APFS Slack Analysis and Detection of Hidden Data

Security and Network Engineering, University of Amsterdam, The Netherlands

Sunday 9th February, 2020

Axel Koolhaas
axel.koolhaas@os3.nl

Woudt van Steenbergen
woudt.vansteenbergen@os3.nl

Supervisor: Danny Kielman
danny.kielman@fox-it.com

***Abstract*—**
Computer systems store data on storage mediums. File systems are utilized to manage this data. File system specifications can enable unsollicated behaviour, such as hiding data within the data structures or the unused data blocks (slack). In 2016 Apple introduced the Apple File System (APFS) as a next-generation default file system for Apple products. In this paper, we present our findings regarding the automated detection of hidden and/or modified data within APFS data structures and slack. We used a Mac mini to create APFS partitions which were researched using a modified file carving tool named AFRO. We exhaustively analyzed previously discovered hiding techniques by Göbel et al., i.e., the superblock slack hiding technque and also the inode pad hiding. We discovered that there are fields containing data patterns in the superblock slack. These offsets are undefined in the official Apple specification at the time of writing. We speculate that this is currently unspecified APFS functionality. We conclude that it is feasible to detect modifications to the inode padding fields and the slack space of APFS datastructures. Specifically, the container superblock, volume superblock, and object map slack.

***Index Terms*—APFS, Apple, Detection, filesystem slack**

I. Introduction

Apple File System (APFS) was introduced on the 14th of June in 2016 as the next-generation default file system for Apple devices with the Apple proprietary operating systems (OS) [1]. It replaced the Hierarchical File System Plus (HFS+) which was in use since 1998 [2]. Apple OSs comprise about 20% of the OS market share as of December 2019 [3]. This makes APFS a worthwhile target for digital forensics, as systems using APFS are wide in use. APFS is a relatively new file system which hasn't been completely analyzed and researched [4]. Additionally, since it is a proprietary file system, the source code is not freely available, which obscures the file system. However, in 2018 Apple released a partial specification of the file system and some research has been done on reverse engineering the architecture. More on this in Section II. Because of the novelty and obscurity, APFS is an interesting subject to research regarding digital forensics. Research has been done on data hiding in APFS, but there is no research as of yet which involves the detection of hidden data. This paper aims to tackle a subset of this subject by researching the possibility of automating the detection of modified APFS data structures and slack space [5].

II. Related work

The pioneering research to understand the APFS file system was done on a pre-release version in macOS 10.12 Sierra. This version is incompatible with the current version, but its primary data structures have not diverged substantially [6]. An APFS partition consists of a single container, which could stretch across multiple partitions like logical volumes [7], and this container has a corresponding superblock (NXSB). The container contains multiple volumes which have their own accompanying superblocks (APSB). Both the container and volume superblock refer to an Object map (OMAP) which contains references to objects. A simplified overview made by Dewald et al. can be seen in Figure 1.

Each volume contains a directory structure for files and folders. The container superblock is stored multiple times, preserving the state of the container at different points in time in the Checkpoint Area. Block zero contains a superblock that is primarily used during mounting to find the most recent checkpoints. The spacemanager keeps track of what is allocated and what is free. Then comes the storage for objects and file data. These are retained in records which are kept track of using a B-Tree [8]. Each object in a B-Tree contains a 64-bit Fletcher checksum for integrity [9].

According to the official Apple documentation [10], APFS stores data little endian and it is conceptually divided into two layers, namely the container layer and the file-system layer. The container layer has 64-bit aligned boundaries and stores volume metadata, snapshots and the encryption state. The container layer has a one to many relation with the file-system layer. This are the container and volumes in Apple nomenclature. Each container contains multiple volumes and only one container exists per partition. The file-system layer is built-up using data structures for, e.g., directory, metadata, and file content. A visual representation of the APFS structure can be seen in Figure 1 [8].

APFS has been treated in the last release of Levin's Apple OS internals book series [11]. There most of the aforementioned information has been aggregated, but no new research/-analysis is presented regarding APFS internals or slackspace.

Within file systems, sequentially ordered events are numbered. This is called timestamping. The timestamping mechanism of APFS has been explored by Song et al. [12]. They

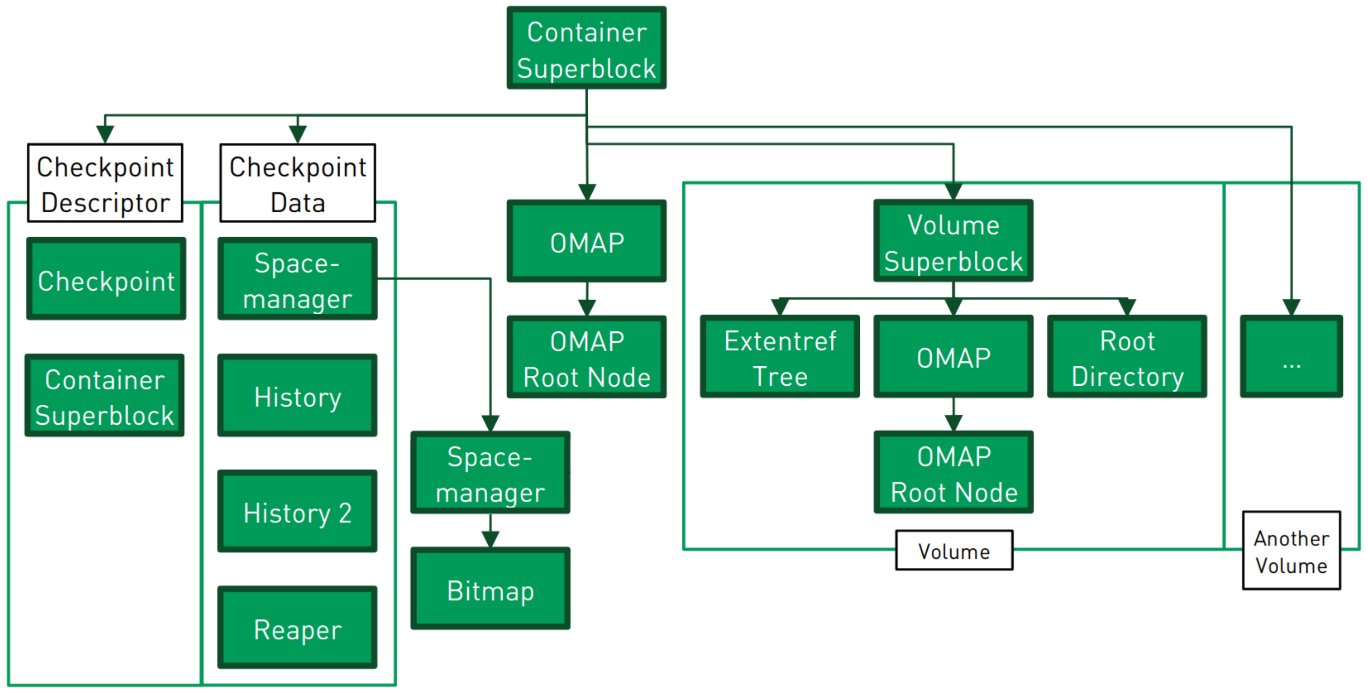

Fig. 1. APFS composition, original from Dewald et al. [8]

documented several operations that caused the system to update the timestamps. In their research, Song et al. observed that APFS behaved differently depending on whether the operations performed on files and directories originated from the terminal or a GUI. This is interesting behavior, since this means that APFS ends up in different states depending on what interface is used.

The data structures in APFS also contain non-critical information for the file system to function. Non-critical information fields can be modified as a technique for data hiding, storing the hidden information within the file system [4]. For example, the inodes [13] which describe files and directories contain four fields with timestamps that are measured in nanoseconds, can be abused to hide small chunks of data. The superblock slack is probably the most interesting data hiding technique when it comes to the size of potential data to be hidden, where the superblocks are used to store data.

Reconstructing lost or deleted files from a file system is an important aspect in forensic computing. This is called carving. There exist tools for carving from file systems internal data structures, e.g., Sleuthkit [14]. Sleuthkit has incomplete functionality for carving APFS. Plum et. Dewald researched different approaches to identify and recover files on APFS [15]. The result of their work has been implemented in a proof of concept called AFRO [16], which derives its name from APFS File RecOvery.

AFRO is a tool written in Python3 that can parse APFS partitions and uses this to carve files. AFRO parses APFS by searching for bytes that correspond with APFS data structures and then creates a fletcher checksum to see if the block is indeed valid. AFRO makes use of Kaitai Struct as a model. Kaitai Struct is a declarative language used to describe binary data structures. A particular format can be described in Kaitai Struct language and can then be compiled with kaitaistructcompiler into source files in one of the supported programming languages, such as Python [17]. This is a clear separation of the model and code, as opposed to working with offsets within the binary data, allowing for cleaner code and better extensibility.

As mentioned by Göbel et al.:

> In addition to improving the framework's ability to hide data, there would be real added value in identifying potentially hidden data. [4]

which brings us to our research and our research question:

> Is it possible to automate the detection of information hidden within APFS datastructures and their slack?

We researched a subset of the hiding techniques that were researched by Göbel et al. Namely, slack space hiding in container superblocks, volume superblocks, and object maps, as well as data hiding in inode pad fields.

III. Methodology

Detecting hidden data within the APFS requires an OS to format drives with the file system, an additional machine to perform a data dump of this unmounted drive, and a tool to parse the image. We used a Mac mini with macOS Catalina 10.15.2. Our system had 1412.61.1 as its APFS driver version. We fetched the APFS driver version using the following command:

```
strings /sbin/fsck_apfs | grep "fsck_apfs ("
```

To detect modifications to APFS data structures, we first had to establish what the data structures would look like under normal circumstances. To do this, we used the open source tool AFRO [16]. We verified that AFRO correctly parses data structures by referencing the Apple APFS Reference [10] and by manually parsing data structures in a hex editor. The original implementation of AFRO contained a non-critical bug which we speculate came as a result of Apple's misleading specification. Namely, the `o_type` field in the object headers is defined as a 32-bit `uint32_t o_type` multipurpose field for `object_type` and `object_type_flags`. The lower 16-bits are used to indicate which object is being described, and the high 16-bits are used to provide information at a higher abstraction layer [10]. However, the examples in the documentation were given in 32-bits which lead to the implementation comparing 16-bit values to 32-bit ones. In addition to parsing the slack, we also implemented a simple hexadecimal view output for AFRO with optional color coding for ease of use.

A. Images

To create an image we formatted a USB flash drive with an APFS partition. A data dump was made of this newly created image before any modifications were made. Next, files and directories were added with dumps taken between each iteration to take a snapshot of the current state of the partition. Dumps were created using `dd` on a GNU/Linux machine. Using a different OS ensured that the APFS partition was not mounted during the dumping process. This is due to the fact that macOS automatically mounts APFS volumes, which may change its internal state.

B. Slack space

Storage blocks on disk often only contain one APFS data structure, with this datastructure not using all the space of a block. The remainder is referred to as slack [5]. For our research we extended the AFRO [16] open source tool to also parse the remainder of a block. We implemented this parsing for container superblocks, volume superblocks and their respective object maps. With AFRO we were able to parse and catalogue the contents of the slack space. More on this in Section IV.

C. Inode pad fields

Node data structures of type inode contain two pad fields for memory alignment. According to the APFS specification these fields should be set to zero on creation, and left unmodified when updating the inode. We modified AFRO to parse all the nodes, and return the contents of the pad fields allowing us to collect information about these fields' contents.

IV. Results

A. Superblock slack

1) Container superblocks: We collected information about the contents of the container superblocks from the multiple images that were created. On analysis of these images we found that the container superblock slack is not empty. Namely, there are three fields which are reoccurring. The first field at offset 0x03DC from the start of the block contains values which differed from all the other container superblocks. In all of our images we did not observe a certain value to be present on more than one superblock at the same time. See Table I for an overview of all the container superblocks on a single image. We can also see that the second observed field has a constant value between the different superblocks. Note that the observed value is also present on the other images that we have analyzed. The third and last field also contains a constant value that does not differ between the container superblocks. All other fields besides the ones mentioned here were observed to contain zero values only. These zero values were observed in each container superblock present on our images.

NXSB block id	**XID**	**Offset 0x03DC**	**Offset 0x0520**	**Offset 0x0568**
0	122	0763	08 0004 0001	050443 125DA440
11561	121	0754	08 0004 0001	050443 125DA440
11563	122	0760	08 0004 0001	050443 125DA440
11565	113	06C4	08 0004 0001	050443 125DA440
11567	114	06D6	08 0004 0001	050443 125DA440
11569	115	06E6	08 0004 0001	050443 125DA440
11571	116	06F6	08 0004 0001	050443 125DA440
11573	117	070C	08 0004 0001	050443 125DA440
11575	118	0720	08 0004 0001	050443 125DA440
11577	119	072F	08 0004 0001	050443 125DA440
11579	120	0741	08 0004 0001	050443 125DA440

TABLE I
The container superblocks found on an image, ordered by their order of occurrence on disk (block id). Included are their version (XID) and the unknown value fields present in the slack space.

2) Volume superblocks: Multiple images were used to collect information of the contents of the volume superblocks their respective slack. In this slack space we observed two fields to contain non zero values. The first field, which exists at offset 0x03D8, was observed to contain a constant value of 0x10. This value was present in all volume superblocks on the same image, as well as those on other images. The second and last field at offset 0x03E0 contained a value which appeared to change between versions, but was not unique when compared to other volume superblocks on the same image. See Table II for an overview of the observed values. All other fields of the volume superblock were zero, this behavior was observed in all volume superblocks regardless of image or superblock version.

APSB block id	XID	Offset 0x03D8	Offset 0x03E0
6	83	10	5200
24	93	10	5C00
37	94	10	5C00
43	95	10	5C00
49	96	10	5C00
65	97	10	5C00
79	104	10	6700
83	98	10	5C00
93	99	10	5C00
131323	53	10	3500

TABLE II
THE VOLUME SUPERBLOCKS FOUND ON AN IMAGE, ORDERED BY THEIR ORDER OF OCCURRENCE ON DISK (BLOCK ID). INCLUDED ARE THE VOLUME SUPERBLOCKS' VERSION (XID), AND THE UNKNOWN VALUE FIELDS PRESENT IN THE SLACK SPACE.

3) Object Maps: We analyzed the slack space of object maps (OMAP). Unlike the container and volume superblock, we observed no unspecified values in this slack space. The slack space of OMAPs contains only zeroes. This applies to OMAPs which were referenced by the container superblock, as well as to the OMAPs which were referenced by the volume superblock.

B. Inode pad fields

The inode pad fields' contents were observed through the use of AFRO. Of all the inode pad fields that were present, we found none which did not have zero values on examination. This applies to all inode pad fields that were present on our images.

V. Discussion

A. Superblock slack

1) Container superblocks: The values that were observed in the container superblock are at an offset from the end of the container superblock data structure, we observed this offset to be constant. The first value was found 0x17C bytes behind the NXSB, this value behaved as a variable of 2 bytes with the values differing between NXSB versions. The other two values found in the NXSB were observed to be constant, at relative offsets 0x2C0 and 0x308. These constant values were observed in images from different machines, which rules out the possibility of user fingerprinting. The function of these bytes will be left up for speculation, or if Apple updates the APFS specification to include these unknown values. Automating detection of data hidden in the NXSB slack space can be defined as a result of our research: first, parse the container superblock up to its slack. If the bytes leading up to offset 0x17C are non zero values, flag the NXSB as suspicious. Ignore the 2 bytes at offset 0x17C. If the values of the bytes between offset 0x17E and 0x2C0 are non zero, also flag the NXSB as suspicious. Compare the bytes at offsets 0x2C0 and 0x308 to the constant values, `0x08 00040001` and `0x00050443 125DA440` respectively, and if they differ then flag the NXSB as suspicious. The remainder of the slack should be zero values under normal circumstances, therefore if the opposite is observed the NXSB should be flagged.

2) Volume superblocks: The values in the volume superblock slack were contiguous to the end of the volume superblock datastructure. This may imply that they are used for new APFS functionality that has not been documented in the public specification at the time of writing. We observed the first value to be a constant, at relative offset 0x0 to the APSB. The second and last value were observed to be variable. The behavior of these fields along with their constant offsets relative to the APSB allow us to define their location in a storage block. Due to this, we reason it will be possible to automate the detection of data hidden within the volume superblock slack. The automated detection process could go as follows: first, parse the volume superblock up to its slack. If the byte at relative offset 0x0 is not equal to `0x10`, flag the APSB as suspicious. Ignore the contents of the byte at relative offset 0x8. The rest of the slack space should contain none other than zero values, if this is not the case the APSB should be flagged as suspicious.

3) OMAP: We observed exclusively zero values in the OMAP slack space. This is regardless of whether the OMAP is used in conjunction with a container superblock, or volume superblock. We can now define a method to detect modifications to the OMAP slack: if the slack space is non zero, flag the OMAP as suspicious.

B. inode pad fields

The pad fields of the inode datastructure potentially provides 10 bytes of hiding space. However, we observed these fields to only contain zero values which means that any data hidden within the pad fields is easily identifiable.

C. Volatility of data structures

APFS implements versioning as a substitute for journaling. Filesystems which implement journalling keep a journal of operations that were performed on the filesystem and committed to the storage device. The journal is a circular log, meaning that old entries are overwritten when the journal has no free entries. APFS has a similar implementation, as there is a limit on the amount of checkpoints/versions it can have. This limit correlates with the size of the container [4]. We observed that old versions are zeroed out instead of overwritten, and the new checkpoint will reside on a different location on disk. This has a direct consequence on hiding techniques which abuse the versioned APFS data structures, among which are the container superblock, volume superblock, OMAP, and inodes. Data that is hidden in one of these data structures will eventually be overwritten if the storage device is mounted as writable on a live machine.

VI. Conclusion

In this paper, we presented techniques to automatically detect hidden data within APFS data structures, namely: the container and volume superblock slack, their accompanying OMAP slack, and of the inode padding fields. Irregularities are trivial to identify once data structures have been correctly parsed. However, we encountered unspecified fields in superblock slack that are being used by APFS, potentially being new, undocumented APFS functionality. We also discovered that when APFS partitions are mounted, old blocks are quickly discarded, making the superblock slack hiding technique volatile. Revisiting our research question: Is it possible to automate the detection of information hidden within APFS datastructures and their slack? The answer to that is yes, when taking the unspecified fields into consideration it will be feasible to detect modifications to APFS data structures and their slack.

We extended AFRO, a tool to carve files from APFS. Our modifications to AFRO will be published after our research [16].

VII. Future work

In the future it would be worthwhile to analyze the values in the superblock slack and other data structures. This could be done by attaching a debugger to Apple APFS drivers and see when these offsets are loaded into memory. The source code of the drivers is however, at the time of writing, not available. This would make it difficult to see what happens since the drivers needs to be (partially) reverse engineered.

It would also be interesting to see if it is possible to hide files that are detached from the file systems using the Spacemanager bitmap, e.g., block aggregation abuse for write protection or removing inode entries from their tree, erasing the file index.

APFS drivers of different platforms have not been compared. Future work could entail the researching of potential differences between APFS drivers of different OSs, e.g., macOS vs. iOS.

References

[1] Nob Hill. Introducing apple file system, June 2016.

[2] Apple Inc. Technical note tn1121. http://developer.apple.com/technotes/tn/tn1121.html, 1998.

[3] StatCounter. Operating system market share worldwide. https://gs.statcounter.com/os-market-share, 2020.

[4] Thomas Göbel, Jan Türr, and Harald Baier. Revisiting data hiding techniques for apple file system. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, page 41. ACM, 2019.

[5] Ewa Huebner, Derek Bem, and Cheong Kai Wee. Data hiding in the ntfs file system. *digital investigation*, 3(4):211–226, 2006.

[6] Kurt H Hansen and Fergus Toolan. Decoding the apfs file system. *Digital Investigation*, 22:107–132, 2017.

[7] Leon Daniel Baranovsky, Luis Felipe Cabrera, Chiehshow Chin, and Robert Rees. Logical volume manager and method having enhanced update capability with dynamic allocation of storage and minimal storage of metadata information, April 27 1999. US Patent 5,897,661.

[8] Andreas Dewald. Ernw whitepaper 65 apfs internals for forensic analysis. https://static.ernw.de/whitepaper/ERNW_Whitepaper65_APFS-forensics_signed.pdf, 2018.

[9] John Fletcher. An arithmetic checksum for serial transmissions. *IEEE transactions on Communications*, 30(1):247–252, 1982.

[10] Apple Inc. Apple file system reference. https://developer.apple.com/support/downloads/Apple-File-System-Reference.pdf, 2019.

[11] Jonathan Levin. **OS Internals, Volume II Kernel Internals*. John Wiley & Sons, 2019.

[12] Jong-Hwa Song, Se Ho Kim, Song Yi Hwang, Seung Gyu Kim, and Sung-Jin Lee. A study on the apfs timestamps in macos. *International Journal of Engineering & Technology*, 7(2.33):133–138, 2018.

[13] Marshall K McKusick, William N Joy, Samuel J Leffler, and Robert S Fabry. A fast file system for unix. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, 1984.

[14] Brian Carrier. The sleuth kit. *TSK–sleuthkit. org*, 2011.

[15] Jonas Plum and Andreas Dewald. Forensic apfs file recovery. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, pages 1–10, 2018.

[16] Jonas Plum and Andreas Dewald. Afro. https://github.com/cugu/afro, 2018.

[17] Kaitai Project. Kaitai struct. http://kaitai.io/, 2020.