Forensic Timeline Analysis of ZFS

Using ZFS Metadata to enhance timeline analysis and detect forged timestamps.

Dylan Leigh <research@dylanleigh.net>,
Hao Shi <hao.shi@vu.edu.au>
Centre for Applied Informatics, College of Engineering and Science,
Victoria University, Melbourne, Australia

BSDCan 2014 - 14 May 2014

During forensic analysis of disks, it may be desirable to construct an account of events over time, including when files were created, modified, accessed and deleted. "Timeline analysis" is the process of collating this data, using file timestamps from the file system and other sources such as log files and internal file metadata.

The Zettabyte File System (ZFS) uses a complex structure to store file data and metadata and the many internal structures of ZFS are another source of timeline information. This internal metadata can also be used to detect timestamps which have been tampered with by the touch command or by changing the system clock.

Contents

1 Introduction

In computer forensics, *timeline analysis* is the collation of timestamps and other event information into an account of what occurred on the computer. Event information such as file timestamps, log files and web browser cache and history is collated into a "super-timeline"[1], with events corroborated from multiple sources.

The Zettabyte File System (ZFS)[2, 3], a relatively new and unusual filesystem, uses a novel pooled disk structure and hierarchical object based mechanism for storing files and metadata. Due to the unusual structure and operation of ZFS, many existing forensics tools and techniques cannot be used to analyse ZFS filesystems. No timeline analysis tools[1, 4] currently make use of ZFS internal metadata.

Furthermore, there are few existing studies of ZFS forensics, none of which include an empirical analysis of timeline forensics. There are no established procedures or guidelines for forensic investigators, with the exception of data recovery and finding known data on a disk [5, 6].

We have examined ZFS documentation, source code and file systems to determine which ZFS metadata can be useful for timeline analysis. Much of the metadata is affected by the order in which the associated files were written to disk. Our analysis has discovered four techniques which can use this metadata to detect falsified timestamps, as well as one which provides extra timeline information.

2 Existing ZFS Forensics Literature

Beebe et al's *Digital Forensic Implications of ZFS* [7] is an overview of the forensic differences between ZFS and more "traditional" file systems; in particular it outlines many forensic advantages and challenges of ZFS. Beebe identifies many properties of ZFS which may be useful to a forensic investigator, highlighting the opportunities for recovery of data from multiple allocated and unallocated copies. However, it is based on theoretical examination of the documentation and source code, noting that "we have yet to verify all statements through our own direct observation and reverse engineering of on-disk behavior".

Beebe also mentions ZFS forensics in [8], noting that many file systems not used with Microsoft and Apple products have recieved inadequate study; ZFS, UFS and ReiserFS are mentioned as particular examples of "file systems that deserve to be the focus of more research".

Data recovery on ZFS has been examined extensively by other researchers. Max Bruning's *On Disk Data Walk* [6] demonstrates how to access ZFS data on disk from first principles; Bruning has also demonstrated techniques for recovering deleted files[9].

Li [5, 10] presents an enhancement to the ZDB filesystem "debugger" which allows it to trace target data within ZFS media without mounting the file system. Finally, some ZFS anti-forensics techniques are examined by Cifuentes and Cano [11].

3 Background: The Zettabyte File System

A brief description of ZFS internals is provided here; for full details please consult Bonwick and Ahrens' *The Zettabyte File System* [2], Sun Microsystems *ZFS On Disk Specification* [12] and Bruning's *ZFS On Disk Data Walk* [6].

ZFS stores all data and metadata within a tree of objects. Each reference to another object can point to up to three identical copies of the target object, which ZFS spreads across multiple devices in the pool if possible to provide redundancy in case of disk failure. ZFS generates checksums on all objects and if any objects are found to be corrupted, they are replaced with one of the redundant copies.

3.1 Basic Structure

A ZFS Pool contains one or more virtual devices ("vdev"), which may contain one or more physical devices. Virtual devices may combine physical devices in a mirror or "RAID-Z" to ensure redundancy. Dedicated devices for cache, ZIL and hot spares may also be allocated to a pool.

Each device contains an array of 128 "Uberblocks"; only one of these is active at one time. The Uberblock array is duplicated 4 times in each vdev for redundancy. Each Uberblock contains a "Block Pointer" structure pointing to the root of the object tree, the "Meta Object Set". Apart from the metadata contained within the Uberblock, all data and metadata is stored within the object tree, including internal metadata such the free space maps and delete queues.

Block Pointers connect all objects and data blocks and include forensically useful metadata including up to three pointers (for redundant copies of the object/data), transaction group ID (TXG) and checksum of the object or data they point to. Where Block Pointers need to refer to multiple blocks (e.g. if not enough contiguous space is available for the object), a tree of "Indirect" Blocks is used. Indirect blocks point to blocks containing further Block Pointers, with the level zero leaf Block Pointers pointing to the actual data. Section 5.2.3 further discusses the forensic uses of Block Pointers.

Each ZFS filesystem in a pool is contained within an object set, or "Dataset". Each pool may have up to 2^{64} Datasets; other types of datasets exist including snapshots and clones. The "Meta Object Set" contains a directory which references all datasets and their hierarchy (e.g. snapshot datasets depend upon the filesystem dataset they were created from). Filesystem datasets may be mounted on the system directory hierarchy independently; they contain several filesystem-specific objects as well as file and directory objects. File objects are examined in detail in section 5.2.

3.2 Transactions, TXG and ZIL

ZFS uses a transactional copy-on-write method for all writes. Objects are never overwritten in place, instead a new tree is created from the bottom up, retaining existing pointers to unmodified objects, with the new root written to the next Uberblock in the array.

Transactions are collated in "Transaction groups" for committing to disk. Transaction groups are identified by a 64 bit transaction group ID (TXG), which is stored in all Block Pointers and some other structures written when the Transaction Group is flushed. Transaction groups are flushed to disk every 5 seconds under normal operation.

Synchronous writes and other transactions which must be committed to stable storage immediately are written to the ZFS Intent Log (ZIL). Transactions in the

ZIL are then written to a transaction group as usual, the ZIL records can then be discarded or overwritten. When recovering from a power failure, uncommitted transactions in the ZIL are replayed. ZIL blocks may be stored on a dedicated device or allocated dynamically within the pool.

3.3 Space Allocation

Each ZFS virtual device is divided up into equal sized regions called “metaslabs”. The metaslab selection algorithm prioritizes metaslabs with higher bandwidth (closer to the outer region of the disk) and those with more free space, but de-prioritizes empty metaslabs. A region within the metaslab is allocated to the object using a modified first-fit algorithm.

The location of free space within a ZFS disk is recorded in a “space map” object. Space maps are stored in a “log-structured” format - data is appended to them when disk space is allocated or released. Space maps are “condensed”[13] when the system detects that there are many allocations and deallocations which cancel out.

4 Experiments

4.1 Method

In our experiments, ZFS metadata was collected from our own systems and test systems created for this project. Disk activity on the test systems was simulated using data obtained from surveys of activity on corporate network file storage [14, 15, 16].

ZFS pools with 1,2,3,4,5,7 and 9 devices were created on the test systems, including pool configurations with all levels of Raid-Z, mirrors and mirror pairs as well as “flat” pools where applicable, giving 22 different pool constructions. The test systems used FreeBSD version 9.1-RELEASE with ZFS Pool version 28.

4.2 Timestamp Tampering

Three experiments were conducted for each pool: a control with no tampering, one where timestamps were altered by changing the system clock backwards one hour for ten seconds, and one where the Unix “touch” command was used to alter the modification and access times of one file backwards by one hour.

4.3 Data Collection

Internal data structures were collected every 30 minutes as well as before and after all experiments, using the “zdb” (ZFS Debugger) command. Three types of metadata were collected:

- All 128 Uberblocks from all virtual devices in the pool.
  - `zdb -P -uuu -l /dev/<device>`
- All Spacemaps for all Metaslabs in each virtual device.
  - `zdb -P -mmmmmm <pool>`
- All objects and Block Pointers from with the dataset with simulated activity.
  - `zdb -P -dddddd -bbbbbb <poolname>/<dataset-name>`

The “Quick Reference”[17] contains further example commands for ZFS timeline forensics.

Listing 1: Example Uberblock dump from ZDB

```
  ...
Uberblock[72]
  ...
Uberblock[73]
        magic = 0000000000bab10c
        version = 28
        txg = 1737
        guid_sum = 4420604878723568201
        timestamp = 1382428544 UTC = Tue Oct 22 18:55:44 2013
        rootbp = DVA[0]=<0:3e0c000:200> DVA[1]=<1:3f57200:200> DVA
            [2]=<2:3593a00:200> [L0 DMU objset] fletcher4 lzjb LE
            contiguous unique triple size=800L/200P birth=1737L/1737P
            fill=42 cksum=15ffed59a7:7e9c9c594b0:17c4c7cb7a7eb:318
            e5de89d442a
Uberblock[74]
   ...
```

5 ZFS Structures for Timeline Analysis

5.1 Uberblocks

An example Uberblock dump from ZDB is shown in Listing 1. The most useful entries for forensic investigators are the TXG and timestamp; this provides one method to link a TXG to the time it was written. This can be used to verify file timetamps and detect false ones. Where a file is modified with the touch command, the timestamp on the file will not match the timestamp in the Uberblock with the same transaction group (TXG).

Consecutive uberblocks contain increasing TXG and timestamps (except for the highest TXG where the array wraps around). Where the system clock was changed, the corresponding changes in the uberblock timestamps before and after the clock changes show that tampering occurred.

Uberblocks could potentially be used to access previous version of the object tree, however they are quickly overwritten (see “Disadvantages” below) so this facility is only useful when they can be collected immediately after tampering has occurred. Other than this, uberblocks do not provide any extra information beyond the file timestamps.

In 50% of all experiments involving tampering we were successfully able to use an Uberblock to detect a falsified file timestamp. However, this was only possible because data was collected every 30 minutes during the experiments; at the end of the simulation the relevant uberblocks were already overwritten.

5.1.1 Uberblocks: Disadvantages

Uberblocks are the simplest metadata for an attacker to tamper with as they are at the top of the hash tree; if modified the attacker only has to change the uberblock's internal checksum to match.

The most serious disadvantage of Uberblocks for forensic use is that for most pools they last only 10.6 minutes under a continuous writing load. This is due to ZFS typically writing a new Uberblock every 5 seconds ($5 \times 128 = 640\, seconds$). In 75% of experiments, the oldest Uberblock in the array of 128 was between 634 and 636 seconds old.

Pools with 4 or more top level virtual devices (i.e. 4 or more disks and a “flat”

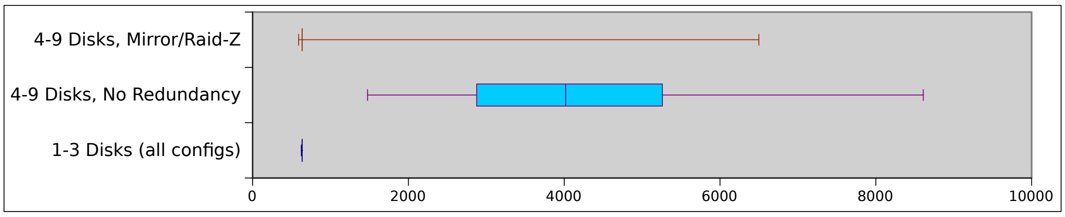

Figure 1: Time in seconds between oldest and newest Uberblock, for different pool configurations.

pool with no redundancy) consistently had a significantly longer period between the oldest and newest Uberblock in the pool; between 1475 and 8609 seconds (median 4018). With more than three devices, some writes may not affect all devices, so a new uberblock will not always be written to all of them for each TXG. Pools using RAID-Z or mirroring are less affected as writes are likely to affect all devices in a more redundant pool.

Figure 1 plots the differences between different types of pool configuration. Pools with less than 4 disks all had a uberblock period of 636 seconds or less, as did most pools with 4 or more disks which used a mirror or RAID-Z configuration.

Due to the short lifespan of Uberblocks they are most useful in a "dawn raid" scenario, where a system is siezed immediately after a suspect may have been deleting or modifying files. In this case they could be used to access previous version of the object tree and determine recent changes.

5.2 File Objects

All objects and Block Pointers associated with a dataset may be extracted using ZDB's `-b` and `-d` options. We have only examined filesystem datasets at this stage. A sample file object is shown in Listing 2.

File objects contain several attributes which are useful for verifying timestamps; in particular, the "gen" TXG and the Object Number which are related to the creation time of the object. Block pointer TXG values in the Block Pointers which point to the file data can also be used to detect falsified timestamps, and in some cases Block Pointers to previously written file data segments may also be used to obtain the time of those modifications.

5.2.1 Object Number

All objects in a dataset have a 64 bit "object number", used as an internal identifier for ZFS. As new object numbers are generated in locally increasing order, they are related to the order of creation of objects, and can therefore be used to detect objects with a falsified creation time.

In all experiments involving a falsified creation time, the object number of the affected files were out of order; this provides a simple method to detect false creation times. Figure 2 (left) shows a plot of object number vs creation time from an experiment where the system clock was adjusted one hour backwards to alter file timestamps; the outliers above the line indicate files with false creation times.

Object numbers may be reused when ZFS detects a large range of unused numbers (e.g. when many files are deleted); this did not occur during our experiments and further study is required to determine how often numbers are reused during production conditions and how this would affect forensic use of object numbers.

Listing 2: Example File Object dump from ZDB

```
    Object  lvl   iblk   dblk  dsize  lsize   %full  type
     15417    1  16384  67072  67072  67072  100.00  ZFS plain file (K
        =inherit) (Z=inherit)
                                           168   bonus  System attributes
        dnode flags: USED_BYTES USERUSED_ACCOUNTED
        dnode maxblkid: 0
        path    /1382428539-to-2147483647-reps-0
        uid     0
        gid     0
        atime   Tue Oct 22 17:55:40 2013
        mtime   Tue Oct 22 17:55:40 2013
        ctime   Tue Oct 22 17:55:40 2013
        crtime  Tue Oct 22 17:55:40 2013
        gen     1737
        mode    100644
        size    66566
        parent  4
        links   1
        pflags  40800000004
Indirect blocks:
               0 L0 DVA[0]=<2:353c200:10600> [L0 ZFS plain file]
                   fletcher4 uncompressed LE contiguous unique single
                   size=10600L/10600P birth=1737L/1737P fill=1 cksum=1
                   e970f0f68f0:3f178f0ed3b3b88:1c0a9b8bd4c82800:
                   eb83cbb696eca800

                 segment [0000000000000000, 0000000000010600) size
                    67072
```

As object numbers are not affected by modifications to the file, they cannot be used to detect falsified modification time, unless the modification time is altered to before the file was created. They cannot be used to provide any extra event information to forensic investigation (beyond verifying the file creation time).

5.2.2 Object Generation TXG

All filesystem objects also store the transaction group ID in which they were generated (this is the "gen" attribute in Listing 2).

Similar to object numbers, as they increase with the order of creation of objects they can also be used to detect objects with falsified creation time. In all experiments involving a falsified creation time, the generation TXG was out of order. The right plot in Figure 2 shows a plot of Generation TXG vs creation time; again the outliers above the line are from the same files which were created when the system clock was reverted by one hour. Although covering the same three hours the TXG graph rises evenly over time as TXG are flushed to disk at a constant rate, whereas file objects are created at a variable rate.

Likewise, they cannot be used to detect falsified modification time unless the modification time is altered to before the file was created, and no not provide any extra information beyond corroborating the creation time.

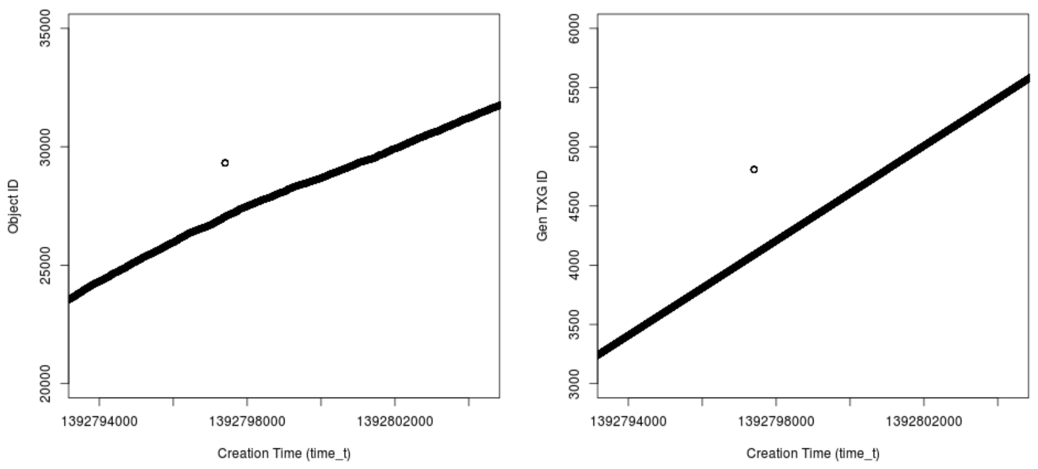

Figure 2: Object number (left) and generation TXG (right) vs. creation time, from file objects in an experiment with false creation times. Time values are time_t (seconds since Unix epoch).

5.2.3 Block Pointer TXG

Unless empty, each file object dumped from ZDB contains at least one Block Pointer pointing to the file data. Each Block Pointer contains the TXG in which the pointer was written. This is the most useful metadata discovered so far for detecting falsified timestamps, and can even be used to determine past modification times in some cases.

As TXG increase with time, the order of writes can be verified by comparing modification times with the file's most recent Block Pointer TXG. If a file contains multiple Block Pointers, the highest level pointer in the Indirect Block(s) will contain the most recent TXG. Figure 3 (left) shows Block Pointer TXG plotted against modification time.

Allowing a 5 second window for the flushing of each TXG to disk, any out-of-order TXG/modification time pairs indicate the timestamp may be falsified. In all experiments inolving tampering, comparison of Block Pointer TXG between files was successful in detecting it.

Determining Past Modification Times

Listing 3 shows an example of a large file with two segments written during different transactions. The file object points to one level 1 Indirect Block containing two level 0 "leaf" Block Pointers, which point to the file data. The data in the first level 0 BP was written during TXG 15464, whereas the second level 0 BP was written in transaction 15853 (requiring the parent BP to be also rewritten).

If there is another file (or other object) in the pool which can link a timestamp to TXG 15464, we can determine the last time that the first segment was written. Note that this can only discover the most recent write to each segment; either segment may have been rewritten multiple times

This technique is most effective for large files which are modified only in small sections at a time (e.g. virtual machine images). In our experiments only 109 per million files required more than one Block Pointer for their data. Our experiments used statistics based on corporate network storage[15], where most files are under 4KB in size. With default settings ZFS uses segments up to 128KB in size thus very few files required more than one segment.

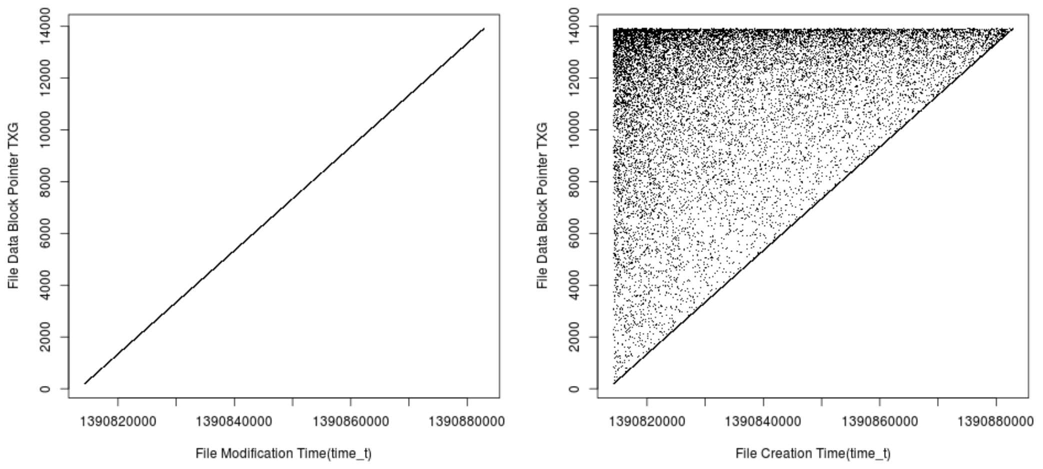

Figure 3: Modification Time (left) and Creation Time (right) of files vs their Block Pointer TXG values. No tampering was involved in this experiment.

Listing 3: Example File Object with multiple Block Pointers to data (many fields removed for clarity)

```
      Object  lvl   iblk  lsize   %full  type
      57296    2  16384  262144  100.00  ZFS plain file
         ...
         mtime   Wed Oct 23 13:21:10
         size    158599
         ...
Indirect blocks:
    0 L1  DVA[0]=<1:202c7400:400> DVA[1]=<2:2183e000:400>
         [L1 ZFS plain file] double size=4000L/400P ...
         birth=15853L/15853P fill=2 ...
    0  L0 DVA[0]=<1:24409600:20000> [L0 ZFS plain file]
         single size=20000L/20000P birth=15464L/15464P fill=1 ...
20000  L0 DVA[0]=<1:24a7da00:20000> [L0 ZFS plain file] ...
         single size=20000L/20000P birth=15853L/15853P fill=1 ...
```

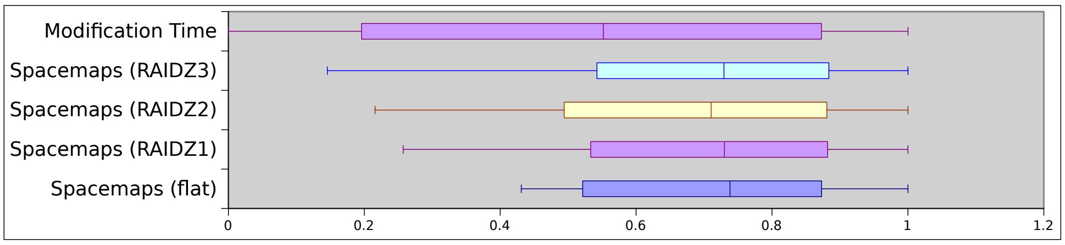

Figure 4: Normalized modification times (top) and TXG of Spacemap segments (lower 4 plots) in 5 disk pools after 24 hours with various RAIDZ levels.

5.3 Spacemaps

Each spacemap segment lists the TXG when the spacemap was written (see examples in Listing 4). However, this is likely to be a more recent TXG than the one in which the space was allocated, because spacemaps are "condensed" and re-written frequently. Therefore, presence of a Block Pointer's allocated space in a segment with a later TXG than the file's Block Pointer TXGs is not evidence of a forged timestamp.

Spacemaps could be used to verify that a given block has not been modified since the spacemap segment TXG (due to the copy-on-write system, blocks are never overwritten in place), although this is of limited forensic use.

Condensation is more frequent than might expected due to the frequency of temporary files created and deleted within seconds [15] and the ZFS metaslab allocation algorithm preferring to fill metaslabs towards full capacity to reduce fragmentation (increasing the chance of recently used metaslabs being condensed).

Figure 4 shows normalized plots of transaction groups from spacemaps in 5 disk pools (of various RAIDZ levels) and the modification time of the files in the pools. While modification time is evenly distributed, slightly skewed towards the present (median 0.55), the spacemap TXGs have a median of 0.72. With increasing redundancy, there are more early outliers remaining, however most spacemap segments are still condensed after the allocated block was written.

In none of the experiments could spacemaps be used to detect falsified timestamps, due to the the relevant allocation being condensed. It may not be possible to reliably reconstruct events from spacemaps in any filesystem with transient files.

5.3.1 Virtual Devices

ZFS uses a round robin algorithm[13] which attempts to write 512KB to each disk before moving to the next. Theoretically this could be used to detect out of order writes. However, due to transient files which may have been created in between other writes, it appears that this behaviour cannot be used to detect out-of-order timestamps.

Even if it is known that files were not modified or deleted, transient internal objects (e.g. spacemaps) may have been modified or destroyed. None of the timestamp falsification in our experiments could be detected by analysing the pattern of virtual device writes.

Listing 4: Example Metaslab/Spacemap dump from ZDB

```
        vdev          0
        metaslabs   119   offset                spacemap          free
        ---------------   -------------------   ---------------
            -------------
        metaslab      0   offset            0   spacemap     35   free
            536655872
                          segments         15   maxsize   536123392   freepct
                                99%
            [     0] ALLOC: txg 229, pass 1
            [     1]    A  range: 0000000000-0000008000  size: 008000
            [     2]    A  range: 0000011000-0000016800  size: 005800
            [     3]    A  range: 000000f800-0000010800  size: 001000
            [     4]    A  range: 000001c800-000001d000  size: 000800
...
            [    14]    A  range: 00000ad000-00000b4800  size: 007800
            [    15]    A  range: 0000093000-000009a800  size: 007800
            [    16]    A  range: 0000043800-0000047800  size: 004000
            [    17] ALLOC: txg 229, pass 2
            [    18]    A  range: 0000008000-000000b000  size: 003000
            [    19] FREE: txg 229, pass 2
            [    20]    F  range: 00000c4000-00000c7000  size: 003000
        metaslab      1   offset     20000000   spacemap      0   free
            536870912
                          segments          1   maxsize   536870912   freepct
                              100%
        metaslab      2   offset     40000000   spacemap      0   free
            536870912
                          segments          1   maxsize   536870912   freepct
                              100%
...
```

6 Discussion

6.1 Results & Observations

A summary of the results is shown in Table 1 below. It can be seen that comparing Block Pointer TXGs is the most effective technique for detecting falsified timestamps and the only method found so far which can obtain extra timeline information beyond the file timestamps. False creation time is much simpler to detect than false modification time, as there are more attributes affected by the creation order of an object.

Table 1: Summary of Results

Forensic Uses	Structures				
	Uberblock	BP TXG	Gen TXG	Obj. ID	Spacemap
Detect Forged Mtime	Sometimes	Yes	No	No	No
Detect Forged CRTime	Sometimes	Yes	Yes	Yes	No
Determine Past Mtime	No	Sometimes	No	No	No

6.2 False Positives

Timestamp mismatches can be caused by normal, innocuous events as well as deliberate tampering and anti-forensics. Care must be taken by forensic investigators to avoid false positives.

Regular corrections to the system clock will affect all timestamps. If the clock is fast and is corrected backwards, timestamps will appear out of order. This occurred in two of the 76 experiments, a overnight clock correction caused object number, Generation TXG and Block Pointer TXGs to appear out of order when they were not.

There are also valid reasons why timestamps may be changed by the user to a previous value, (or a userspace utility), such as unpacking archives or copying files with previous timestamps preserved - the newly created files will appear to have “false” timestamps to forensic tools.

6.3 Modifying Internal Metadata

A determined attacker could tamper with internal ZFS structures to match any modified file timestamps, in order to make the modification more difficult to detect by forensic investigators.

It is most simple to modify uberblocks as they are at the head of the tree, and if modified only their own checksum needs to be corrected. However, this may not be worthwhile for the attacker as uberblocks are quickly overwritten.

ID numbers (including Object number, Gen TXG and Block Pointer TXGs) could be falsified fairly simply if a suitable old ID can be inserted. On a typical filesystem with many transient files there would probably be many IDs which could be “reused” this way. Once this change is made, the attacker must then correct the checksums in all parent objects and Block Pointers up the tree and the uberblock, to prevent the tampering from appearing as disk corruption.

If an old ID cannot be used, tampering would be possible but may require rewriting all files in the filesystem to fit the desired IDs. In the worst case (or best case for the forensics team), where no plausible TXGs could be inserted, the entire pool would have to be re-created. Apart from the time and effort involved, rewriting the disk may leave its own traces detectable to the forensic investigators (such as lack of fragmentation).

6.4 Future work

Although this is the first time internal ZFS metadata has been examined for timeline forensics, the scope of this study is extremely limited and much more work needs to be done to validate the techniques presented here under varied conditions.

Most importantly, data from production systems needs to be obtained to verify our results with real systems. The next phase of this project will involve a survey to obtain metadata from real systems. Volunteers would be asked to submit anonymized ZDB data (with paths and names removed)[1].

Another important task is to incorporate the detection algorithms described into existing forensic tools such as log2timeline[1] so they can be tested in the field.

6.4.1 Other ZFS Structures

We have only examined a few ZFS structures at this stage. There are many other ZFS objects and other structures which could be used for timeline analysis.

As identified by Beebe et al [7], the ZFS Intent Log (ZIL) contains metadata relating to filesystem events. As entries in the ZIL are discarded (but not destroyed) after they are committed to a TXG, information from ZIL entries be recoverable long after they are written.

There are several per-dataset objects and attributes which are likely to be useful for timeline forensics, particularly the Delete Queue. Snapshot and clone datasets

[1]We intend to announce the call for volunteers on freebsd-fs and other mailing lists when the survey is ready.

have not yet been examined and are likely to provide a great deal of time-related metadata to investigators. The Meta-Object-Set and its dataset directories may also contain useful timeline data.

As mentioned in the Uberblock section, previous Object Sets from previous uberblocks could also be traced to determine recent changes. Although we have used the Uberblock array to detect TXG/timestamp inconsistencies, we have not yet examined them further.

Finally, regions of the disk which are no longer referenced by any Block Pointer could be examined to see if they contain old ZFS objects or BPs. Most ZFS structures contain magic numbers so that they can be easily identified, and TXG so the time they were written could be determined. If this sort of analysis would provide any useful data to forensic investigators it still needs to be determined empirically.

6.4.2 Other Pool Configurations and Workloads

Our experiments used simulated file activity based on a corporate/engineering network file workload[15, 16]. Many factors in our analysis, especially the condensation of spacemaps and allocation of large files, are affected by the patterns of file activity. Other types of systems (e.g. desktop, home media storage, webserver) could be expected to have different patterns of file activity which may make some techniques more or less viable.

Our experiments also used only a single filesystem dataset, whereas real systems would use many filesystems, with different patterns of file activity. Other dataset types (snapshots, clones, ZVOLs) were not examined. Many ZFS features such as compression and deduplication were not used. We did not examine pools with dedicated log or cache devices.

7 Conclusion

We have determined that multiple ZFS internal structures can be used as a source for timeline analysis, including the detection of forged timestamps. Block Pointers to file data are particularly useful and can sometimes be used to determine the time of previous writes to a file.

Although these techniques have been shown to work under laboratory conditions they are yet to be tested in the field and on a wide variety of systems and configurations.

Clock corrections and other normal behaviour could appear to be deliberate tampering with timestamps, and investigators should consider this possibility.

7.1 Acknowledgements

We are extremely grateful to BSDCan for funding travel to and accommodation at the conference and providing us with an opportunity to speak to the BSD community. We are also grateful to many staff and students at Victoria University who have provided advice regarding presentation and academic writing, especially Prof. Xun Yi, Dr Gitesh Raikundalia, Dr Russell Paulet and Prof. Yuan Miao, as well as Dr Ron van Schyndel of RMIT University, Melbourne, Australia. Finally, we would like to thank Matt Ahrens of Delphix for corrections and advice regarding ZFS processes and terminology.

References

[1] Guðjónsson K. Mastering the super timeline with log2timeline. SANS Institute. 2010;Available from: http://www.sans.org/reading_room/whitepapers/logging/mastering-super-timeline-log2timeline_33438.

[2] Bonwick J, Ahrens M, Henson V, Maybee M, Shellenbaum M. The zettabyte file system. In: Proc. of the 2nd Usenix Conference on File and Storage Technologies; 2003. Available from: http://users.soe.ucsc.edu/~scott/courses/Fall04/221/zfs_overview.pdf.

[3] Bonwick J, Moore B. ZFS: The last word in file systems. SNIA Software Developers Conference. 2008;Available from: http://wiki.illumos.org/download/attachments/1146951/zfs_last.pdf.

[4] Carrier B. Mactime. [manual]; 2013. Available from: http://www.sleuthkit.org/sleuthkit/man/mactime.html.

[5] Li A. Digital Crime Scene Investigation for the Zettabyte File System. Macquarie University; 2009. Available from: http://web.science.mq.edu.au/~rdale/teaching/itec810/2009H1/FinalReports/Li_Andrew_FinalReport.pdf.

[6] Bruning M. ZFS On-Disk Data Walk. In: OpenSolaris Developer Conference; 2008. Available from: http://www.osdevcon.org/2008/files/osdevcon2008-max.pdf.

[7] Beebe NL, Stacy SD, Stuckey D. Digital forensic implications of ZFS. Digital Investigation. 2009;6:S99–S107. Available from: http://www.sciencedirect.com/science/article/pii/S1742287609000449.

[8] Beebe N. Digital forensic research - The good, the bad and the unaddressed. Advances in Digital Forensics V. 2009;p. 17–36. Available from: http://link.springer.com/chapter/10.1007/978-3-642-04155-6_2.

[9] Bruning M. Max Bruning's weblog: Recovering removed file on zfs disk; 2008. Available from: http://mbruning.blogspot.com.au/2008/08/recovering-removed-file-on-zfs-disk.html.

[10] Li A. Zettabyte File System Autopsy: Digital Crime Scene Investigation for Zettabyte File System; 2009. Available from: http://web.science.mq.edu.au/~rdale/teaching/itec810/2009H1/WorkshopPapers/Li_Andrew_FinalWorkshopPaper.pdf.

[11] Cifuentes J, Cano J. Analysis and Implementation of Anti-Forensics Techniques on ZFS. Latin America Transactions, IEEE (Revista IEEE America Latina). 2012;10(3):1757–1766. Available from: "http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6222582".

[12] Anonymous. ZFS On Disk Specification. Sun Microsystems; 2006. Available from: https://maczfs.googlecode.com/files/ZFSOnDiskFormat.pdf.

[13] Bonwick J. ZFS Block Allocation. [online]; 2006. Available from: https://blogs.oracle.com/bonwick/en_US/entry/zfs_block_allocation.

[14] Leung AW, Pasupathy S, Goodson GR, Miller EL. Measurement and Analysis of Large-Scale Network File System Workloads. In: USENIX Annual Technical Conference. vol. 1; 2008. p. 5–2. Available from: http://www.ssrc.ucsc.edu/Papers/leung-usenix08.pdf.

[15] Agrawal N, Bolosky WJ, Douceur JR, Lorch JR. A five-year study of file-system metadata. ACM Transactions on Storage (TOS). 2007;3(3):9. Available from: "http://dl.acm.org/citation.cfm?id=1288788".

[16] Roselli DS, Lorch JR, Anderson TE, et al. A Comparison of File System Workloads. In: USENIX Annual Technical Conference, General Track; 2000. p. 41–54. Available from: https://www.usenix.org/legacy/event/usenix2000/general/full_papers/roselli/roselli.pdf.

[17] Leigh D. ZFS Timeline Forensics Quick Reference; 2014. Available from: http://www.research.dylanleigh.net/zfs-bsdcan-2014/zfs-timeline-quickref.pdf.

Biography

Dylan Leigh is an honours student at Victoria University, Melbourne, where his research focus is filesystem forensics. He completed Computer Science and Engineering degrees at RMIT University, where he teaches several subjects including Computer & Internet Forensics. He has been a FreeBSD user since 2004.

Dr Hao Shi joined Victoria University as Lecturer in 1992 and is now Associate Professor in the College of Engineering & Science at Victoria University. She completed her PhD in the area of Computer Engineering at University of Wollongong, Australia and obtained her Bachelor of Engineering degree at Shanghai Jiao Tong University, China.