Chapter 23

SECURITY ANALYSIS AND DECRYPTION OF FILEVAULT 2

Omar Choudary, Felix Grobert and Joachim Metz

Abstract This paper describes the first security evaluation of FileVault 2, a volume encryption mechanism that was introduced in Mac OS X 10.7 (Lion). The evaluation results include the identification of the algorithms and data structures needed to successfully read an encrypted volume. Based on the analysis, an open-source tool named `libfvde` was developed to decrypt and mount volumes encrypted with FileVault 2. The tool can be used to perform forensic investigations on FileVault 2 encrypted volumes. Additionally, the evaluation discovered that part of the user data was left unencrypted; this was subsequently fixed in the CVE-2011-3212 operating system update.

Keywords: Volume encryption, full disk encryption, FileVault 2

1. Introduction

The FileVault 2 volume encryption software was first included in Mac OS X version 10.7 (Lion). While the earlier version of FileVault (introduced in Mac OS X 10.3) only encrypts the home folder, FileVault 2 can encrypt the entire volume containing the operating system – referred to as "full disk encryption." This has two major implications. The first is that there is a new functional layer between the encrypted volume and the original filesystem (typically a version of HFS Plus). This new functional layer is actually a full volume manager, which Apple calls CoreStorage. Although the full volume manager could be used for more than volume encryption (e.g., mirroring, snapshots and online storage migration), we do not know of any other applications. Therefore, in the rest of this paper we use the term CoreStorage to refer to the combination of the encrypted volume and the functional layer that links the volume to the HFS Plus filesystem. The second implication is that

G. Peterson and S. Shenoi (Eds.): Advances in Digital Forensics IX, IFIP AICT 410, pp. 349–363, 2013.

the boot process is modified because the user password or some other recovery token must be retrieved before the operating system can be decrypted.

Mac OS X volume encryption is similar to other volume encryption solutions. These include PGP Whole Disk Encryption [22], TrueCrypt [23], Sophos SafeGuard [21], Credant [6], WinMagic SecureDoc [24] and Check Point FDE [4].

This paper presents a detailed analysis of the FileVault 2 full disk encryption architecture, including the key derivation mechanisms and the data structures needed for decryption. Based on the analysis, an open source cross-platform library named `libfvde` was developed to decrypt and mount CoreStorage volumes. The library and the detailed documentation of the data structures used by FileVault 2 are available online [5]. The library can be used to analyze the contents of a particular file or block in an encrypted volume without having to use Mac OS and even without gaining physical access to the Apple computer in question (e.g., by booting from a Linux live CD and connecting to the machine via the Internet).

1.1 Motivation

Our main goal was to determine how FileVault 2 operates. This task involved finding how and where the encrypted volume master key is stored, how the key can be obtained from the user password or token, what other data (metadata) is available and how is used, and finally, how disk encryption and decryption are performed.

We were motivated by two factors. First, we needed a tool for digital forensic investigations. When a computer is suspected of having malware or having been the target of malicious access, it is necessary to obtain certain files from the disk. If the computer is at a distant location, it may not be feasible or convenient to transport the computer or disk to a forensic laboratory for analysis (it is not easy to remove the disk from a MacBook Air). Even if physical access is available to the computer, the native operating system cannot be trusted to extract the necessary files because of the presence of malware. Furthermore, although it is possible to access a FileVault 2 encrypted volume using another Mac computer connected via FireWire, this is a very limiting context. Our open source library (`libfvde`) does not have any of these limitations.

Our second motivation was to have a security evaluation of the system. Since FileVault 2 could be used on sensitive corporate machines, it is necessary to verify that no serious vulnerabilities exist.

2. Background

A hard disk is generally organized in multiple sections called partitions or volumes. These volumes are often structured according to a filesystem format (e.g., NTFS, FAT or HFS). It is possible to have a single disk with three 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, and the third volume is formatted with FAT and only contains data (no operating system). Interested readers are referred to [3] for details about this topic.

Volume encryption is a mechanism used to encrypt the contents of an entire volume. This is sometimes incorrectly referred to as "full disk encryption." We will use the term "volume encryption" in this paper to refer to the encryption performed by FileVault 2.

One of the main problems with volume encryption is that the operating system data is also encrypted, so there is no code left to boot the system. Therefore, the minimum code required to decrypt the operating system (or enough to initialize the filesystem and decrypt the rest of the operating system) must reside elsewhere. This is a problem that is tackled differently by the various volume encryption solutions.

Another important aspect of volume encryption is key derivation. The volume is generally encrypted using an algorithm that relies on AES or other symmetric cipher (asymmetric cryptography would have too much of an impact on read/write performance). Therefore, there must be a key that can unlock the encrypted volume. FileVault 2 uses 128-bit AES keys and has a layered architecture that allows multiple users to decrypt the same volume master key.

The next important aspect of volume encryption is the encryption operation itself. This operation must be carefully designed because there are several problems that can make a straightforward implementation insecure [11]. Also, the nature of disk encryption has special requirements with regard to speed (encryption should not introduce noticeable delays) and size (individual fixed-size sectors must be encrypted individually so they can be accessed independently).

The last important problem deals with the storage of the volume master key during the system operation and sleep modes. During the boot process, the volume master key is derived from the user password or token. After this key is derived, the operating system stores it in memory in order to read and write blocks efficiently without having to derive the key on every disk access. Halderman, *et al.* [13] have shown that an attacker with temporary access to a running system can scan the memory to retrieve the volume master key and then decrypt

Figure 1. Recovery password shown when FileVault 2 is enabled.

the disk contents. This is still a general problem, but it is possible to use proprietary methods such as on-board tamper resistant memory to mitigate these attacks.

3. FileVault 2 Architecture

This section describes the architecture and key features of FileVault 2.

3.1 Enabling FileVault 2

After FileVault 2 is enabled, a series of events take place. First, the user is presented with a 24-character recovery password (see Figure 1) that can be used to access the encrypted volume, even if the user password is lost.

Next, the filesystem in the main volume is converted from the native HFS Plus type to CoreStorage (encrypted). During this operation, the user can still use the system and the ConversionStatus field in the `EncryptedRoot.plist` file (details are provided below) contains the string "Converting." After the encryption process is complete, the string is changed to "Complete." At this time, we do not know how the operating system keeps track of the encrypted blocks during the conversion process, so our tool cannot correctly mount volumes that are in the Converting state. We are continuing to investigate this situation.

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

Running the command `diskutil list` on a Mac OS installation with FileVault 2 enabled yields an output similar to that shown in Figure 2. The output contains the encrypted volume (`disk0s2`), Recovery HD volume (`disk0s3`), original unmodified EFI volume (`disk0s1`) and also the unlocked (unencrypted) version of the main volume (`disk1`). There

```
> 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 2. Output of `diskutil list` run on a Mac Book Air with FileVault 2 enabled.

is also an additional partition (`disk0s4`), which we created only for testing purposes.

The Recovery HD volume contains a series of new files, including new EFI boot code to deal with the encrypted volume. Among all the files available in the new volume, the most important for FileVault 2 operation is the `EncryptedRoot.plist.wipekey` file, which is found at: *com.apple.boot.X/System/Library/Caches/com.apple.corestorage* where X changes between R, S and P. This file contains all the information needed to extract the volume master key from the user password or recovery token.

The `EncryptedRoot.plist.wipekey` file is encrypted using AES-XTS with an all-zeros "tweak key," but the encryption key is easily available in the header (first block) of the CoreStorage volume. The header block also contains other data, including the size of the entire volume (including metadata), another UUID that is used as a key to decrypt part of the metadata, and a CRC32 checksum. The checksum uses a weak CRC calculation based on the Castagnoli (CRC-32C) polynomial. The same checksum is used to validate the values in the FileVault 2 metadata structures.

After it is decrypted, the `EncryptedRoot.plist` file includes the following important entries: recovery password PassphraseWrappedKEKStruct, user password PassphraseWrappedKEKStruct and wrapped volume master key KEKWrappedVolumeKeyStruct. If multiple users are registered on the same machine, then the `EncryptedRoot.plist` file has a separate PassphraseWrappedKEK structure for each user.

3.2 Key Derivation

The volume master key is derived from the user password, private key token or recovery password. Since the volume master key is the same but the input may be 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

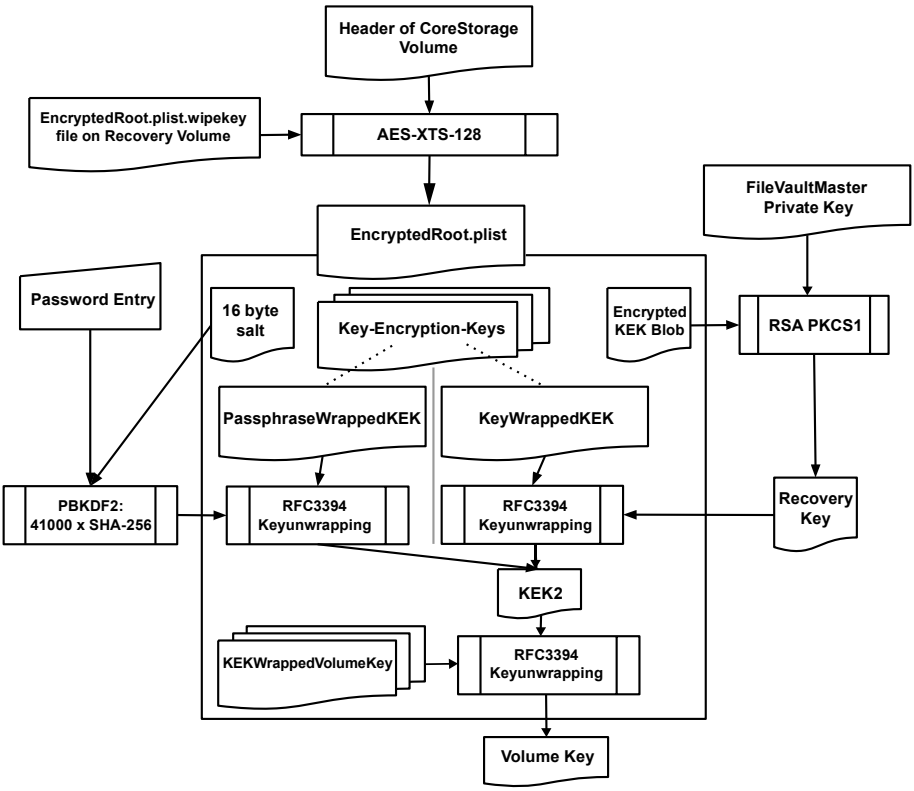

Figure 3. FileVault 2 key derivation process.

volume Key-Encryption-Key (KEK). The overall key derivation process is shown in Figure 3.

We now describe the key derivation process when the volume master key is derived from the user password. The first step is to derive the intermediary volume KEK from the given password; this is accomplished in two stages. First, PBKDF2 is used to derive a key from the user password. Then, the AES-wrapped version of the volume KEK is decrypted using this key.

The PBKDF2 Algorithm [14] derives a standard cryptographic key (of any length) from an arbitrary password string. The main objective of the algorithm is to make it difficult for an attacker to brute force all possible values of the user password. This is accomplished using two countermeasures: a salt to prevent rainbow table attacks and a large number of iterations of a pseudorandom function to increase the computational time. In the case of FileVault 2, both the salt and the number of iterations are available in the PassphraseWrappedKEK structure from

the `EncryptedRoot.plist` file. We observed that FileVault 2 uses a fixed number of 41,000 iterations of HMAC-SHA256, although we found code that allows a variable number of iterations that are multiples of 1,000 (but this code does not appear to be used).

After the PBKDF2 key is derived, it can be used to unwrap the volume KEK using the AES Wrap Algorithm [20]. The wrapped version of the volume KEK is also found in the PassphraseWrappedKEK structure. After obtaining the volume KEK, the same AES Wrap Algorithm can be applied to the data from the KEKWrappedVolumeKeyStruct structure to obtain the volume master key.

In both the unwrapping operations, the wrapped data has 24 bytes, but the unwrapped key has only sixteen bytes. This is because the first eight bytes of the unwrapped data actually contain the initial value of `0xA6` (a different value indicates a wrong decryption key), leaving only sixteen bytes for the actual unwrapped key.

The recovery password shown in Figure 1 can be used exactly as the user password. Note that the dashes are part of the password and must be included.

As an alternative decryption token – especially for organizations that need key escrow – a private key in the FileVaultMaster certificate may be used. This certificate is generally installed in */Library/Keychains/FileVaultMaster.cer*. In this case, the volume KEK is obtained from the KeyWrappedKEK structure instead of the PassphraseWrappedKEK structure. The KeyWrappedKEK structure contains a wrapped version of the volume KEK. To unwrap it, it is necessary to use a recovery key that is saved as an encrypted blob (in file `EncryptedRoot.plist`) that is protected using RSA and PKCS#1 padding. Note that the encrypted blob is added along with ExternalKeyProps to `EncryptedRoot.plist` when a certificate is used for recovery. The recovery key can be extracted from the blob using the private key from the FileVaultMaster certificate.

3.3 AES-XTS

FileVault 2 uses the AES-XTS Algorithm [16] to encrypt data. AES-XTS is a type of "tweakable encryption" that uses AES [17] as the block cipher. The XTS construction for tweakable encryption is based on XEX [19]. It uses a tweak value to encrypt each block on the volume differently, even if the plaintext is the same.

The AES-XTS encryption operation is performed per block. The blocks can be of arbitrary size, although multiples of 128 bits are generally used. For each data block to be encrypted, the algorithm expects

two keys named key_1 and key_2 (tweak key) that are 128 or 256 bits long, and a 128-bit tweak value i that is usually derived from the block offset.

AES-XTS has several advantages over alternatives such as AES-CBC: there is no requirement for an initialization vector because the tweak key can be derived from the block number; each block is encrypted differently based on the tweak value; furthermore, 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. 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 zeroes (i.e., 16 zero bytes); the tweak value is zero and the entire file is treated as one large block. AES-XTS is also used to encrypt part of the metadata, 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, AES-XTS is used to encrypt the main volume.

In the previous section, we discussed how to derive the volume master key, which is used as key_1 with AES-XTS. However, key_2 (tweak key) is also required. Finding this tweak key was one of our most difficult tasks. Eventually, we discovered that the tweak key is derived from the volume master key and another value, Logical Volume Family UUID, which is found in the encrypted metadata.

AES-XTS encryption might be used in other places as well. We found unknown key values when performing live debugging for unknown data. We have indications that these keys are used to encrypt paged data and other memory contents.

3.4 Metadata Structures

The structures needed to decrypt the main volume, along with a header and other additional information, are stored in the CoreStorage volume listed in Figure 2. The layout of these structures within the volume is shown in Figure 4.

Two essential metadata structures are needed to decrypt the main volume: the Disk Label metadata and the encrypted metadata. The Disk Label metadata block offset and size can be found in the CoreStorage volume header. The Disk Label metadata block contains a pointer to other metadata blocks that are encrypted, so we refer to these blocks as encrypted metadata. The contents can be decrypted using AES-XTS with key_1 and key_2 from the CoreStorage volume header, starting with

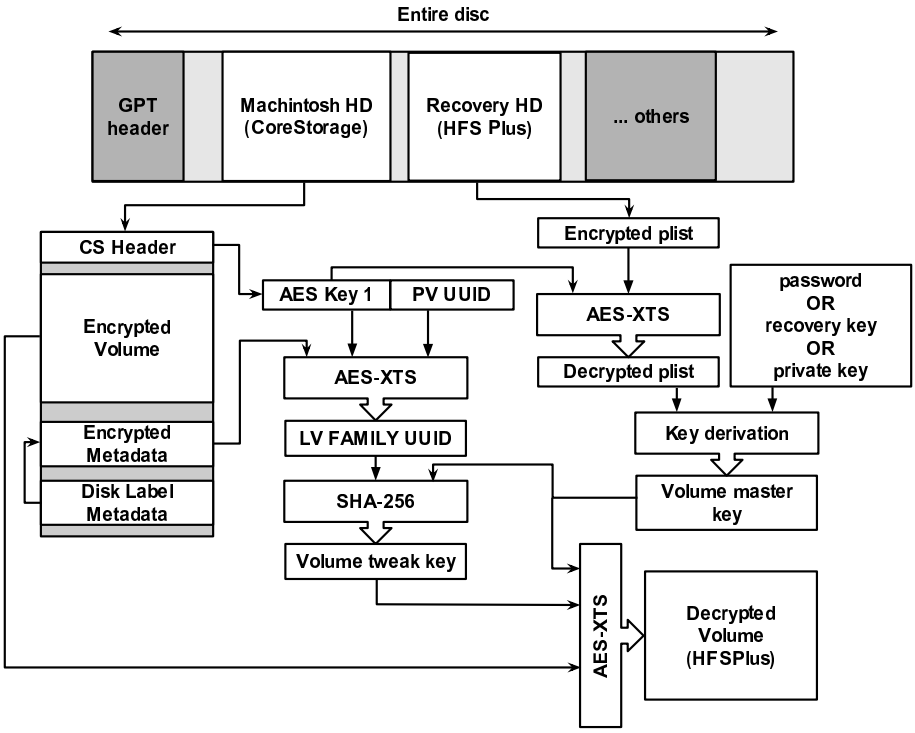

Figure 4. FileVault 2 architecture.

a tweak value of zero and using a block size of 8,192-bytes (size of an encrypted metadata block).

We are still investigating the exact structure of the metadata, which is likely related to the structure of CoreStorage. Readers who are interested in the latest information about these structures and other aspects of our work are referred to the project website [5].

The encrypted metadata contains, among other things, an XML structure with a Logical Volume Family UUID (lv.familyUUID). This UUID can be used to derive the volume tweak key by applying SHA-256 to the concatenation of the volume master key and the UUID, and retaining the first 16 bytes of the result:

$$key_2 = \mathrm{MSB}_{16}(\mathrm{SHA}_{256}(\text{volume master key} \mid \text{lv.familyUUID}))$$

3.5 Full Disk Encryption and Decryption

Having presented the building blocks of FileVault 2, we can describe the entire volume decryption process, which is illustrated in Figure 4. First, the `EncryptedRoot.plist` file is decrypted using the key from

the volume header. Then, the user password or recovery token is used to extract the volume master key. Following this, the volume tweak key is derived from the encrypted metadata and the volume master key. At this point, AES-XTS is used with the volume master key as key_1 and the volume tweak key as key_2 to decrypt the main volume, with a tweak value starting from zero and a block size of 512 bytes.

4. Security Analysis

This section presents the results of our security analysis of FileVault 2.

4.1 Recovery Password

When activating FileVault 2, the System Preferences application displays a randomly-generated 120-bit password (base32 encoded) to the end user and advises that the password should be stored securely for data recovery (see Figure 1).

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 pseudorandom number generator (PRNG) used in Mac OS X Lion for `/dev/random`. Mac OS relies on Counterpane's implementation of the Yarrow PRNG [15] with modifications by Apple available as open source [1]. The Yarrow PRNG design has been rendered obsolete by Fortuna [10], which was written by the original authors.

We evaluated the seeding of the PRNG to evaluate the strength of Apple's implementation of Yarrow. Because the state of the PRNG is kept between reboots, we assume a scenario in which an end user activates FileVault 2 right after the first boot after the operating system is installed. This is the worst-case scenario where the PRNG is 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. An attacker who guesses the seed correctly could recreate the PRNG state and predict its output, thereby determining the recovery password. The sources for the seeding are as follows:

- **Boot Seed (8 Bytes):** This seed is deterministic because it is the value of the current `microtime()` during boot.
- **Boot Seed (20 Bytes):** This seed is read from the `SystemEntropyCache` file, which contains the previous state of the PRNG before reboot. The file is written by EntropyManager every six hours and during shutdown. It contains a 20-byte output from `/dev/random`.

This seed is deterministic in our scenario because the system is booted for the first time.

- **Boot and Periodic Seed (328 Bytes):** This seed is triggered by `securityd` and the contents of the seed are collected in the kernel function `kdbg_getentropy()`. It corresponds to the core seed for the PRNG. The data contains 41 samples of `mach_absolute_time()` that returns an 8-byte nano-precision time offset for different kernel threads. We sampled the entropy seed of the PRNG over 1,000 reboots. Our estimate is that the total seeding entropy is 40 samples of eight 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 the higher bytes have repeating patterns. Thus, this represents a suboptimal search space, i.e., not every input to the seed is unpredictable and the amount of entropy input is less than what other operating systems seed [8, 12]. However, the search space is large enough to prevent it from being brute forced.

In a security-critical scenario, the PRNG should be reseeded by manually writing entropy to `/dev/random` before activating FileVault 2.

4.2 Plaintext Bits in Encrypted Volume

Having discovered most of the details about the operation of FileVault 2, we computed the entropy of each 512-byte block of the CoreStorage volume to verify our assumptions and also to ensure that we did not miss any data.

Figure 5 shows a bitmap of the volume encrypted with Mac OS X 10.7.2. Each pixel corresponds to a 512-byte block. Blue (dark) regions correspond to plaintext (low entropy), white regions correspond to zero or constant data such as all bytes with `0x00` or `0xFF` (zero entropy), and red (bright) regions correspond to encrypted data (very high entropy).

The bitmap shows a large block of zero data (with the exception of the first header block) at the beginning of the disk. This is followed by a large amount of encrypted data corresponding to the encrypted volume. At the end, there is a mix of plaintext, encrypted and zero data corresponding to the metadata, encrypted metadata and related structures, and the backup header (last block).

Upon examining the encrypted data portion, we discovered that there is a significant portion of plaintext (around 250 MB) in the middle of the encrypted volume. This plaintext blob contains code, dictionaries, journal metadata, error messages, debug messages and some user data.

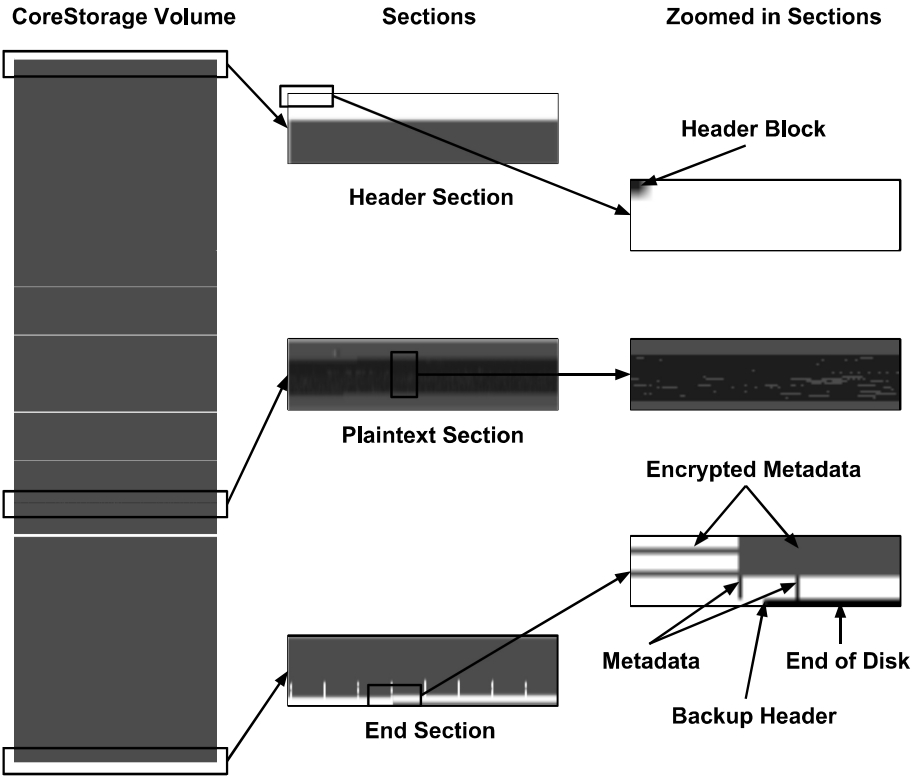

Figure 5. Entropy bitmap of the CoreStorage volume created by Mac OS X 10.7.2.

Our best guess is that this data is from the base operating system installation that was encrypted elsewhere, but that had not been wiped from the disk. Furthermore, we determined that long-used clear volumes could contain personal (possibly sensitive) data after the activation of FileVault 2. We advised Apple about this problem on February 9, 2012 (ticket ID 191364581), and it was fixed in the next update.

4.3 Possible User Password Attacks

PBKDF2 is used to slow down brute-force attacks on user passwords. Raeburn [18] reports that, for N iterations and a known salt, a 2 GHz machine can perform approximately $2^{17}/N$ PBKDF2 iterations per second. Therefore, the 2 GHz machine would require about 34 years to brute force a FileVault 2 password using a data set of 2^{32} words. However, if the password is a weak six-character common word, it could be determined within five hours. This should be taken into considera-

tion before assuming that user data is completely secure simply because FileVault 2 is enabled.

The details provided in this paper make it possible to verify that FileVault 2 users have secure passwords without requiring them to disclose their passwords. A systems administrator could use a tool that brute forces user passwords using a data set. If a password is revealed, then the administrator could request the user to choose a better password and perform the check again. This would ensure that users do not employ known weak passwords with FileVault 2.

4.4 Extracting Keys from Memory

Halderman, *et al.* [13] have shown that is possible to extract encryption keys from memory under many circumstances. FileVault 2 is also vulnerable to this attack: we could retrieve all the necessary keys from memory using the standard GNU debugger (`gdb`).

Compared with Bitlocker [9] in the TPM mode, FileVault 2 is more resistant to key extraction attacks when the computer is turned off. This 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 needing the user password. Note that Bitlocker can also be used with a recovery key instead of the TPM, but this is not very common.

Dornseif [7] has shown that keys can be extracted from memory using FireWire in the DMA mode. This enables an attacker with physical access to a running system to extract the memory contents, bypassing the operating system and CPU because the transfer takes place via DMA. Fortunately, Apple addressed this problem in the OS X 10.7.2 update [2].

5. Conclusions

This paper has presented the first detailed analysis of the FileVault 2 volume encryption system that was introduced in Mac OS X 10.7 (Lion), including the key derivation mechanisms and the data structures needed for decryption. As a result of the analysis, an open source cross-platform library named `libfvde` was developed to decrypt and mount FileVault 2 encrypted volumes when the user password or recovery token are available. The library and the detailed documentation of the data structures used by FileVault 2 are available at the project website [5]. Two major points revealed by the security analysis are that the entropy of the recovery password can be increased and that a portion of user data is available as plaintext.

Acknowledgements

We thank Darren Bilby for his support of this work. We also thank Germano Caronni, Michael Cohen, Jan Monsch and Frank Stajano for their comments. This research was supported by a Google Europe Fellowship in Mobile Security awarded to Omar Choudary.

References

[1] Apple, Source Browser, Cupertino, California (`opensource.apple.com/source/xnu/xnu-1699.24.8/bsd/dev/random`), 2010.

[2] Apple, About the security content of OS X Lion v10.7.2 and security update 2011-006, Cupertino, California (`support.apple.com/kb/HT5002`), 2011.

[3] B. Carrier, *File System Forensic Analysis*, Pearson Education, Upper Saddle River, New Jersey, 2005.

[4] Check Point Software Technologies, Check Point Full Disk Encryption, San Carlos, California (`www.checkpoint.com/products/full-disk-encryption`), 2013.

[5] O. Choudary and J. Metz, `libfvde`: Library and tools to access FileVault Drive Encryption (FVDE) encrypted volumes (`code.google.com/p/libfvde`), 2013.

[6] Dell, Credant Enterprise Edition for Mac, Round Rock, Texas (`www.credant.com/ products/cmg-enterprise-edition/cmg-enterprise-edition-for-mac.html`), 2013.

[7] M. Dornseif, Owned by an iPod, presented at the *PacSec Conference*, 2004.

[8] L. Dorrendorf, Z. Gutterman and B. Pinkas, Cryptanalysis of the random number generator of the Windows operating system, *ACM Transactions on Information and System Security*, vol. 13(1), article no. 10, 2009.

[9] N. Ferguson, AES-CBC + Elephant Difusser: A Disk Encryption Algorithm for Windows Vista, Technical Report, Microsoft, Redmond, Washington, 2006.

[10] N. Ferguson and B. Schneier, *Practical Cryptography*, Wiley, Indianapolis, Indiana, 2003.

[11] C. Fruhwirth, New Methods in Hard Disk Encryption, Theory and Logic Group, Institute for Computer Languages, Vienna University of Technology, Vienna, Austria (`clemens.endorphin.org/nmihde/nmihde-A4-ds.pdf`), 2005.

[12] Z. Gutterman, B. Pinkas and T. Reinman, Analysis of the Linux random number generator, *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 371–385, 2006.

[13] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum and E. Felten, Lest we remember: Cold boot attacks on encryption keys, *Communications of the ACM*, vol. 52(5), pp. 91–98, 2009.

[14] B. Kalisky, PKCS #5: Password-Based Cryptography Specification Version 2.0, RFC 2898, 2000.

[15] J. Kelsey, B. Schneier and N. Ferguson, Yarrow-160: Notes on the design and analysis of the Yarrow cryptographic pseudorandom number generator, *Proceedings of the Sixth International Workshop on Selected Areas in Cryptography*, pp. 13–33, 2000.

[16] L. Martin, XTS: A mode of AES for encrypting hard disks, *IEEE Security and Privacy*, vol. 8(3), pp. 68–69, 2010.

[17] National Institute of Standards and Technology, Specification for the Advanced Encryption Standard (AES), Federal Information Processing Standards Publication 197, Gaithersburg, Maryland, 2001.

[18] K. Raeburn, Advanced Encryption Standard (AES) Encryption for Kerberos 5, RFC 3962, 2005.

[19] P. Rogaway, Efficient instantiations of tweakable block ciphers and refinements to modes OCB and PMAC, *Proceedings of the Tenth International Conference on the Theory and Application of Cryptology and Information Security*, pp. 16–31, 2004.

[20] J. Schaad and R. Housley, Advanced Encryption Standard (AES) Key Wrap Algorithm, RFC 3394, 2002.

[21] Sophos, SafeGuard Enterprise, Abingdon, United Kingdom (`www.sophos.com/en-us/products/encryption/safeguard-enterprise.aspx`), 2013.

[22] Symantec, Symantec Drive Encryption, Mountain View, California (`www.symantec.com/drive-encryption`), 2013.

[23] TrueCrypt Foundation, TrueCrypt (`www.truecrypt.org`), 2012.

[24] WinMagic, SecureDoc for Mac, Mississauga, Canada (`www.winmagic.com/products/full-disk-encryption-for-mac`), 2013.