SANS | GIAC CERTIFICATIONS

WHITE PAPER

Mastering the Super Timeline With log2timeline

Kristinn Guðjónsson

Mastering the Super Timeline With log2timeline

GIAC (GCFA) Gold Certification

Author: Kristinn Guðjónsson, kristinn@log2timeline.net
Advisor: Charles Hornat

Accepted: June 29, 2010

Abstract

Traditional timeline analysis can be extremely useful yet it sometimes misses important events that are stored inside files or OS artifacts on the suspect. By solely depending on traditional filesystem timeline the investigator misses context that is necessary to get a complete and accurate description of the events that took place. To achieve this goal of enlightenment we need to dig deeper and incorporate information found inside artifacts or log files into our timeline analysis and create some sort of super timeline. These artifacts or log files could reside on the suspect system itself or in another device, such as a firewall or a proxy. This paper presents a framework, log2timeline that addresses this problem in an automatic fashion. It is a framework, built to parse different log files and artifacts and produce a super timeline in an easy automatic fashion to assist investigators in their timeline analysis.

1. Introduction

1.1. Background

Timeline analysis is a crucial part of every traditional criminal investigation. The need to know at what time a particular event took place, and in which order can be extremely valuable information to the investigator. The same applies in the digital world, timeline information can provide a computer forensic expert crucial information that can either solve the case or shorten the investigation time by assisting with data reduction and pointing the investigator to evidence that needs further processing. Timeline analysis can also point the investigator to evidence that he or she might not have found using other traditional methods.

The focus of traditional timeline analysis has been extracting and analyzing timestamps from the filesystem that stores the digital data. Although not all filesystems store the same timestamp information about files they usually have some timestamps in common such as information about the file's last access and modification time. Some filesystems also store information about the file's deletion or creation time or even the modification time of the file's metadata. Metadata can be described as data about data, or in other words data that describes or otherwise adds information to the actual content. In filesystem context this is usually information about the name of the file, location of the data blocks inside the filesystem as well as information about the parent folder that stores the data. Since the SANS forensics course, 508, explains in detail how to create and analyze a traditional filesystem timeline there will only be light references to such analysis in this paper.

1.1.1. Problems with Traditional Timeline Analysis

Although traditional timeline analysis can provide the investigator with great insights into the events that took place on a suspect drive, it does not come without its problems. One of which is the frequent modification of timestamps during what can be called "normal" user or operating system behavior. Timestamps, such as the last access time of file, might be updated during activity such as when an antivirus product scans the hard drive searching for malware. This leads to quickly degrading value of the

information that those timestamps provide. Another problem is that some operating systems do not update last access time for performance reasons. Examples of such operating systems are the latest ones from Microsoft, Vista and Windows 7. Another option is to turn off the updating of the last access time with a simple operation. This can be achieved in Windows by editing a single registry key, that is to either altering or creating a DWORD entry called NtfsDisableLastAccessUpdate with the value of 1 in the key HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem (NtfsDisableLastAccessUpdate, 2010). There is an option to the mount command in Linux to disable the updates of last access time, called the noatime, which can be made permanent by modifying the fstab. Disabling the last access time is often suggested in various optimization guides, since it can increase performance of the filesystem (How-To-Geek, 2008) (mindstate, 2009).

These are not the only issues with traditional timeline analysis. Another important shortcoming is the fact that relying solely on filesystem timestamps lacks context that might only be written inside log files or found within the file's metadata. Last write time for /var/log/messages or the security event log for instance has little meaning since these files are constantly written to by the underlying operating system. The true value in these files is found within the actual content, which contains time stamped data that might of relevance to the investigation. Including the content of these files adds important information to the investigation and might even discover evidence that otherwise had been missed during traditional forensic analysis.

Timestamps in general all share the same inherent problem, which is the volatile nature of the evidence. In the digital world, often only the last occurrence of a particular event is stored. This might not cause any problems if the investigator arrives at the scene immediately after the alleged crime or event of interest took place. However as time passes the likelihood that some timestamps of interest get overwritten increases, making timeline analysis often less reliable or even misleading.

1.1.2. Anti-forensics

The fact that the timestamps are stored in a digital media creates another dilemma. The fact that all digital data can be altered, even timestamps, makes timeline analysis

often vulnerable to attacks. Anti-forensics tools such as timestomp (Vincent, Liu, 2005) have been created specifically to alter timestamp data to trick or mislead forensic investigators. Timestomp is designed to alter timestamps in the NTFS filesystem, such as the file's last access, modification, entry modified and creation time (MACB). Similar tool to timestomp is the old *NIX tool touch. Other tools exist that have the ability to modify metadata information found inside some files.

Some anti-forensics techniques involve modifying magic values in files to fool forensics tools and to inject false information into log files, or even to include regular expressions in log files to crash forensic software. The possibility of anti-forensics makes it vital that the investigator has access to information gathered from multiple locations to verify his or hers results as well as the insights and training needed to spot the anomalies that such techniques usually introduce to the system.

1.1.3. Extending the Timeline

One of the solutions to the shortcomings of traditional timeline analysis is expanding it with information from multiple sources to get a better picture of the events that took place as well as to minimize the impact of anti-forensic techniques. By incorporating timestamps from sources that do not have any methods to modify the timestamps, such as an API or any other mechanism, into the timeline provides valuable information to the investigator.

Commercially available forensic software has not yet incorporated super timelines into its toolkits. Part of the reason might be the shear complexity of including such information or that the power of such analysis has not yet reached their eyes. Timestamps are stored using various formats inside files, making it more difficult to extract them all using a single tool. A tool that extracts such timestamps needs to be capable of parsing various artifacts; both binary and text based ones as well as understand the various formats that timestamps are stored and modify them to a single one for representation.

Many document formats includes metadata that describes both the content of the document as well as actions taken, such as the last time the document was printed, modified and who modified it. Including the information found in the metadata of these documents can provide vital information in the investigation. The last time a document

was printed or the author that modified it might very well be the evidence that the investigator needs to solve the case in hand. Registry files in the Windows operating system also include useful information to include in timeline analysis, such as last write time of keys in the UserAssist, entries that contain information about the computer behavior of the registry file's owner. There are several other important files that contain timestamp information, such as browser history files, instant-messaging logs, anti virus logs, emails and various other files scattered throughout the operating system.

By extending the timeline into a super timeline with the inclusion of these artifacts; that is entries in log files, metadata information from documents as well as timeline information from the Windows registry provides the investigator with important information and context around events that are not available in the traditional filesystem timeline. It also works as an alternative method to fight anti-forensics with the inclusion of timestamp information from multiple sources. Additionally if timestamps from other sources, such as a proxy or a firewall, sources the suspect or an attacker does not have access to can provide vital information to the case as well as to corroborate other timestamp information from the suspect drive.

The creation of a super timeline has the possibility to shorten the investigation time considerably. By analyzing the super timeline the investigator gets a very good overview of the suspect drive and has the possibility to quickly find the necessary evidence that needs further analysis. There aren't many reliable data sources that explicitly show the advantage of using a super timeline versus traditional analysis yet, however a questionnaire conducted by Olsson and Boldt about the usage of their tool CyberForensicsTimeLab showed that their test subjects solved a hypothetical case that was focused on behavior during a particular timeframe was solved considerably faster than using a traditional named brand forensic application, that is in 14 minutes versus 45 minutes (Olsson, Boldt, 2009).

The amount of data collected in a super timeline can be overwhelming to an investigator especially when in the vast majority of cases there are only a handful of entries in the timeline that the investigator is looking for. That is the relevant timeline entries are often said to be few straws in a very large haystack. There is a great need to

create a tool that can both extract all available timeline data in an easy manner and create means to easily reduce the data set to speed up the investigation.

Timeline analysis is and has always been a very manual process since each investigation is usually completely different than the previous one. That is the relevant entries are almost never consistent between two different investigations; entries that are completely irrelevant to one investigation might be the only entries that matter in the next one. This makes creating a single tool that reduces the data set in an automatic fashion very difficult and has forced manual investigation of the timeline in each case. There is a great need to create a tool that can either eliminate some static or irrelevant entries from the timeline or at least to create a mean to easily filter out those entries that are not part of the current investigation and add a powerful search to find those that are relevant. The tool would need to have the capability to search for an event that contains some or any words from a "dirty" word list as well as being able to limit the search within a particular timeframe. Visual representation of the timeline can also be a very valuable aspect in such a tool, making it both easier to read the timeline after filtering and data reduction has taken place and to make reports more appealing and understandable to the investigator and the non technical people who are often the receiver of forensic reports.

1.2. Prior Work

Extraction of temporal data from hard drives is not a new concept. The tool mactime and grave-robber first appeared with the collection of programs by Dan Farmer and Wietse Venema called the Coroner's Toolkit (TCT) in 1999. Grave-robber collected filesystem timestamps from a mounted filesystem to produce a bodyfile that could be read by the tool mactime. Brian Carrier further developed these tools among others and mactime is now part of the Sleuthkit (TSK) while mac-robber (former grave-robber) is available as a stand-alone tool from TSK web site. TSK also includes other tools to grab filesystem timestamps, such as *fls* and *ils* that read image files and extract filesystem timestamps from either deleted inodes (*ils*) or from the filename layer of the filesystem (*fls*). These tools also produce a bodyfile in the mactime format. Rob Lee also wrote mac-daddy, a tool that is based on grave-robber which is similar to mac-robber except written in Perl. Most of the other forensic software, such as Encase, FTK and others are also

capable of extracting the filesystem timestamp from an image file however they have not added any capability to extract timestamp information from artifacts found on the filesystem.

The notion of writing a tool for extended timelines has been discussed every now and then for few years, yet there are few projects that have tried implementing those ideas. Most of these attempts have been focused on creating separate scripts that each parses only a single file format. Although some of these tools are of great quality their usage is limited in the sense that they need lot of manual work to be used properly and they do not provide the same flexibility as a framework does. There has been at least two attempts to create a framework not unlike the proposed one in this paper, Mike Cloppert wrote a SANS Gold paper called Ex-Tip which describes a suggested framework for extended timelines (Cloppert, 2008) yet unfortunately the tool has not been maintained since it's first release in May 2008. Don Weber wrote the tool System Combo Timeline (Weber, Don, 2010) in Python that already supports parsing several artifacts, a project that is still active.

Harlan Carvey has also created several individual scripts designed to extract timestamps from artifact or log files, each one outputting its data in his own format, called TLN.

Other tools have focused on visually representing the timeline data, tools such as the CyberForensics TimeLab (CFTL) (Olsson, Boldt, 2009) a tool that still has not been released to the public. CFTL consists of a GUI application that can read an image file and parse several artifacts found within it to produce a XML document that is then read by the GUI that visually represents the data. Zeitline "is a tool that lets a user import events from various sources and manage those events in one or more timelines" (Buchhokz, Florian, 2005) according to the author of Zeitline. Although promising, Zeitline has not been updated sine June 2006.

2. The log2timeline Framework

2.1. Why a framework

There are several benefits of creating a framework instead of writing separate scripts to extract timeline information from log files or artifacts. A framework can address the many problems that arise when creating separate scripts for each parser, such as repetition of code. What tends to happen with writing specialized scripts is that there might not be any consistency in the usage of it, that is that the parameters to the scripts are often different depending on the author and time it was written, requiring the investigator to know the functionality of each script. The final problem lies in inability of the scripts written so far to recursively search through a directory to find artifacts that the script is able to parse, forcing the investigator to manually locate each file of interest.

The creation of a framework makes development of new parsers considerably easier. The framework separates the functionality of the tool into independent modules that can be written without consideration about other parts of the script, such as parameter parsing or output capabilities. This in turns makes the development of new modules more focused. Another benefit of separating the functionality is that each created module can be used instantly with all portion of the framework. The modular approach also introduces the ability to build into the framework a recursive scanner without the need to rewrite all the parsers. With each new parser added to the framework it is automatically added to the recursive scanner.

A framework design forces all development of modules to follow certain guidelines to fit into the tool, which can either make development easier or create constraints on the developer.

2.2. Framework Structure

The framework being described here reflects to version 0.5x of the log2timeline framework. It is built into four main modules; a front-end, shared libraries, an input module and an output module.

Each module is independent of each other and can be developed separately. This makes it easy for other developers to contribute to the project by developing a module.

The front-end takes care of processing parameters, loading other modules and controlling the flow of operation. The shared libraries contain functions that are used by more than one module and are created to avoid repeating code segments. The input module takes care of parsing the artifact, whether that by a file or a directory, and prepare a timestamp object that is then used by the output module to print a file containing either a timeline or a file that can be imported directly into a tool that creates timelines, whether that be visually or textually.

The timestamp object is implemented as a hash in Perl. "A hash is an unordered collection of values, each of which is identified by a unique key" (Menon-Sen, Abhijit, 2002). The timestamp object, called internally *t_line*, contains several values that are pertinent to the artifact being parsed. Table 2 explains the current structure of the *t_line* hash and includes all the available keys and what they store. The input module creates this timestamp object and most of the keys within it. Since timestamps are stored using various methods within the operating system all timestamps are normalized to a single format, and in the case of log2timeline the chosen format is Epoch time, or seconds since January 1, 1970 stored in UTC. This makes defining the local time zone very important when running the tool.

After parsing the artifact the input module returns a reference to the timestamp object to the main module that further processes it, adjusts time offsets of the artifact and inserts other information before passing the reference to an output module that in turn parses the structure to form a line it can print, either to screen or to a file.

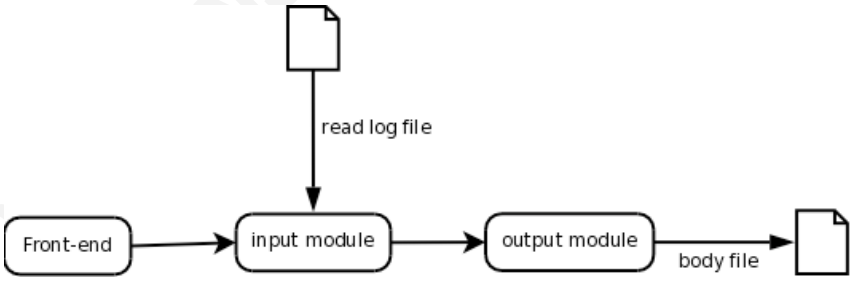

Figure 1: A simple flow chart of the structure of log2timeline

Figure 1 shows a simple flow chart that explains the flow between different modules of log2timeline. The front-end prepares the framework by parsing parameters,

loading the input module that takes care of parsing the log file or artifact and then to invoke the output module to print out a timeline or a body file.

2.3. Available Front-Ends

There are currently three front-ends that are available in the framework; log2timeline, the main command line front-end, glog2timeline, a GUI front-end and timescanner which is the automation part of the framework or a recursive scanner that extracts timeline from every file and directory that the input modules are capable of parsing.

Each of them will be further described in the subsequent chapters.

2.3.1. Log2timeline, Command Line Front-End

Log2timeline is the main front-end of the framework and contains the most features. It is like all other parts of the framework written in Perl.

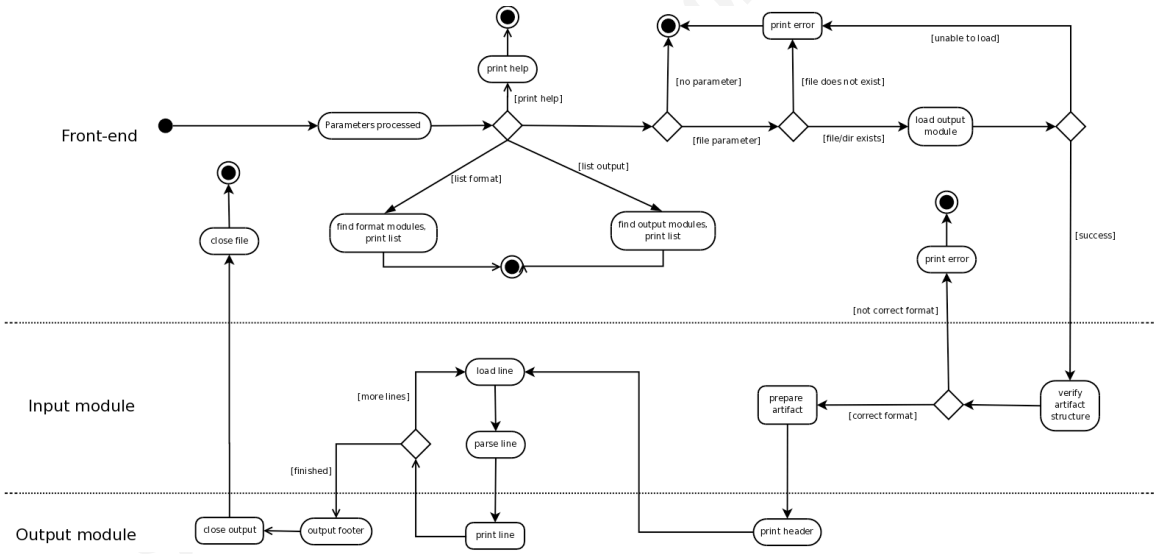

Figure 2: UML chart showing the process of log2timeline, main CLI front-end

Figure 2 shows a UML chart that depicts the process of parsing an artifact as it is done by log2timeline's front-end. The first phase of operation is parameter processing. If the parameters do not cause the tool to stop (help file shown, output or input modules listed, version check, etc.) the tool verifies that the artifact exists. If it does, that is the file or the directory exists, then both the input and output modules are loaded. If the modules load successfully the artifact is then sent to the input module for verification.

Upon successful verification it is prepared for parsing before sending the flow back to the output module to print the header, if one is needed. The input module then loads each line of the artifact and parses it. After each line is parsed a reference to a hash that contains the timestamp object is sent to the output module. Finally after parsing the last line all open file handles to the artifact are closed and the output module prints footer if needed.

2.3.2. Glog2timeline, Graphical Interface

This is the graphical user interface (GUI) counterpart of log2timeline. The purpose of glog2timeline is to provide a simple GUI that can be used to extract timestamp information from an artifact found on a suspect drive. Glog2timeline works in the same manner as the CLI counterpart, log2timeline, except that it uses a graphical interface to read in all the parameters.

Figure 3: Shows the glog2timeline, or the GUI front-end to the framework

The GUI is written in Perl just as other parts of the framework using the Glib and GTK2 libraries, making it only available on a X11 system (such as *NIX or a Mac OS X with X11 installed).

2.3.3. Timescanner, A Recursive Command Line Scanner

The third front-end of log2timeline is called timescanner. The purpose of timescanner is to load up all the selected input modules and search recursively through

the user-supplied directory parsing every file and directory that the input modules are capable of.

It follows the same flow of operation as the main front-end, log2timeline, except that instead of loading a single input module it loads the ones that are selected, which is by default all of them. It also only accepts a directory as its input instead of either a file or directory depending on the chosen input module, as is the case of the other front-ends. The tool then reads the content of that directory and recursively goes through it and tries to verify the structure of each file and directory found within the supplied directory against the entire set of input modules.

The way that input modules are selected into the tool is through the -f switch. The user has a choice of either supplying a comma separated list of all the input modules that should be run, a list of the ones that should not be run or the name of a predefined list file containing the names of the chosen input modules.

This front-end provides the investigator with an automatic tool to collect all available timestamp information from an image file in an easy manner. The investigator simply needs to mount the image file and point timescanner to the mount point as well as to supply the tool with the time zone settings of the suspect drive. Timescanner can also be used to quickly extract all timestamp information from a log file directory by pointing the tool to it.

2.4. Input Module Structure

To understand the structure of the input modules it is best to examine the purpose of the module in the combined flow of operation for the tool.

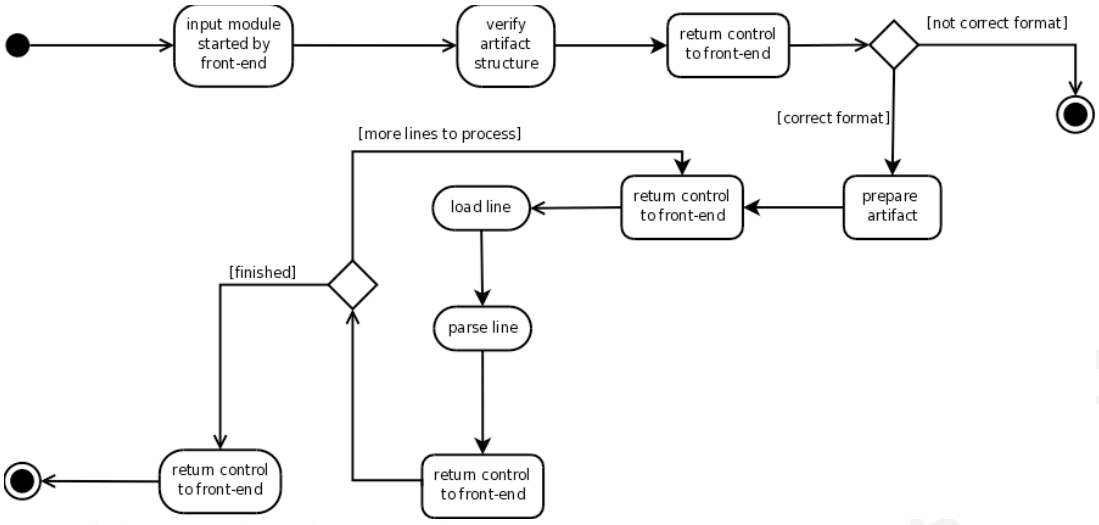

Figure 4: Shows the flow of operation for an input module

Figure 4 shows how the input module takes care of parsing each input file. There are four functions that every input module has to implement to function within the framework.

- **Verify artifact structure.**

  This functionality is implemented by the subroutine *get_version()*. The routine takes as an input the name of the file or directory to check and then examines it to verify if it contains the correct structure that the input module is able to parse and extract timestamps from.

  It is very important to properly validating the file's structure before continuing to ensure the input module does not try to parse a file it was not built to do. This could cause the tool to die or exit with an error and not completing its task. This is especially true when timescanner, the recursive scanner, is used since it will scan every file and directory in the supplied folder and try to parse it.

  Timescanner is heavily depended upon the validation function to eliminate the need to parse files it is not capable of.

  The verification process needs to be optimized to minimize the performance impact since timescanner calls this function for each input module on every file and directory. If the verification process is not optimized it will have great impact on the tools overall performance.

  The verification process returns a reference to a hash to the main module that

contains two entries. An integer indicating whether or not the artifact could be verified and a string containing a message on why the file did not pass the verification phase (or a string indicating a success).

- **Prepare the artifact.**

  This functionality is implemented by the subroutine *prepare_file()* that takes as an input the file name of the file or directory to be parsed as well as an array containing the parameters passed to the input module. Since the artifact has already been verified the preparation phase takes care of preparing the file for parsing. It also takes care of parsing the parameters passed to the input module to further configure the input module itself.

  The purpose of this functionality differs between input modules, depending if the artifact to be parsed is a simple log file or a complex binary file. In some cases the preparation phase consists solely of opening a file handle to a text file and in other cases it consists of parsing the structure of the file and create an object (Perl hash) containing each timestamp entry that is further processed in later phases.

- **Load line.**

  This functionality is implemented by the subroutine *load_line()* that takes care of verifying if there are unparsed timestamp entries in the artifact. The routine notifies the main module whether there are any timestamps left to parse and either loads the next line into a global variable found within the input module or increments a global index variable into a hash that contains each timestamp entry found within the artifact.

- **Parse line.**

  This functionality is implemented by the subroutine *parse_line()*. The main module calls this routine once for each new timestamp entry. The routine then takes care of either reading a line in a log file that has been loaded inside a global variable or reading from a hash or an array that contains the timestamp information.

  The main purpose of the *parse_line* subroutine is to create some sort of timestamp object to pass to the main module. The timestamp object consists of a Perl hash

that contains all the needed information to output the value in any way that the output module is designed to do. The subroutine *parse_line* creates this hash and then returns a reference to it, which is used by the main module and the output module for further processing.

The timestamp object, or the hash *t_line* is structured in the following way by the *parse_line()* routine.

```
time
->    index
      ->    value
      ->    type
      ->    legacy
desc
short
source
sourcetype
version
[notes]
extra
->    [filename]
->    [host]
->    [user]
->    […]
```

A description of each field is further explained in the tables below.

Name	Description
time	This is a reference to a hash value that stores the actual timestamp information. The hash itself is structured in the following way: index -value - type - legacy Where the "value" field contains the actual Epoch timestamp. Type is a short description of the timestamp value, such as "Last Written", "Modified", etc. And finally the legacy field is a binary representing the MACB value that this timestamp gets assigned. The legacy field is a binary value (0b0000) where each bit in the number

<table>
<tr><td></td><td>represent a letter in the MACB.
<table><tr><th>B</th><th>C</th><th>A</th><th>M</th></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>An example value of the field is 12 in decimal, or 0b1100, meaning that the event should be categorized as [..CB].</td></tr>
<tr><td>desc</td><td>This variable contains the description of the timestamp object (extracted and prepared text).</td></tr>
<tr><td>short</td><td>This variable contains the shorter version of the description of the timestamp object.</td></tr>
<tr><td>source</td><td>The short acronym for the source of the timestamp data. Such as WEBHIST for every file that has to do with web history, REG for registry, etc.</td></tr>
<tr><td>sourcetype</td><td>Longer description of the source. For instance “Opera Browser History” instead of only “WEBHIST”, “SOFTWARE Registry” instead of “REG”, etc.</td></tr>
<tr><td>version</td><td>The current version of the timestamp object. The version of the timestamp object used by version 0.5x of log2timeline is version 2.</td></tr>
<tr><td>[notes]</td><td>An optional field that contains any notes about the timestamp object. This can be either filled automatically by the tool or done manually using other analysis tools later on the process.</td></tr>
<tr><td>extra</td><td>A reference to a hash that stores all additional fields that might be used. All information that is input module specific goes in here, such as usernames extracted if they are available and applicable, etc.</td></tr>
</table>

Table 1: Content of the *t_line* timestamp object after being created by the routine *parse_line*

The following fields are left as optional and can be added to the extra field depending on the input module and parameters passed to the tool.

Name	Description
md5	A MD5 sum of the file being parsed. Currently this is not done by default for

	performance reasons however it can be forced by using the –c parameter to the front-end (calculating the MD5 sum for every file is very time consuming and not necessarily recommended).
user	The username of the user that owns the artifact in question (or the username found inside it, that is the username that is referred to inside the file)
host	The hostname of the host that the data came from
inode	The inode or MFT number of the parsed artifact.
mode	The permission mode of the artifact.
uid	The user ID of the owner of the artifact
gid	The group ID of the artifact.
size	The size of the artifact, that is the file itself or the data that is being referred to

Table 2: Some of the values that is stored in the extra field within the timestamp object

In most of the input modules the timestamps that are extracted are not easily categorized into the standard MACB timestamps. The input modules therefore put the same timestamp value in every variable, more detailed information about the choices of timestamp values can be found in chapter 4.1.

The MACB timestamps correspond to the *atime*, *mtime*, *ctime* and *crtime* of the mactime bodyfile format (*Carrier, Brian*, 2009).

The main module then modifies or adds some values to the timestamp object. The changes that are made are the following:

Name	**Change**	**Description**
name	Modified	The path separator is changed from \ to /, to make the output more consistent.
extra|host	Modified	The user supplied text is used to replace the content of the host variable if the option of "-n HOST" was used to call the front-end

extra|path	Added	Added this variable if the option of "-m TEXT" is passed on the tool. This option is used to prepend the path variable with user-supplied text (something like –m C: to indicate that the information came from the C drive).
extra|filename	Added	This variable contains the filename that the input module parsed.
extra|format	Added	The name of the input module used to parse the artifact
extra|inode	Added	Add information about the inode/MFT of the parsed file
time|<INDEX>|value	Modified	If use supplied time offset was used then it is added to the time (or subtracted). This is done for each defined timestamp within the timestamp object.

Table 3: The added or modified values of the timestamp object by the main module

There are few necessary subroutines that have to be implemented by each input module that are not part of the main functionality.

- *get_version()*
  This subroutine returns the version number of the input module. The version number is defined inside the variable $VERSION.
- *get_description()*
  This subroutine returns a string containing a short definition of the format that the input module is capable of parsing. The information that the subroutine returns is used when a list of all available input modules is printed out by the front-end.
- *close_file()*
  This subroutine is called when all timestamp entries have been parsed and outputted from the artifact. Usually this subroutine closes the file, disconnects database connections or otherwise closes open handles to the artifact.

- *get_help()*

  This subroutine returns a string containing longer description of the input module. This subroutine is called by the front-end when there is an error in parsing the artifact and when the front-end is called with a request to get a more detailed description of the input module.

  It sometimes contains a list of all dependencies and possibly some instruction on how to install them on the system to make it easier to install the input module.

To ease the development of new input modules a file called input_structure.pm is placed inside the "dev" folder of the tool. This file contains all the necessary structure and subroutines that need to be implemented to create a new input module.

2.5. Output Module Structure

Figure 5 shows the flow of operation for the output module. The structure is somewhat simpler than the input one.

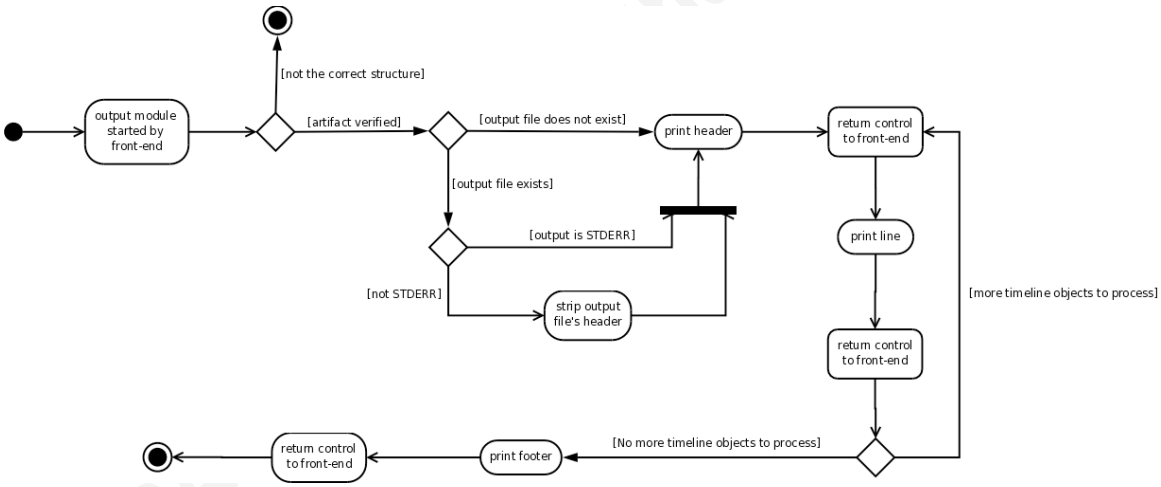

Figure 5: The flow of operation for the output module

The output module depends upon the implementation of the subroutine *print_line()* that is implemented in the main module or the front-end. This subroutine takes as an input a string and prints it out using the relevant output matter, whether that is to a file or to STDOUT.

The output module has to implement the following functions:

- **Initialize the output module.**

  The main module begins with creating a new output object using the "*new()*"

method. This method accepts an array as a parameter that contains user supplied parameters to configure the output module.

- **Print header.**
  This function is implemented using the subroutine *print_header()*. The subroutine calls the subroutine *print_line()* that is implemented in the front-end with a string containing the necessary header for the document format the output module is designed for. If the output module does not require a header to be printed, it simply returns a 1 to the front-end.

- **Process a timeline object for printing.**
  This is the main function of the output module and is implemented in the subroutine *print_line()*. That is to take as an input a reference to a timeline object (a hash) and structuring it in such a way that it can be outputted to a file.

- **Print footer.**
  This function is implemented using the subroutine *print_footer()*. The subroutine calls the subroutine *print_line()* that is implemented in the front-end with a string containing the necessary footer for the document format the output module is designed for. If the output module does not require a footer to be printed, it simply returns a 1 to the front-end.

There are few additional functions that each output module has to implement to properly function within the framework.

- *get_version()*
  This subroutine returns the version number of the output module. The version number is usually defined inside the variable $VERSION.

- *get_description()*
  This subroutine returns a string containing a short definition of the format that the output module provides. The information that the subroutine returns is used when a list of all available output modules is printed out by the front-end.

- *get_footer()*
  This subroutine returns to the main function the content of the footer. The main

module uses the information gathered from this subroutine to remove the footer of a previously saved body file. That way the framework can append information to a body file, even though it contains footer.

- *get_help()*
  This subroutine returns a string containing longer description of the output module. This subroutine is called by the front-end when there is an error in outputting the line and when the front-end is called with a request to get a more detailed description of the output module.
  It sometimes contains a list of all dependencies and possibly some instruction on how to install them on the system to make it easier to install the output module.

To ease the development of new output modules a file called output_structure.pm is placed inside the "dev" folder of the tool. This file contains all the necessary structure and subroutines that need to be implemented to create a new output module.

3. Supported Artifacts

3.1. Input Modules

In the currently released version of log2timeline at the time of publication, version 0.50, there 26 supported input modules. To get a full list of the available input modules in the tool use the parameter "-f list" in the log2timeline front-end.

3.1.1. Google Chrome – Browser History

Google Chrome is a relatively new browser on the market yet it has quickly become very popular. During forensic investigations users browser history is often one of the most frequently examined artifact making it important to understand the format of Chrome's browser history.

Google Chrome stores its browser history in a SQLite database called History. The History database file is stored in the following location:

- **Linux.**
  There are two different versions available. The official Google Chrome from the Google Chrome website and the Linux distribution Chromium.

Google Chrome: /home/$USER/.config/google-chrome/Default/History

Chromium: /home/$USER/.config/chromium/Default/History

- **Windows Vista (and Win 7)**

  C:\Users\[USERNAME]\AppData\Local\Google\Chrome\

- **Windows XP**

  C:\Documents and Settings\[USERNAME]\Local Settings\Application Data\Google\Chrome\

The database file that contains the browsing history is stored under the Default folder as "History" and can be examined using any SQLlite browser there is such as sqlite3. The available tables are:

- downloads
- presentation
- urls
- keyword_search_terms
- segment_usage
- visits
- meta
- segments

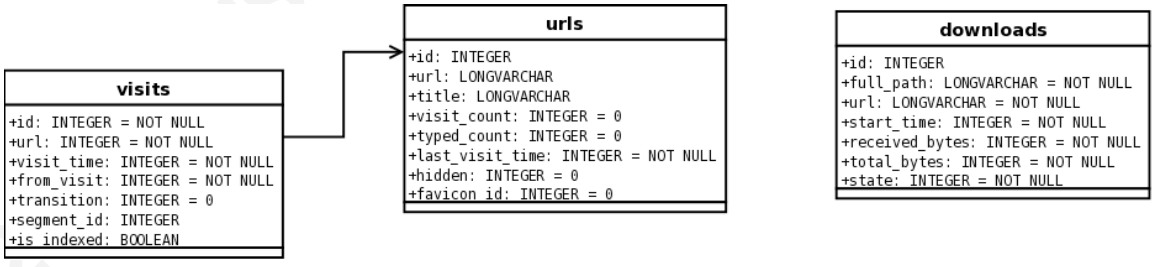

Figure 6: Shows the relationship and structure of three tables in the Chrome History database

The most relevant tables for browsing history are the "*urls*" table that contains all the visited URLs, the "*visits*" table that contains among other information the type of visit and the timestamps and finally the "downloads" table that contains a list of downloaded files.

The relation between the tables "*urls*" and "*visits*" can be seen in Figure 6. It is now possible to construct a SQL statement to extract some of the relevant information of a Chrome browser history:

```
SELECT urls.url, urls.title, urls.visit_count, urls.typed_count,
urls.last_visit_time, urls.hidden, visits.visit_time,
visits.from_visit, visits.transition
FROM urls, visits
WHERE
urls.id = visits.url
```

This SQL statement extracts all the URLs the user visited alongside the visit count, type and timestamps.

Google Chrome stores its timestamp values in different formats depending on table type. In the visits table Chrome uses the number of microseconds since midnight UTC of 1 January 1601 while in the download table standard Epoch time is used.

The row "*transition*" inside the "*visits*" table defines how the user came to browse that particular URL. It is therefore important to get a better understanding of the "*transition*" parameter. Full explanation of it can be found inside the Chrome's source code (page_transition_types.h, 2009). The core parameters are:

- **LINK**. User went to the page by clicking a link.
- **TYPED**. User typed the URL in the URL bar.
- **AUTO_BOOKMARK**. User got to this page through a suggestion in the UI, for example through the destinations page
- **AUTO_SUBFRAME**. Any content that is automatically loaded in a non-top level frame. User might not realize this is a separate frame so he might not now he browsed there.
- **MANUAL_SUBFRAME**. "For sub frame navigations that are explicitly requested by the user and generate new navigation entries in the back/forward list." (page_transition_types.h, 2009)
- **GENERATED**. User got to this page by typing in the URL bar and selecting an entry that did not look like a URL.

- **START_PAGE**. The user's start page or home page or URL passed along the command line (Chrome started with this URL from the command line)
- **FORM_SUBMIT**. The user filled out values in a form and submitted it.
- **RELOAD**. The user reloaded the page whether by hitting the re-load button, pushed the enter button in the URL bar or by restoring a session.
- **KEYWORD**. The URL was generated from a replaceable keyword other than the default search provider
- **KEYWORD_GENERATED**. Corresponds to a visit generated for a keyword.

The "*transition*" variable consists of more values than just the actual transition core parameters. It contains additional information, so called qualifiers such as whether or not this was a client or server redirect and if the beginning and the end of a navigation chain.

When reading the transition from the database and extracting just the core parameter the variable CORE_MASK has to be used to AND with the value found inside the database.

The Chrome input module reads the Chrome database using the previous SQL statement, calculates the date from either Epoch time or microseconds since midnight UTC of 1 January 1601. It then parses the core parameters from the "*transition*" variable and prints this out in a human readable format. The user name is found by extracting it from the current path where the file was found (this can be overwritten using a parameter to the tool). An example format can be seen here:

```
0|[Chrome History] (URL visited) User: john
http://tools.google.com/chrome/intl/en/welcome.html (Get started with
Google Chrome) [count: 1] Host: tools.google.com type: [START_PAGE -
The start page of the browser] (URL not typed directly) (file:
History)|20752894|0|0|0|0|1261044829|1261044829|1261044829|1261044829
```

3.1.2. Event Log (EVT)

The Windows Event Logs store information gathered from various parts of the Windows operating system, such as when a user logs into a machine, from where, and how as well as when services are started or stopped. It also stores error messages from

applications and the operating system itself as well as various other potentially important messages.

The evt input module is based on the script evtparse.pl written by H. Carvey and published as a part of his timeline toolkit (Carvey 2009). A good description of the Event Log file format can be found in H. Carvey's book Windows Forensic Analysis 2E (Carvey, 2009). In short the Event Log is a binary format that is split into a header and a series of records. The header of an event record is 56 bytes long and contains a magic value of *LfLe*. The input module scans the event log file searching for this magic value. As soon as it detects one it starts parsing it and extracting the necessary information to construct a timestamp object. Since the input module scans the entire event log file for the correct magic value it has the possibility to detect deleted events that are still stored inside the event log file as well as being independent of the Windows API.

An example format can be seen here:

```
18|0|12320768|1263814277|1263814277|1263814277|1263814277
0|[Event Log] (Time generated/Time written) <SMITHSLAPTOP>
LGTO_Sync/1;Info; - The Driver was loaded successfully  (file:
SysEvent.Evt)|21318746|0|0|0|12320768|1265204189|1265204199|1265204189|
12652041
```

3.1.3. Event Log (EVTX)

The Windows Event Logs store information gathered from various parts of the Windows operating system, such as when a user logs into a machine, from where, and how as well as when services are started or stopped. It also stores error messages from applications and the operating system itself as well as various other potentially important messages. As of Microsoft Vista the Event Log is stored using a different format called EVTX making it necessary to create a new input. This input module uses the libraries Parse::Evtx developed by Andreas Schuster's (Schuster, Andreas, 2009) to parse the newly created EVTX format that Windows Vista and later Windows system use.

The EVTX format is a binary file that contains an XML structure. By using the libraries developed by Schuster it is possible to extract the XML structure from the document instead of reading the binary one. After extracting the XML schema it looks somewhat like this:

```
<Events>
```

```
        <Event>
                <System> . . . </System>
                <EventData> . . . </EventData>
        </Event>
        <Event>
                <System> . . . </System>
                <UserData> . . . </UserData>
        </Event>
        <Event>
        ...
        </Event>
</Events>
```

The "Event" tag is wrapped around the "Events" tag that spans the whole file. Inside each "Event" tag there can be several different tags, most notably the "System" and "EventData" (Schuster, Andreas, 2007). Microsoft describes the schema for the "Event" tag here (EventType Complex Type, 2009) and "System" (SystemPropertiesType Complex, 2009) and finally the "EventData" (EventDataType Complex Type, 2009).

The first version of the evtx input module only extracts certain information from the "System" tag, while other tags are just concentrated on a single line for printout. The "System" tag provides very useful information about the event itself, such as the EventSourceName, TimeCreated, EventId, Channel (that is the source of the event) and Computer (the hostname) to name a few.

The input module uses XML::LibXML to parse the XML structure and extract few information about the event from the "System" tag (the previously mentioned variables) as well processing other structures with the following format

```
NodeName -> (attributeName = attributeValue, …) NodeValue, ...
```

The NodeName in the format mentioned above refers to the tag itself and the NodeValue is the text content that is surrounded with the tag. An example XML structure and the parsing of the script follow:

```
<Event xmlns="http://schemas.microsoft.com/win/2004/08/events/event">
<System>
      <Provider Name="Avira AntiVir" />
      <EventID Qualifiers="32768">4113</EventID>
      <Level>3</Level>
      <Task>2</Task>
      <Keywords>0x0080000000000000</Keywords>
      <TimeCreated SystemTime="2010-01-07T10:06:55.0Z" />
      <EventRecordID>2829</EventRecordID>
```

```
      <Channel>Application</Channel>
      <Computer>mine</Computer>
      <Security UserID="S-1-5-18" /></System>

<EventData>
      <Data>[0] SPR/Tool.NetCat.B
            [1] C:\nc\nc.exe</Data>
      <Binary></Binary>
</EventData>
</Event>
```

And the output from log2timeline

```
0|[Application] (Event Logged) <mine> Application/Avira AntiVir ID
[4113] :EventData/Data -> [0] SPR/Tool.NetCat.B[1] C:/nc/nc.exe-
EventData/Binary -> empty  (file:
app.evtx)|20883782|0|0|0|0|1262858815|1262858815|1262858815|1262858815
```

3.1.4. EXIF

Exchangeable image file format or EXIF is a specification for image file format. Among other things it contains metadata, or tags that describe the image file. There is numerous amount of information that exist within the EXIF data, such as the date the image was taken, modified, etc. The information found within an EXIF can be very valuable to an investigator since they may contain information about the user, location, make and model of camera, or any other information extracted from image files. There exists an excellent library in Perl to extract this information called Image::ExifTool that log2timeline uses to parse metadata information from media files. ExifTool is not only able to parse metadata from various media files; it is also capable of extracting some metadata information from PDF documents, EXE files, etc.

Log2timeline starts by forcing the ExifTool library to structure dates in a fixed format. It then tries to extract information from the given document using the ExifTool library. If successful it will go through each of the given metadata tags to determine if it contains a date. Each date element is then inserted into a hash variable that is then passed on to other functions of log2timeline to produce an output. Since the name of the metadata tags that are defined in any given media file varies between different modules of cameras and photo-editing software the module only returns the actual name of the metadata that contains the date information.

An example file that ExifTool parses is a word document, such as this document (at one point):

```
ExifTool Version Number         : 8.00
File Name                       : Mastering the Super Timeline With
log2timeline.doc
Directory                       : .
File Size                       : 2.1 MB
File Modification Date/Time     : 2010:06:29 11:07:57+00:00
File Type                       : DOC
MIME Type                       : application/msword
Title                           : Mastering the Super Timeline With
log2timeline
Subject                         : Mastering the Super Timeline With
log2timeline
Author                          : Kristinn Guðjónsson,
kristinn@log2timeline.net
Keywords                        :
Template                        : Normal.dotm
Last Saved By                   : Kristinn Guðjónsson
Revision Number                 : 91
Software                        : Microsoft Macintosh Word
Last Printed                    : 2010:04:16 10:04:00
Create Date                     : 2010:02:09 12:28:00
Modify Date                     : 2010:06:29 11:07:00
Page Count                      : 82
Word Count                      : 22771
Char Count                      : 129800
Category                        : GIAC Gold
Manager                         : Charles Hornat
Company                         : SANS
Lines                           : 1081
Paragraphs                      : 259
Char Count With Spaces          : 159403
App Version                     : 12.0000
Title Of Parts                  : Mastering the Super Timeline With
log2timeline
Code Page                       : Mac Roman (Western European)
```

And the output of log2timeline using the exif input module and mactime output:

```
0|[EXIF metadata] (FlashPix/LastPrinted) LastPrinted (file: Mastering
the Super Timeline With
log2timeline.doc)|1524693|0|0|0|0|1271412240|1271412240|1271412240|1271
412240
0|[EXIF metadata] (FlashPix/CreateDate) CreateDate (file: Mastering the
Super Timeline With
log2timeline.doc)|1524693|0|0|0|0|1265718480|1265718480|1265718480|1265
718480
0|[EXIF metadata] (FlashPix/ModifyDate) ModifyDate (file: Mastering the
Super Timeline With
log2timeline.doc)|1524693|0|0|0|0|1277809620|1277809620|1277809620|1277
809620
```

3.1.5. Firefox Bookmark File

Bookmarks can sometimes provide the investigator with meaningful information, perhaps about the sites that a user favors or otherwise provide a timeframe when a user originally found some perhaps illegal web site (and bookmarked it). Firefox version 2 and

older stored their bookmarks in a HTML file called bookmarks.html in the user profile (Bookmarks.html, 2008). Starting with version 3.0 of Firefox the bookmarks were moved to a SQLite database called places.sqlite.

The bookmark file is constructed as a simple HTML file

```
<DT>
      <A HREF="http://www.w3.org/TR/2004/CR-CSS21-20040225/"
ADD_DATE="1079992089" LAST_VISIT="1132078168" LAST_CHARSET=  "ISO-8859-
1" ID="rdf:#$kLc2Z">Cascading Style Sheets, level   2 revision 1</A>
```

Log2timeline uses the HTML::Parser library to parse the HTML structure of the bookmark file to gather all the timestamps that are stored in the file. The following timestamps are available inside a bookmark file:

Name	Source	Value
LAST_MODIFIED	Bookmark file	The last time the bookmark file itself was modified
ADD_DATE	Folder	The date when the bookmark folder was created
LAST_MODIFIED	Folder	The date when the bookmark folder was modified
ADD_DATE	Bookmark	The date when the bookmark was added
LAST_VISIT	Bookmark	The date when the bookmark was last visited

Table 4: Shows the available sources of timestamps within a Firefox bookmark file

An example output of log2timeline using the Firefox bookmark input module ff_bookmark:

```
0|[Firefox] (bookmark created) User: john ISC: Tip #1
[http://isc.sans.org/diary.html?storyid=3438] (file:
bookmarks.html)|21091335|0|0|0|0|1191448977|1191448977|1191448977|11914
48977
```

3.1.6. Firefox 2 Browser History

As noted before when discussing Google Chrome, users browser history is often one of the most frequently examined artifacts on a suspect machine. And the Firefox browser has been the second most popular web browser for quite some time, making it essential to understand how it stores it browsing history. Up to version 3.0, Firefox stored it's browser history in a flat text file based on the Mork structure, which is essentially a "*table* (or synonymously, a sparse *matrix*) composed of *rows* containing

cells, where each cell is a member of exactly one *column*" (McCusker, 2008). The browsing history is stored in a file called history.dat that is stored inside the user's profile folder.

An example output of log2timeline using the Firefox 2 browser history input module, firefox2:

```
0|[Firefox 2] (Only Visit) User: john Visited http://mbl.is/mm/frettir/
(mbl.is) Title: (mbl.is - Fréttir) (file:
history.dat)|24708330|0|0|0|0|1269960983|1269960983|1269960983|12699609
83
```

3.1.7. Firefox 3 Browser History

As of version 3.0 Firefox changed the way it stored it's browser history. It moved away from the previous Mork structure into a SQLite database file called places.sqlite, which includes more detailed information about the browsing history. The database contains several tables:

- moz_anno_attributes
- moz_favicons
- moz_keywords
- moz_annos
- moz_historyvisits
- moz_places
- moz_bookmarks
- moz_inputhistory
- moz_bookmarks_roots
- moz_items_annos

The tables that contain timestamp information are the moz_places, moz_historyvisits, moz_items_annos and moz_bookmarks. The tables that contain the browsing history as well as the associated timestamps of those visits are the moz_places and moz_historyvisits.

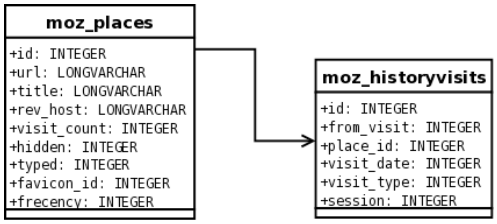

Figure 7: Relationship schema for the two of the tables in places.sqlite SQLite database

Most of the relevant information regarding user's browsing history is stored inside the table *moz_places* including the full URL, host name, type of visit as well as the title of web site. The table *moz_historyvisits* contains vital information as well, such as the actual timestamp and the type of visit. The column *visit_type* inside the table *moz_historyvisits* contains the type of web page visit. The available types are according to Mozilla:

"TRANSITION_LINK. This transition type means the user followed a link and got a new top-level window.

TRANSITION_TYPED. This transition type is set when the user typed the URL to get to the page.

TRANSITION_BOOKMARK. This transition type is set when the user followed a bookmark to get to the page.

TRANSITION_EMBED. This transition type is set when some inner content is loaded. This is true of all images on a page, and the contents of the iframe. It is also true of any content in a frame, regardless if whether or not the user clicked something to get there.

TRANSITION_REDIRECT_PERMANENT. This transition type is set when the transition was a permanent redirect.

TRANSITION_REDIRECT_TEMPORARY. This transition type is set when the transition was a temporary redirect.

TRANSITION_DOWNLOAD. This transition type is set when the transition is a download."

("NsINavHistoryService - MDC," 2009)

To extract the needed timestamp information log2timeline uses database libraries to connect to the SQLite database and issues the following SQL command:

```
SELECT moz_historyvisits.id, url, title, visit_count, visit_date,
from_visit, rev_host, hidden,moz_places.typed,
moz_historyvisits.visit_type
FROM moz_places, moz_historyvisits
WHERE
        moz_places.id = moz_historyvisits.place_id
```

What this SQL command does is to combine results of both the *moz_historyvisits* table and the *moz_places* table. It extracts all the URL's visited from the *moz_places* table and joins the information with the time and type of visit as well as the title of the visited page and the hostname. An example extracted line by the firefox3 input module and a mactime output module looks like this:

```
0|[Firefox 3 history] (URL visited) User: john http://isc.sans.org/
(SANS Internet Storm Center; Cooperative Network Security Community -
Internet Security) [count: 1] Host: isc.sans.org (URL not typed
directly) type: LINK (file:
places.sqlite)|16911245|0|0|0|0|1245874794|1245874794|1245874794|124587
4794
```

The extracted line clearly shows that the user "joe" visited a particular web site with its title displayed inside parenthesis. The line also shows the value of the page count, which indicates how many times the user has visited that particular web site along with the hostname and the type of visit. The type of visit indicates that the user did not type the URL directly in the browser.

Since of version 3 of the Firefox browser bookmarks were no longer stored inside the bookmark file and instead stored within the places.sqlite SQLite database. All bookmark information is stored within two tables, *moz_places* and *moz_bookmarks*. To extract the necessary information from the table the following SQL command is used:

```
SELECT
moz_bookmarks.type,moz_bookmarks.title,moz_bookmarks.dateAdded,moz_book
marks.lastModified,moz_places.url,moz_places.title,moz_places.rev_host,
moz_places.visit_count
FROM moz_places, moz_bookmarks
WHERE
        moz_bookmarks.fk = moz_places.id
        AND moz_bookmarks.type <> 3
```

The field type within the *moz_bookmarks* table has three possible values

- 1 = A bookmark file, or an URL.

- 2 = A bookmark folder.
- 3 = A separator.

The above query will get results where the bookmark has a value within the *moz_places* table, which folders do not. So we only get the bookmark files. To get the folders as well, the following query is issued:

```
SELECT
moz_bookmarks.title,moz_bookmarks.dateAdded,moz_bookmarks.lastModified
FROM moz_bookmarks
WHERE
        moz_bookmarks.type = 2
```

With Firefox 3 it is possible to write annotations or description of the bookmarks

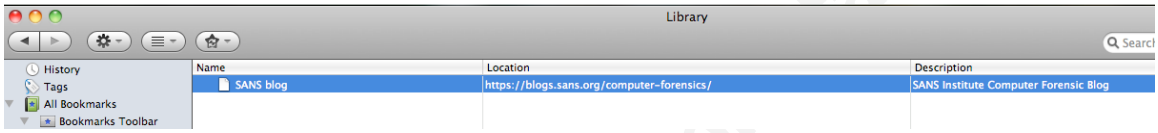

Figure 8: Shows a bookmark in Firefox 3 with a description or an annotation

The description or the annotation is stored inside the table *moz_items_annos*. The table contains various fields, such as lastModified, dateAdded and item_id. The item_id value refers to the id value found inside the *moz_bookmarks* table. To extract the annotations from the *moz_places* the following SQL command was constructed:

```
SELECT moz_items_annos.content,
moz_items_annos.dateAdded,moz_items_annos.lastModified,moz_bookmarks.ti
tle, moz_places.url,moz_places.rev_host
FROM moz_items_annos,moz_bookmarks,moz_places
WHERE
        moz_items_annos.item_id = moz_bookmarks.id
        AND moz_bookmarks.fk = moz_places.id
```

An example output of the Firefox input module reading the bookmarks and the bookmark annotation is the following:

```
0|[Firefox 3 history] (dateAdded/LastModified) User: john Bookmark URL
SANS Forensics Blog (https://blogs.sans.org/computer-forensics/) [SANS
Computer Forensics- Investigation- and Response » A team of GIAC
Certified Forensic Analysts (GCFA) and their thoughts on Digital
Forensic and Incident Response techniques and trends.] count 0 (file:
places.sqlite)|16911245|0|0|0|0|1245704602|1245704602|1245704617|124570
4617
0|[Firefox 3 history] (dateAdded/LastModified) User: john Bookmark
Annotation: [A team of GIAC Certified Forensic Analysts (GCFA) and
their thoughts on Digital Forensic and Incident Response techniques and
```

```
trends.] to bookmark [SANS Forensics Blog]
(https://blogs.sans.org/computer-forensics/) (file:
places.sqlite)|16911245|0|0|0|0|1245704602|1245704602|1245704602|124570
4602
```

3.1.8. Internet Explorer Browser History

Again, during forensic investigations user's browser history is often one of the most frequently examined artifact and Internet Explorer is still considered to be the most widely used browser in the world. The browser stores its history in a binary file called index.dat that can be found in several places depending on the version of the Windows operating system. The structure of the index.dat file has been reversed engineered by Keith Jones and a full description of the inner working can be found in his white paper (Jones, Keith J., 2003). The input module in log2timeline is based on the information found in the white paper by Keith.

The input module starts by reading the file's header before moving in on reading information about available directories that store cached data. Then the location of the first hash table is read. The hash table is an array of data that contains entries pointing to the relevant activity data within the index.dat (Jones, Keith J., 2003). The information about visited URL's is stored within activity records that the hash points to. The hash also contains an offset to the next hash in the file making it easy to follow every hash there is within the file.

There are two timestamps within each URL activity record. The meaning of the timestamps differs depending on the location of the index.dat file. Within the history folder, the one that stores the browser history files, there are several index.dat files. Both of the timestamps refer to the last time a particular URL was visited, the difference lies in the way that the timestamp is stored, that is to say if the timestamp is stored in UTC or in local time. The master file contains timestamps that are stored in UTC while both the weekly and daily history files contain timestamps both stored in local time zone and in UTC. The timestamps of the cache information, which is also stored in an index.dat file, refer to the last time the cache was written to the disk and the last time that cache was accessed. Both of these timestamps are stored in UTC. And the cookie index.dat file contains the timestamps referring to the last time the cookie was passed on to a website and the last time a website modified the cookie. These timestamps are also stored in

UTC. The SANS security 408 course, Computer Forensics Essential, goes into the difference between the meaning of different index.dat files in much more detail than is done here.

There are other index.dat files that can be found inside the operating system, and the meaning of these timestamps differ as well. The input module takes both these timestamps and includes them in the timestamp object as well as reading in the path from which it was stored to determine both the meaning of them as well as the time zone that they were stored in. In MACB representation timestamp two is stored as the last modification time and timestamp one gets assigned [.ACB].

```
0|[Internet Explorer] (Last Access) User: joe
URL:file:///C:/Documents%20and%20Settings/Administrator/My%20Documents/
Not%20to%20be%20seen%20document.txt cache stored in: /URL  -  (file:
Documents and Settings/joe/Local
Settings/History/History.IE5/index.dat)|17112452|0|0|0|0|1249399163|124
9399163|1249399163|1249399163
0|[Internet Explorer] (Last Access) User: joe
URL:http://download.mozilla.org/?product=firefox-3.5.2&os=win&lang=is
cache stored in: /URL  -  (file: Documents and Settings/joe/Local
Settings/History/History.IE5/index.dat)|17112452|0|0|0|0|1249398890|124
9398890|1249398890|1249398890
```

3.1.9. IIS W3C Log Files

The Microsoft Internet Information Server or IIS stores its log files using the W3C extended log file format (W3C Extended Log File Format (IIS 6.0), 2010). The log file includes information about each request made to the server, information that can be of great value, especially when investigating a possibly compromised web server running IIS. There are several fields within the W3C format that can be used within any given log file, but the structure can be changed in the configuration for the web site. The structure of the file is given in its header. An example header is:

```
#Software: Microsoft Internet Information Services 6.0
#Version: 1.0
#Date: 2007-12-17 00:03:13
#Fields: date time s-sitename s-ip cs-method cs-uri-stem cs-uri-query
s-port cs-username c-ip cs(User-Agent) sc-status sc-substatus sc-win32-
status
```

The header contains the marker "#Fields:" which defines both the fields that are used within the log file as well as their order in it. The entries are then space delimited.

The input module starts by reading the header to see which fields are used and in which order. Each line is then parsed according to the structure.

An example output is:

```
0|[IIS Log File] (Entry written) <10.1.1.2> User: 10.1.1.24 10.1.1.24
connect to '10.1.1.2:80' URI: GET /img/picture.jpg using
Mozilla/4.0+(compatible;+MSIE+7.0;+Windows+NT+5.1)- status code 304
(file: ex071003.log)|
17153431|0|0|0|0|1191582960|1191582960|1191582960|1191582960
```

3.1.10. ISA Text Export

Microsoft ISA server maintains a firewall log that contains information about every connection attempt made to the server. The information it contains can be of great value since it can both corroborate information found inside the user's browser history as well as to provide the investigator with a browser history in the absence of such data on the user's machine. With the increased use of privacy software that cleans browsing history as well as privacy settings in browsers that prevent any history to be written to disk it becomes very important to have some sort of method to construct the users browsing history.

The firewall log in an ISA server can be viewed using the "Monitor/Logging" capability of the server. After a query has been constructed and completed it is possible to use the "Copy All Results to Clipboard" tasks as can be seen in Figure 9 and then paste the content of the clipboard to a text file, perhaps using a text editor like Notepad. This exported text file can then read by log2timeline.

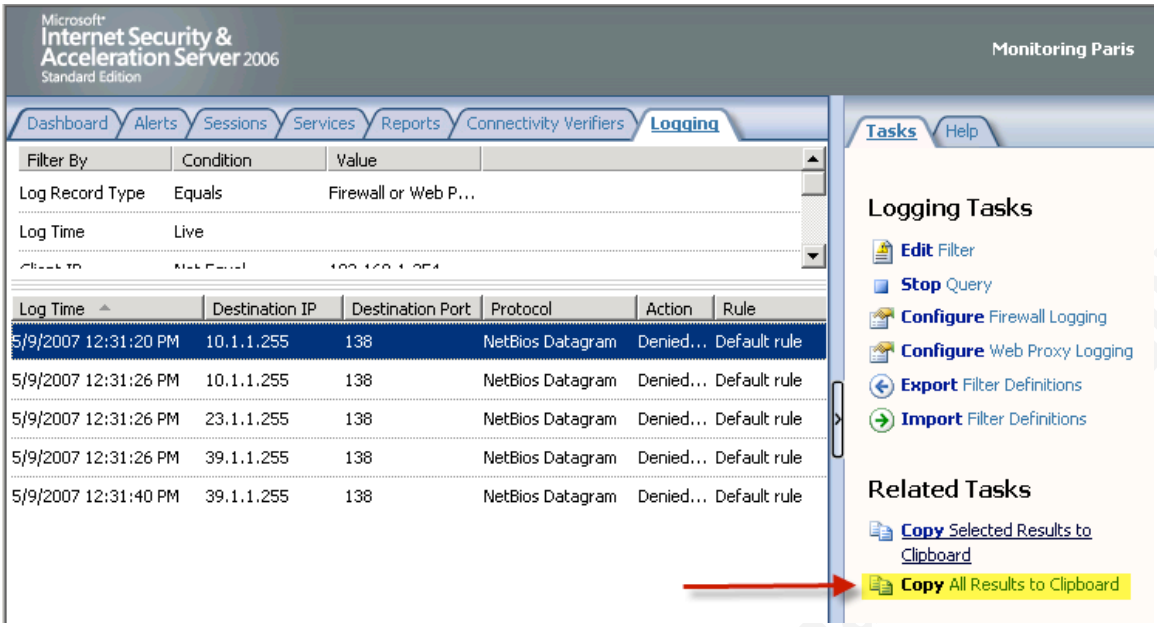

Figure 9: Shows a screen capture from an ISA server

The input module parses the default structure of the text file, which includes timestamps stored in UTC, to produce a timeline object.

An example output from log2timeline using the isatxt input module:

```
0|[ISA text export] (Entry written) <10.1.1.24> User: MYDOMAIN\SMITH
(10.1.1.24) Connected to unknown host [212.58.226.138:80] with URL:
http://news.bbc.co.uk/somesite from referring {http://news.bbc.co.uk/}
using GET [200] - Allowed Connection - Mozilla/4.0 (compatible; MSIE
7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727; .NET CLR
3.0.04506.30; .NET CLR 3.0... (file:
myisaexport.txt)|4998|0|0|0|937|1255648430|1255648430|1255648430|125564
8430
```

3.1.11. Mactime

The current version of the framework does not extract traditional timestamps from the filesystem. This makes it necessary to be able import data from other tools that possess that capability until it has been implemented. Tools such as fls and mac-robber from the Sleuthkit (TSK) use the mactime format to store their filesystem timestamps in a body file. Therefore an input module capable of reading mactime format was created. The structure as defined by the Sleuthkit (TSK) has changed since of version 3.+ of the tool. The format for version 3.+ is the following:

```
MD5|name|inode|mode_as_string|UID|GID|size|atime|mtime|ctime|crtime
```

And the older format that is the format that is used by TSK version 1.x and 2.x is the following:

```
 MD5 | path/name | device | inode | mode_as_value | mode_as_string |
num_of_links  | UID | GID | rdev | size | atime | mtime | ctime |
block_size | num_of_blocks
```

3.1.12. McAfee AntiVirus Log File

Antivirus log files often contain time stamped data that indicates when the antivirus product was last run, last updated as well as information about which files have been quarantined, deleted, repaired or otherwise been flagged by the antivirus product. McAfee, being one of the largest antivirus vendors, maintains three log files; AccessProtectionLog.txt, OnAccessScanLog.txt and OnDemandScanLog.txt.

The mcafee input module in log2timeline starts by verifying which version of the log file it is parsing and then parses each line in the file or several lines if they pertain to the same event and outputs them.

An example output from the input module is:

```
0|[McAfee AV Log] (Entry written) User: MYDOMAIN\smith OnAccessScan:
action: Not scanned (scan timed out) - Virus: - file
c:/windows/system32/as6p.exe - C:/Program Files/Microsoft Virtual
Server/WebSite/VirtualServer/VSWebApp.exe (file:
OnAccessScanLog.txt)|20920477|0|0|0|0|1219677580|1219677580|1219677580|
1219677580
```

3.1.13. Opera Browser History

The Opera browser stores the browser history in two files, a global history file that contains all the visits to web pages that are not directly typed in and the direct history that contains information about visits that have been typed in the browser. Opera browser is one of the most commonly used browsers today making it necessary to take Opera history into the account when examining users browser history.

The global history file is a text file with the following structure:

```
Title of the web site (as displayed in the title bar)
The URL of the visited site
Time of visit (in Epoch time format)
An integer, representing the popularity of the web site
```

The direct history file is a XML file

```
<?xml version="1.0" encoding="ENCODING"?>
<typed_history>
      <typed_history_item
            content="URL TYPED IN"
            type="text"
            last_typed="DATE"/>
</typed_history>
```

The input module starts by verifying which version of the history file the input module is dealing with before parsing each line, either with a simple text parser or with a XML parser. An example output of the tool when the direct history is parsed:

```
0|[Opera] (URL visited) User: smith typed the URL sans.org directly
into the browser (type "text") (file:
typed_history.xml)|4958533|0|0|0|0|1262001379|1262001379|1262001379|126
2001379
```

And an example when the tool is run against the global history file:

```
0|[Opera] (URL visited) User: smith URL visited http://sans.org/ (SANS:
Computer Security Training- Network Security Research- InfoSec
Resources) [-1] (file:
global_history.dat)|4958454|0|0|0|0|1262001382|1262001382|1262001382|12
62001382
```

3.1.14. Open XML

Microsoft Office stores important information about their documents in a metadata structure. The information stored inside these documents can be of great value in an investigation, making it crucial to include timestamps extracted from these documents in the timeline.

The format of Microsoft Office documents changed as of version 2007 to the Open XML standard. The Open XML standard is described partly in the article "Introducing the Office (2007) Open XML Formats" (Introducing the Office (2007) Open XML Formats, 2010). Each Open XML document contains several XML files that either contain the information itself or describe them. The XML files are then compressed using standard ZIP algorithm to reduce the storage space.

Open XML documents can be verified both by verifying that they are a ZIP file (using magic value in the beginning of the file) as well as to read the ZIP header and examine the file name value inside it. The file name value should be "[Content_Types].xml". After verifying that the compressed file is in fact an Open XML document the input module searches for the "_rels/.rels " file that contains the metadata

information that describes the Open XML document. An example .rels files contains the following lines:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Relationships
xmlns="http://schemas.openxmlformats.org/package/2006/relationships">
      <Relationship Id="rId4"
            Type="http://schemas.openxmlformats.org/officeDocument/2006
      /relationships/extended-properties"
            Target="docProps/app.xml"/>

      <Relationship Id="rId1"
            Type="http://schemas.openxmlformats.org/officeDocument/2006
      /relationships/officeDocument"
            Target="word/document.xml"/>
      <Relationship Id="rId2"
            Type="http://schemas.openxmlformats.org/package/2006/relati
      onships/metadata/thumbnail"
            Target="docProps/thumbnail.jpeg"/>
      <Relationship Id="rId3"
            Type="http://schemas.openxmlformats.org/package/2006/relati
      onships/metadata/core-properties"
            Target="docProps/core.xml"/>
</Relationships>
```

The input module then parses the .rels XML file to find all the XML files that contain the metadata type. There are two XML files that contain metadata that is present in documents by default:

- DOC.ENDING/docProps/app.xml
- DOC.ENDING/docProps/core.xml

There is sometimes additional metadata information stored in a custom XML document that is also parsed if it is present. All timestamps with the metadata documents are stored using the ISO 8601 standard and are converted to Epoch time by the input module.

An example output of the input module is the following:

```
0|[Open XML Metadata] (modified) User: Kristinn Gudjonsson  (The title
of the document) - And the subject -  - And this is the comment
section... -  - Application: Microsoft Macintosh Word - Company: My
company - AppVersion: 12.0000 - Desc: And this is the comment
section... (file:
test_document.docx)|24879329|0|0|0|0|1271685900|1271685900|1271685900|1
271685900
```

3.1.15. PCAP

Network captures can sometimes contain vital information about what happened on a suspect drive. Network traffic can be captured using various methods, one of which is to use tools like Wireshark or tcpdump to capture it and save it using a PCAP format. The PCAP format contains a header that consists of among other things a timestamp of when the packet was captured.

The PCAP input module in log2timeline uses the Perl library Net::Pcap to parse the structure of a PCAP file and print information gathered about every packet there is inside it. The current version is quite dump in the perspective that it simply prints each and every timestamp available, and it does no attempts of TCP re-assembly or any other processing. The input module accepts standard BPF filters as a parameter to the module to limit the results if needed.

An example output is:

```
0|[PCAP file] (Time Written) <10.1.1.15> TCP SYN packet 10.1.1.15:51938
-> 66.35.45.201:80 seq [2370686007] (file:
/tmp/test.dump)|24879642|0|0|0|0|1271686708|1271686708|1271686708|12716
86708
```

3.1.16. PDF Metadata

The Portable Document Format (PDF) is an open document standard (ISO/IEC 32000-1:2008) that was originally developed by Adobe Systems. The document standard implements two types of metadata, one being XMP or the Extensible Metadata Platform that is yet another standard created by Adobe to for processing and storing metadata information. The other metadata source within PDF files is the inherent metadata information as defined by the standard. The log2timeline input module for pdf metadata parses the second type, or the native inherent metadata.

According to the standard all “date values used in a PDF shall conform to a standard date format, which closely follows that of the international standard ASN.1 (Abstract Syntax Notation One), defined in ISO/IEC 8824. A date shall be a text string of the form (D : YYYYMMDDHHmmSSOHH ' mm)” (Adobe System, 2010).

An example line that contains metadata is the following:

```
endstream^Mendobj^M161 0 obj<</ModDate(D:20060927095735-
04'00')/CreationDate(D:20060927095735-04'00')/Title(Microsoft
PowerPoint - 20060927.ppt)/Creator(PScript5.dll Version
5.2.2)/Producer(Acrobat Distiller 6.0.1
\(Windows\))/Author(aswanger)>>^Mendobj^Mxref^M
```

The pdf input module in log2timeline takes the document described above and parses it into the following line:

```
0|[PDF Metadata] (creationdate) User: aswanger File created. Title :
(Microsoft PowerPoint - 20060927.ppt) Author: [aswanger] Creator:
[PScript5.dll Version 5.2.2] produced by: [Acrobat Distiller 6.0.1
/(Windows/] (file:
webcast_vm.pdf)|16816552|0|0|0|0|1159365455|1159365455|1159365455|11593
65455
```

3.1.17. Prefetch and Superfetch

Windows XP introduced the concept of prefetch files to speed up application startup process. Each time an application is started on a Windows system a prefetch file is created (or updated if it already exists), making it a great source of recently run software on the machine. Prefetch files are stored a folder called %windir%\Prefetch along with Layout.ini that contains information about loaded files and directories that are used during the boot process. The information in the Layout.ini file are used by the automatic run of the Windows defragment tool to speed up the boot process by making the files that are loaded during the boot in a more sequential order on the hard drive.

The prefetch file itself is stored in a binary format that contains information about the executable file. Within the prefetch file are information about the last time the executable was run as well as the run count, loaded drivers and various other information.

The prefetch input module takes as an input the prefetch folder itself. It starts by parsing the Layout.ini file and stores all files that are executables. It then parses each of the prefetch files found within the folder and extracts the run count, the timestamp of the last time the executable was run, the name of if and a list of the drivers it loaded.

The input module originally contained part of the script pref.pl, written by Harlan Carvey, to parse the binary structure of each prefetch file. The code has since been modified to extract further information from the prefetch file than the script provided. Support for Windows Vista and Windows 7 superfetch files, which replace the older prefetch file format, has also been included in the latest version.

The binary structure starts with an indication of the prefetch file version (one byte), and then a magic value starting at offset 4, containing the letters SCCA in ASCII. The version number for a Windows XP prefetch file is 0x11 while 0x17 represents Vista superfetch files. An example header from a superfetch file is:

```
1700 0000 5343 4341
```

The input module then uses the versioning information extracted from the header to determine the correct offsets into the binary structure to extract the rest of the information.

The indication that a particular application was loaded might not be sufficient for the investigator. Sometimes additional context has to be provided to get a better picture of what has really been run. For instance, the loading of the application mmc.exe does not necessarily mean that the user changed audit configuration on the machine. However the prefetch file contains which drivers that particular application loaded, which might give the investigator the information needed, such as to which operation was loaded up in mmc. Therefore in the latest version of the input module information about the loaded drivers is included.

An example output of the input module is:

```
0|[XP Prefetch] (Last run) BCWIPE3.EXE-08C9B14D.pf - [BCWIPE3.EXE] was
executed - run count [1]- full path: [<path not found in Layout.ini>] -
DLLs loaded: {WINDOWS/SYSTEM32/NTDLL.DLL -
WINDOWS/SYSTEM32/KERNEL32.DLL - WINDOWS/SYSTEM32/COMCTL32.DLL -
WINDOWS/SYSTEM32/ADVAPI32.DLL - WINDOWS/SYSTEM32/RPCRT4.DLL -
WINDOWS/SYSTEM32/SECUR32.DLL - WINDOWS/SYSTEM32/GDI32.DLL -
WINDOWS/SYSTEM32/USER32.DLL - WINDOWS/SYSTEM32/IMM32.DLL -
WINDOWS/SYSTEM32/UXTHEME.DLL - WINDOWS/SYSTEM32/MSVCRT.DLL -
WINDOWS/SYSTEM32/MSCTF.DLL - WINDOWS/SYSTEM32/VERSION.DLL -
WINDOWS/SYSTEM32/OLE32.DLL - WINDOWS/SYSTEM32/MSIMTF.DLL -
WINDOWS/SYSTEM32/APPHELP.DLL} (file: ./BCWIPE3.EXE-
08C9B14D.pf)|17108475|0|0|0|0|1249399276|1249399276|1249399276|12493992
76
```

3.1.18. Recycle Bin

The recycle bin contains files that have been deleted using the Explorer in the Windows operating system (as long as the recycle bin has not been emptied). The file structure of Windows operating systems until Vista appeared are described in the paper "Forensic Analysis of Microsoft Windows Recycle Bin Records" written by Keith Jones

(Jones, Keith J., 2003). The recycle bin is contained in the folder RECYCLER, which is stored at the root directory of each partition. Within the RECYCLER directory are directories that belong to each user that has used the recycle bin. The name of the user directory is the same as the user's SID, which is the unique identification number used by Microsoft Windows.

Inside the user's directory are the deleted files, named Dc#.EXT where # is an integer in sequential order and EXT is the original's files extension. A file called INFO2 contains information about the original file, including the original name and path, date of deletion, drive information and the size of the file.

The structure changed in Windows Vista and later operating systems. The recycle bin name has changed to $Recycle.bin (the location is the same). Instead of storing the files as Dc#.EXT each file is stored as a $R*random*.EXT where EXT is the original file's extension. The INFO2 file no longer exits and instead the information about the original file is stored in a file called $I*random*.EXT, where random is the same random value as the $R*random*.EXT file has.

The input module starts by reading the user's folder within the recycle bin and determines whether or not the INFO2 file exists. If it does, it proceeds with parsing it as it was a XP or older recycle bin. If the file does not exist it verifies that $I and $R files exist and parses each of $I files to extract the needed information about deleted files.

An example output of the input module is:

```
0|[RECYCLER] (File deleted) nr: Dc1 - C:/Documents and
Settings/Administrator/My Documents/Not to be seen document.txt (drive
C) (file: .)|
17108606|0|0|0|4096|1249399169|1249399169|1249399169|1249399169
```

3.1.19. Restore Points

Windows restore points are a collection of system files that are backed up to make it possible to restore to a previous state if something fails. Restore points are created for various reasons within the operating system. They are created both on a predefined regular basis (normally once a day) or when software is installed that complies with the system restore API, before an unsigned driver is installed, when OS updates and security

patches are applied and if the user manually creates one. These are the main reasons for which a restore point is created, although there are more.

The restore points are stored inside the folder System Volume Information in the root path of the system drive. Inside it is a folder called "_restore{GID}" that contains all the restore points. The input module reads each folder within the restore point directory that contains the name RP, which stands for restore point and parses the content of the rp.log file found within it. Inside the rp.log file is information about the reason for which the restore point was created as well as the date of the creation.

An example output from the restore point input module:

```
0|[Restore Point] (Created) Restore point RP25 created - Installed
Java(TM) 6 Update 11 (file:
.)|17108613|16877|501|501|442|1247513570|1247513570|1247513570|12475135
70
```

3.1.20. SetupAPI

Microsoft Windows XP introduced a file called SetupAPI.log that contains information about device, service-pack and hot fix installations. The file also contains information about device and driver changes and major system changes (Microsoft Corporation, 2003).

The file is a plain text file that is structured basically in two sections:

- Header
- Device or Driver installation section

The header contains the following format:

```
[SetupAPI Log]
OS Version =
Platform ID =
Service Pack =
Suite =
Product Type =
Architecture =
```

Each section is setup in the following format:

```
[DATE TIME PID.INSTANCE MESSAGE_DESCRIPTION]
#XYYY Description of event
#XYYY…
…
```

The X is either a "-", "I" for information or "E" for error. The YYY or the message code is an integer indicating the type of information provided in the description.

```
[2009/09/06 16:49:51 668.35 Driver Install]
#-019 Searching for hardware ID(s): nmwcd\vid_0421&pid_00b0&if_lc
#-198 Command line processed: C:\WINDOWS\system32\services.exe
#I022 Found "NMWCD\VID_0421&PID_00b0&IF_LC" in
C:\WINDOWS\inf\oem21.inf; Device: "Nokia E66 USB LCIF"; Driver: "Nokia
E66 USB LCIF"; Provider: "Nokia"; Mfg: "Nokia"; Section name: "OtherX".
```

The input module starts by processing the header for global variables contained within the setupapi.log file before it reads each device or driver installation section. Because of the share amount of information that is included within the SetupAPI file the description field in the timeline can get quite long (shortening it requires somewhat more processing by the input module). An example output of the setupapi input module is:

```
0|[SetupAPI Log] (Timestamp) Context: Driver install entered (through
services.exe). Information: . Context: Reported hardware ID(s) from
device parent bus. Context: Reported compatible identifiers from device
parent bus. Information: Compatible INF file found. Information:
Install section. Context: Processing a DIF_SELECTBESTCOMPATDRV request.
Information:  [c:/program files/vmware/vmware
tools/drivers/scsi/vmscsi.inf]. Information: . Information: .
Information: . Context: Copy-only installation
[PCI/VEN_104B&DEV_1040&SUBSYS_1040104B&REV_01/3&61AAA01&0&80]. Context:
Processing a DIF_SELECTBESTCOMPATDRV request. Information: . Context:
Processing a DIF_SELECTBESTCOMPATDRV request. Context: Installation in
progress [c:/program files/vmware/vmware
tools/drivers/scsi/vmscsi.inf]. Information: . Context: Processing a
DIF_SELECTBESTCOMPATDRV request. Information:
[PCI/VEN_104B&DEV_1040&SUBSYS_1040104B&REV_01/3&61AAA01&0&80]. Warning:
[SCSI/Disk&Ven_VMware_&Prod_VMware_Virtual_S&Rev_1.0/4&5fcaafc&0&000].
Warning: . Information: Device successfully setup
[PCI/VEN_104B&DEV_1040&SUBSYS_1040104B&REV_01/3&61AAA01&0&80].  (file:
setupapi.log)|18318719|0|0|0|0|1240698145|1240698145|1240698145|1240698
145
```

3.1.21. Local Shared Object ("Flash Cookies")

If the commonly used Adobe Flash player needs to save data locally to a disk it does so by using so called local shared objects or Flash cookies as they are so often named. A Flash cookie is a file that contains serialized shared objects that save the current state or configurations of a Flash application, and is often used to store information such as user settings. Web developers and advertisers have increasingly started to use these files to store other information than just the volume and video player settings. These files are now more often used to store the same information as can be

found inside traditional browser cookies. The benefit of storing the cookie information inside a local shared object is that it is not affected when the user clears his or hers cookies and it can be shared independent of the browser that is used. This also means that despite the privacy modes of many browsers today, where they do not store any type of browser history on the local disk, local shared objects or flash cookies can still be collected, stored and examined to determine the browser history. This might change though with the introduction of version 10.1 of the Flash player, where it respects private settings of browsers and only stores local shared object files within the computers memory (RAM).

The LSO is stored as a binary file in a network or big-endian style. The file is structured in three sections:

- First 16 bytes are the file's header.
- The objects name in UTF-8
- A payload encoded as an Action Message Format (AMF) (Adobe, 2006), either version zero or three.

As noted above the first 16 bytes compromise the LSO header. It is structured in the following way:

- Two bytes magic value (should be 0x00bf)
- Four bytes length of LSO file excluding the length and magic value. So to get the correct total length this value should be increased by six.
- Four bytes of magic value. This value should be the ASCII value of TCSO.
- Two bytes indicating the length of the size variable, should always equal to 0x00 04. The following four bytes then represent the version number of the AMF message that is encoded inside the LSO file. Version number 0x00 00 00 00 represents AMF0, while 0x00 00 00 03 means that the file uses AMF3.

An example header is:

```
<00bf> {0000 004b} [5443 534f] <0004 0000 0000>
```

If the header above is examined a bit more closely it can bee seen that the length of the LSO file is 0x00 00 00 4b = 75. If we add the length of the magic bytes in the beginning and the length of the int32 value that represent the length variable we get that the total length of the LSO file is 75+6 or 81 bytes (0x51).

Following the header is the objects name. The structure is simple:

- First two bytes represent the length of the name part
- The name part in UTF-8, the length according to previous value
- Four bytes of pad (should equal to 0x00 00 00 00)

An example name part is the following:

```
0014 <796f 7574 7562 652e 636f 6d2f 7365 7474 696e 6773> {0000 0000}
```

The length of the name is the bold part, 0x14 or 20 bytes. The next 20 bytes then represent the name part, in this case youtube.com/settings.

After the name part ends we get to the actual payload, which is either encoded using AMF0 or AMF3. The current version of log2timeline only supports version AMF0, so this chapter only explains the details of that particular format. The vast majority of Flash cookie files that have been seen on web sites use AMF0, at least at the time when this paper is written, however support for AMF3 should get into log2timeline in near future versions of the tool.

The payload is structured in the following way:

- Two bytes represent the length of the name in UTF-8
- The name of the data/variable
- Data marker indicating the type of the data/variable
- Additional data, depending on the data type

There are several different data types available, each one represented differently. The available data type markers are:

- 0x00 – NUMBER. This is a double number so the next eight bytes represent the number.

- 0x01 – BOOLEAN. One byte is read, either it is TRUE (value of 0x01) or FALSE (value of 0x00).
- 0x02 – STRING. Contains two bytes representing the length of the string, followed by that amount of bytes in UTF-8.
- 0x03 – OBJECT. An object contains various data types wrapped inside a single data type. It has the same structure as other data types, that is it begins with two bytes representing the length of the name part of the variable, followed by the name in UTF-8 and then one byte representing the data type, which is then read the same way as any other data type is read. The object then ends with the magic value of an empty UTF-8 string followed by the object end marker, or the byte code "0x00 00 09".
- 0x04 – MOVIECLIP. This data type is reserved for future use, so no further processing.
- 0x05 – NULL. This is just a null object.
- 0x06 – UNDEFINED. This is also a null object, since it has not been defined.
- 0x07 – REFERENCE. If a complex data type object has to be mentioned twice in the same file, a reference is passed instead of repeating the object. A reference consists of an unsigned 16-bit integer that points to an index in a table of previously defined objects.
- 0x08 – ECMA ARRAY. The array contains the exactly the same structure as an object, except that it starts with four bytes representing the number of elements stored inside the array. As with the objects it ends with the magic value of "0x00 00 09".
- 0x09 – OBJECT END MARKER. This is the object end marker, and if pre-pended with an empty UTF-8 string forms a magic value to mark the end of complex objects, such as arrays or typed objects.
- 0x0A – STRICT ARRAY. Is constructed much the same way as an associated or ECMA array. It starts with four bytes that contain the number of elements the

array stores, followed by each array type. A strict array ends the same way as an ECMA array, with an empty UTF-8 string followed by the object end marker.

- 0x0B – DATE. A date object is ten bytes, beginning with a double number contained in the first eight bytes followed by a signed short integer (two bytes) that represents the time offset in minutes from Epoch time. The double number represents an Epoch time in milliseconds.
- 0x0C – LONG STRING. The problem with the string object is that the length is represented as two bytes, restricting the possible length of the string. The long string object has the same structure as the string, except that it uses four bytes for the size instead of two.
- 0x0D – UNSUPPORTED. This represents an unsupported type, and no further processing is needed.
- 0x0F – XML. An XML structure begins with two bytes that should equal to zero (0x00) followed by a string value (first two bytes represent the length of the XML, then an UTF-8 text containing the XML text).
- 0x10 – TYPED OBJECT. The typed object is very similar to that of an array or an object. The only difference is a type object starts with a name of the object. The first two bytes represent the length of the name, followed by the name of the object in UTF-8. Then as usual additional data types that are contained within the typed object, ending with a magic value of "0x00 00 09".

The sol input module starts by parsing the LSO header before moving on to read the name of the LSO. The input module then enters a loop while it reads all the available data types that exist within the LSO. If the input module encounters a date object or a double number that represents a valid date it includes that in a date array. When the LSO has been completely parsed it goes through the date array and prints out the available dates from it, explaining which date is being displayed. The input module uses the filesystem timestamps if there are no valid timestamps found within the LSO file. An example output of the module is:

```
0|[LSO] LSO created -> File: youtube.com/soundData.sol and object name:
soundData variable: {mute = (FALSE), volume = (40)}
|20835066|0|0|0|0|1262603852|1262603852|1262603852|1262603852
…
0|[LSO] lastVisit -> File: www.site.com/aa_user.sol and object name:
aa_user variable: {usr_rName = (XXXXX), mainList = (Bruno_DVD_UK =>
banner2 => count => 1, hasClicked => no, lastVisit => Mon Nov 23 2009
15:43:37 GMT, count => 1, hasClicked => no), usr_rgdt = (Mon Nov 23
2009 15:43:37 GMT), usr_kbsr =  (0)}
|20835107|0|0|0|0|1258991017|1258991017|1258991017|1258991017
```

3.1.22. Squid Access Log

The Squid proxy stores information about every request made to the server in a file called access.log. The default format of this text file, which is the only format that the input module in its current version supports, is the following:

```
Timestamp Elapsed Client Action/Code Size Method URI Ident
Hierarchy/From Content
```

The timestamp is stored in Epoch time format, although with milliseconds included. Each field is space delimited, meaning that there is a space between each field. An example log line is the following:

```
1245643325.771    109 10.0.0.1 TCP_MISS/200 564 GET
http://myserver.com/admin/login.php? - DIRECT/myserver.com text/xml
```

The input module starts by removing all double or more instances of space to make each field separation exactly one space before splitting the line into the appropriate variables. An example output from the module is:

```
0|[Squid access log] (Entry written) User: 10.1.1.25 10.1.1.25
connected to 'http://10.1.1.100/admin/private/index.php?' using GET
[TCP_MISS/200] from DIRECT/10.1.1.100- content text/xml (file:
access.log)|16825458|0|0|0|564|1245643333|1245643333|1245643333|1245643
333
```

3.1.23. TLN

Harlan Carvey has created several scripts to extract timestamps from various document formats. All of his scripts output using the TLN format, which he created. The TLN input module takes timestamps from a TLN body file and parse it so that information gathered using one of his scripts can be incorporated into the overall timeline. The format of the TLN file is describe in a blog post by Carvey, http://windowsir.blogspot.com/2010/02/timeline-analysisdo-we-need-standard.html. Although the standard does not directly imply on how the fields should be presented the

default behavior is to use an ASCII output that is pipe delimited. The defined fields in the standard are:

```
Time|Source|Host|User|Description|TZ|Notes
```

The first five fields are mandatory while the last two are optional. If by any chance a mandatory field is not available it should be populated with a '-' while the optional fields should be left blank.

The time value is stored using Epoch time. An example output from one of the scripts, recbin.pl, which reads the INFO2 document inside a user's recycle bin:

```
recbin.pl -u administrator -s joesmachine -t -i INFO2

1249399169|INFO2|joesmachine|administrator|DELETED - C:\Documents and
Settings\Administrator\My Documents\Not to be seen document.txt
```

And the corresponding line using the TLN input module to parse the tln body file:

```
0|[INFO2 from TLN] (Entry) <joesmachine> User: administrator DELETED -
C:/Documents and Settings/Administrator/My Documents/Not to be seen
document.txt (file:
recbin_body.tln)|24885266|0|0|0|0|1249399169|1249399169|1249399169|1249
399169
```

3.1.24. UserAssist

The UserAssist key in the Windows registry stores information about user activity, such as the last execution time of applications started using the GUI (Windows Explorer, not from the command line) and the run count of these applications.

The input module was originally built on the userassist.pm plug-in in the RegRipper tool written by Harlan Carvey yet has since been modified both to fit into the log2timeline framework as well as to add support to the newer versions of the UserAssist key in Windows 7 and newer operating systems.

The UserAssist key is stored in the user specific registry file NTUSER.DAT. The location of the key differs from different operating systems, yet the all store the key within the Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist. The UserAssist values in Windows XP are stored underneath {75048700-EF1F-11D0-9888-006097DEACF9}\Count while Windows 7 stores them under the keys{CEBFF5CD-ACE2-4F4F-9178-9926F41749EA}\\Count and {F4E57C4B-2036-45F0-A9AB-443BCFE33D9F}\\Count.

The names of each key within the UserAssist portion of the registry are all encoded using ROT-13, which can be easily decoded independent on the operating system that was used. The binary structure that is the value of the keys differs though, and the newer format is described in an article written by Didier Stevens (Stevens, 2009).

The userassist input module starts by validating which sub keys are available within the UserAssist key inside the NTUSER.DAT registry file, to determine which version of the UserAssist binary structure it is dealing with. After that each key within the UserAssist key are read and parsed.

Another change in the newer format of the UserAssist key is the heavy use of globally unique identifiers (GUID) that replace many of the paths that were used in the older format of the binary structure. The actual values of the GUID's are stored in several locations within the registry, both inside the user registry file and the operating system ones. Instead of trying to read those values directly from the registry the input module depends upon the default values of these GUID's. This might cause some paths to be wrongly represented for that particular operating system, however in the majority of the cases it would be the correct value. For that reason both the default value of the path is written to the output as well as the virtual name for it.

An example output from the userassist input module for a Windows XP NTUSER.DAT file is:

```
0|[UserAssist key] (LastWritten) User: joe UEME_RUNPIDL:%csidl2%/BCWipe
3.0 [Count: 1] (file:
NTUSER.DAT)|17112640|0|0|0|0|1249399312|1249399312|1249399312|124939931
2
```

And for a Windows 7 operating system:

```
0|[UserAssist key] (LastWritten) User: kiddi [Program Files] [X86]
%ProgramFiles% (%SystemDrive%/Program Files)/Microsoft SQL
Server/90/Tools/Binn/VSShell/Common7/IDE/ssmsee.exe [Count: 2]  nr. of
times app had focus: 15 and duration of focus: 285279ms (file:
ntuser.kiddi.dat)|21312002|0|0|0|0|1257755350|1257755350|1257755350|125
7755350
```

3.1.25. Windows Shortcut Files (LNK)

The Windows operating system creates shortcut files for various reasons, one of which is to remember recently opened documents or applications. The recent folder in

Windows often contains valuable information to an investigator, such as information about which documents where opened on a USB stick or other removable drive.

The shortcut files, or LNK files are stored in a binary format that has been reversed engineered by Jesse Hager (Hager, J., 1998). In his white paper, The Windows Shortcut File Format, the format of the shortcut files is explained in great detail. The win_link input module was originally based on a script written by Harlan Carvey, lslnk.pl however it was heavily modified to both fix character encoding issues and to incorporate further information gathered from the file as described in the white paper by Jesse.

An example output from the input module:

```
0|[LNK] C:/VirtualMachines/SIFT Workstation v1.3/Forensic
Workstation.vmx <-Forensic Workstation.lnk, which is stored on a local
vol type - Fixed, SN 1913823382 - Rel path:
../../../../../../../VirtualMachines/SIFT Workstation v1.3/Forensic
Workstation.vmx Working dir: C:/VirtualMachines/SIFT Workstation v1.3
[a rel. path str,SI ID exists,working dir.,points to a file or dir] -
mod since last backup
|49|0|0|0|979|1265649978|1239992300|1239992300|1239992300
```

3.1.26. XP Firewall Logs

A firewall was included with Windows XP with the introduction of the service pack 2. Reviewing the firewall log can often be very valuable since it can include information about every dropped or allowed connection (depending on how it was configured).

The XP firewall log file is stored by default in the C:\Windows directory as the file pfirewall.log. Its structure is based on the W3C log format, which implies that the structure of the field is given in the file's header. By default the structure includes the following fields (Internet connection firewall security log file overview, 2010):

- Date. The day and year of the event in question, the format is YY-MM-DD.
- Time. The time when the event took place, formatted in the following manner: HH:MM:SS (24 hour format). The time zone information is defined in the header.

- Action. Specifies the operation that was observed by the firewall, such as OEPN,DROP, etc.
- Protocol. The protocol that used during the communication.
- src-ip. The source IP of the communication.
- dst-ip. The destination IP of the communication.
- src-port. The source port used by the sending computer.
- dst-port. The port of the destination computer.
- Size. Specifies the packet size in bytes.
- Tcpflags. The TCP control flags found in the TCP header of an IP packet.
- Tcpsyn. Specifies the TCP sequence number in the packet.
- Tcpack. Specifies the TCP acknowledgement number in the packet.
- Tcpwin. Specifies the TCP window size in bytes in the packet.
- Icmptype. Specifies a number that represents the type field of an ICMP message.
- Icmpcode. Specifies the code field of an ICMP message.
- Info. The informational entry that describes the type of action that occurred.

The input module begins with parsing the header of the file to get the available fields within the file, and to configure their location within the log file. It then proceeds to parsing every line within the log file, extracting the timestamps and relevant information.

An example output from the input module:

```
0|[XP Firewall] DROP UDP 192.168.1.6:68 > 255.255.255.255:67
|546|0|0|0|328|1173251940|1173251940|1173251940|1173251940
0
```

3.2. Output Modules

In the currently released version of log2timeline at the time of publication, version 0.50, there are ten supported output modules. To get a full list of the available output modules in the tool use the parameter "-o list" in the log2timeline front-end.

3.2.1. BeeDocs

BeeDocs is a timeline visualization tool designed for the Mac OS X platform. It can import data using a TDF or a tab delimited file. The structure of the TDF is the following:

```
Label        Start Time          End Time
```

An example timeline when exported from log2timeline:

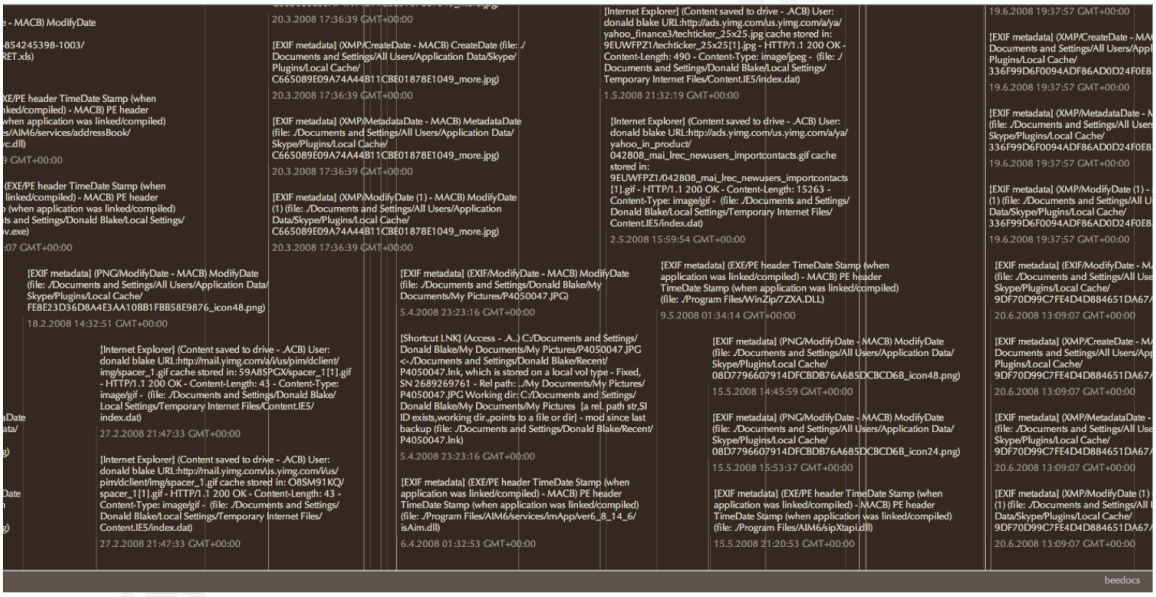

Figure 10: An example timeline in BeeDocs

3.2.2. Common Event Format (CEF)

The Common Event Format or CEF is an open log management standard created by ArcSight to standardize log file output and make it easier to import data from different sources into a enterprise management system. The CEF output module was created so that the output of log2timeline could be imported into already existing tools for log analysis that support the CEF format.

The format used by log2timeline is the following:

```
CEF:0|log2timeline|timeline_cef_output|0.1|<SOURCE>|Timeline
Event|5|dvc=<IP ADDRESS> dvchost=<HOSTNAME> smac=<MAC ADDRESS>
fsize=<SIZE OF FILE> filePermission=<PERMISSION OR MODE> suid=<USER ID>
fileID=<MFT OR INODE INFORMATION> fname=<FILENAME> suser=<USERNAME>
act=EventTime rt=<TIMESTAMP> msg=<MESSAGE>
```

There are few variables found within the CEF output module that are not currently extracted from the artifacts. To use the CEF properly the user has to supply values to some of these variables with parameters to the front-end. The parameters that the output module accepts are:

- dvc
- dvchost
- smac
- suser

By design all parameters are processed by the front-end and those parameters that are not understood by the front-end themselves produce errors. Therefore to pass a parameter to the output module it is necessary to separate the parameters passed to log2timeline from those that are sent to the input and output module with the parameter --. An example usage of the CEF output module using log2timeline front-end is:

```
log2timeline -f userassist -o cef -z local NTUSER.DAT -- -
smac=00:11:22:33 -host=SMITHSLAPTOP -dvc=10.0.0.1
…
```

```
CEF:0|log2timeline|timeline_cef_output|0.2|REG|Timeline Event|5|dvc=10.0.0.0.1
dvchost=SMITHSLAPTOP smac=00:11:22:33  fsize=0 filePermission=0 suid=0 fileID=1534979
fname=NTUSER.DAT suser=joe act=LastWritten rt=1249398822000 msg=[MACB]
UEME_RUNPATH:C:/Program Files/Internet Explorer/iexplore.exe [Count: 1]
```

3.2.3. Computer Forensics TimeLab (CFTL)

Computer Forenics TimeLab or CFTL is a similar tool as log2timeline although it is based on a GUI that uses visualization as its main presentation method of the timeline (Olsson, Boldt, 2009). Despite not being released to the general public yet the XML schema that the tool uses has been released in the white paper that describes it. CFTL uses the following XML schema:

```
<EvidenceCollection>
      <Evidence title=CDATA type=CDATA id=CDATA parent=CDATA>
            <Chunk from=CDATA to=CDATA />
            <Timestamp type=CDATA value=CDATA origin=CDATA>
```

```
            <Data name=CDATA value=CDATA>
      </Evidence>
</EvidenceCollection>
```

The output module cftl outputs the timestamp information using this XML schema. An example output is:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EvidenceCollection SYSTEM "CyberForensicsTimeLab.dtd">
<!--
Created by log2timeline for CyberForensics TimeLab.
Copyright(C) 2009 Kristinn Gudjonsson (log2timeline)
Copyright(C) 2008 Jens Olsson (CFTL)
-->
<EvidenceCollection>
…
<Evidence
      title="[UserAssist] User: joe - UEME_RUNPATH:C:\Program
Files\Internet Explorer\iexplore.exe [Count: 1]"
      type="UserAssist"
      id="6" parent="">
      <Chunk from="0" to="0"/>
      <Timestamp type="Timestamp extracted" value="2009-08-04
15:13:42.0" origin="UserAssist" />
      <Data />
</Evidence>
…
</EvidenceCollection>
```

3.2.4. CSV

This output module is very simple. It creates a file where each value found within the timeline object is printed using comma as a separator. The only processing that is done is to convert all timestamps into a human readable format and to include them first. Other keys and values of the timestamp object are printed in the same order they are stored within the timestamp object. An example output is:

```
time,timezone,type,macb,sourcetype,source,short,version,desc,user,host,
filename,inode,notes,format,extra
Tue Aug 04 2009 15:19:23,GMT,LastWritten,MACB,UserAssist
key,REG,UEME_RUNPATH:C:/WINDOWS/system32/NOTEPAD.EXE,2,UEME_RUNPATH:C:/
WINDOWS/system32/NOTEPAD.EXE [Count: 2],joe,-,NTUSER.DAT,1534979,-
,Log2t::input::userassist,-
Tue Aug 04 2009 15:21:52,GMT,LastWritten,MACB,UserAssist
key,REG,UEME_RUNPIDL:%csidl2%/BCWipe 3.0,2,UEME_RUNPIDL:%csidl2%/BCWipe
3.0 [Count: 1],joe,-,NTUSER.DAT,1534979,-,Log2t::input::userassist,-
```

3.2.5. Mactime

The tool was originally built to support mactime output that explains many of the keys that are supported within the timestamp object. The mactime format or the content

of a bodyfile as defined by the Sleuthkit (TSK) has changed since of version 3.+ of the tool. The format for version 3.+ is the following:

```
MD5|name|inode|mode_as_string|UID|GID|size|atime|mtime|ctime|crtime
```

And the older format, which is the format used by TSK version 1.x and 2.x is the following:

```
 MD5 | path/name | device | inode | mode_as_value | mode_as_string |
num_of_links  | UID | GID | rdev | size | atime | mtime | ctime |
block_size | num_of_blocks
```

En example output of both of these formats is the following:

```
0|[UserAssist key] (LastWritten) User: joe UEME_RUNPATH:C:/Program
Files/Internet Explorer/iexplore.exe [Count: 1] (file:
NTUSER.DAT)|1534979|0|0|0|0|1249398822|1249398822|1249398822|1249398822
```

And for the older version:

```
0|[UserAssist key] (LastWritten) User: joe UEME_RUNPATH:C:/Program
Files/Internet Explorer/iexplore.exe [Count:
1]|0|1534979|0|0|0|0|0|0|0|1249398822|1249398822|1249398822|0|0
```

3.2.6. SQLite

One of the original ideas with the creation of log2timeline was the ability to create another tool to assist with the analysis after gathering all the events. The SQLite output module creates a SQLite database that can be used for this purpose. The table schema can be seen in Figure 11:

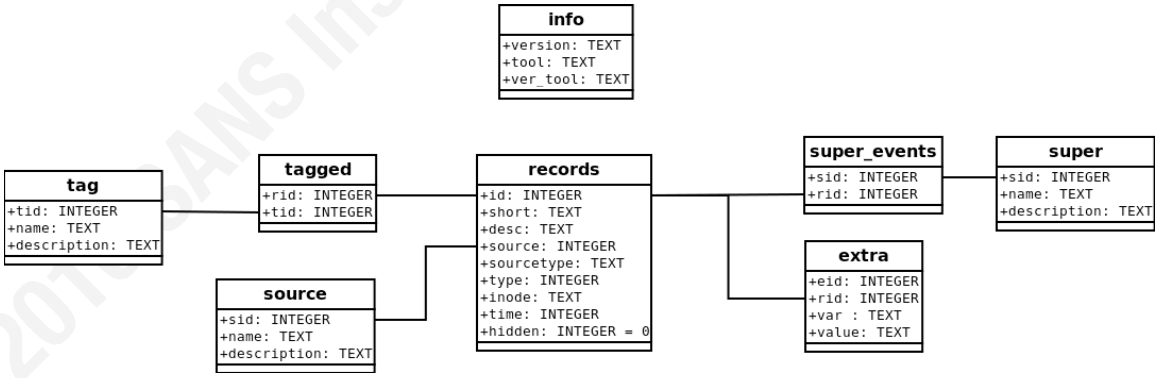

Figure 11: Shows the database schema for the SQLite database output

This proposed SQLite schema may change in the future when the tool that is supposed to read from this database is created.

This particular schema contains tables to store host information, records, source and tags. Another table of interest is the super and super_events that contains information about super events, or group of events that are part of the same activity.

An example content found within the records table of the SQLite database is:

```
sqlite> SELECT * FROM records;
…
7|'[[X86] System32] %windir%/system32/mspaint.exe'|'[[X86] System32]
%windir%/system32/mspaint.exe [Count: 8] nr. of times app had focus: 9
and duration of focus: 180000ms'|1|'UserAssist
key'|15|NULL|'1254652530'|0
```

3.2.7. SIMILE

There are several options available for timestamp visualization. One example of such visualization techniques is a SIMILE widget that is according to the tools website (simile-widgets.org/): "Free, Open-Source Data Visualization Web Widgets, and More". The tool reads the timestamp information from a XML file with the following structure:

```
<data wiki-url="http://simile.mit.edu/shelf/" wikiSection=CDATA>
      <event start=CDATA durationEvent=CDATA title=CDATA>CDATA</event>
      …
      <event>…</event>
</data>
```

An example XML file is:

```
<data wiki-url="http://simile.mit.edu/shelf/" wikiSection="Timeline
produced from log2timeline">
        <!-- Sources:
                log2timeline produced timeline data
        -->
…
      <event start="Tue Aug 04 2009 15:18:27 GMT"
            durationEvent="false"
            title="[Firefox3] User: kristinn visited the URL:
      http://www.jetico.com/ (Jetico - Military-Standard Data
      Protection Software - Wiping, Encryption, Firewall) [count: 1]
      Host: www.jetico.com visited from:
      http://tech.yahoo.com/blogs/null/50840 (URL not typed directly)
      type: LINK"
      >
      [Firefox3] User: kristinn visited the URL: http://www.jetico.com/
(Jetico - Military-Standard Data Protection Software - Wiping,
Encryption, Firewall) [count: 1] Host: www.jetico.com visited from:
http://tech.yahoo.com/blogs/null/50840 (URL not typed directly) type:
LINK
      </event>
…
</data>
```

To use the SIMILE XML file a HTML file has to be created that uses the XML file. An example file is included with the tool in the examples/simile_visual folder.

An example output widget after creating the HTML file:

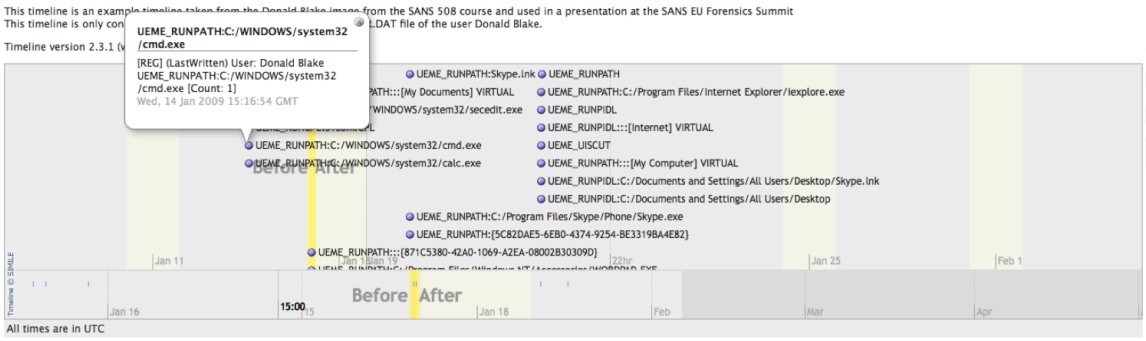

Figure 12: An example SIMILE widget

3.2.8. Timeline Format (TLN)

This output module outputs the timestamp object in the TIMELINE or TLN format that H. Carvey defined and is further discussed in chapter 3.1.23.

An example output is:

```
1254652530|REG|-|joe|(UserAssist key) [LastWritten] [[X86] System32]
%windir%/system32/mspaint.exe [Count: 8] nr. of times app had focus: 9
and duration of focus: 180000ms|GMT|File:ntuser.dat inode:1527387
```

3.2.9. Timeline Format (TLN) in XML

This output module outputs the timestamp object in the TIMELINE or TLN format that H. Carvey defined. More detailed description of the format can be found in chapter 0.

The standard itself does not contain any description on how it should be outputted, making it optional. The standard describes one example, which is an ASCII representation with a pipe symbol in between fields. Another method of presenting the TLN format is to use XML. The output module tlnx does that, which is to output the TLN format as a XML document.

An example output is:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Created by log2timeline.
Copyright(C) 2009-2010 Kristinn Gudjonsson (log2timeline)
```

```
-->
<Events>
<Event>
        <time>1254652530</time>
        <source>REG</source>
        <host>-</host>
        <user>joe</user>
        <description>(UserAssist key) [LastWritten] [[X86] System32]
%windir%/system32/mspaint.exe [Count: 8]  nr. of times app had focus: 9
and duration of focus: 180000ms</description>
        <tz>GMT</tz>
        <notes>File:ntuser.dat inode:1527387</notes>
</Event>
</Events>
```

4. Timeline Analysis

Timeline analysis focuses around the technique of examining the timeline that has been produced using either manual methods or from an automatic tool such as log2timeline. To properly analyze the timeline it is vital to properly understand it's structure and meaning. This chapter is an introduction into the subject of timeline analysis and therefore it will only touch on the subject of few techniques that can assist investigators in their timeline analysis.

One thing to keep in mind regarding timeline analysis is that although it has the potential to reduce investigation time, it requires the investigator to know what he or she is doing. The data needs to be reviewed and the context of the events understood for it to really assist with the investigation. Although timelines can be useful in most investigations, it might not suit them all, requiring the investigator to make an educative decision whether or not to create a timeline. Another think to keep in mind is data reduction, for some investigations it might not be required to simply extract all available timestamps, since that could potentially drown the investigator with unnecessary information. Instead it might be more valuable to only include limited sources in the timeline, feature that is currently available in timescanner. For other investigations however it might make perfect sense to simply get as much information as possible, and then sift through them afterwards using data reduction techniques.

4.1. Timestamp Values

The available timestamps within TSK are the atime, mtime, ctime and the crtime. The framework log2timeline uses the same timestamps to accommodate the mactime format, although a more descriptive indication is given in the filename field. The meaning of these fields differs between both artifacts parsed by log2timeline as well as different filesystems using TSK. The meaning in the context of three popular filesystems that is the NTFS, FAT and EXT3 is:

- **atime**
  This timestamp represents the last access time to a file.
- **mtime**
  This timestamp represents the last modification time of a file.
- **ctime**
  This timestamp represents different values depending on the filesystem context. In NTFS it represents the last time the MFT was modified, while in EXT3 it represents inode changes. FAT uses this timestamp to represent when the file was created.
- **crtime**
  This timestamp represents the time a file was created in NTFS while it is used for file deletion in EXT3 and not used at all in FAT.

The SANS course, Computer Forensic Investigations and Incident Response, goes over traditional timeline analysis in quite some detail, leaving this paper to rather explain the addition of extended events. It is important to know what a particular timestamps means in the context that is given in the timeline. The default output of log2timeline is to use the mactime body format, which can then be used in conjunction with other tools to augment the timeline, tools such as TSK. If the default output of mactime is used then it is necessary to understand the meaning of MACB for each module, since the timestamps might have different meanings depending upon which input module is used. Most of the artifacts that are parsed by the input modules only contain one timestamp, causing all

timestamp types to be filled with the same value. The following table illustrates the meaning of the four timestamps in the currently available input modules:

Input module	**atime**	**mtime**	**ctime**	**crtime**
chrome	The time a URL was visited or a file downloaded			
evt	The time when the event is registered in the Event log file			
evtx	The time when the event is registered in the Event log file			
exif	The time when the event is registered in the Even log file			
ff_bookmark	The time when a bookmark was created or visited or when a bookmark folder was created or modified (the type is contained within the description)			
firefox2 (two entries)	The time when a URL was first visited	The time when a URL was last visited.		
firefox2 (one entry)	The when a URL was visited			
firefox3 (URL record)	The time when a URL was visited			
firefox3 (bookmark)	When a bookmark or a bookmark folder was last modified		When a bookmark or a bookmark folder was added	
iehistory	Meaning different depending on the location of the file – cache meaning when the file was saved on the hard	When the user visited the URL – but can differ between index.dat files		

	drive			
iis	The time when the entry was logged down			
isatxt	The time when the entry was logged down			
mactime	atime of the record	mtime of the record	ctime of the record	crtime of the record
mcafee	Entry written			
opera	The time a user visited the URL			
oxml	The time of the action that is referred to in the text			
pcap	The time when the packet was recorded			
pdf	The time of the action that is referred to in the text			
prefetch	The time when the executable that the prefetch file points to was executed last			
recycler	The time the file was sent to the recycle bin			
restore	The time when the restore point was created according to the timestamp in the rp.log file			
setupapi	The timestamp in the log file for each entry, indicating the time when the information was logged down			
sol	The time of the event that was found inside the file			
sol (no timestamp found)	The last access time of the sol file	The last modification time of the sol file	The last inode or metadata change time of the sol file	
squid	The time when the entry was logged down			
tln	The timestamp within the TLN entry			

userassist	The extracted timestamp from the binary value of the registry entry		
win_link	The extracted last access time from the LNK file	The extracted last modified time from the LNK file	The extracted last creation time from the LNK file
xpfirewall	The time when the entry was logged down		

Table 5: Shows the meaning of timestamp values in the available input modules

One of the problems with timeline analysis is the amount of timestamp events within any given suspect drive. Just the traditional filesystem timestamps could yield to few hundred thousand events with the addition of perhaps a hundred thousand events from log2timeline (on an average size laptop hard drive). Some investigations have a narrow time frame that needs to be examined and in those cases the amount of events might not be a problem, yet in other cases data reduction becomes a vital part of timeline analysis. Timestamp entries of interest to any given investigation usually turn up to be the event of least frequency of occurrence (LFO) (Silberman, P, 2010), which leads to the necessity of a thorough inspection of every event in the timeline. Although this paper does not propose any magic solutions to this problem it will discuss some general techniques and possible methods of data reduction and analysis.

4.2. Event Context

Although the meaning of each timestamp is important it is necessary to examine each timestamp in conjunction with other surrounding events to get the context needed to determine the true reason behind the timestamp entry.

```
0|[XP Prefetch] (Last run) DFRGNTFS.EXE-269967DF.pf - [DFRGNTFS.EXE]
was executed - run count [12]- full path:
[C:/WINDOWS/SYSTEM32/DFRGNTFS.EXE] - DLLs loaded:
{WINDOWS/SYSTEM32/NTDLL.DLL - WINDOWS/SYSTEM32/KERNEL32.DLL -
WINDOWS/SYSTEM32/MSVCRT.DLL - WINDOWS/SYSTEM32/ADVAPI32.DLL -
WINDOWS/SYSTEM32/RPCRT4.DLL - WINDOWS/SYSTEM32/SECUR32.DLL -
WINDOWS/SYSTEM32/GDI32.DLL - WINDOWS/SYSTEM32/USER32.DLL -
WINDOWS/SYSTEM32/COMCTL32.DLL - WINDOWS/SYSTEM32/SHELL32.DLL -
WINDOWS/SYSTEM32/SHLWAPI.DLL - WINDOWS/SYSTEM32/OLE32.DLL -
WINDOWS/SYSTEM32/VSSAPI.DLL - WINDOWS/SYSTEM32/ATL.DLL -
WINDOWS/SYSTEM32/OLEAUT32.DLL - WINDOWS/SYSTEM32/NETAPI32.DLL -
WINDOWS/SYSTEM32/SHIMENG.DLL - WINDOWS/APPPATCH/ACGENRAL.DLL -
WINDOWS/SYSTEM32/WINMM.DLL - WINDOWS/SYSTEM32/MSACM32.DLL -
WINDOWS/SYSTEM32/VERSION.DLL - WINDOWS/SYSTEM32/USERENV.DLL -
```

```
WINDOWS/SYSTEM32/UXTHEME.DLL - WINDOWS/SYSTEM32/IMM32.DLL -
WINDOWS/WINSXS/X86_MICROSOFT.WINDOWS.COMMON-
CONTROLS_6595B64144CCF1DF_6.0.2600.5512_X-WW_35D4CE83/COMCTL32.DLL -
WINDOWS/SYSTEM32/RPCSS.DLL - WINDOWS/SYSTEM32/DFRGRES.DLL -
WINDOWS/SYSTEM32/CLBCATQ.DLL - WINDOWS/SYSTEM32/COMRES.DLL -
WINDOWS/SYSTEM32/XPSP2RES.DLL - WINDOWS/SYSTEM32/WINSTA.DLL -
WINDOWS/SYSTEM32/MSCTF.DLL} (file: ./DFRGNTFS.EXE-
269967DF.pf)|1535139|0|0|0|0|1248014556|1248014556|1248014556|124801455
6
```

It can be very difficult to determine the true meaning of each event in the timeline without the context of surrounding events. For example the event above might suggest that the user had initiated a defragment of the drive or it could refer to an automatic defragment that the operating system starts regularly. To determine whether or not the defragment was started automatically or user initiated we need to examine the timeline more closely and examine events surrounding the DFRGNTFS.EXE reference. Items in the timeline that might suggest user initiated action involves running mmc.exe (which loads drivers that are connected to defragment), entries within the UserAssist key in the user registry or the use of CMD.EXE prior to the run of the event.

```
time,timezone,type,macb,sourcetype,source,short,version,desc,user,host,
filename,inode,notes,format,extra
Fri Jan 16 2009 18:26:50,EST5EDT,Created,..CB,Shortcut LNK,LNK,F:/TIVO
Research - CONFIDENTIAL - BACKUP2.doc,2,F:/TIVO Research - CONFIDENTIAL
- BACKUP2.doc <-./Documents and Settings/Donald Blake/Recent/TIVO
Research - CONFIDENTIAL - BACKUP2.lnk- which is stored on a local vol
type - Removable- SN 0xb4384803 - Working dir: F:/  [SI ID exists-
working dir.-points to a file or dir] - mod since last backup,-,-
,./Documents and Settings/Donald Blake/Recent/TIVO Research -
CONFIDENTIAL - BACKUP2.lnk,8180,-,HASH(0x100bf6158),path:  size: 376
```

The event above suggests that a file was opened on the F:\ drive on the operating system, leading to a shortcut file being created in the recent document folder. The file is stored on a local volume, which is removable, according to the entry in the timeline. Yet we do not know which type of device this is, so the need for a better context is needed. If the events surrounding the LNK are examined more closely events from the SetupAPI are seen:

```
Fri Jan 16 2009 18:24:06,EST5EDT,Entry written,MACB,SetupAPI
Log,LOG,DriverContextual information. Contextual information.
Contextual information.  Information msg 022.  Information msg 023.
Contextual information.  Information msg 063.  Information msg 320.
Information msg 060.  Information msg 058. Contextual information.
Information msg 124. Contextual information.  Information msg 056.
Contextual information. Contextual information.  Information msg 054.
Contextual information.  Information msg 123.  Information msg 121.
,2,DriverContext: Reported hardware ID(s) from device parent bus.
```

```
Context: Reported compatible identifiers from device parent bus.
Context: Driver install entered (through services.exe). Information:
Compatible INF file found. Information: Install section. Context:
Processing a DIF_SELECTBESTCOMPATDRV request. Information:
[c:/windows/inf/usbstor.inf]. Information: . Information: .
Information: . Context: Processing a DIF_SELECTBESTCOMPATDRV request.
Information: Copy-only installation
[USB/VID_05AC&PID_1301/000A270010C4E86E]. Context: Processing a
DIF_SELECTBESTCOMPATDRV request. Information: . Context: Processing a
DIF_SELECTBESTCOMPATDRV request. Context: Installation in progress
[c:/windows/inf/usbstor.inf]. Information: . Context: Processing a
DIF_SELECTBESTCOMPATDRV request. Information:
[USB/VID_05AC&PID_1301/000A270010C4E86E]. Information: Device
successfully setup [USB/VID_05AC&PID_1301/000A270010C4E86E]. ,-,-
,WINDOWS/setupapi.log,3596,-,Log2t::input::setupapi,-

Fri Jan 16 2009 18:24:10,EST5EDT,Entry written,MACB,SetupAPI
Log,LOG,DriverContextual information. Contextual information.
Contextual information.  Information msg 022.  Information msg 023.
Contextual information.  Information msg 063.  Information msg 320.
Information msg 060.  Information msg 058. Contextual information.
Information msg 124. Contextual information.  Information msg 056.
Contextual information. Contextual information.  Information msg 054.
Contextual information.  Information msg 123.  Information msg 121.
,2,DriverContext: Reported hardware ID(s) from device parent bus.
Context: Reported compatible identifiers from device parent bus.
Context: Driver install entered (through services.exe). Information:
Compatible INF file found. Information: Install section. Context:
Processing a DIF_SELECTBESTCOMPATDRV request. Information:
[c:/windows/inf/disk.inf]. Information: . Information: . Information: .
Context: Processing a DIF_SELECTBESTCOMPATDRV request. Information:
Copy-only installation
[USBSTOR/DISK&VEN_APPLE&PROD_IPOD&REV_2.70/000A270010C4E86E&0].
Context: Processing a DIF_SELECTBESTCOMPATDRV request. Information: .
Context: Processing a DIF_SELECTBESTCOMPATDRV request. Context:
Installation in progress [c:/windows/inf/disk.inf]. Information: .
Context: Processing a DIF_SELECTBESTCOMPATDRV request. Information:
[USBSTOR/DISK&VEN_APPLE&PROD_IPOD&REV_2.70/000A270010C4E86E&0].
Information: Device successfully setup
[USBSTOR/DISK&VEN_APPLE&PROD_IPOD&REV_2.70/000A270010C4E86E&0]. ,-,-
,WINDOWS/setupapi.log,3596,-,Log2t::input::setupapi,-
```

The above events indicate that a USB drive had been plugged into the system, and more specifically an Apple iPod. Further examination of the suspect user's NTUSER.DAT file along with the operating system registry files would reveal further evidence of the USB drive activity, such as to confirm the drive's mount point (drive letter), etc.

If we examine the user's registry file using RegRipper we can see in the MountPoints2 key the following value:

```
    {1647bd00-e26e-11dd-a06e-00096b5312dd}  Fri Jan 16 18:24:54 2009
(UTC)
```

If we then examine the SYSTEM registry and look for the MountedDevices key, we can see the following data:

```
Device: \??\STORAGE#RemovableMedia#7&1d6490f9&0&RM#{53f5630d-b6bf-11d0-
94f2-00a0c91efb8b}
        \??\Volume{1647bd00-e26e-11dd-a06e-00096b5312dd}
        \DosDevices\F:
```

Here we've confirmed the volume ID for the drive that was used by the user Donald Blake at the time of the shortcut file being created, and that it was using the F drive letter. If we then further examine the SYSTEM registry file looking for the ParentID that is given in the previous entry we can see in the USBStor key the following values:

```
Disk&Ven_Apple&Prod_iPod&Rev_2.70 [Fri Jan 16 18:24:10 2009]
  S/N: 000A270010C4E86E&0 [Fri Jan 16 18:24:15 2009]
    FriendlyName  : Apple iPod USB Device
    ParentIdPrefix: 7&1d6490f9&0
```

This again leads us to strengthen our believe in what the timeline told us, that an Apple iPod was connected to the machine and a document called "TIVO Research - CONFIDENTIAL - BACKUP2.doc" was opened.

```
Thu Jan 21 2010 11:33:36,130048,.a..,r/rrwxrwxrwx,0,0,17771-128-
5,C:/Documents and Settings/smith/My
Documents/Downloads/aimupdate_7.1.6.475.exe
```

The above event is taken from a traditional filesystem timeline. It is difficult to determine whether the file was truly executed, something that an updated access time to an executable might mean, or if it was accessed through a different event, such as an antivirus scan or something completely different. If the super timeline is examined around this event we can see:

```
Thu Jan 21 2010 11:30:08,130048,macb,0,0,0,0,[Chrome] User smith
downloaded file
http://update.aol.com.yhffd4.com.pl/products/aimupdate_7.1.6.475.exe
(C:\Documents and Settings\smith\My
Documents\Downloads\aimupdate_7.1.6.475.exe).  Total bytes received :
130048 (total: 130048)
…
Thu Jan 21 2010 11:33:36,130048,.a..,r/rrwxrwxrwx,0,0,17771-128-
5,C:/Documents and Settings/smith/My
Documents/Downloads/aimupdate_7.1.6.475.exe

Thu Jan 21 2010 11:33:39,0,macb,0,0,0,0,[UserAssist] User: smith -
UEME_RUNPATH:C:\Documents and Settings\smith\My
Documents\Downloads\aimupdate_7.1.6.475.exe [Count: 1]
```

The above events show more clearly what truly happened. The user smith downloaded the file aimupdate_7.1.6.475.exe using the Chrome browser and executed the file, according to the users browser history and the UserAssist key of the user's smith registry file. And if we examine the timeline little more closely the source of this malware can be seen:

```
Thu Jan 21 2010 11:28:51,0,macb,0,0,0,0,[Chrome] User: smith URL
visited: https://WEBMAIL.DOMAIN.COM// (My webmail :: Inbox) [count: 2]
Host: unknown visited from: http://109.95.114.251/usr5432/xd/sNode.php
type: [FORM_SUBMIT - A form the user has submitted values to] (URL not
typed directly)

Thu Jan 21 2010 11:28:51,0,macb,0,0,0,0,[Chrome] User: smith URL
visited: https://WEBMAIL.DOMAIN.COM//skins/default/watermark.html ()
[count: 2] Host: unknown type: [AUTO_SUBFRAME - Content automatically
loaded in a non-toplevel frame - user may not realize] (url hidden)
(URL not typed directly)

Thu Jan 21 2010 11:28:57,0,macb,0,0,0,0,[Chrome] User: smith URL
visited:
https://WEBMAIL.DOMAIN.COM//?_task=mail&_action=preview&_uid=1&_mbox=IN
BOX&_framed=1 () [count: 1] Host: unknown type: [MANUAL_SUBFRAME -
Subframe explicitly requested by the user] (url hidden) (URL not typed
directly)

Thu Jan 21 2010 11:29:16,2928,.ac.,r/rrwxrwxrwx,0,0,35176-128-
4,C:/Documents and Settings/smith/Local Settings/Application
Data/Google/Chrome/User Data/Default/Last Tabs

Thu Jan 21 2010 11:29:40,0,macb,0,0,0,0,[Chrome] User: smith URL
visited:
http://update.aol.com.yhffd4.com.pl/products/aimController.php?code=453
8143611065376028340141505947 25&email=user@email.com (AIM) [count: 1]
Host: update.aol.com.yhffd4.com.pl type: [TYPED - User typed the URL in
the URL bar] (directly typed)

Thu Jan 21 2010 11:29:41,0,macb,0,0,0,0,[Chrome] User: smith URL
visited: http://109.95.114.251/usr5432/in.php () [count: 1] Host:
109.95.114.251 type: [AUTO_SUBFRAME - Content automatically loaded in a
non-toplevel frame - user may not realize] (url hidden) (URL not typed
directly)
```

The user seems to have read an E-mail using a web browser (webmail) and then copied the link from the e-mail to the browser, hence the directly typed link pointing to the malware.

4.3. Temporal Proximity

The technique of examining objects in the timeline that are not directly related to each other yet are within temporal proximity often results in the discovery of additional

evidence that needs further inspection, evidence the investigator might have missed if no timeline analysis had been performed.

This technique is often used in malware investigations where the analyst often knows one part of the attack vector, or is able to determine that during the analysis. When the investigator has discovered something that is a part of the attack he or she can look for other objects in the timeline that are in temporal proximity to the known event. If we examine a bit further the timeline above, with the aimupdate_7.1.6.475.exe (which is a malware) we can see:

```
Thu Jan 21 2010 11:31:14,0,...b,r/rrwxrwxrwx,0,0,10809-128-
1,C:/WINDOWS/system32/lowsec/user.ds

Thu Jan 21 2010 11:31:14,464,...b,d/dr-xr-xr-x,0,0,18319-144-
1,C:/WINDOWS/system32/lowsec

Thu Jan 21 2010 11:31:14,0,macb,r/rrwxrwxrwx,0,0,18390-128-
1,C:/WINDOWS/system32/lowsec/local.ds

Thu Jan 21 2010 11:31:14,5752,...b,r/rrwxrwxrwx,0,0,18433-128-
3,C:/WINDOWS/system32/lowsec/user.ds.lll

Thu Jan 21 2010 11:31:28,196608,macb,0,S-1-5-18,0,0,[EventLog] From
host: SIMTTO-LAPTOP: Service Control Manager/7035;Info;Computer
Browser,start
```

The events above suggest that few seconds after the file aimupdate_7.1.6.475.exe was first executed there are few files created in the filesystem: C:/WINDOWS/system32/lowsec/user.ds.lll and C:/WINDOWS/system32/lowsec/local.ds. These files require further analysis by the investigator to determine whether they are relevant to the investigation or not.

Temporal proximity can be a very useful method if timestamps of files haven't been modified as a mean of fooling the investigator (anti-forensics technique). The super timeline assists in this matter by including timestamps from various sources, some of which might not have any direct methods of modification, instead of relying solely on the filesystem timestamps.

4.4. Frequency of Events – Spikes in Timeline

Depending upon the investigation in hand, sometimes it might come in handy to examine if there are any spikes in timeline activity. Although many investigations will

follow the rule of least frequency of occurrence (LFO) (Silberman, P, 2010), some might benefit from such analysis.

There are many different methods of extracting timeline spikes, one of which requires the use of a combination of the *NIX tools *gnuplot*, *awk* and *grep* as demonstrated in a simple Bash script called *extract_timespikes*.

```
#!/bin/bash
##############################################################################
###
#           EXTRACT_TIMESPIKES
##############################################################################
###
# A simple script to extract timespikes from a CSV file created by
mactime
#
# Usage: extract_timespikes CSVFILE MONTH YEAR
#
# Author: Kristinn Gudjonsson


if [ $# -ne 3 ]
then
      echo "Wrong usage: `basename $0` CSVFILE MONTH YEAR"
      exit 12
fi

# we have the correct number of arguments
CSV=$1
MONTH=$2
YEAR=$3

# check if the CSV file exists
if [ ! -f "$CSV" ]
then
      echo "Wrong usage: File '$CSV' does not exist"
      exit 10
fi

# start extracting timespikes
cat $CSV | grep -i "$MONTH [0-9][0-9] $YEAR" | awk -F ',' '{print $1}'
| awk '{print $1,$2,$3,$4}' | uniq -c > monthly.dat
```

The script is then called using the following command:

```
extract_spikes mywebserver_supertimeline.csv may 2010
```

This results in a file called “monthly.dat” being created that contains the statistic for May 2010 for a web server that got compromised. To be able to visually inspect the time spikes we can use *Gnuplot*. A simple *Gnuplot* script was written, and saved as *extract.gnu*:

```
#############################################################
# A gnuplot file for creating a PNG image of timeline spikes
#############################################################
# Author: Kristinn Gudjonsson

set output "timeline_spike_month.png"
set terminal png
set key below      # enable the legend and place it below the plot

# we define our line/point combinations
set style line 1 lt 1 lw 2
set style line 2 lt 2 lw 1 pt 4 ps 2
set style line 3 lt 4 lw 1 pt 6 ps 2
set style line 4 lt 1 lw 3 pt 2 ps 2
set style line 5 lt 7 lw 1 pt 12 ps 2
set style line 6 lt 3 lw 1.5 pt 3 ps 2

set xzeroaxis lt 2 lw 2

set mxtics 2                 # I want two minor tic marks on the x axis
set grid xtics ytics mxtics  # enable the grid with major tic marks

# and now the text
set title "Timeline Spikes"
set xlabel "Day"
set ylabel "Events"
set xrange [1:]        # choose the x data range on the plot

# count day     - the setup of T1
plot “monthly.dat" using 4:1 with lines t "Number of Events" ls 1;
```

And to plot the data we simply use the command *Gnuplot*:

```
gnuplot extract.gnu
```

The resulting graph can be seen in Figure 13. In this particular case we had a web server that got scanned prior to be being compromised. The timeline contains among other entries from the IIS web server log, making it ideal to spot anomalies such as scanning activity, using frequency of events. This particular server got scanned on the 16th of May 2010, and then later during that day compromised.

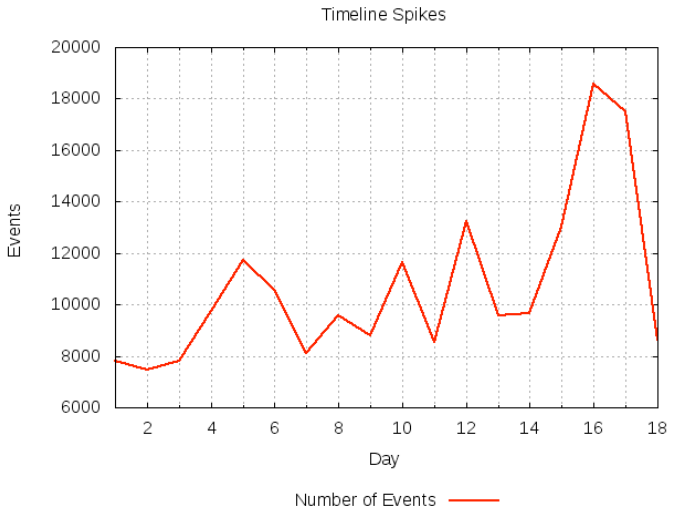

Figure 13: A simple graph showing the frequency of timestamp events in a month period

4.5. Using Analysis Tools

A super timeline often contains too many events for the investigator to fully analyze, making data reduction or an easier method of examining the timeline essential. Often investigators resort to the use of simple tools such as the UNIX tool *grep* or to export the timeline in CSV format so it can be opened in a spreadsheet application. This methodology does not scale very well, especially when multiple sources are added to the timeline.

There are other tools that exist that can be of value during timeline analysis. Tools such as Splunk can be utilized to make indexing and searching through the timeline easier. Another option would be to use host based intrusion systems, such as OSSEC to detect known "bad" parts of the timeline using pre-defined patterns.

Splunk can be used to correlate timelines from multiple images as well as other information, such as firewall log files or any other log file that Splunk is capable of importing. The log2timeline framework allows for an easier integration with Splunk using the Common Event Format (CEF) output module. Currently Splunk does not know how to properly decode the output from log2timeline however work is undergone to create either a Splunk application or some other mean to properly import and display the timeline data. When that application is completed it will hopefully provide a better and

more productive method of correlating timeline data with other sources of information along with visualization of the extracted timestamps.

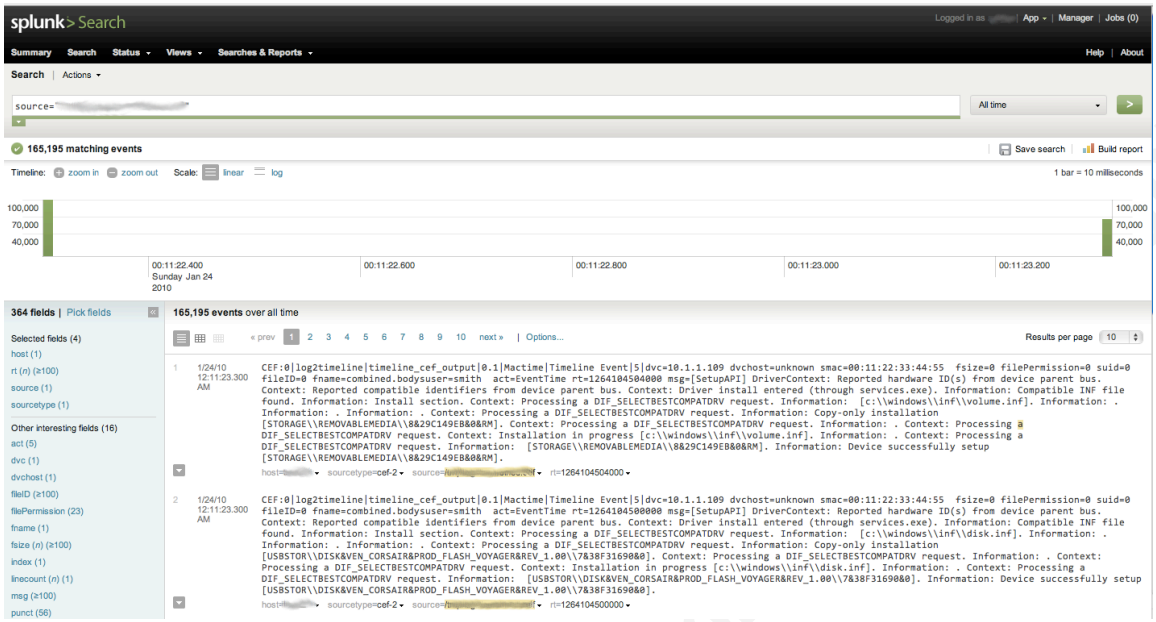

Figure 14: Shows a picture of an imported CEF file into Splunk

OSSEC is capable of parsing through several log files and compare them to pattern files to detect known "evil" or otherwise suspicious events. In the latest snapshot version (feature available as of version 2.4) OSSEC includes the tool ossec-logtest that is capable of testing a single log file, such as a timeline created using log2timeline. It is possible to create a new decoder in OSSEC that is capable of parsing through a timeline from log2timeline and run it against its patterns or rules. That could provide a great starting point for investigators, especially during a malware investigation or other intrusion analysis. OSSEC could quickly go through the timeline and alert the investigator about potential suspicious entries, such as additions to known startup keys in registry, use of anti-forensics tools, etc. Other techniques of timeline analysis would then follow since pattern-matching techniques are not capable of finding all "evil".

5. Future Development

The framework is far from being complete and in fact it will never really be, since there are always new operating systems, new file types, etc, that will contain timestamps that need to be parsed and added to the super timeline. The roadmap of the tool does give

a pretty good indication of where the tool is heading in the near future, although being a very dynamic one.

In the near future most of what the roadmap focuses on is additions of new input modules, to make the tool capable of parsing more artifacts to include into the timeline. Other future developments include code optimization and general improvements of the code. The time required for timescanner to sift through an entire mounted image file increases as each new input module is introduced into the tool. This is obvious since the tool scans each file and directory in the filesystem and tries to verify the structure using every available input module, making optimization a very crucial part of the development. One method of improving the speed of timescanner is to make the tool multi-threaded. Such an implementation could imply that a new thread would be created for each file that passes the verification phase, so that the parser itself is a single thread. That way the tool could continue it's search through the mounted image while the artifact is being processed. Another option would be to limit the amount of input modules to test against or in other words to provide some sort of sorting algorithm to the tool so that each file is only evaluated against artifacts the input module is likely to be able to parse. That way the run time could be reduced by limiting the number of input modules that need to be run against each found file or directory within the image file. Other optimization tricks exist, such as to improve algorithms, use references instead of copying variables to functions, optimize while loops, etc.

The tool was originally written for use on Linux and Mac OS X since those are the platforms that the author of this paper uses for most of his investigations. In the future it is not unlikely that it will be ported to other platforms, such as the Windows operating system. Most of the code is already compatible with Windows, yet there are some parts of it that are *NIX specific that need to be modified so the tool can be used properly.

The front-end timescanner was originally written as a separate front-end from the two others. In the future it will be merged into the main tool to reduce the front-ends to two; a CLI version and a GUI one. This will also affect the updating mechanism that is currently in place. One of the items in the roadmap discusses moving the update

mechanism so that it will be incorporated into the front-ends instead of being an independent bash script.

The GUI needs serious overhaul to be really useful. The GUI will most likely be completely rewritten in another programming language than Perl, perhaps in Java so it can be used on other platforms as well. It needs to be made more accessible as well as to provide more functions into it, making it easy to create the timelines using the push of a button.

Improvements that are scheduled for log2timeline include options such as to guess the log file that is to point the tool at a single log file and make the tool guess the format of the file and to provide aliases for the input modules. Currently the user has to know that index.dat files that belong to Internet Explorer has to be parsed using iehistory input module. Perhaps it would be better to provide aliases, so that the index.dat file could be parsed with other names as well.

One of the problems so to speak with creating a super timeline is the vast amount of data that you are including in the timeline. With each new input module and especially if input modules are developed for IM conversations or e-mail, the amount of events increases exponentially. This creates an extra burden on the investigator that needs to sift through seemingly endless amount of data, quite possibly making him miss the relevant entries in the timeline. As this tool progresses the need for some sort of data reduction becomes even more crucial. One of the ideas to reduce this dataset is to use already existing tools, such as Splunk, to make it easier to sift through the data as well as to corroborate timeline data with other log files. Work is undergone to get Splunk to support importing data from log2timeline as well as to use OSSEC to go through the initial timeline to find signs of obvious entries that might suggest malware or other signs of "known bad". Yet tools like Splunk do not really address the problem, since it is not specifically developed for timeline data. The need to create a perhaps separate tool that assists with the data reduction might be in the future of the super timeline. Log2timeline already supports writing the timeline into a database, an option that makes it quite easy to develop tool that reads from this database schema and provide the investigator with an easy method to filter out what he believes is irrelevant to the investigation. The tool also

has to have the ability to mark or tag those events that the investigator considers to be relevant and to group events that belong to the same action into a new collective or group event (so called super event). The investigator also needs to be able to include other events of interest to the case, such as timestamps found during interviews, or in the physical world (suspect apprehended, machine confiscated, etc.).

Hiding events that are not of interest to the particular investigation, highlighting those that are, grouping events that belong to the same action, and creating a mean to effectively search through the timeline would make such a tool a very useful application. Such a tool could also provide means to extract reports based on the timeline as well as to visually represent the timeline, for instance within the reports, or on tagged events. That would make the visualization more focused and useful for both the investigator as well as those non-technical individuals that might read the report.

6. Conclusion

Timeline analysis has been used and will be used in various investigations, whether they involve physical crimes or those that are committed in the digital world. To this day in the digital world investigators have mostly focused on the analysis of filesystems timestamps along with the manual inclusions of other timestamps found during investigations. The move to super timelines that include information found inside files on suspect drives is something that has been missing in the forensic community. Despite being discussed for several years only few attempts have been made to create tools to assist investigators into creating such timelines. The extension of the timeline to a super timeline provides the investigator with considerably better overview of the suspect drive as well as the potentials to reduce the investigation time.

Limited resources along with the usually long time each investigation takes makes any method that can reduce the time it takes to complete an investigation extremely valuable to every investigator. Although there have been few researches that show direct connection with extended timelines and reduced investigation time it has all the potentials to reduce the investigation time considerably. Automating the gathering of all timestamp information from a suspect drive into a super timeline and making it accessible to the

investigator could potentially shorten the backlog that plagues many forensic analysts. However it should be noted that timeline analysis requires the investigator to know and understand the context of why a particular event might occur in a given timeline, something that comes with knowledge and practice, and without that knowledge timeline analysis might sometimes not be as efficient as it potentially can be. If the investigator knows how to reduce the dataset of a given timeline, whether that is by only limiting the information included in the timeline or by using other analysis techniques to quickly minimize the timeline, timeline analysis becomes a very efficient and quick mechanism to speed up investigations and reduce backlogs.

The proposed framework, log2timeline, already supports automatic extraction of several key artifacts found on Windows systems, making it easy for the investigator to create the timeline. Despite its early stages it has already reduced analysis time for few investigators that have provided feedback to the author. With the continued work on the framework it will only get better and include more information that might be beneficial to investigations.

The need for a tool that can assist with the analysis after the creation of the super timeline is becoming a vital part for this project to properly succeed. If a tool that fulfills all the requirements as listed in the future work chapter there is the chance that investigators can reduce their investigation time considerably by focusing on further examinations of the relevant data on hard drives, instead of sifting through what sometimes seems to be endless amount of information.

7. References

Carrier, Brian (2009 04 27). Body file. Retrieved January 7, 2010, from the Sleuthkit web site: http://wiki.sleuthkit.org/index.php?title=Body_file

Vincent, Liu (2005). Timestomp. Tool's web site: http://www.metasploit.com/research/projects/antiforensics/

nsINavHistoryService - MDC. (2009, April 16). Retrieved January 9, 2010, from https://developer.mozilla.org/en/NsINavHistoryService

Schuster, Andreas (2009 12 22). Evtx Parser Version 1.0.1. Retrieved January 9, 2010, from http://computer.forensikblog.de/en/2009/12/evtx_parser_1_0_1.html#more

Schuster, Andreas (2007 07 30). Vista Event Log The Inner Structure. Retrieved January 9, 2010 from http://computer.forensikblog.de/en/2007/07/the_inner_structure.html#more

page_transition_types.h (2009). Chrome Source Code. Retrieved January 9 2010, from http://src.chromium.org/viewvc/chrome/trunk/src/chrome/common/page_transition_types.h

Carvey, Harlan. (2009). Windows forensic analysis DVD toolkit 2E. Burlington, MA: Syngress.

Bookmarks.html (2008 November 25). Retrieved January 9, 2010, from http://kb.mozillazine.org/Bookmarks.html

Stevens, D. (2009, January 1). Windows 7 userassist registry keys. Into The Boxes, (0x00), Retrieved from http://intotheboxes.wordpress.com

EventType Complex Type. (2009, November 12). Retrieved January 9, 2010, from http://msdn.microsoft.com/en-us/library/aa384584%28VS.85%29.aspx

SystemPropertiesType Complex. (2009, November 12). Retrieved January 9, 2010, from http://msdn.microsoft.com/en-us/library/aa385206%28VS.85%29.aspx

EventDataType Complex Type. (2009, November 12). Retrieved January 9, 2010, from http://msdn.microsoft.com/en-us/library/aa384373%28VS.85%29.aspx

Carvey, Harlan (2009 02 29). Timeline analysis, pt III. Retrieved August 17, 2009, from Windows Incident Response Web site: http://windowsir.blogspot.com/2009/02/timeline-analysis-pt-iii.html

Menon-Sen, Abhijit (2002 October 01). How Hashes Really Work. Retrieved January 15, 2010, from: http://www.perl.com/lpt/a/679

How-To-Geek (2008 February 5). Speed Up Disk Access by Disabling Last Access Updating in Windows XP. Retrieved January 15, 2010, from http://www.howtogeek.com/howto/windows/speed-up-disk-access-by-disabling-last-access-updating-in-windows-xp/

mindstate (2009 December 04). How to Optimize NTFS Performance. Retrieved January 15, 2010 , from http://www.ehow.com/how_5713047_optimize-ntfs-performance.html

Olsson, Boldt, J., M. (2009). Computer forensic timeline visualization tool. digital investigation, 6, 78-87.

W3C Extended Log File Format (IIS 6.0) (2010). Retrieved January 19, 2010, from http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/676400bc-8969-4aa7-851a-9319490a9bbb.mspx

NtfsDisableLastAccessUpdate (2010). Retrieved January 15,2010, from http://technet.microsoft.com/en-us/library/cc959914.aspx

Carvey, Harlan (2009). timeline_tools.zip. Retrieved January 9, 2010 from http://tech.groups.yahoo.com/group/win4n6/files/

Cloppert, M. (Ed.). (2008 May 16). Ex-tip: an extensible timeline analysis framework in perl. SANS Institute.

Weber, Don (2010 January 5). System Combo Timeline. Retrieved January 18, 2010, from http://www.cutawaysecurity.com/blog/system-combo-timeline

Strunk, William Jr. & White, E. B. . The Elements of Style. New York, NY: Longman

Warlick, David (2004). Son of citation machine. Retrieved February 17, 2009, from Son of citation machine Web site: http://www.citationmachine.net

Carvey, Harlan (2009 02 29). Timeline analysis, pt III. Retrieved August 17, 2009, from Windows Incident Response Web site: http://windowsir.blogspot.com/2009/02/timeline-analysis-pt-iii.html

Buchhokz,Florian (2005 August 16). CERIAS – Zeitline: a forensic timeline editor. Retrieved January 18, 2010 from http://projects.cerias.purdue.edu/forensics/timeline.php

Jones, Keith J. (2003 19 March). Forensic analysis of internet explorer activity files [Rev.]. Retrieved from http://www.foundstone.com/us/pdf/wp_index_dat.pdf

Jones, Keith J. (2003 6 May). Forensic analysis of Microsoft Windows Recycle Bin Records [Rev.]. Retrieved from http://surfnet.dl.sourceforge.net/sourceforge/odessa/Recycler_Bin_Record_Reconstruction.pdf

Hager, Jesse (1998 December 11). The windows shortcut file format. Retrieved from: http://www.i2s-lab.com/Papers/The_Windows_Shortcut_File_Format.pdf

Adobe Systems Inc (2006). Action Message Format – AMF0. Retrieved from: http://opensource.adobe.com/wiki/download/attachments/1114283/amf0_spec_121207.pdf

McCusker, David (2008 March 9). Mork Structure. Retrieved from: https://developer.mozilla.org/en/Mork_Structure

Microsoft Corporation (2003 October 3). Troubleshooting Device Installation with the SetupAPI Log File. Retrieved from http://download.microsoft.com/download/e/b/a/eba1050f-a31d-436b-9281-92cdfeae4b45/SetupAPILog.doc

Silberman, P (2010 January 21). DOD Cyber Crime: New Audit Viewer/Memoryze. Retrieved February 9, 2010 from http://blog.mandiant.com/archives/741

Internet connection firewall security log file overview (2010). Retrieved February 9, 2010 from http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/hnw_firewall_log_understanding.mspx?mfr=true

Adobe System (2010 June). Adobe Supplement to the ISO 32000. Edition 1.0. Retrieved April 19th, 2010 from http://www.adobe.com/devnet/pdf/pdf_reference.html

Introducing the Office (2007) Open XML Formats (2010). Retrieved from http://msdn.microsoft.com/en-us/library/aa338205.aspx