Reverse Engineering Virtual Machine File System 6 (VMFS 6)

Michael Smith

Reverse Engineering Virtual Machine File System 6 (VMFS 6)

GIAC (GCFA) Gold Certification

Author: Michael Edward Smith, mesmith1@protonmail.com
Advisor: David Hoelzer

Accepted: September 30, 2020

Abstract

Virtual Machine File System (VMFS) 6 is a proprietary file system. The file system's proprietary nature means that many forensic applications are unable to parse the file system. There is a lack of support because proprietary file systems do not have to follow an accepted standard and can make modifications that break forensic tools with any release. This instability means that maintaining parsers for these file systems can become costly very quickly. This vacuum of support for proprietary file systems has created an opportunity for open-source utilities to grow in ways that support parsing these file systems. Skilled forensic examiners scour the open-source community and publicly available research for parsers and digital artifacts analyses when they encounter file systems or files unsupported by large forensic applications. The goal of this research is two-fold. First, to increase the understanding of VMFS 6 with its myriad digital artifacts. Second, to conclusively determine the recoverability of a deleted file.

1. Introduction

Reverse engineering analyzes a subject to identify components and interrelationships and represent the system in another form (Chikofsky, 1990). Redocumentation and design recovery are two sub-disciplines of reverse engineering, according to Chikofsky. Redocumentation is a non-intrusive process of representing the processes of a system in alternate views. According to Biggerstaff, cited by Chikofsky: "Design recovery recreates design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains."

In the context of reverse engineering sub-disciplines: this research focuses on design recovery. Redocumentation requires the existing source code and is more interested in newly representing code elements. Meanwhile, design recovery includes analyzing multiple unofficial open-source utilities and official documentation, such as release notes and capability documentation.

Chikofsky also mentions two other software life-cycle phases on the same level as reverse engineering: restructuring and reengineering. The research conducted in this paper represents restructuring by adapting the source code of vmfs-tools and its forked repositories from C++ to Python. This project utilizes reengineering by comparing the source code of the open-source utilities and the VMFS 6 file system's raw data.

This paper hypothesizes that reverse engineering VMFS 6 will identify new metadata useful to security researchers and forensic analysts in filling information gaps and determine if file recovery with metadata is possible.

1.1. Virtual Machine File System (VMFS)

This project seeks to analyze the virtual machine file system. The acronym VMFS has a confusing history because the first two releases of VMFS used by VMware stood for VMware File System and did not have the same level of capability (Henry, 2012). The Virtual Machine File System was a significant advancement for VMFS and debuted as VMFS 3 in March 2009. Satyam Vaghani, the new virtual machine file system architect, published a paper on the new filesystem in 2010. This paper, published by

Vaghani was beneficial during the VMFS file system's reverse engineering even after two major releases. It significantly contextualized certain variables, such as copy-on-write (cow) and zeroed (tbz), among others.

1.2. File Recovery

There are two forms of file recovery for a file deleted on a disk: carving and metadata.

Carving is a straightforward and costly endeavor. It is the equivalent of panning for gold, except the dirt and sand are bytes, and the gold is a file signature. Just like panning for gold, false positives are extremely likely in file carving. File carving has two significant weaknesses: the lack of metadata and the inability to recover fragmented files.

The act of recovering a deleted file via file system metadata addresses both of the weaknesses of carving. A prerequisite of this style of recovery is file system metadata. File system metadata provides information to recover a file as well as contextualize it. Contextual information includes file names, timestamps, file type, file size, file owner, and access control. In VMFS, the file system is not user-oriented, but vSphere-oriented. As a result, the/a file owner is not present, and the closest thing to access control is host heartbeats and file locks. The file metadata stored by a file system that is leveraged to recover a file is its record of file allocation. The file system maintains this allocation data so that even if a file does become fragmented, the file system can present the user with the entire file. If file deletion preserves this allocation metadata, a forensic tool can leverage it to recover fragmented files.

2. Research Method

2.1. Generation of Research data

The methodology employed consists of two approaches: manipulating files on a partition to perform a comparative analysis of the changes to the file metadata and generating partitions of varying sizes to perform a comparative analysis of the VMFS file system dynamic bitmap sizing.

Michael Smith, mesmith1@protonmail.com

2.1.1. File Fragmentation and Deletion

The first exercise is to test the feasibility of file recovery. This test uses a standalone machine, a thumb drive, and ESXi 7. ESXi 7 is installed on the thumb drive to isolate the boot partition from the VMFS partition. In most production environments, architects separate boot disks of hosts from storage arrays.

As discussed in 1.2, one of the features of metadata-based file recovery is handling fragmented files. Thus, the first scenario requires a fragmented file. This fragmentation is achieved by creating a thin-provisioned virtual machine (VM1), creating a second virtual machine (VM2), then expanding VM1 by adding data. This method will result in VM1 having data provisioned before and after VM2 on the disk. Finally, to test file recovery, VM1 is deleted. After each step in this process, the host is powered off, and a forensic image is acquired (E01) using Guymager distributed with Kali Linux 2020.2.

2.1.2. Maximal and Minimal File Systems

The second test involves the dynamic provisioning of VMFS file system metadata to scale with varying size partitions. As a result, the corpus includes two additional volumes created with a virtualized ESXi host using VMware Fusion 11.5. The smallest partition tested is 2GB, and the largest is 8TB.

2.2. Examination of Open-Source Utilities

This research includes an analysis of open-source utilities potentially associated with VMFS parsing. The study concludes that none of the open-source parsers supported file recovery from VMFS 6.

2.2.1. QEMU

The scope of QEMU does not encompass the VMFS file system. QEMU imports VMware VMs by interpreting VMX files and VMDKs.

2.2.2. Open Source VMFS Driver

The open source VMFS driver supports VMFS 3, is developed and maintained by fluid Operations, and was last updated in 2010 (FluidOps, 2010).

2.2.3. VMFS-TOOLS

Mike Hommey loosely based a VMFS file system parser on the fluid Operations version. Hommey's parser, vmfs-tools, has limited support for VMFS 5 and was last updated in 2016 (Hommey, 2016).

2.2.4. VMFS6-TOOLS

Weafon Tsao forked Hommey's project to create his derivative parser named vmfs6-tools. Tsao's parser provides limited support to VMFS 6 (Tsao, 2019).

2.3. Analysis Method

The analysis performed during this study will proceed as follows. Convert the open-source vmfs-tools by Mike Hommey to Python. During this conversion, verify and validate vmfs-tool's assumptions and document additional information or deviances found. Identify new data structures or fields where possible. Additionally, determine the feasibility of recovering a file with file system metadata.

3. VMFS 6 Data Types and Structures

VMFS 6 uses five primary data types: uint32, uint64, UUID, u_char, and char.

Data Type	Data Type Representation	Bytes	Bits
Unsigned Character	u_char	1	8
Unsigned Integer	uint32	4	32
Unsigned Integer	uint64	8	64
Universally Unique Identifier	UUID	16	128
Character	char	1	8

Figure 1. VMFS 6 Common Data Types

While only using five data types, the file system can support many interpretations of the data. For example, in VMFS 6, some uint64 values are Linux epoch timestamps in microseconds, while some uint32 values are Linux epoch timestamps in seconds.

VMFS Block IDs are not simply unsigned integers, but rather a combination of non-byte-based values. Non-byte-based values are incapable of explicit representation in 8-bit chunks. For example, a simple flag to indicate true or false only needs one bit: 0, for false, and 1 for true. As a result, it is not uncommon for data structures to pack eight flags

into a single byte (8-bits) to conserve space. A technique known as bit masking effectively parses non-byte-based data.

3.1. Position Field

The position field is frequently referred to in vmfs-tools as "pos." A forensic analyst's first impression upon seeing a position data field is that it would contain the offset to the position on the disk. Alternatively, the position can be from the beginning of the VMFS volume. The position field implemented by VMFS 6 presents a bit of a challenge. Unlike initial assumptions, it represents the offset from the 17th block of the volume.

Analysis of heartbeats and file descriptor entries revealed this 17MB offset. The first populated heartbeat entry contains a pos value of 4,063,232 bytes. The first file descriptor bitmap entry contains a hb_pos value of 4,063,232. This correlation of the pos in the heartbeat and the file descriptor hb_pos makes this a strong value to determine position value calculations. The beginning of the heartbeat data structure is at position 4,063,232. The disk offset of the heartbeat is 22,937,604 bytes. One would hypothesize that the static block offset to apply to pos values is 18 blocks, 22 - 4. If we do the math:

22937604 – (18 * 1048576) = 4063236

Calculating the file descriptors position further confirms this hypothesis. The file descriptor pos is 7,405,568, and the disk offset is 26279936, performing the same formula as before:

26279936 – (18 * 1048576) = 7405568

To find the disk offset from the pos value, add 18 blocks. A more reliable value, especially to define a static variable, is to determine the blocks from the beginning of the VMFS 6 volume. This static variable is more trustworthy since a VMFS 6 volume position on a disk is not immutable. In test cases for this research, the VMFS 6 volume consistently starts one block into the disk (see Section 4.1). As a result, the static position modifier is 17 blocks from the beginning of the VMFS 6 partition.

4. VMFS 6 Volume Analysis

The most recent open-source utilities for parsing VMFS 5 and 6 volumes are written and available on GitHub in the C programming language. As a result, these codebases were re-written in Python to facilitate experimentation and comprehension of the parsed data structures.

4.1. GUID Partition Table (GPT)

According to VMware's VMFS Datastore documentation (Svoskobo, n.d.), the adoption of GPT for VMFS began in VMFS 5.

Mike Hommey's vmfs-tools only supports the Master Boot Record (MBR) DOS Partition Table format. Neither Mike Hommey's vmfs-tools nor Weafon Tsao's vmfs6-tools accounted for the transition of VMFS 5 and 6 to the GPT format.

This reliance on MBR explains why the observed offset for the file signature for VMFS Volume Information does not correspond with the code implemented in vmfs-tools. The code in vmfs-tools references the start LBA for the partition presented by the MBR to find the VMFS Volume's beginning. When using GPT, the format replaces the MBR with a Protective MBR. The protective MBR contains a start LBA that references the beginning of the GPT, not the partition (Carrier, 2005, p. 140). The GPT is then analyzed to find the beginning of the VMFS volume. The result of vmfs-tools reliance on MBR is that the VMFS volume is mistakenly understood by the program to start at the GUID Partition Table.

When converting a VMFS 3 volume to VMFS 5, MBR is maintained instead of converting the partition table to GPT (Svoskobo, n.d.). This conversion makes vmfs-tools functional on some VMFS 5 volumes while not supporting GPT. Vmfs6-tools compensated for this by modifying the static base variable for the VMFS volume to bypass the GPT.

The lack of GPT support led to writing a custom Python GPT parser leveraging the data structures outlined in Brian Carrier's "File System Forensic Analysis." GPT format classifies partitions by referencing GUIDs for specific partition types. The analysis revealed the GPT partition type for VMFS in Figure 2 (Hogan, 2011).

GUID Type Value	Description
AA31E02A-400F-11DB-9590-000C2911D1B8	VMFS

Figure 2. VMFS GPT Partition Type GUID

4.2. VMFS Volume Information – Block 2

Since the block size is static as of VMFS 5 and VMFS 6, the section headers include the block number to clarify how the VMFS partition is structured. The VMFS Volume information is in Block 2, and the first block of the volume is empty. A null span of data is not uncommon in filesystems. File systems include this unused disk area to protect the rest of the volume.

The VMFS volume data structure analysis revealed some unverifiable information and a new field that interferes with quite a few of the structures documented in vmfs-tools. The LUN number is no longer correct, nor is the size field valid. There is also a new field called UUID in the host interface (Figure 3).

This UUID value is present in the VMFS Volume Information in the byte-range 18-83. The byte-range occupied by the new UUID effectively invalidates the name and _unknown2 fields (Figure 4).

```
051a636627f1863abf1914f33058d73c1846fc0b3dc2607ed6834df38bd1ed4f40
```

Figure 3. UUID Value captured from the ESXi host

Byte Range	Structure	Name	Valid
0-3	uint32	magic	Yes
4-7	uint32	ver	Yes
8-13	u_char	_unknown0[6]	Unconfirmed
14-14	u_char	lun	No
15-17	u_char	_unknown0[3]	Unconfirmed
18-45	char	name[28]	No
46-94	u_char	_unknown2[49]	No
*18-83	uuid66	UUID	Yes
95-98	uint32	size	No
99-129	u_char	_unknown3[31]	Unconfirmed
130-145	uuid	uuid	Mostly
146-153	uint64	ctime	Yes
154-161	uint64	mtime	Yes

Figure 4. Validation of packed structure of volume from vmfs-tools

4.2.1. Logical Volume Manager (LVM) Information – Block 2

The observations of this research conclude the LVM information data structure to be reliable (Figure 5). The handful of entries that remain unconfirmed are not a reflection on the data structure, but rather the inability to find a means of confirming the value promptly.

Byte Range	Structure	Name	Valid
0-7	uint64	size	Yes
8-15	uint64	blocks	Yes
16-19	uint32	_unknown0	Unconfirmed
20-54	char	uuid_str[35]	Mostly
55-83	u_char	_unknown1[29]	Null
84-99	uuid	uuid	Mostly
100-103	uint32	_unknown2	Unconfirmed
104-111	uint64	ctime	Yes
112-115	uint32	_unknown3	Null
116-119	uint32	num_segments	Yes
120-123	uint32	first_segment	Yes
124-127	uint32	_unknown4	Null
128-131	uint32	last_segment	Yes
132-135	uint32	_unknown5	Null
136-143	uint64	mtime	Yes
144-147	uint32	num_extents	Yes

Figure 5. Validation of packed structure of LVM from vmfs-tools

4.2.2. Device Name – Block 2

The device name is a single field at cluster offset 384, byte 0. Vmfs-tools and vmfs6-tools do not parse this field. The data contains the device name, as seen in the ESXi host in ASCII.

4.3. VMFS File System – Block 20

The structure of the VMFS 6 File System Metadata remains mostly consistent with the structure in VMFS 5, as presented in Mike Hommey's parser vmfs-tools. The reported offsets in Tsao's vmfs6-tools are off due to VMFS 5 and 6's implementation of the GUID Partition Table (refer to Section 4.1).

Byte Range	Structure	Name	Valid	Alternative
0-3	uint32	magic	Yes	
4-7	uint32	volver	Questionable	Unknown
8	u_char	ver	Questionable	minor_ver
9-24	uuid	uuid	Yes	
25-28	uint32	mode	No	major_ver
29-156	char	label[128]	Yes	
157-160	uint32	dev_blocksize	Yes	
161-168	uint64	blocksize	Yes	
169-172	uint32	ctime	Yes	
173-176	uint32	unknown3	Questionable	Mode?
177-192	uuid	lvm_uuid	Yes	
193-208	u_char	unknown4[16]	Questionable	Mode?
209-212	uint32	fdc_header_size	Yes	
213-216	uint32	fdc_bitmap_count	Yes	
217-220	uint32	subblock_size	Yes	
221-236	unknown		N/A	
237-347	unknown		N/A	

Figure 6. VMFS 6 File System changes

Regardless of the size of the volume, most of the values for the metadata remain the same. Given the test samples of a 2GB, 232GB, and 8TB VMFS Volumes, only the values for UUIDs, timestamps, and label change.

Some of the structured data changed from VMFS 5 to 6. For example, Concatenation of the vmfs-tools values for mode, 6, and version, 82, yields 6.82. The VMFS version tested is 6.82. This confirmation that these two values make the version

number means the mode field represents the major version and the version field represents the minor version.

Variable unknown3 or unknown4 must now represent the mode field. According to vmfs-tools, the valid options for mode are 0 for private, 1 and 3 for shared, 2 for the public. The only variables with legitimate values in all the test volumes are unknown3, and the first 32 bits of unknown4.

Bytes 221-236 are undocumented in vmfs-tools; however, the 16 bytes are static across all the test volumes. The additional bytes in 247-347 vary between the different volumes and require further analysis.

4.3.1. Block Size

This research found that the VMFS 6 Volume information metadata confirms that the file system uses a block size of 1MB. The 1MB block size is standard since the release of VMFS 5 (Svoskobo, n.d.). VMFS 3 offered four different block sizes to compensate for using the MBR partition schema: 1, 2, 4, and 8 MB. However, the ability to vary block size among datastores introduced problems and incompatibilities among some of VMware's products (AndreTheGiant, 2011).

4.3.2. Sub-Block Size

The metadata in the VMFS Volume information confirms that the file system uses a sub-block size of 64KB. VMFS 5 previously used a sub-block size of 8KB. VMFS 3 used the same sub-block size as VMFS 6, 64KB (Hogan, 2017).

4.4. VMFS Heartbeats – Blocks 21-24

The heartbeats in VMFS are a method of maintaining and validating a host and storage device's connectivity and availability. The heartbeat data structure is in Section 6.1.

Vmfs-tools has a static variable for the VMFS heartbeats base offset as three 1MB Blocks. This base offset did not help find the heartbeats. As a result, the initial analysis consisted of scanning the volume in 512-byte sectors to determine the location and number of heartbeats present in the VMFS 6 file system. Assisting this low-level scan is

the fact that VMFS has reliable heartbeat file signatures (Figure 7). This scan revealed that VMFS 6 has 1024 heartbeats that are 4KB in size. The heartbeats are present in the volume as a contiguous data structure consuming four total 1MB blocks from block 21 to block 24.

Signature (magic)	Signature Meaning
0xABCDEF01	Heartbeat Off
0xABCDEF02	Heartbeat On

Figure 7. VMFS Heartbeat File Signatures

4.5. VMFS Bitmaps – Block 25

VMFS 5 has four bitmaps: file descriptors, file blocks, sub-blocks, and pointer blocks. VMFS 6 added a fifth bitmap, pointer block 2.

The size of the bitmaps in VMFS is dependent on the size of the VMFS volume. This variability in size is due to the bitmap representing the consumed and available blocks in the disk data section. As a result, a larger disk with more available blocks will necessitate a larger bitmap to address that space.

This variability in bitmap size means each bitmap header defines the scope of the volume's bitmap.

4.6. File Descriptor Cluster System File (fdc.sf) – Block 25

The file descriptor bitmap contains inodes (Hommey, 2012). These inodes contain metadata associated with files and directories. Section 5.4 documents the data structure for inodes.

The beginning of every bitmap contains the bitmap header. The bitmap header defines the organization of bitmap elements and how much space they consume. Section 5.2 describes the data structure for the bitmap header.

The FDC bitmap header has the same values in the 2GB, 250GB, and 8TB VMFS volumes. The FDC bitmap's fixed-size nature makes sense due to the file descriptor inode's independence from the disk size. For other bitmaps, this will not be the case.

One significant challenge to parsing the very first FDC bitmap area is that the bitmaps are files. VMFS needs the heartbeat, file descriptor, inode, directory entry, and block bitmap to quantify a file.

At this point in block 25, the VMFS file system only has the heartbeat, file descriptor, and inode. A comprehensive metadata structure of a file needs the directory entry and the block bitmap as well. As a result, the initial blocks of the VMFS file system are in the following order: the first block of the FDC (Block 25), the Root Directory (Block 26), and the first block of the File Block Bitmap (FBB) (Block 27).

Data Block Type	File System Block	Data Block Size (Blocks)
Initial File Descriptor Bitmap	25	>16
Root Directory	26	unknown
Initial File Block Bitmap	27	>1

Figure 8. Representation of the Initial Bitmap Fragmentation

This positioning of the FDC, Root Directory and FBB Block is recognized as fragmentation because each area of the FDC bitmap is 16 blocks and one sub-block. This fragmentation means the initial FDC bitmap has been split by two blocks to accommodate the Root Directory's first block and the File Block bitmap's first block.

4.7. Root Directory – Block 26

The root directory contains essential information for all the bitmaps and the critical file system files (Figure 10). The directory block contains directory entries see Section 5.6. The discovery of the Root Directory in block 26 involved creating a file in the root directory and writing a block scanner to find the file name within the bytes. For this research, the aptly named file is "Test Directory." Manual analysis of the raw hex identified that the new directory entry size is 288, confirmed by validating with code. At offset 953, we find the Linux symbolic link entries for the current directory and parent directory (Figure 9).

Bytes	Symbolic Link	Block ID	Record ID	Type
952-1239	. (current dir)	4	1	2
1240-1527	.. (parent dir)	4	1	2

Figure 9. VMFS Root Directory Symbolic Links

This directory is the root directory because it is the parent of itself according to the symbolic links (Figure 10). Oddly, the directory structure separates the symbolic links from the children's directory entries. The symbolic links are present in the first cluster (4096 bytes) of the block. However, the child directory entries do not begin until 3008 bytes into the second sub-block at offset 68544. Not all entries have valid names; as a result, the entries start at 69696.

Bytes	File Name	Block ID	Record ID	Type
69696-69983	.fbb.sf	4194308	1	5
69984-70271	.fdc.sf	8388612	1	5
70272-70559	.pbc.sf	12582916	1	5
70560-70847	.sbc.sf	16777220	1	5
70848-70559	.vh.sf	20971524	1	5
71136-71423	.pb2.sf	25165828	1	5
71424-71711	.sdd.sf	29360132	1	5
71712-71999	.jbc.sf	33554436	1	5
72000-72287	ISOs	37748740	1	2
72288-72575	Test Directory	41943044	2	2

Figure 10. VMFS Root Directory Children of the 2GB VMFS 6 Volume

There are some familiar-looking file names when comparing the system files (.sf) to the bitmaps identified in vmfs-tools and vmfs6-tools (Figure 11).

File Name	Bitmap Name	Signature (Magic)
.fbb.sf	File Block Bitmap	0x10C00002 (Hommey)
.fdc.sf	File Descriptor Cluster	0x10C00005 (Hommey)
.pbc.sf	Pointer Block Cluster	0x10C00004 (Hommey)
.sbc.sf	Sub-Block Cluster	0x10C00003 (Hommey)
.vh.sf	Volume Header ?	N/A
.pb2.sf	Pointer Block 2	0x10C00006 (Tsao)
.sdd.sf	Unknown	0x10C00008 ?
.jbc.sf	Jumbo Block Cluster ?	0x10C00007 ?

Figure 11. VMFS Root Directory Children of the 2GB VMFS 6 Volume

Sector scanning of all three volumes revealed two new signatures incremented on the signatures used by inodes and the bitmap entries. Signature 0x10C00007 appears in the VMFS 6 volume similar to a traditional bitmap signature. This pattern of behavior means that it is most likely associated with ".jbc.sf". The "jbc" naming convention more closely resembles the Block Cluster (bc) found in ".pbc" and ".sbc".

The presence of signature 0x10C00008 appears more sporadically in the 8TB file system. Since ".sdd.sf" does not immediately correlate to any other system files, it is most likely associated with it.

4.7.1. .vh.sf System File

A comment on the VMware community forums by username "continuum" clarifies some details specific to the system file ".vh.sf" (Continuum, 2020). The post states that the ".vh.sf" file is located at block 17 and consumes 7MB of data according to vmkfstools, a VMware utility. This tool output means that "vh" stands for volume header and contains the volume filesystem information and the heartbeat blocks. The volume header starting at block 17 also fits nicely with the position values within VMFS 6 initializing at block 17 (Section 3.1).

4.8. File Block Bitmap – Block 27

The file block bitmap is the first variable-sized bitmap encountered thus far. The file descriptor bitmap remained the same size for volumes 2GB to 8TB in size.

Name	2GB Volume	250GB Volume	8TB Volume
items_per_bitmap_entry	16	16	16
bmp_entries_per_area	8	8	8
hdr_size	65536	65536	65536
data_size	8192	8192	8192
area_size	1114112	1114112	1114112
total_items	4	466	16384
area_count	1	4	128
_unknown0	0	0	0
_unknown1	1919315300	1919315300	1919315300
_unknown2	1	1	1
_unknown3	53	53	53
_unknown4	3408712	3408712	3408712
_unknown5	3	3	3
Size (from LVM_INFO)	1879048192	249913409536	8795824586752

Figure 12. Comparison of File Block Bitmap Headers

Regardless of the volume size, up to 8TBs, the file block bitmap's basic structure does not change. The goal of a dynamic bitmap is scalability, not that the fundamental architecture changes with each deployment.

The dynamic nature of VMFS bitmaps manifests itself in the changing total items value across the differently sized volumes. Section 5.2 validated the file descriptor bitmap header by multiplying the area_count, bmp_entries_per_area, and items_per_bitmap_entry to find the total_items. This test fails on the file block bitmap headers.

Name	2GB Volume	250GB Volume	8TB Volume
items_per_bitmap_entry	16	16	16
bmp_entries_per_area	8	8	8
area_count	1	4	128
Calculated total_items	128	512	16384
Bitmap header total_items	4	466	16384
Size / total_items	448MB	511.45MB	511.98MB

Figure 13. Test of File Block Bitmap Total Items

The test fails because the total_items field reflects the items needed to address the volume's disk space, not maximum potential items. The clear trend towards 512MBs of disk handled per item which demonstrates how the bitmap increases in size to accommodate the additional disk space.

5. VMFS 6 Data Structures

5.1. VMFS Heartbeat Data Structure

The research environment for this project makes validation of the data structure of the heartbeat entries difficult. This project created the volumes for this research in a home lab with only one ESXi host active at a time. As a result, this research observed minimal usage of heartbeats.

Analysis of the heartbeat data structure did identify one new development in the information provided within the heartbeat metadata—the presence of the IPv4 address of the host.

Byte Range	Structure	Name	Valid	Alternative
0-3	uint32	magic	Yes	
4-11	uint64	pos	Questionable	4063232,
12-19	uint64	seq	Questionable	
20-27	uint64	uptime	Questionable	
28-43	uuid	uuid	Questionable	
44-47	uint32	journal_block	Questionable	
48-51	uint32	vol_version	Questionable	
52-55	uint32	version	No	Replaced with host ipv4 address
+54-68	u_char	host_ipv4_address	New	
69-4096	null			

Figure 14. VMFS Heartbeat Data Structure

In addition to this research occurring in a home lab, this project did not deploy vSphere to interact with the VMFS volumes. This caveat is important because all VMFS volumes generated during this project have heartbeat entries with null UUIDs. In an

environment with vSphere implemented, one would expect the UUID fields to be populated.

The value populated by the vol_version, 24, is consistent across all VMFS volumes created and also consistent with the value reported in the VMFS file system information as volver. More research is needed to determine what 24 in this context indicates.

The newly discovered field by this research of the heartbeat data structure is the host IPv4 address's presence. This value consumes half of the vmfs-tool's version field. The IPv4 address value is the character representation of the IPv4 and not the packed binary that would only need 4-bytes.

The pos value for the first heartbeat is the same across the 2GiB, 250GiB, and 8TiB volumes. However, the seq value varies, which concludes that the value is no longer reliable or has a more nuanced meaning than the sequence number of heartbeats.

The uptime value does not directly correlate to the uptime of the host. The host was powered off and powered on to renew this uptime value, and the value was not reset.

The journal_block value was zero in all of the created volumes.

5.2. VMFS Bitmap Header Data Structure

The bitmap header appears at the beginning of every bitmap and defines the provisioning of the bitmap.

The initial structure of the bitmap headers has not changed since VMFS 5. However, this project has observed some new unparsed data.

Byte Range	Structure	Name	Valid	fdc_value
0-3	uint32	items_per_bitmap_entry	Yes	256
4-7	uint32	bmp_entries_per_area	Yes	8
8-11	uint32	hdr_size	Yes	65536
12-15	uint32	data_size	Yes	8192
16-19	uint32	area_size	Yes	16842752
20-23	uint32	total_items	Yes	16384
24-27	uint32	area_count	Yes	8
28-31	uint32	unknown0	New	0
32-35	uint32	unknown1	New	1919315300
36-39	uint32	unknown2	New	1
40-43	uint32	unknown3	New	4
44-47	uint32	unknown4	New	65792
48-51	uint32	unknown5	New	1

Figure 15. VMFS Bitmap Header Data Structure

Fortunately, validating the bitmap header's initial structure is straightforward, with some calculations (Figure 15). The analysis takes the static values found in the file descriptor cluster (FDC) bitmap to validate the fundamental structure. The total_items in the FDC is 16,384. Multiplying the items_per_bitmap_entry (256), bmp_entries_per_area (8), and area_count (8) yields the same value as the total items. This calculation validates four of the seven fields.

Calculation of the area_size using total_items, data_size, and hdr_size validates the remaining three fields. The value for area_size is 16,842,752. The area_size is not the entire FDC bitmap; rather, it reflects the size of one area, which there are 8, according to area_count. Additionally, each bitmap area has one header. The area_size is confirmed by adding the hdr_size (65,536) to the multiplication of bmp_entries_per_area (8), items_per_bitmap_entry (256), and data_size (8,192):

65536 + (8 * 256 * 81922) = 16,842,752

After validating the initial data structures, the bitmap contains additional unparsed data. Figure 15 presents this unparsed data in 32-bit chunks. The unknown1 value yields "dmfr" when cast as an ASCII string. In the sense of internet queries and documentation scanning, open source research found no feature or utility named "dmfr". The additional data structures are static in all three test scenarios: 2GB, 250GB, and 8TB.

Michael Smith, mesmith1@protonmail.com

5.3. VMFS Bitmap Entry Data Structure

The bitmap entry is a structure that consumes the data_size as indicated in the bitmap header, which is consistently 8,192 bytes. The first half of the bitmap entry is the metadata header discussed in Section 5.9, while this section only discusses the second half. The bitmap entries are present to identify the allocation of data in the volume.

Byte Range	Structure	Name	Valid
0-3	uint32	id	No
4-7	uint32	total	No
8-11	uint32	free	No
12-15	uint32	ffree	No
16-*	uint8*	bitmap	Unknown

Figure 16. VMFS Bitmap Entry Data Structure

The bitmap entry data structure has some severe issues. In all of the bitmap entries parsed in researching VMFS 6, none of the stated data structures in vmfs-tools are populated. They are all null bytes, with the single exception of the bitmap. The bitmap contains data, but it is unclear if the information is valid.

5.4. VMFS Inode Data Structure

Like the bitmap entry, the inode consumes the data_size, as indicated in the bitmap header, which is consistently 8,192 bytes. This data_size includes 4096 bytes for the metadata header and 4096 bytes for the inode metadata. The previous VMFS Inode size was 2048-bytes, according to Mike Hommey's vmfs-tools. The switch to 4096-byte metadata structures in VMFS 6 makes sense due to VMware's stated intent to transition to 4K native eventually.

The inode metadata is critical to the file system's functioning and for understanding a given file's disk footprint. The inode data is the first element of metadata, where the average user will be familiar with the data captured. It includes data fields like file size and modified, accessed, and created (MAC) times. For file recovery, inodes contain data regarding the outlay of the data in the volume. Inode metadata tells the user how many blocks a file consumes and includes a blocks field with block ids for

the individual blocks the file uses in the volume. If this data is intact after deletion, it will make the recovery of the file significantly easier.

The inode data structure broadly still aligns with the open-source vmfs-tools and vmfs6-tools parsers.

Byte Range	Structure	Name	Valid
0-3	uint32	id	Yes
4-11	uint32	id2	Yes
8-19	uint32	nlink	Unknown
12-27	uint32	type	Yes
16-35	uint32	flags	Unknown
20-39	uint64	size	Yes
28-55	uint64	blk_size	Yes
36-63	uint64	blk_count	Yes
44-59	uint32	mtime	Yes
48	uint32	ctime	Yes
52	uint32	atime	Yes
56	uint32	uid	Unknown
60	uint32	gid	Unknown
64	uint32	mode	Unknown
68	uint32	zla	Unknown
72	uint32	tbz	Yes
76	uint32	cow	Yes
80-511	u_char	unknown	Unknown
512-	uint32 array	blocks[]	Yes
512-515	uint32	rdm_id	Unknown
512-?	char	content	Yes

Figure 17. VMFS Inode Data Structure

5.4.1. VMFS Inode User Friendly Metadata

The data that represents information that is easily verifiable by a user is discussed in this section. These fields include the file type (type), file size (size), modified time (mtime), created time (ctime), and accessed time (atime) values. Fortunately, these data structures are still valid.

5.4.2. VMFS Inode File Handling Data

This subsection discusses the inode metadata fields that handle how a file system should interact with the file. These fields include the copy-on-write (cow) and "to be zeroed (tbz).

The copy-on-write (cow) field tells the file system if any write occurs against this block in the file, the file system copies the block to a new block, and subsequent writes are applied to the new block. This cow functionality is the essence of how snapshots are performed (Vaghani, 2010, p. 51).

The "to be zeroed" value indicates the file is pre-allocated, but not zeroed out. This tbz value means that if the file system performs a read operation on a non-zeroed area of the file, the read operation returns zeroes instead of reading the disk (Vaghani, 2010, p. 51).

5.4.3. VMFS Inode File Distribution Metadata

This section includes the metadata for file allocations in a VMFS volume. These fields include the id, blk_size, blk_count, size, and blocks values.

The block size is the size of the blocks used by the file to accommodate its size. The file system will initially allocate the smallest block size capable of accommodating the file, including the content block of the inode itself. If the size of a file is less than the field's size reserved for the blocks/rdm_id/content field, then the inode blocks/rdm_id/content field contains the file's contents. Suppose the file size is too big to hold in the inode but smaller than a sub-block, then the file system stores it in a sub-block. If the file is larger than a sub-block, the file system stores it in enough file blocks to accommodate its contents.

The tests performed for this research included changing files' values to see how it impacted the metadata as the files are modified. One such test included creating a thin-provisioned virtual machine and later increasing its size (Section 2.1.1).

Experimentation	Block Count	Block Size	Size
Prior to VM creation	1	65536	7225
Post VM creation	9901	1048576	107374182400
Post VM fragmentation	15527	1048576	107374182400
Post VM deletion	0	1048576	0

Figure 18. Differences in VMFS Inode block count, block size, and size

Since the virtual machine is thin-provisioned, the file size is never fully allocated for the virtual machine. The VM's size is 102,400MB, while the allocated blocks are only 9,901MB. The increased block count as the VM is fragmented is the only indication that the size of the allocated space for the virtual machine has increased. Upon deleting the VM, the file system zeroes the block count and the size, but the block size remains.

For details on how the file id links the inode to the directory entry, see Section 6.6. This link to the directory entry facilitates identifying the logical distribution, i.e., the file's path.

The blocks/rdm_id/content field is essential for identifying the file distribution. It may not be obvious, but the blocks, rdm_id, and content all occupy the same data space (Figure 17). The value that is present is determined by evaluating the rest of the metadata. Of chief concern is when the inode uses this data field in the capacity of inventorying blocks for a file volume allocation. Blocks help identify the file blocks' physical location in the volume data used to store the file contents.

Unfortunately, when the file system deletes a file, it also zeroes out the blocks field, making file recovery via metadata not feasible.

5.4.4. VMFS Inode Flags and Zla Metadata

According to the vmfs-tools, the zla identifies the block type for the inode. Vmfs-tools does not address the use of the flags field.

When the VM referenced in Section 5.4.3 was manipulated, some exciting modifications to the flags and zla fields in the metadata occurred.

Experimentation	Flags	Zla
Prior to VM creation	1	2
Post VM creation	9	3
Post VM fragmentation	9	3
Post VM deletion	8	3

Figure 19. Differences in VMFS Inode flags and zla

In Figure 19, the flags change from 9 to 8 upon the deletion of the VM. This observation supports the conclusion that the first bit is a flag for "not deleted/deleted", or perhaps more likely "in use / not in use." If the inode before the VM creation is a file, then this conclusion remains valid.

The zla transition from 2 to 3 upon creating the VM means the block type of the inode has changed. This zla transition, coupled with the file size change in Figure 18, implies the inode was previously provisioned to another file but reprovisioned to the VM upon creation.

5.5. VMFS Metadata Header Data Structure

The VMFS metadata header is an abstracted component that is 4096-bytes and is the first half of data objects, such as Inodes and Bitmap Entries; this is evident in vmfs-tools. The FDC bitmap header consistently has a data_size value of 8192, and the metadata header is the abstracted first half the data entry. The metadata header contains information, including: the signature, heartbeat information, and data position.

Byte Range	Structure	Name	Valid
0-3	uint32	magic	Yes
4-11	uint64	pos	Yes
12-19	uint64	hb_pos	Yes
20-27	uint64	hb_seq	No
28-35	uint64	obj_seq	Unknown
36-39	uint32	hb_lock	Yes
40-55	uuid	hb_uuid	Yes
56-63	uint64	mtime	No
56-59	uint32		New

Figure 20. VMFS Metadata Header

The vmfs-tools parser remains largely accurate in parsing the data in the metadata header. The magic, pos, hb_pos, hb_lock, and hb_uuid were all confirmed in the first FDC bitmap entry analysis. The first FDC bitmap entry was the only metadata structure that had an active heartbeat entry. The research also noted the following observations:

- The hb_seq is no longer valid. When the hb_pos value matches the heartbeat's pos value, the hb_seq does not match the heartbeat's seq value.
- The obj_seq field is inconclusive. It is unclear precisely what the obj_seq is referring to; additional analysis is necessary to determine if it is referencing an inode or some other metadata.
- The time value is no longer valid. The replacement entry is curious because it appears to increment with the bitmap entries and inodes, some of the time.

5.6. Directory Entry Data Structure

The only significant change to the directory entry data structure is that the entry size increased from 140 bytes to 288 bytes.

Byte Range	Structure	Name	Valid
0-3	uint32	type	Yes
4-7	uint32	block_id	Yes
8-11	uint32	record_id	Yes
12-287	char	name	No

Figure 21. VMFS Metadata Header

The block_id of the directory entry can correlate with the id of the inode in the FDC bitmap. According to vmfs-tools, linking an inode to a directory entry requires matching the following fields:

- inode type = type
- inode id = block_id
- inode id2 = record_id

Once confirmed, the file metadata can be reliably associated with the file name and path.

6. Findings and Discussion (Exposition of the Data)

6.1. File Recovery

This research concludes that the goal of file recovery using metadata is not feasible. The VMFS file system's comparative analysis before and after deletion identified that the file system zeroes all of the blocks determining the disk allocation upon file deletion. The inode data structure contains the blocks mentioned above (5.4.5).

6.2. Metadata Discoveries

VMFS does not have the same level of directly attributable metadata as a file system used by a personal computer. However, it's metadata can still be very useful in identifying relationships. Some of the newly found metadata include:

- The device name located in the Volume Information Block (4.2)
- The host IPv4 address located in the heartbeat entry (5.1)

6.3. VMFS-Tools Observations

Open-source utilities were instrumental in facilitating this research. Unfortunately, the abstraction and lack of documentation did challenge the understanding of the utilities' inner workings. When the data structures were relevant, the detail provided by vmfs-tools is extremely helpful in giving a starting point for analyzing the packed data. Some of the more useful findings from converting vmfs-tools to Python are the following:

- Position values are relative to the 17th block (17MB) (3.1)
- No support for GUID Partition Table (4.1)
- The File System Information contained novel unhandled packed data (4.3)
- The File Descriptor Bitmap values remained static in volumes up to 8TB (4.6)

- The root directory contained two new system files: jbc.sf and sdd.sf (4.7)
- The existence of two new inode/bitmap entry signatures: 0x10C00007 and 0x10C00008 (4.7)
- The Bitmap Entries were all null, and new unhandled packed data was present (5.3)

6.4. File System Fragmentation

The most challenging obstacle to reverse engineering VMFS 6 is the fragmentation of the file system metadata files. The fragmentation of the initial file descriptor bitmap, root directory, and file block bitmap absorbed many hours of research (4.6). This lost time resulted in an inability to conduct a more rigorous examination of block ids, bitmaps, block arrays, and other unparsed VMFS data.

7. Recommendations and Implications

7.1. Recommendations for Practice

The new metadata found through this research provides a means of identifying volume names and host IP addresses. If the IP addresses for hosts in the file system heartbeats are present in VMFS volumes managed by vSphere, it presents a potential security risk. If an attacker gains access to a host, they will not need to rely on network protocols to inventory the other hosts using the datastores. A scan of the IPv4s in the heartbeats of the attached storage should be enough information to pivot.

The observations that vmfs-tools does not support GPT is fascinating because it means that the utility would only support VMFS 3 partitions and VMFS 5 partitions converted from VMFS 3.

7.2. Implications for Future Research

There remains an opportunity for further research on VMFS 6. This research identified metadata not handled by vmfs-tools added to the file system information and the bitmaps.

Additionally, since this research observed that the data structure specified by vmfs-tools for bitmap entries is null, then the bitmap entries need to be reverse-engineered from scratch.

Also, the bitmap files' fragmented nature is an obstacle to additional research. It is conceivable that parsing the block array for the files in the File Descriptor Bitmap file block may be the key to mapping the fragmented bitmaps.

The two new system files found in the Root Directory and the corresponding signatures present a compelling challenge to expand the scope of vmfs-tools.

8. Conclusion

Reverse engineering VMFS 6 yielded some previously unparsed metadata fields: the volume name and host IP addresses. Additionally, this research verified that file recovery of deleted files is not possible through mapping file allocations that are available in metadata entries of deleted files. There was some hope because the metadata fields for deleted files are not themselves deleted; however, the wiping of the block array used to determine block allocation eliminates the ability to recover the file using this information.

Michael Smith, mesmith1@protonmail.com

References

AndreTheGiant. (2011, July 13). *VMFS block size |VMware communities*. Retrieved August 31, 2020, from https://communities.vmware.com/docs/DOC-11920

Biggerstaff, T. (1989). Design recovery for maintenance and reuse. *Computer*, *22*(7), 36-49. https://doi.org/10.1109/2.30731

Carrier, B. (2005). *File system forensic analysis*. Addison-Wesley Professional.

Chikofsky, E. J., & Cross, J. H. (1990). Reverse engineering and design recovery: a taxonomy. IEEE Software, 7(1), 13–17. doi:10.1109/52.43044Continuum. (2020, July 7). *How can a normal vSphere admin figure out where... |VMware communities*. https://communities.vmware.com. Retrieved September 8, 2020, from https://communities.vmware.com/thread/639241

FluidOps. (2010). *Open Source VMFS Driver*. Google Code. https://code.google.com/archive/p/vmfs/

Henry, P. (2012, August 10). *Data sanitization in the virtual realm and cloud*. Information Security Training | SANS Cyber Security Certifications & Research. https://www.sans.org/blog/data-sanitization-in-the-virtual-realm-and-cloud/

Hogan, C. (2011, August 8). VSphere 5.0 storage features Part 7 - VMFS-5 & GPT - VMware vSphere blog. *VMware vSphere Blog*. https://blogs.vmware.com/vsphere/2011/08/vsphere-50-storage-features-part-7-gpt.html

Hogan, C. (2017, August 17). *A change to sub-blocks on VMFS-6*. CormacHogan.com. https://cormachogan.com/2017/08/17/change-sub-blocks-vmfs-6/

Hommey, M. (2012, March 24). *DF result mismatch · Issue #5 · glandium/vmfs-tools*. GitHub. https://github.com/glandium/vmfs-tools/issues/5

Michael Smith, mesmith1@protonmail.com

Hommey, M. (2016, January 16). *Glandium/vmfs-tools*. GitHub. Retrieved April 13, 2020, from https://github.com/glandium/vmfs-tools

Kinchla, B. (2015). *Forensic Recovery of Evidence from Deleted VMware vSphere Hypervisor Virtual Machines* [Master's thesis]. ProQuest Dissertations and Theses Global.

Mustafa, Z. S. (2016). *Assessing the Evidential Value of Artefacts Recovered from the Cloud*[Doctoral dissertation]. https://dspace.lib.cranfield.ac.uk/bitstream/handle/1826/12017/Mustafa%20Z,%20Thesis.pdf?sequence=1

Svoskobo. (n.d.). *Versions of VMFS Datastores*. VMware Docs Home. Retrieved July 27, 2020, from https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.storage.doc/GUID-7552DAD4-1809-4687-B46E-ED9BB42CE277.html

Tsao, W. (2019, January 29). *Weafon/vmfs6-tool*. GitHub. Retrieved April 13, 2020, from https://github.com/weafon/vmfs6-tool

Vaghani, S. B. (2010). Virtual machine file system. *ACM SIGOPS Operating Systems Review*, *44*(4), 57-70. https://doi.org/10.1145/1899928.1899935