

TZWorks® \$MFT and \$Logfile Analysis (*mala*) Users Guide



Abstract

mala is a standalone, command-line tool that parses the Windows \$Logfile artifact. The \$Logfile is a transaction log used for the NTFS filesystem. The tool makes use of the \$MFT file for support information to help add additional context to the log entries. This tool has working versions for Windows, Linux and macOS.

Copyright © TZWorks LLC

www.tzworks.com

Contact Info: info@tzworks.com

Document applies to v0.24 of ***mala***

Updated: Apr 24, 2025

Table of Contents

1	Introduction	2
2	NTFS Transactional Log Internals	3
2.1	Operation Log Record	3
2.2	Operation Types Using in Log Records	4
2.3	Data Structure interdependencies.....	5
2.4	Log records and partial MFT attribute data.....	6
2.5	Transactions and operations.....	8
2.6	Time is not explicitly recorded in the records.....	9
3	How to Use <i>mala</i>	10
3.1	Parsing with only the \$Logfile for analysis.....	10
3.2	Reporting.....	11
4	Pulling Artifacts off a Live System	13
5	Available Options	13
6	Authentication and the License File.....	14
7	References	14

TZWorks® \$MFT and \$LogFile Analysis (*mala*) Users Guide

Copyright © TZWorks LLC

Webpage: http://www.tzworks.com/prototype_page.php?proto_id=46

Contact Information: info@tzworks.com

1 Introduction

The Windows NTFS file system has a transactional architecture that is used to ensure that the operating system can recover from a crash into a known good state. Aside from the NTFS file system kernel driver failing, Windows does a good job at maintaining data consistency after critical failures that cause the system to shut down unexpectedly. Specifically, NTFS logs file transactions when:

- Creating a file
- Deleting a file
- Extending a file
- Truncating a file
- Setting file information
- Renaming a file
- Change the security applied to a file

To achieve this level of reliability, Windows NTFS employs a journaling technique that records the sequence of file operations within the *\$LogFile*. After the sequence of operations is completed, the operating system commits the changes and the transaction is done. In this way, if the system should crash prior to a transaction being committed to disk, the system can read the sequence of changes from the *\$LogFile* and then perform (if necessary) any 'undo' operations to get the system into a known good, stable state.

From a forensic standpoint, analyzing the *\$LogFile* can yield a chronological list of historical transactions that were done. The *\$LogFile* is fixed in size, so once it is filled, additional data is wrapped and the old data overwritten with new transactions. Depending on the frequency of file changes made on a system the number of historical transactions will vary. The size of the *\$LogFile* is typically 64 MB for a volume, however, it can be resized based on need. Using the standard default size and normal usage, one should expect a few hours of activity recorded in a *\$LogFile*. This time estimate, is highly subjective and will vary depending on the frequency of the file system changes.

To determine the size of the *\$LogFile* on a volume, type the following:

➤ ***chkdsk /L***

To adjust the size of the *\$LogFile* on a volume, use the following. The example adjusts the c: volume to 128MB, the value is in terms of KB:

➤ ***chkdsk c: /L:131072***

The transaction log artifacts are located in the root of any NTFS partition. These are the files that are of interest. (note: transactional record data can also be retrieved from unallocated clusters).

Artifact	Path
NTFS Transaction log	<NTFS partition>\\$Logfile
Master File Table	<NTFS partition>\\$MFT (changes to the \$MFT are in the \$Logfile)
Change log journal	<NTFS partition>\\$Extend\UsnJrnl:\$J (much of this data is already in the \$Logfile)

2 NTFS Transactional Log Internals

Aside from the documentation that is available on the Internet from a number of open sources, there is some extensive documentation provided by *Microsoft* in various sources, including the *Windows Internals* series of books. This section is not meant by any means to provide detailed analysis of the internals of the transaction log. The purpose here is to try to go over some of the transactional log internals, with the intent: (a) to try to help the analyst understand how the *mala* tool parses the data, and (b) to provide background information why the data is presented as it is in the reports the *mala* tool generates.

2.1 Operation Log Record

The header that is present for each operational log record, contains of the fields shown in the table below. The first two fields are the *Current LSN* and the *Previous LSN*. *LSN* is defined here as a *Log Sequence Number*, and it serves as a unique identifier for each log entry. Using the *LSN*, each operational record will have a link to the previous operation in the transaction. In this way, one can traverse the list from the newest operation to the oldest operation as it pertains to a single transaction. The operation that was done is indicated in the *Redo Operation*, and its converse, the *Undo Operation* is also identified should the system need to undo any operations if a crash occurs. The *mala* tool parses both the 'redo' and 'undo' operations, but it only reports on the 'redo', since that is the operation that took place. To get more details about an operation, one needs to parse the respective payload at the end of the header. The payload identifies the data that actually used to make the change. More information about the payload and the type of data it contains is discussed throughout this document.

Field	# bytes	Meaning
Current LSN	8	This operation's sequence number
Previous LSN	8	Sequence number of the previous operation in the transaction
Client Undo LSN	8	Used for recovery from a crash, similar to Previous LSN
Client Data length	4	Add 0x30 bytes to this field = size of the header + payloads
Record Type	4	Indicates type of record (checkpoint, general, etc)
Transaction ID	4	Type of log transaction
Flags	2	Indicates whether a continuation page(s) was used or not
Redo Operation	2	Operation that was performed
Undo Operation	2	Specifies what is needed to do undo the operation
Redo Offset	2	Offset to the Redo payload
Redo Length	2	Size in bytes of the Redo payload
Undo Offset	2	Offset to the Undo payload
Undo Length	2	Size in bytes of the Undo payload
Target Attribute	2	
LCNs to follow	2	Specifies how many logical cluster numbers (LCNs) to follow

Record Offset	2	Offset of the MFT record (if operation is MFT related)
Attribute Offset	2	Offset within the MFT attribute affected (if operation is MFT related)
MFT Cluster Index	2	Helps determine the inode given the offset in the \$MFT file
Target VCN	8	Virtual cluster number (VCN) for the operation
Target LCN	8	Logical cluster number (LCN) for the operation
Payload for Redo Oper	[see Redo Length]	This has the data specific to the Redo operation. The data is structured differently depending on operation.
Payload for Undo Oper	[see Undo Length]	Same as above, applied to Undo operation.

2.2 Operation Types Using in Log Records

The table below enumerates the various types of operations the *\$LogFile* can make use of when identifying a change. Some operations have extra data associated with them and this gets stored at the end of the record header (which is referred to as the payload for the operation in this document). For those familiar with the *\$MFT* file and the attributes associated with an *inode filerecord* (or *\$MFT* record), many of the operational names below will match those attribute names.

Code	Operation
0	Noop
1	CompensationLogRecord
2	InitializeFileRecordSegment
3	DeallocateFileRecordSegment
4	WriteEndOfFileRecordSegment
5	CreateAttribute
6	DeleteAttribute
7	UpdateResidentValue
8	UpdateNonResidentValue
9	UpdateMappingPairs
10	DeleteDirtyClusters
11	SetNewAttributeSizes
12	AddIndexEntryRoot
13	DeleteIndexEntryRoot
14	AddIndexEntryAllocation
15	DeleteIndexEntryAllocation
16	WriteEndOfIndexBuffer
17	SetIndexEntryVcnRoot
18	SetIndexEntryVcnAllocation
19	UpdateFilenameRoot
20	UpdateFilenameAllocation
21	SetBitsInNonresidentBitMap
22	ClearBitsInNonresidentBitMap
23	HotFix
24	EndTopLevelAction
25	PrepareTransaction
26	CommitTransaction
27	ForgetTransaction
28	OpenNonresidentAttribute
29	OpenAttributeTableDump
30	AttributeNamesDump
31	DirtyPageTableDump
32	TransactionTableDump
33	UpdateRecordDataRoot
34	UpdateRecordDataAllocation
35	UpdateRelativeDataIndex
36	UpdateRelativeDataAllocation

37	ZeroEndOfFileRecord
38	LastAction

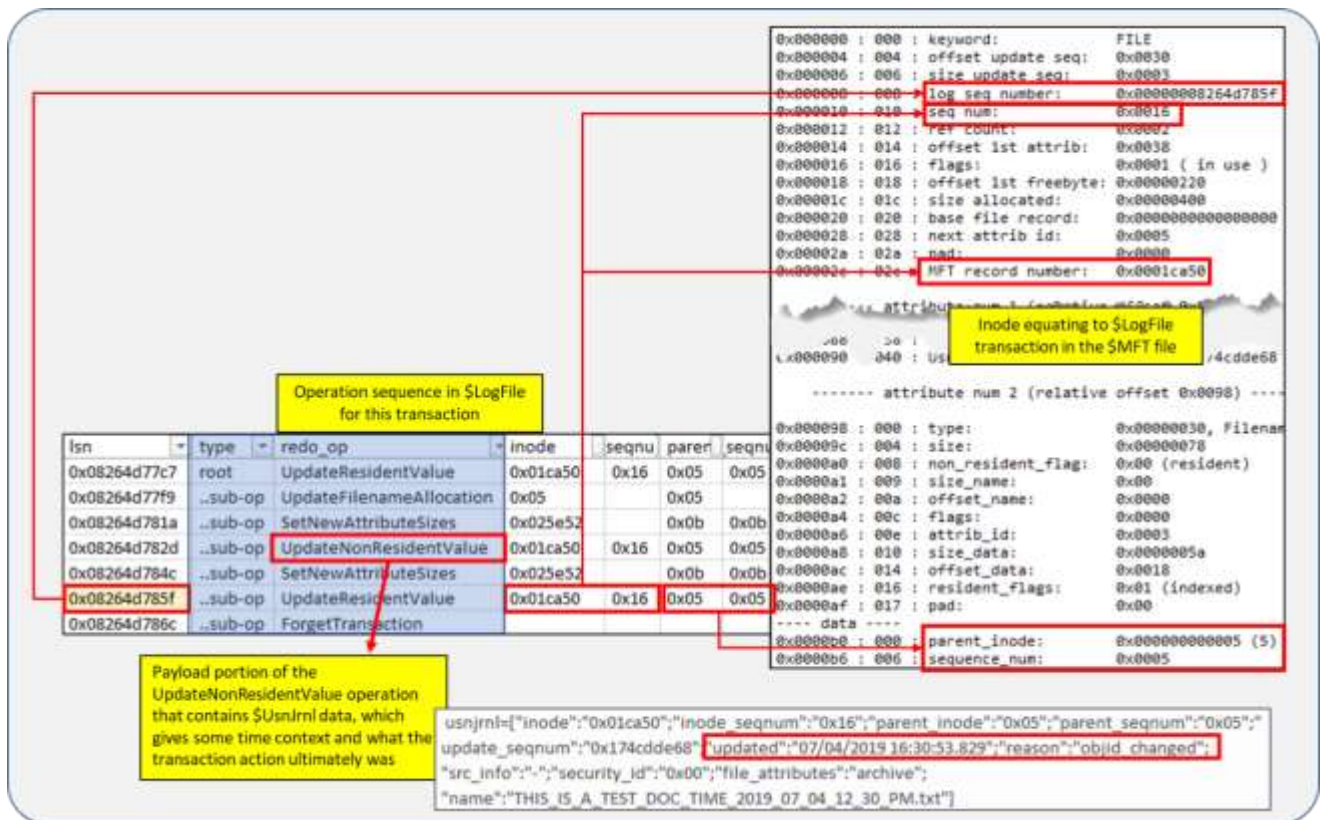
To date, some of the metadata associated with certain operations below have not been sufficiently analyzed for the ***mala*** tool to parse them. For those types, ***mala*** will report the operation but the payload data should appear as a sequence of hex bytes. The *code to operation* mapping shown in the table was obtained from various sources available on the Internet. In many cases, the *code to operation* was verified independently with internal testing during the development of this tool, however, a few others were not. This was because no empirical data could be obtained for some of the operation types. The warning here is that some of these entries may be incorrect.

2.3 Data Structure interdependencies

Using the portions of data structures from the *\$LogFile*, *\$MFT* and *\$UsnJrnl:\$J* artifacts, one can show various interdependencies between the structures used in the same artifacts. To show some of these relationships we took some data from our test system to provide an example. The images that follow are from the parsed data from each of the artifacts mentioned above.

Referring to the image below, on the far right is an inode record (or *\$MFT* record) with a record number of 0x1ca60 and a sequence number of 0x16. This record also has a Log Sequence Number (LSN) embedded into it which is 0x08264d785f. Moving to the next artifact output, which is in the middle, left of the image, is the output of one transaction that was parsed from ***mala***. The transaction consists of seven operations where the sixth operation relates has the same LSN referenced in the inode record. Specifically, one can see the inode and sequence number matches between the two artifact types. Finally, if one examines the payload from the fourth operation in the transaction, one can see a *change log journal* entry was made to the *\$UsnJrnl:\$J* file. The *change log journal* entry is complete with timestamp, inode number, sequence number, file name, and why the change was made. In this case the change refers to an Object ID change, presumably to document that the operating system created a LNK file for the target file.

From this brief example, one can see: (a) the *\$LogFile* contains a sequence of operations for each transaction, (b) the operating system then updates the *\$MFT* record to reflect the changes, and (c) the *change log journal* (*\$UsnJrnl:\$J*) is updated.



2.4 Log records and partial MFT attribute data

For each operation entry in the *\$LogFile* that points to an attribute modification, one will note that sometimes the payload will only contain a partial set of data for the attribute change. This is intentional since the *\$LogFile* is only recording delta changes within an attribute. However, the parser needs to be able to reconstruct which data in the attribute was changed and thus provide some context to the analyst.

To show an example of this in action, below are some images from a *\$LogFile* viewer that was created to help reverse the internal structures of the *\$LogFile* records. These screenshots are only shown in this user's guide to help explain some of the internals and the why the reported data produced by *mala* only shows portions of the data per operation.

In this example, we take the same operational log entry used in the previous example when showing the data interdependencies (eg. log sequence number 0x8264d785f). When breaking out the internals of log header and payload data structures, one can see that the operation is meant to update the Standard Information (StdInfo) attribute in the associated MFT record. (The header portion of the log entry is the top portion of the image, whereas the payload is the bottom portion of the image). When looking at the payload data, it skips the first 0x40 bytes of the StdInfo data and only addresses the last 8 bytes. The StdInfo structure, in this case, contains 0x48 bytes. These 8 bytes are the *change log record index*

number which identifies the *\$UsnJrnl:\$J* entry, which gets mapped directly into the appropriate StdInfo field of the inode record.

Header Info

```

0x26bc2f8 : 000 : m_current_lsn: 0x00000000264d785f [35002349663]
0x26bc300 : 008 : m_previous_lsn: 0x00000000264d784c [35002349644]
0x26bc308 : 010 : m_client_undo_lsn: 0x00000000264d784c [35002349644]
0x26bc310 : 018 : m_client_data_length: 0x00000038 [56]
0x26bc314 : 01c : m_client_id: 0x00000000 [0]
0x26bc318 : 020 : m_record_type: 0x00000001 [1] [General]
0x26bc320 : 024 : m_action_id: 0x00000040 [64]
0x26bc322 : 02a : m_pad: 00 00 00 00 00 00
0x26bc328 : 02a : m_redo_op: 0x0007 [7] [UpdateResidentValue]
0x26bc32a : 02c : m_undo_op: 0x0007 [7] [UpdateResidentValue]
0x26bc32c : 02e : m_redo_offset: 0
0x26bc32e : 030 : m_redo_length: 0
0x26bc330 : 032 : m_undo_offset: 0
0x26bc332 : 034 : m_undo_length: 0
0x26bc334 : 036 : m_target_attribute: 0
0x26bc336 : 038 : m_lcms_to_follow: 0
0x26bc338 : 03a : m_record_offset: 0
0x26bc33a : 03c : m_attribute_offset: 0
0x26bc33c : 03e : m_mft_entries_index: 0

```

Operation entry header

Sequence of operations for this transaction

Operation entry payload

Payload Redo data for opcode: 7 [UpdateResidentValue]

Translated output

```

0x26bc3e0 : 000 : time file creation: no data
0x26bc3e8 : 008 : time file altered: no data
0x26bc3f0 : 010 : time mft changed: no data
0x26bc3f8 : 018 : time file accessed: no data
0x26bc400 : 020 : file attributes: no data
0x26bc404 : 024 : max versions: no data
0x26bc408 : 028 : version num: no data
0x26bc40c : 02c : class id: no data
0x26bc410 : 030 : $Quota:$O key: no data
0x26bc414 : 034 : $Secure:$SII key: no data
0x26bc418 : 038 : quota charged: no data
0x26bc420 : 040 : UsnJrnl index num: 0x0000000174cdde68

```

Raw dump of Redo payload

```

026b c350: 68 de cd 74 01 00 00 00

```

No Payload data for these attribute fields

Only these 8 bytes are populated

As a second example, we continue to look at the same transaction, but look at one of the previous operations that lead up to the operation that was just discussed above. In this case, we select the *SetNewAttributeSizes* operation. This particular operation is applied to the actual *\$UsnJrnl:\$J* file that reflects the changes to the target file in question. This change only requires 32 bytes to record the data changes, even though the attribute for the header portion of the *\$Data* attribute is 72 bytes in this case. The data changes are the size modification changes for the *\$UsnJrnl:\$J* file to accommodate the new *change log record* that was added for the target file.

Header Info

```

0x26bc0d0 : 000 : m_current_lsn: 0x00000008264d781a [35002349594] [
0x26bc0d8 : 008 : m_previous_lsn: 0x00000008264d77f9 [35002349561] [
0x26bc0e0 : 010 : m_client_undo_lsn: 0x00000008264d77f9 [35002349561] [
0x26bc0e8 : 018 : m_client_data_length: 0x00000068 [104]
0x26bc0ec : 01c : m_client_id: 0x00000000 [0]
0x26bc0f0 : 020 : m_record_type: 0x00000001 [1] [General]
0x26bc0f8 : 028 : m_record_offset: 0x00000040 [64]
0x26bc0fa : 02a : m_redo_op: 0x000b [11] [SetNewAttributeSizes]
0x26bc100 : 02c : m_undo_op: 0x000b [11] [SetNewAttributeSizes]
0x26bc104 : 02e : m_redo_offset: 0x0000 [0]
0x26bc106 : 030 : m_redo_length: 0x0000 [0]
0x26bc108 : 032 : m_undo_offset: 0x0000 [0]
0x26bc10a : 034 : m_undo_length: 0x0000 [0]
0x26bc10c : 036 : m_target_attribute: 0x0000 [0]
0x26bc10e : 038 : m_lcn_to_follow: 0x0000 [0]
0x26bc110 : 03a : m_record_offset: 0x0000 [0]
0x26bc112 : 03c : m_attribute_offset: 0x0000 [0]

```

Operation entry header

Operation entry payload

Sequence of operations for this transaction

No Payload data for these attribute fields

Only these 32 bytes are populated

Translated output

```

0x26bc128 : 000 : type: no data
0x26bc12c : 004 : size: no data
0x26bc130 : 008 : non_resident_flag: no data
0x26bc131 : 009 : size_name: no data
0x26bc132 : 00a : offset_name: no data
0x26bc134 : 00c : flags: no data
0x26bc136 : 00e : attrib_id: no data
0x26bc138 : 010 : starting_vcn: no data
0x26bc140 : 018 : ending_vcn: no data
0x26bc148 : 020 : run_array_offset: no data
0x26bc14a : 022 : compression_flag: no data
0x26bc14b : 023 : pad (5 bytes): no data
0x26bc150 : 028 : size_data_allocated: 0x0000000174d30000
0x26bc158 : 030 : size_data_real: 0x0000000174cdd668
0x26bc160 : 038 : size_valid_data: 0x0000000174cddf08
0x26bc168 : 040 : size_compressed: 0x0000000025300000

```

Raw dump of Redo payload

```

026b c128: 00 00 d3 74 01 00 00 00 68 de cd 74 01 00 00 00 ...t...h..
026b c138: 08 df cd 74 01 00 00 00 00 00 53 02 00 00 00 00 ...t.....5

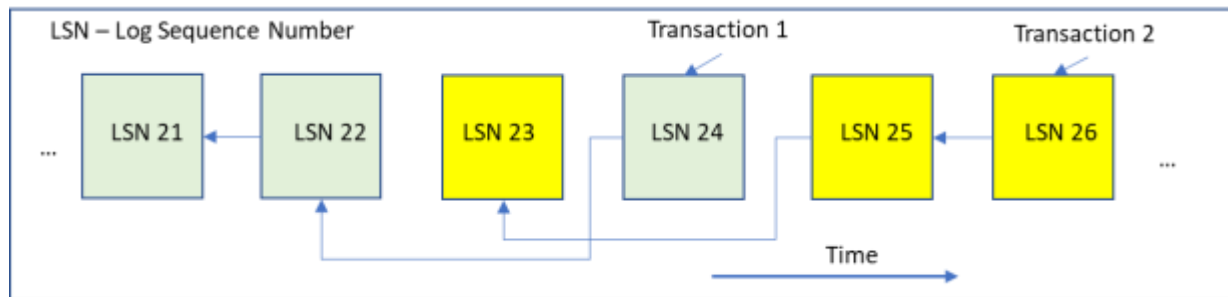
```

The intent of this document is not to delve into too much detail with these examples; it is only to show that the **mala** tool is able pull these partial data chunks and map them into the proper structure so the data can be reviewed in the proper context.

2.5 Transactions and operations

When looking at what NTFS calls a transaction and how it translates to the records in the *\$LogFile*, one sees that there are multiple operations that occur in sequence, and when combined together, are labeled a transaction. **mala** looks at all the records/operations and pieces together which records are chained together to form a single transaction. This is quickly identifiable, since each record has a pointer to the previous record in the chain for its transaction. This pointer while monotonic (always increases), doesn't necessarily mean that the next record is part of the current transaction. Thus, consecutive records can be interleaved between multiple transactions that are occurring (presumably for mutually exclusive transactions). This is shown below where transaction 1, includes log sequence numbers (LSNs) 21, 22, 24; and transaction 2 includes LSNs 23, 25, 26. (*note: the LSN numbers shown*

below are consecutive which is not normal with valid data; LSN numbers are incremented as a function of the size of the current record to compute the next LSN number; the consecutive sequence below is only shown to keep the numbers simplistic and easy to follow for this example).



When **mala** reports on these operations, it groups the appropriate ones into their own separate transaction's in the output so it is clear what the sequence of operations were per transaction.

2.6 Time is not explicitly recorded in the records

Unlike other artifacts used in forensics, the *\$LogFile* does not have a timestamp embedded into its normal record's data structure. This makes it difficult for the forensic analyst to try to correlate the time when a transaction occurred. However, one can infer time by looking at of the combination of records that comprise timestamp within its payload data. One can also infer time by looking at the inode records that match the *\$LogFile* records logical sequence numbers.

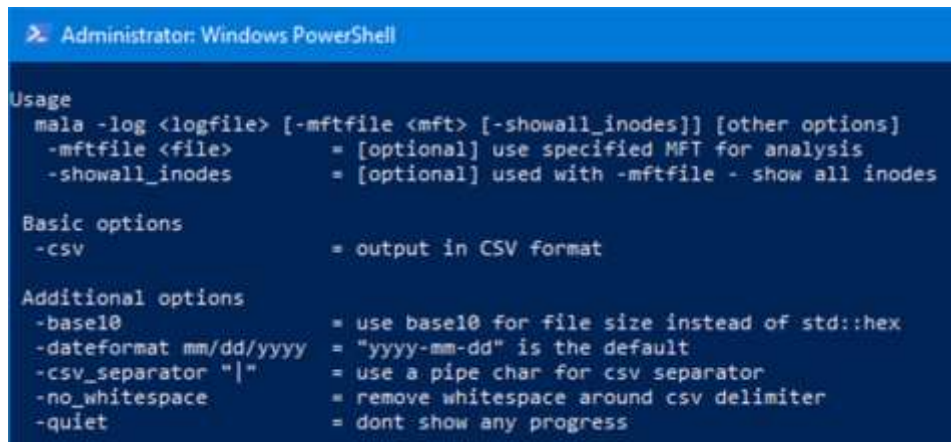
For example, if any of the records in a sequence contains a payload of *\$UsnJrnl* change log data, then one can parse that payload data and pull the timestamp embedded in the *change log record*. This is only because the *\$UsnJrnl* change log data has a timestamp as part of its metadata. If this metadata is contained within a *\$LogFile* record, then it can be extracted and parsed, giving the time inferred for the *\$LogFile* transaction. Further, if one parses the *\$MFT* file records in parallel with the *\$LogFile* records and if there happens to be an entry in the *\$MFT* file that matches the *\$LogFile* record in question, one can pull the latest timestamp from the *\$MFT* attributes. In this way, one can estimate the time the log entry was made.

To aid the analyst in this inference of time, **mala** tries to do this on a best effort basis and reports an 'extrapolated timestamp' for each record. Internally, **mala** keeps track of the last time that was reported (either via a *\$UsnJrnl* entry or a *\$MFT* entry) and reports that time for the next log sequence number (LSN) record reported. This extrapolated time is just a *guess* or *estimate*, even though the precision is shown in the 100 nano second resolution. Sometimes this estimate is very accurate when a *change log entry* was recorded and sometimes it is not very accurate at all. The latter case happens when a large number of log records have passed without a timestamp being observed to infer during the parse operation. Once a timestamp is found, the accuracy is again good until the next timestamp found.

As an indicator to the analyst, the more closely aligned timestamps are shown in the 'ref' field of the report.

3 How to Use *mala*

The current options for the *mala* tool are shown the screenshot below.



```
Administrator: Windows PowerShell

Usage
mala -log <logfile> [-mftfile <mft> [-showall_inodes]] [other options]
-mftfile <file>           = [optional] use specified MFT for analysis
-showall_inodes          = [optional] used with -mftfile - show all inodes

Basic options
-csv                     = output in CSV format

Additional options
-base10                  = use base10 for file size instead of std::hex
-dateformat mm/dd/yyyy  = "yyyy-mm-dd" is the default
-csv_separator "|"       = use a pipe char for csv separator
-no_whitespace           = remove whitespace around csv delimiter
-quiet                   = dont show any progress
```

The required syntax is to pass in a *\$Logfile* via the **-log <file>** option as shown below. The syntax below shows an **-out <file>** parameter, but one can redirect the output to any file as well. As a side note, if you are using Windows *PowerShell* instead of a command prompt, then we recommend the use of use single quotes around any path/filename that contain a '\$' character.

```
mala64 -log 'c:\test\$Logfile' -csv -out results.csv
```

To gain more context information, one can also pass in the companion *\$MFT* file via the **-mftfile <file>** option. The tool will merge the parsed *\$MFT* data into the *\$Logfile* data to generate more complete records.

```
mala64 -log 'c:\test\$Logfile' -mftfile 'c:\test\$MFT' -csv -out results.csv
```

When multiple artifact files are used in the analysis (eg. using both *\$LogFile* and *\$MFT* files), *mala* will spawn multiple threads to handle each artifact in parallel so as to parse the data quickly. The resulting output from each thread will then be combined into one output file.

3.1 Parsing with only the *\$LogFile* for analysis

While it is preferable to use both the *\$LogFile* and *\$MFT* artifact files to maximize the usefulness of the report generated by *mala*, some *use-cases* involve corrupted *\$MFT* files. If the corruption of the *\$MFT* file is sufficient to prevent the *mala* tool from parsing it, then running *mala* with just the *\$LogFile* still provides useful, albeit, degraded results. To explain this in more detail, one needs to analyze the

differences in the reporting of running **mala** with just using the *\$Logfile* to that of running it in conjunction with the *\$MFT* file. The main issues without using the *\$MFT* file in the analysis are: (a) the *path* for the target file being created, changed or deleted is not present; and (b) any logical sequence number matches between the *\$Logfile* records and *\$MFT* records are not available which consequently results in less possible context data in the output. There are other artifact data that is lost as well, but the ones listed above are the main ones.

Aside from these issues, parsing the *\$Logfile* by itself is still useable for those cases where the *\$MFT* file is not present. Why? The *\$Logfile* has embedded into its records the *\$UsnJrnl:\$J* change log entries that were done. More specifically, the *\$Logfile* has an entry for each change recorded in the *\$UsnJrnl:\$J* as part of a transaction since the change log journal is a file as well. Since *\$UsnJrnl:\$J* entries are preserved in the *\$Logfile* records, one can extract and parse these records which has meaningful data, even though the path of the file is still missing.

3.2 Reporting

The output generated by **mala** is in a delimited format where the delimiter can be either a comma (CSV format), pipe, or tab character. So as to limit the number of fields and provide uniformity across different operations, the last field is a *quasi-json* format that allows the tool to use a condensed notation and be extensible so as to allow for an unlimited combination of data types. In this way dissimilar data can be concisely put into a format that is easily digested by a spreadsheet program (like excel) or into a database. Most of the data put into this 'catch-all' column is the payload data associated with the operation, as well as, any supporting information provided by the *\$MFT* file (if available).

Below are the delimited fields that are included in the reporting:

Field	Meaning
extrapolated_timestamp	Internally, mala keeps track of the last time that was reported (either via a <i>\$UsnJrnl</i> entry or a <i>\$MFT</i> entry) and reports that time for the next log sequence number (LSN) record reported. This extrapolated time is just a <i>guess</i> or <i>estimate</i> .
ref	Indicator when the time was updated based on explicit timestamp data in the payload or a referenced <i>\$MFT</i> record
change_reason	Relates to the <i>\$UsnJrnl</i> entry embedded into one of the operations associated with that transaction, or if the <i>\$UsnJrnl</i> entry is not available it is derived from the type of operation.
lsn	Log Sequence Number
type	Specifies whether it is the start of the transaction or one of the operations in the transaction
op_pattern	Operational code (or sequence of codes if the initial start of the transaction)
redo_op	The translated operation name for the redo operation (doesn't show the undo operation).
target_lcn	Logical Cluster number of the target that is affected
inode	MFT record entry. Combination of either (a) explicitly listed inode or (b) computed based on the offset, cluster size, and MFT record size
inode_seqnum	MFT record entry sequence number. From data that explicitly listed the sequence

	number
parent_inode	Parent MFT record entry. From data that explicitly listed this.
parent_seqnum	Parent MFT record entry sequence number. From data that explicitly listed this.
path	Relies on the \$MFT file to build the absolute path.
comment	General purpose field that displays the parsed payload data and/or other data from support files.

For the *quasi-json* formatted data, there are some keywords used. The main ones are listed below. The purpose of using keywords is to try to group *like-data* segments from various sources so as to allow one to have more insight as to where they came from. For example, any data that comes from a \$MFT supporting file, will be preceded by the 'mftfile' keyword. Likewise, if the data came from a \$UsnJrnl:\$J data, it would be preceded by the 'usnjrnl' keyword. Sometimes the payload data is truncated, which can be derived when the size of the payload disagrees with the actual number of bytes left before the start of the next log record. For these cases, the 'data_truncated' keyword is used and the data is parsed to the extent possible given that it was truncated.

Keyword	Meaning
open_record	Relates to the payload data associated with the OpenNonresidentAttribute operation
file_record	Relates to the payload data associated with the InitializeFileRecordSegment operation
Name is derived from one of the MFT attributes	Partial data within the payload data that can affect any of the MFT attributes, including: \$filename, \$stdinfo, \$indx_direntry, \$data, etc
usnjrnl	Contains a \$UsnJrnl:\$J record embedded in the payload data
hex_bytes	Contains an unparsed series of bytes in the payload data
cluster_run_data	Contains a cluster run embedded in the payload data
bitmap_set	Relates to the payload data associated with the SetBitsInNonresidentBitMap operation
set_size	Relates to the payload data associated with the SetNewAttributeSizes operation
mftfile	Comes directly from any inode data from a separate \$MFT file. If listed for an operation, it directly relates to the operation log entry
metadata	This is the metadata associated during the parsing of the operation record.
data_truncated	This relates to the operation's record reference to the payload data and the fact that the size for the payload doesn't reflect the number of bytes present. Some payload data exists, but the data is truncated.

Below is a sample output showing an entire transaction and the relationship of the normal fields and the comments field.

extrapolated_timestamp	ref	change_ref	lsn	type	redo_op	target	path	comment
07/04/2019 16:30:53.758		file_created	0x08264d7103	root	SetBitsInNonresidentBitMap	0x0		bitmap_set=["hex_bytes":"50-4a-00-00-01-00-00-00"]
07/04/2019 16:30:53.758		file_created	0x08264d710f	..sub-op	Noop	0x0	[root]\THIS_IS_A_TEST_DOC.txt	mftfile=["filename":"THIS_IS_A_TEST_DOC.txt","path":["root"]]
07/04/2019 16:30:53.758		file_created	0x08264d713c	..sub-op	AddIndexEntryAllocation	0x0	[root]\THIS_IS_A_TEST_DOC.txt	\$indx_direntry=["inode":"0x01ca50","inode_seqnum":"0x13"]
07/04/2019 16:30:53.758		file_created	0x08264d7155	..sub-op	InitializeFileRecordSegment	0x0	[root]\THIS_IS_A_TEST_DOC.txt	file_record=["inode_seqnum":"0x8264d710f","inode":"0x01ca50"]
07/04/2019 16:30:53.758		file_created	0x08264d719d	..sub-op	SetNewAttributeSizes	0x0	[root]\\$Extend\Usnjrnl	set_size=["allocated":"0x174d30000","real":"0x174cdd8e8"]
07/04/2019 16:30:53.758	-c usnjrnl	file_created	0x08264d71b0	..sub-op	UpdateNonResidentValue	0x0	[root]\THIS_IS_A_TEST_DOC.txt	usnjrnl=["inode":"0x01ca50","inode_seqnum":"0x13","parent_inode":"0x08264d710f"]
07/04/2019 16:30:53.758		file_created	0x08264d71cf	..sub-op	SetNewAttributeSizes	0x0	[root]\\$Extend\Usnjrnl	set_size=["allocated":"0x174d30000","real":"0x174cdd8e8"]
07/04/2019 16:30:53.758		file_created	0x08264d71e2	..sub-op	ForgetTransaction	0x0		free log entries for this sequence

If we expand the comments field for the operation that contains the embedded \$UsnJrnl data, the parsed data is shown below prefixed by the "usnjrnl" keyword. The output also includes the "mftfile" keyword for the filename and path information.


```
usnjrnl=["inode":"0x01ca50","inode_seqnum":"0x15","parent_inode":"0x05","parent_seqnum":"0x05";  
"update_seqnum":"0x174cdd8e8","updated":"07/04/2019 16:30:53.758","reason":"file_created";  
"src_info":"-","security_id":"0x00","file_attributes":"archive","name":"THIS_IS_A_TEST_DOC.txt");  
mftfile=["filename":"THIS_IS_A_TEST_DOC.txt","path":["root"]\THIS_IS_A_TEST_DOC.txt"]
```

4 Pulling Artifacts off a Live System

The raw artifact files used by **mala** (eg. *\$LogFile* and *\$MFT*) are locked down if trying to access them from the running system. One solution is to look to other tools to copy the appropriate artifact files. If you are on a Windows machine, one can use the TZWorks' tool **dup** (Disk Utility and Packer). It will allow one to copy a file, or an entire directory, even if some of the files are locked down by the operating system. To use **dup** to target the system files used by **mala**, one could use the following command:

```
dup -copygroup -pull_sysfiles -out <results folder>
```

The above command will also pull other system files not needed by **mala**, but all the files used by mala will be extracted.

5 Available Options

Option	Description
-log	Specifies which <i>\$LogFile</i> to act on. The syntax is: -log <file>
-mftfile	Use the specified <i>\$MFT</i> file for <i>\$LogFile</i> analysis. The syntax is: -mftfile <file> . There is a sub-option [-showall_inodes] to display all the inodes in the output
-csv	Outputs the data fields delimited by commas. Since filenames can have commas, to ensure the fields are uniquely separated, any commas in the filenames get converted to spaces.
-base10	Ensure all size/address output is displayed in base-10 format versus hexadecimal (base-16) format. Default is hexadecimal format.
-no_whitespace	Used in conjunction with -csv option to remove any whitespace between the field value and the CSV separator.
-csv_separator	Used in conjunction with the -csv option to change the CSV separator from the default comma to something else. Syntax is -csv_separator "/" to change the CSV separator to the pipe character. To use the tab as a separator, one can use the -csv_separator "tab" OR -csv_separator "\t" options.

-dateformat	Output the date using the specified format. Default behavior is <i>-dateformat "yyyy-mm-dd"</i> . Using this option allows one to adjust the format to mm/dd/yy, dd/mm/yy, etc. The restriction with this option is the forward slash (/) or dash (-) symbol needs to separate month, day and year and the month is in digit (1-12) form versus abbreviated name form.
-quiet	Show no progress during the parsing operation.
-utf8_bom	All output is in Unicode UTF-8 format. If desired, one can prefix an UTF-8 <i>byte order mark</i> to the CSV output using this option.

6 Authentication and the License File

This tool has authentication built into the binary. The primary authentication mechanism is the digital X509 code signing certificate embedded into the binary (Windows and macOS).

The other mechanism is the runtime authentication, which applies to all the versions of the tools (Windows, Linux and macOS). The runtime authentication ensures that the tool has a valid license. The license needs to be in the same directory of the tool for it to authenticate. Furthermore, any modification to the license, either to its name or contents, will invalidate the license.

7 References

1. NTFS Log Tracker, blueangel, forensic-note.blogspot.kr, JungHoon Oh briefing charts, <http://forensicsinsight.org/wp-content/uploads/2013/06/F-INSIGHT-NTFS-Log-TrackerEnglish.pdf>
2. NTFS.com, NTFS Transaction Journal [<https://www.ntfs.com/transaction.htm>]
3. G-C Partners, File System Journal Analysis, David Cowen and Matthew Seyer and ANJPv3.11.07_FE.exe tool
4. LogFileParser, <https://github.com/jschicht/LogFileParser>
5. Windows Internals, Microsoft Press