also available at later offset but is not essential. Can be found by using a command like `grep xml` in the metadata contents