Global Information Assurance Certification Paper

Forensic Timeline Analysis using Wireshark
GIAC (GCFA) Gold Certification

Author: David Fletcher, david.fletcher.6@us.af.mil

Advisor: Richard Carbone

Accepted: July 10, 2015

Abstract

The objective of this paper is to demonstrate analysis of timeline evidence using the Wireshark protocol analyzer. To accomplish this, sample timelines will be generated using tools from The Sleuth Kit (TSK) as well as Log2Timeline. The sample timelines will then be converted into Packet Capture (PCAP) format. Once in this format, Wireshark's native analysis capabilities will be demonstrated in the context of forensic timeline analysis. The underlying hypothesis is that Wireshark can provide a suitable interface for enhancing analyst's ability. This is accomplished through use of built-in features such as analysis profiles, filtering, colorization, marking, and annotation.

1. Introduction

Correlation of events can be a challenging task during crime investigations. Timeline analysis is used to ease this burden by ordering each piece of evidence by its time of occurrence. Doing so aids the investigator by enforcing organization, providing event context, revealing inconsistencies, and creating a frame of reference for the overall effort. (Luttgens, 2014) This ultimately enhances the ability to formulate hypotheses and ultimately solve the crime. This same method can be applied to computer forensic examinations to attempt to piece together the actions that take place during a computer incident. To illustrate the process, commonly accepted tools for creating timeline evidence will be explored. After discussion of timeline generation is complete, an analysis capability will be presented using the Wireshark network analysis tool. This capability will be supported using various scripts to convert and present timeline data within the Wireshark tool.

A computer filesystem is a complex environment that contains a great deal of evidence. Among this evidence is timestamp information contained in the metadata layer of the filesystem. Typical filesystems maintain three or more separate timestamps for each file stored. These timestamps are updated based on activity performed against the structures that describe the file or its data. Generally, the following timestamp activities may generate evidence based on the filesystem in question:

Modified – This timestamp is updated when the file contents are updated.

Accessed – This timestamp is updated when the file contents are accessed.

Changed – This timestamp is updated when the metadata of a file is updated.

Birth – This timestamp is updated when the file is created (NTFS only).

Delete – This timestamp is updated when the file is deleted (Ext2/3/4 only).

These timestamps are represented within a field in the timeline using the acronym MACB (Lee, 2011). Alternatively, Incident Response & Computer Forensics, Third Edition refers to the timestamp data as MACE – Modified, Accessed, Created, and Entry Modified rather than MACB – Modified, Accessed, Metadata Change, and Birth.

David R. Fletcher Jr., david.fletcher.6@us.af.mil

Reference material should be consulted to determine the activities that are valid for a particular filesystem prior to analysis. For instance, Linux ext2/3/4 filesystems do not record a birth timestamp (Lee, 2012) but do support a delete timestamp (Carbone, 2011) while FAT filesystems do not record a change timestamp (Lee, 2012). To complicate the matter further, configuration data should be consulted to ensure that expected timestamp information is supported in the operational environment. For example, recent versions of windows support use of access timestamps but they are not enabled by default (Lee, 2012).

Within a typical Windows filesystem an even greater amount of time-related evidence exists in the form of log files, file metadata, system databases (such as the Windows registry), and filesystem overlays (such as Volume Shadow Copy). These files serve to provide additional correlative and contextual clues to the analyst. Using this information, the investigator may be able to more accurately reconstruct the chain of events leading up to an incident. In addition, this added evidence may serve to resolve attempts made to employ anti-forensic techniques such as time-stomping.

Timeline evidence collection is typically preceded by identification of an event of interest. Using this event as a reference point, individual timestamps are read from the filesystem and collected into a single file to represent the timeline of activity surrounding that event. The analyst then interprets combinations of timestamp updates that represent operating system actions such as file creation, modification, copy, move, access, and delete within the target operating system. The operating system actions are then correlated to determine facts about the event in question.

The standard timeline is typically created using tools such as *fls* and *mactime* from The Sleuth Kit (TSK) which consists of filesystem metadata layer timestamp information only (Lee, 2012)(Carrier, 2005). This restricts the analyst to seeing only those events that cause changes in file data, or metadata. Inability to correlate filesystem activity with additional temporal information, such as log files, in an automated fashion increases the work factor for analysis. This is because any correlation would need to be performed manually by mounting the filesystem and inspecting various sources by hand.

The super-timeline is created using the *log2timeline* tool. *Log2timeline* is capable of integrating time-based activity from multiple sources that includes standard timeline data in the form of a Bodyfile. This tool does not parse filesystem level evidence so the most effective method of use is to integrate output from both *log2timeline* and *fls*/*mactime*. Prior to investigating an event, the analyst must identify the sources of time data that are of interest. This is accomplished using a list file or command line options passed to the *log2timeline* command. While *log2timeline* has shortcomings in evidence parsing this paper is focused on providing an alternate method of analyzing timeline output (Carbone, 2011). Since it is the most widely used super-timeline generation tool, it will be the sole focus of this paper.

The resulting timeline output may be presented in a number of formats but text formats such as Mactime and CSV are common. These formats are popular because command line tools such as *grep*, *sed*, and *awk* can be used to parse and search for evidence (Carbone, 2011). However, the resulting timeline can represent a significant amount of information that is difficult to interpret without visual cues. Rob Lee, the forensic curriculum lead for the SANS Institute, has created an Excel spreadsheet that provides visual information to the analyst (Lee, 2011). Microsoft Excel is not the ideal environment for sifting through large volumes of text. This is where Wireshark may be able to advance timeline analysis.

Wireshark is a protocol-parsing tool that is typically used to interpret the various protocols of the Internet Protocol (IP) suite. Among other features, this tool provides the ability to colorize, mark, comment, sort, and filter packet information for easy interpretation by a network analyst (Chappell, 2010).

So how does timeline data relate to IP packets? The IP protocol has several fields that may be used to provide an analog to timeline data. In addition, among the many protocols that Wireshark understands is the Transmission Control Protocol (TCP). Once again, several of the TCP protocol fields may be used to represent timeline information. One such field is the TCP flags field, which maintains connection state during a conversation. These flags resemble the MACB field in the timeline output. By carefully selecting timeline fields for conversion to equivalent IP and TCP fields, one can take

advantage of the advanced capabilities that the Wireshark tool provides. Finally, timeline data can be encapsulated in the payload portion of the TCP packet. This allows interpretation of the payload data using a custom Wireshark protocol dissector.

2. Background

2.1 File System Forensic Artifacts and Manual Analysis

The filesystem of a computer represents a treasure trove of information. As previously described, timestamp information is critical to recreating the activity surrounding an event or incident. The filesystem also represents a much greater amount of information than the typical user may realize.

When a file is deleted, several artifacts remain until the areas where both the filename information and data of the file are reused. Even then, small portions of a file may remain inside of other files in what is known as slack space. Slack space is one or more sectors within a new file that contain old data due to both the cluster size and manner in which the filesystem attributes sectors to a new file. This evidence may be difficult to access and process in a manual fashion that nevertheless remains on the disk (Carrier, 2005).

During manual analysis, the analyst must first create a Bodyfile using the *fls* tool from (TSK). This tool outputs timeline information but the data it generates is unordered and in a less than human readable format. After the Bodyfile is created, the *mactime* tool is used to produce the human readable timeline (Lee, 2011). The basic timeline contains information regarding allocated files, deleted files, and unallocated file data. Once the timeline is produced, the analyst must identify events of interest and manually correlate this information with other artifacts of investigation.

Identifying events of interest within the timeline can be a challenging task since elapsed time represents decay in the fidelity of timeline evidence. The timeline will only include the last timestamp change evidence (modify, access, change, and birth) in the filesystem. Only the last change is available since any prior changes are destroyed during the most recent update (Lee, 2011).

Correlation of events becomes an even more daunting task. Operating systems and applications collect a mountain of evidence in the form of log files, file metadata, and application/system databases that may be used to further explain an event of interest. This evidence comes in a myriad of formats such as text, binary, and XML that presents another challenge for collection and correlation. In addition, special files such as the Windows registry and Volume Shadow Copies present even more information that is somewhat difficult to manually access and parse in a timely fashion (Luttgens, Mandia, & Pepe, 2014).

2.2 Log2Timeline

The *log2timeline* tool is a framework for creating timeline evidence from various sources found within the filesystem, operating system, and application (or application service) layers. The creators of *log2timeline* have automated the process of evidence gathering and correlation for a great number of different artifacts and formats. In addition, the *log2timeline* tool presents this information in a single consolidated timeline file (Metz, 2015).

The *log2timeline* output is known as the super-timeline and can be produced in one of several different formats such as CSV, XML, HTML, SQLite, and Bodyfile. For the purposes of this paper, the CSV format will be used due to its ease of parsing. Other format parsers may be added in the future based on the utility and applicability of the Wireshark tool. The super-timeline maintains the same general format as the standard timeline but adds new fields such as source, source type, and type to differentiate information gathered from the different evidence sources (Metz, 2015).

Consolidation of this information makes it much easier to correlate events and find evidence that would otherwise be missing at the filesystem level. In order to gather the right information for a particular case, the analyst must include appropriate log types in a list file or specify them as command line parameters. This process is outlined in detail on the SANS DFIR blog (*"Digital Forensic SIFTing: SUPER Timeline Creation using log2timeline"*, 2011).

2.3 Wireshark

Wireshark is a network protocol analyzer that has many powerful features for supporting and interpreting protocol and network behavior. It is capable of parsing network traffic at layer two of the OSI model and above. This includes the ability to understand individual packets, network conversations, protocol sessions, and application interaction. These capabilities are provided based on integrated features of the software as well as protocol specific extensions known as protocol dissectors. This allows the analyst to focus on traffic behavior rather than the semantics of a given protocol. In order to ease the analysis process, several other features beyond basic protocol dissection are also provided (Chappell, 2010).

This section provides an overview of the features that may be useful in the context of timeline analysis. It is not meant to provide complete coverage of the features or analysis techniques supported by Wireshark. For a full explanation of these capabilities, please see "Wireshark Network Analysis" or an equivalent Wireshark reference.

The first feature of interest is Wireshark's powerful filtering language. This language allows an analyst to zero in on the specific activity of interest. The filtering language is protocol aware so the analyst can identify a specific protocol, protocol field, and target value. Using these examples, an analyst might look for packets from an individual host, packets using a specific protocol, or packets containing specific values or strings. In the context of timeline analysis, this may correlate to specific MACB field combinations, filenames, or strings. An example display filter can be seen in Figure 1.

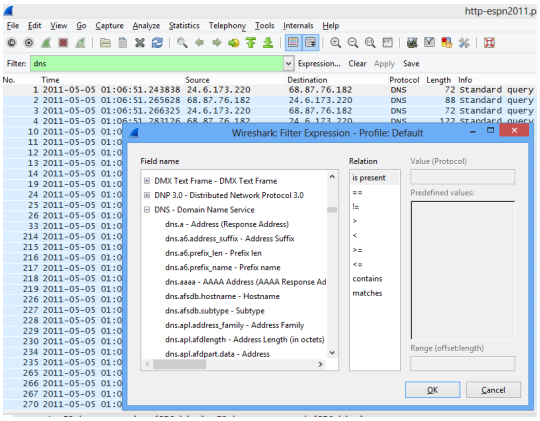


Figure 1: Wireshark filter bar and expression builder.

In Figure 1, the Wireshark display is being filtered for Domain Name Service (DNS) packets. In addition, the expression builder is visible (front window). This gives the user a flexible interface for developing and testing filter expressions. As a visual cue, the filter bar turns green when a syntactically correct filter expression has been entered. If an appropriate protocol dissector is available then the user can filter on the fields exposed within that protocol. This can be seen within the expression builder window. In this case, the DNS fields are exposed showing the user valid protocol field values available for building a filter expression.

To further aid the analyst in interpreting activity, Wireshark supports profiles and colorization rules. The profile is simply a group of settings that provide a consistent environment between analysis sessions. Colorization rules are one of the most powerful analysis features available in Wireshark. Since colorization rules are most effectively used when stored in a saved profile, the two features will be discussed together. Colorization allows the analyst to provide visual cues for identifying packet characteristics. Wireshark comes with a set of default colorization rules that identify anomalous conditions such as invalid checksums and retransmissions.

For the purpose of timeline analysis it would be useful to key off specific MACB field combinations to color categorize file activity such as creation, access, modification, move, copy, etc. Ideally, a profile would be created for each individual filesystem to be analyzed since behavior varies based on the filesystem in question. In addition, different colorization rules can be used for different analysis contexts and switching profiles can occur dynamically. The profile selector and default colorization rules can be seen in Figure 2.

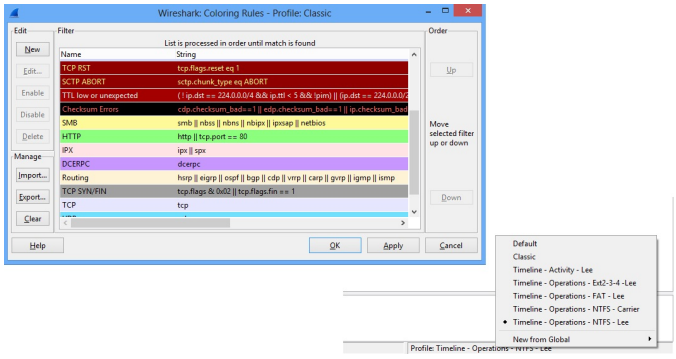


Figure 2: Colorization rules and profile selector.

Another feature that is important for analysis is the ability to add and remove columns from the packet list window. Within Wireshark, in order to troubleshoot network problems, it is often necessary to add protocol fields as columns to the display.

Available columns include standard network and transport layer protocol fields such as IP and TCP that are built into the tool. In addition, fields exposed by a dissector may also be used. Finally, calculated fields are available which do not exist in the packet capture or the underlying protocols being analyzed. Instead, the Wireshark engine calculates these values as the packet trace is being loaded into the application.

One of the available calculated fields is the Delta Time field, which identifies the time gap between receipts of packets. This can help the analyst determine where bottlenecks or performance affecting processes may reside in a client-server application. This type of analysis is equally important in timeline interpretation. A scripted or malware attack may become obvious based on the time elapsed between filesystem actions. Using Delta Time a forensic analyst may filter on a threshold value to reveal activity that occurs in rapid succession. Figure 3 shows the addition of the Delta Time column and its application to a packet capture. Sorting on this column can also identify the longest and shortest delays between packet reception.

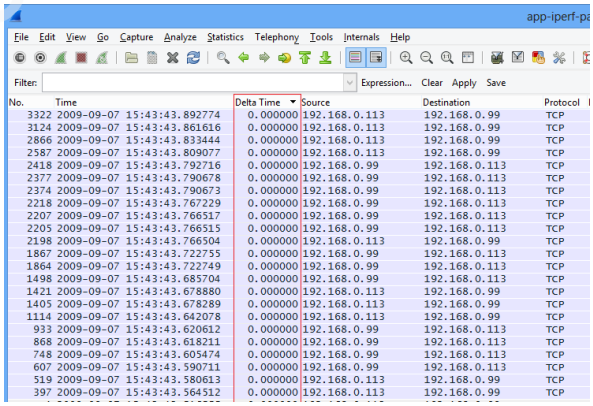


Figure 3: Delta Time column applied to packet capture.

Packet marking allows an analyst to identify packets of interest that may or may not be important in the context of an investigation. Typically, during the course of troubleshooting, the analyst will mark multiple packets that may represent activity of

significance for further review. Wireshark allows an analyst to arbitrarily mark packets in a capture file for future reference. This feature is equally relevant to forensic investigation in that the forensic analyst may mark individual file activities as potentially malicious or important. After processing the entire timeline, the analyst can then filter to show only those entries that were marked. This could be a valuable tool in the reporting phase to identify those timeline entries that are important in the context of an investigation without altering or extracting original timeline information. Figure 4 illustrates the ease with which packets may be marked.

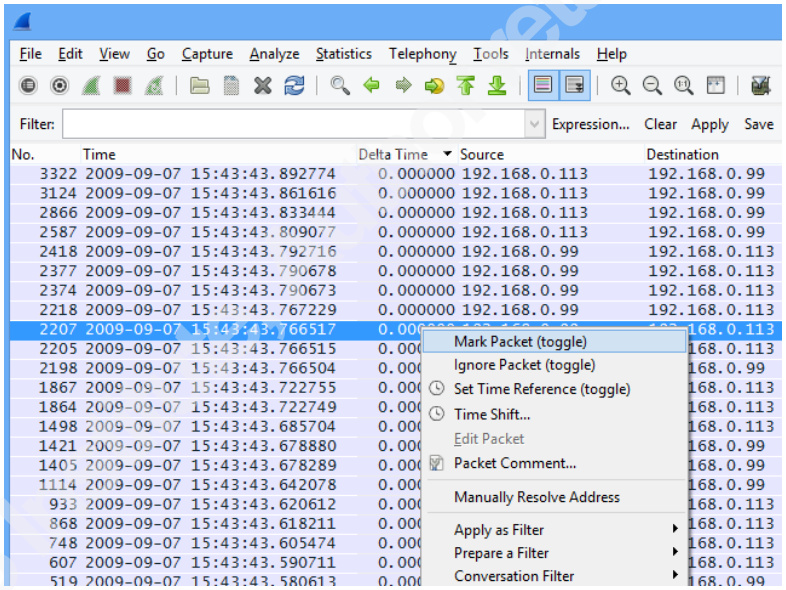


Figure 4: Packet marking in the packet list pane.

Another useful feature is the ability to add packet comments. This feature allows an analyst to attach notes to individual packets in a capture file. Using this feature along with packet marking allows the analyst to identify reasons that the packet was marked or hypotheses to be validated at the outset of initial investigation. Note taking is critical when interpreting activity from such a large volume of information. Once analysis is complete, display filtering can be used to restrict the displayed packets to only those marked or containing comments using the filters *"frame.marked == true"* or *"frame.comment != """* , respectively. Figure 5 shows the context menu for adding packet comments while Figure 6 shows the packet comment entry field.

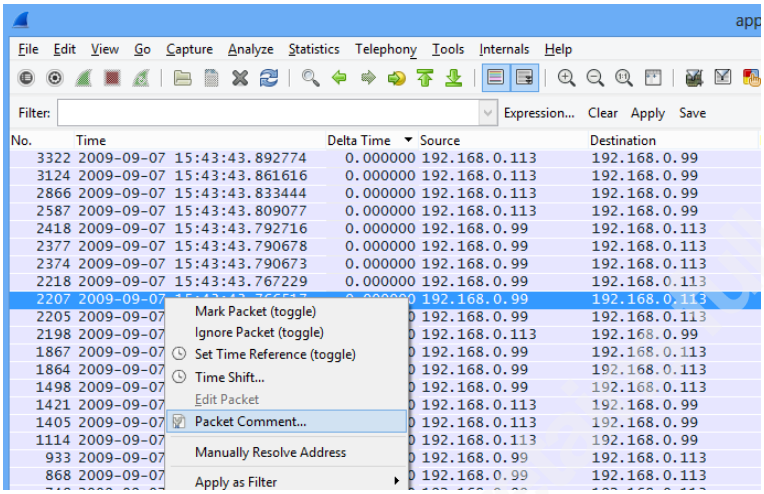


Figure 5: Packet Comment context menu selection.

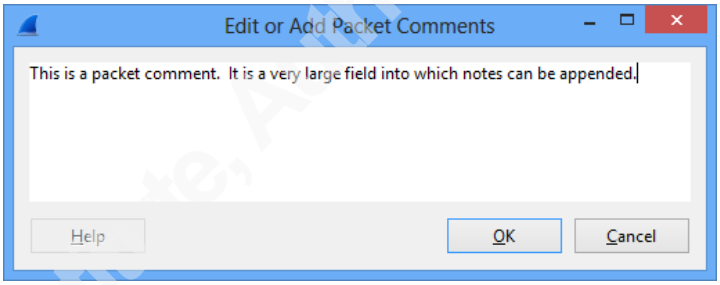


Figure 6: Packet comment add/edit dialogue box.

The final set of features that Wireshark provides is its statistics menu. The statistics menu hosts a mountain of features that are valuable in the context of network analysis. Only a small subset of those features is relevant in the context of timeline analysis. First, the comments summary feature aggregates all packet level comments into a single text field along with the frame number that the comment references. This can be seen in Figure 7. Review of the comment summary allows the analyst to review the activity at a macro level and clip information for reporting. In addition, the frame number corresponds to the line number in the original timeline in the event that the source document must be referenced.

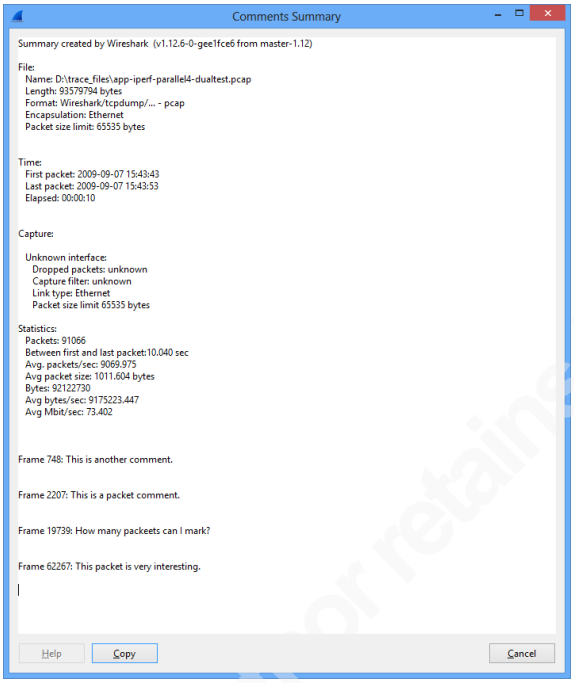


Figure 7: Comments summary window.

Next, the Conversations and Endpoints options allow an analyst to see aggregated communication sessions and filter on any single session to see the details. In the context of timeline analysis, this would allow the analyst to see where a file had been referenced multiple times over the course of the timeline. The Packets field will indicate the number of times that the host system (address A) saw activity on the target file (address B). This is shown in Figure 8.

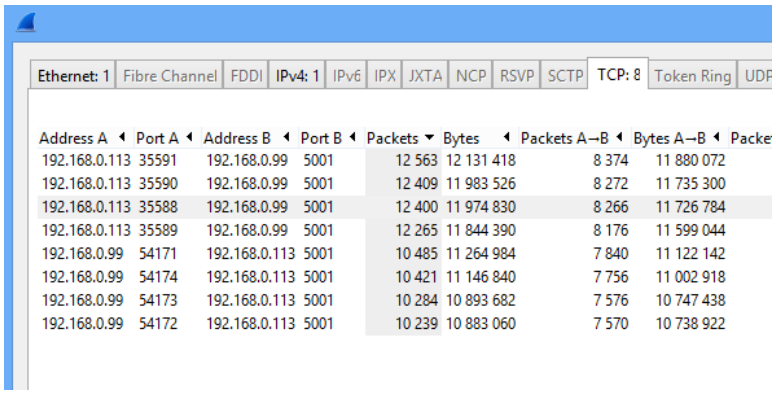


Figure 8: Conversations statistics window.

Finally, the most useful macro level feature is the IO Graph option. This option plots throughput over time. Since the timeline entries are of a somewhat standard size, an increase in throughput corresponds to increased filesystem activity. Zeroing in on sharp increases in filesystem activity may reveal the presence of malware or scripted exploitation. This concept is illustrated in Figure 9.

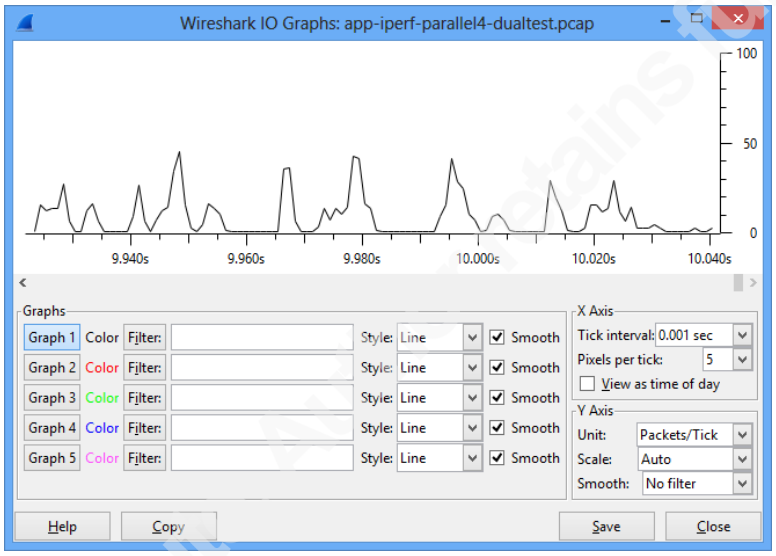


Figure 9: IO Graph showing throughput peaks and troughs.

2.4 Scapy

In order to take advantage of the advanced features that Wireshark provides, the timeline output must be converted to packet capture (Pcap) format. The Scapy Python framework is perfect for tackling this challenge. Scapy provides the ability to craft arbitrary packets using either the Scapy interactive interface or by importing the Scapy library into a custom Python script.

To make packet-crafting simple, the Scapy framework exposes the Internet Protocol and its encapsulated protocols by breaking them down into their constituent fields. In packet-crafting, one simply provides values for the fields of interest, stacks the desired protocols on top of one another to construct a valid packet/payload, and determines disposition. Disposition for a packet may include untracked transmission, transmission and reception of a reply, or output to a packet capture file for future

analysis. The latter will be the method employed for timeline conversion (SecDev.org, 2007).

To support timeline conversion, Scapy will be used in a custom Python script developed for parsing the timeline contents (Python Software Foundation, 2015). The script will manipulate the packet timestamp, IP, and TCP fields of interest for easier timeline analysis and embed a representation of the timeline entry for parsing by a custom Wireshark dissector. The packets will then be written to a packet capture file that will represent the packetized forensic timeline.

2.5 Supporting Tools

In order to make the timeline easier to read within the Wireshark interface two additional tools will be necessary.

The first tool is a custom host file. A host file is used to provide name resolution in the absence of Domain Name System (DNS) servers. For this effort, the host file will be used to turn IP addresses representing the host under analysis and target file name into human readable format. The host file is a simple tab delimited file with an IP address column and a text column (Chappell, 2010). The Python script described in Section 2.4 will generate this host file. An example can be seen in Figure 10.

```
172.16.0.1  abc123
172.16.0.3  FPEXT.MSG
172.16.0.4  MSOWS409.DLL
172.16.0.5  FP4AWEC.DLL
172.16.0.6  MSDAPML.DLL
172.16.0.7  MSONSEXT.DLL
172.16.0.8  Ir_begin.wav
172.16.0.9  ir_end.wav
172.16.0.10 ir_inter.wav
172.16.0.11 sRGB Color Space Profile.icm
172.16.0.12 sharedaccess.ini
172.16.0.13 digiras.chm
172.16.0.14 cyycoins.chm
172.16.0.15 cyzcoins.chm
172.16.0.16 kodak_dc.icm
172.16.0.17 cpqdap01.sys
172.16.0.18 nikedrv.sys
172.16.0.19 rio8drv.sys
172.16.0.20 riodrv.sys
172.16.0.21 rasirda.sys
172.16.0.22 cdaudio.sys
172.16.0.23 fsvga.sys
172.16.0.24 battc.sys
172.16.0.25 compbatt.sys
172.16.0.26 audstub.sys
172.16.0.27 tosdvd.sys
```

Figure 10: Example host file entries.

The second tool is a custom Wireshark protocol dissector. Dissectors extend the functionality of Wireshark and enable it to display details of protocols it does not natively

know how to parse. Dissectors can be written in the C programming language or the Lua scripting language. For the purposes of this paper, the latter method will be used to demonstrate timeline analysis capabilities. Once a dissector is imported into Wireshark, the dissector is registered with an expected protocol and port. This allows Wireshark to "recognize" a protocol and attempt to parse its fields using traffic characteristics (Kaplan, 2015).

3. Methodology

In order to use Wireshark as a timeline analysis tool there must be a correlation between timeline entries and network traffic. In this case, that correlation includes use of fields in both the Internet Protocol (IP) and Transmission Control Protocol (TCP).

At the IP level, the source address can be used to identify the computer/user under investigation while the destination address can be used to identify the target of the operation (typically a file) (Stevens & Fall, 2011). The remaining fields will be identified using TCP as the embedded protocol in the packet.

At the TCP layer of the packet, the source port can be used to identify either the timeline or the super-timeline packet payload. By picking a specific port for each timeline format, a protocol dissector can be built and related to the chosen port. The dissector can then be used to parse the TCP payload providing even greater flexibility for analysis. Next, the TCP flags field will be used to represent the MACB flags found in the timeline entry. Finally, the TCP payload field will contain the full contents of the timeline entry as a series of zero terminated strings. This will allow the dissector to distinguish the timeline fields within the packet payload (Trinh, 2012).

The following process assumes that the analyst has already performed scoping, identified timeline artifacts of interest, and generated a valid timeline output file. The output file must be in Mactime format for standard and CSV format for super timeline analysis. The process described will convert the generated timeline into a packet capture file (Pcap) and companion host file for use in evidence analysis using Wireshark. Once the packet capture has been generated, a Wireshark dissector will be created to handle the

notional timeline and stimeline protocols[1]. The notional protocols correlate directly to the standard and super timeline formats respectively. This will enhance analysis capabilities by exposing the full complement of Wireshark features to the process. As a final step, multiple example Wireshark profiles will be generated to illustrate the ability to switch analysis context on the fly. These will include colorization profiles that highlight differences in filesystem characteristics and analysis methods.

3.1 Timeline Pcap Generating Scripts

The general process of timeline conversion can be seen in the flowchart found in Figure 11. The conversion will read each line of the timeline file, parse that line into its constituent fields, assign an IP address to the subject of the entry, create a host file entry, generate an individual packet using Scapy, assign the appropriate timestamp to the packet, add the packet to a packet list, and continue until the end of file marker is reached. Once the end of file marker is reached, the resulting Pcap file will be written out to disk. The full scripts that perform this activity can be found in Appendix A and B for standard timeline (Mactime) and super-timeline parsing (CSV), respectively.

[1] The timeline and stimline protocols are a side effect of using Wireshark for analysis. The protocols describe the internal message format of the timeline payload of the packet. In addition, when associated with a TCP port, Wireshark will automatically apply the dissector script to packets matching that characteristic.

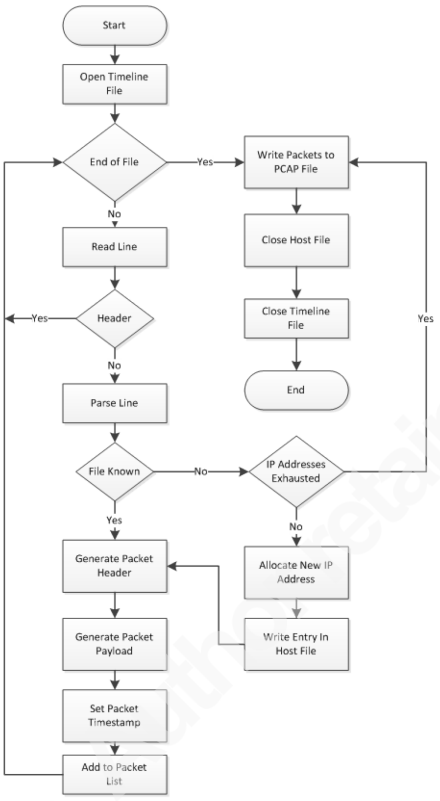


Figure 11: Timeline parsing process.

Prerequisites for using the scripts include installation of the Scapy and Pytz libraries. These libraries are imported in the parsing scripts to craft timeline packets and construct valid timestamps from the parsed timeline data, respectively (Bishop, 2015). In addition, the Argparse library is used to present and parse command line arguments to the script.

Both scripts take the same command line arguments for processing a timeline file. These arguments include the path to the input timeline file (--file), the timezone to be applied (--tzone), the output location for the resulting Pcap file (--outfile), the hostname of the system under investigation (--hostname), and output location for the resulting host file (--hostfile). The *tzone*, *outfile*, *hostname*, and *hostfile* arguments are all optional and each is given a default value if not specified. A valid command line invocation of the super-timeline processing script can be seen in Figure 12.

```
$ Python parse_stimeline.py --file ~/timeline.CSV --tzone
UTC --outfile ~/timeline.Pcap
--hostname case_1234_server --hostfile ~/hosts
```

Figure 12: Super-timeline parser command line invocation.

The parser operates in a straightforward fashion by reading the entries from the specified timeline file in the appropriate format (CSV for super timeline). Each line is then expanded into its individual timeline fields by splitting the line using the comma as a delimiter.

Within the main loop, additional processing is applied to several of the timeline fields. First, the path is stripped off the filename for inclusion in the host file. Next, the time/date elements are split into values that are acceptable for the Python datetime constructor. The MACB field is then used to construct a string representing the TCP flags that should be set in the resulting packet. Finally, the description field length is checked to determine whether the packet will exceed 65,536 bytes (super-timeline only).

If the description field does cause the packet to exceed this length, then the field is truncated. This is required due to the length limit for an IP packet (Stevens & Fall, 2011). The vast majority of timeline entries are a fraction of this length but Prefetch files contain a vast amount of metadata in this particular field. This process represents a compromise in capability. It is possible to store more than 65,536 bytes in a TCP stream but this would require the analyst to re-assemble these packets and limit the utility of Wireshark as a timeline analysis tool.

After the packet length is confirmed not to exceed allowed limits, the timeline entry is re-assembled as a series of zero terminated strings. This sets the stage for processing the payload using a custom Wireshark dissector in Lua. This process is described in Section 3.2.

Before a packet can be generated, it must be assigned a source IP address, destination IP address, source port, and destination port. The source IP address resolves to the hostname specified at the command line. This value is always the IP address 172.16.0.1. The destination port can be chosen at random (it is set to 9999 in the

prototype script). The destination IP address represents the file identified in the timeline entry.

Using the full filename as the key, a Python dictionary object is used to keep track of files that have been processed. If the current file has been previously processed, the IP address stored in the dictionary is reused. If the filename does not exist in the dictionary, then a new IP address is allocated and a host file entry is written. Using IP version 4 it is possible to create a timeline file with approximately 4.3 billion unique files (Stevens & Fall, 2011). The prototype script is currently restricted to processing 65,535 unique files. However, this restriction can be easily lifted by altering the IP address selection logic.

With the addressing details of the IP packet complete, the power of the Scapy library can be put to use. In order to assemble the packet, variables at the IP, TCP, and payload layers are assigned as seen in the code snippet in Figure 13. Each layer's variables are specified within parenthesis and each layer is separated by a forward slash. After assembly, the appropriate timestamp is assigned to the packet and the packet is added to a Scapy packetList object. The packetList is used to contain a collection of packets in memory (SecDev.org, 2007).

```
# Create an IP packet with TCP transport and data payload
p=IP(src=srcIP,dst=ip)/TCP(sport=timelinePort,dport=destPort
,flags=flags)/Raw(load=data)
# Set packet timestamp to constructed value
p.time = int(pktTime.strftime("%s"))

# Append packet to packet list
pkts.append(p)
```

Figure 13: Python packet assembly using Scapy.

Once the end of the timeline file has been reached, the packetList object is written out to disk at the location specified using the outfile argument. The remaining open resources are then closed and script execution terminates.

Prior to viewing the Pcap in Wireshark, it is necessary to move the host file created by the parsing script to the proper location. Within Wireshark, this location can be identified by navigating to *Help > About Wireshark* and selecting the *Folders* tab as seen in Figure 14. The personal configuration folder is where profiles are created. The root of this folder represents the default profile and subfolders will be named for the profiles that a given user has created (Chappell, 2010).

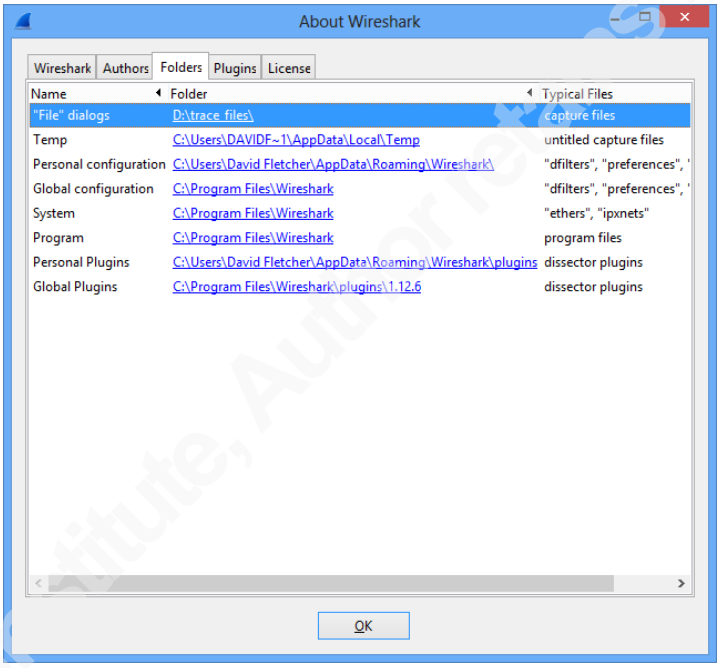


Figure 14: Wireshark folder location dialogue box.

Wireshark must be instructed to use the generated host file for name resolution. This is accomplished by navigating to *Edit > Preferences,* selecting *Name Resolution* and checking the *Resolve Network (IP) Addresses* and Only use the profile "hosts" file entries. This dialogue can be seen in Figure 15.

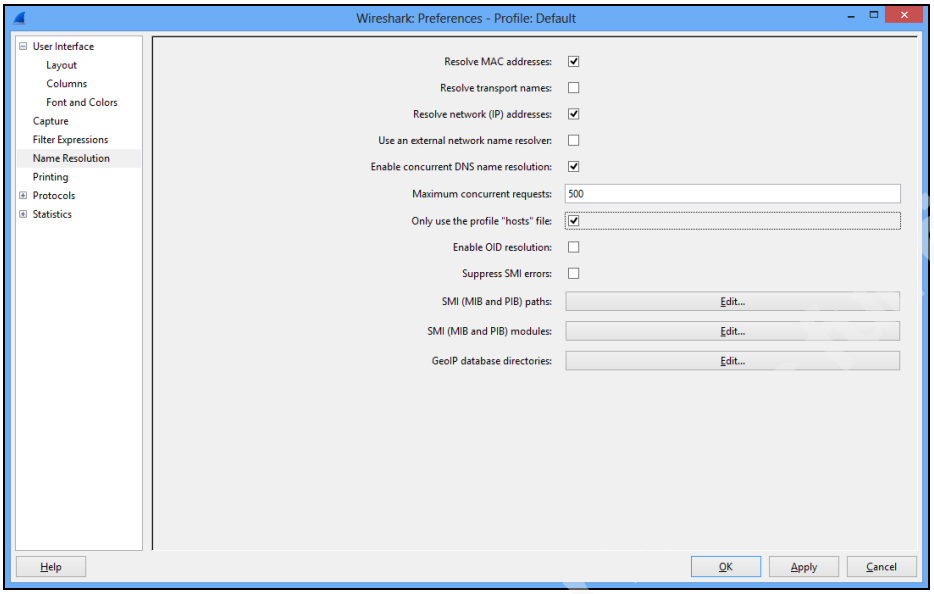


Figure 15: Wireshark name resolution options.

Even with name resolution enabled, the default Wireshark display is not initially conducive to timeline analysis. In order to fix this, columns will be removed, added and re-arranged. Right clicking on the column heading reveals the *Remove Column* option. In the display, remove all but the time, source, destination, and length columns. Next, select a single packet in the packet list pane (top), expand the *Transmission Control Protocol* entry in the packet details pane (middle), right click on the "Flags" field and select *Add as Column*.

The default display is still not analyst friendly. Change the time display format by selecting *View > Time Display Format > Date and Time of Day*. Finally, add the *Delta Time* field by navigating to *Edit > Preferences* selecting *Columns* and clicking the add button. Set the column title to *Delta time* and set the field type to *Delta Time*. This option setting can be seen in Figure 16.

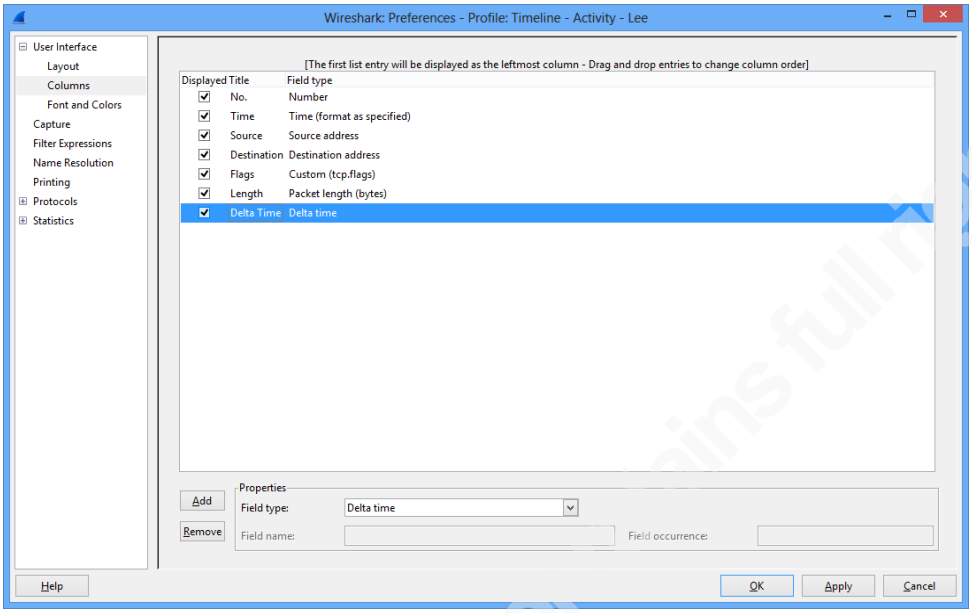
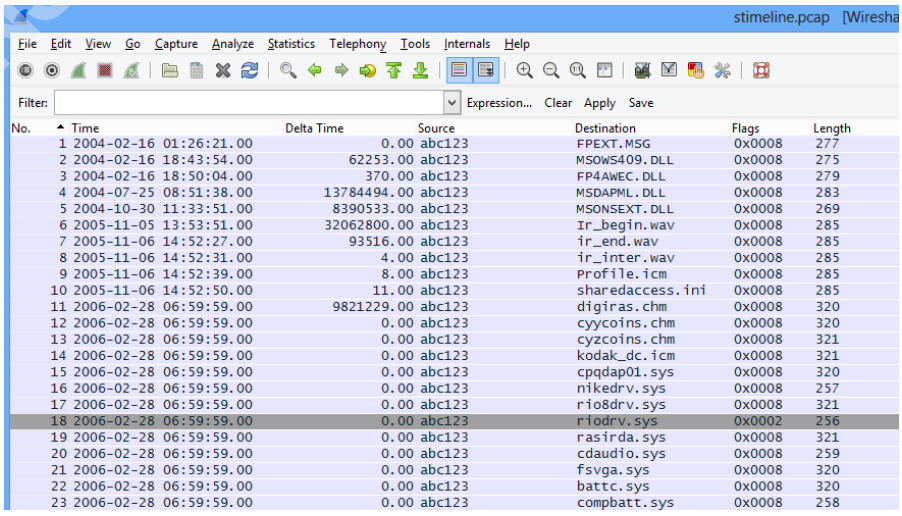


Figure 16: Delta Time column addition.

After all these actions are completed, the Wireshark display should look like Figure 17. The settings above will not need to be repeated as profile changes are saved upon exit. This creates a marginally useable interface for timeline analysis. An investigator can see when file activity takes place, the time between file actions, the host on which the file activity occurs, the target file, the MACB flags in a cryptic format, and the length of the packet (and relative length of the timeline entry).



No.	Time	Delta Time	Source	Destination	Flags	Length
1	2004-02-16 01:26:21.00	0.00	abc123	FPEXT.MSG	0x0008	277
2	2004-02-16 18:43:54.00	62253.00	abc123	MSOWS409.DLL	0x0008	275
3	2004-02-16 18:50:04.00	370.00	abc123	FP4AWEC.DLL	0x0008	279
4	2004-07-25 08:51:38.00	13784494.00	abc123	MSDAPML.DLL	0x0008	283
5	2004-10-30 11:33:51.00	8390533.00	abc123	MSONSEXT.DLL	0x0008	269
6	2005-11-05 13:53:51.00	32062800.00	abc123	Ir_begin.wav	0x0008	285
7	2005-11-06 14:52:27.00	93516.00	abc123	ir_end.wav	0x0008	285
8	2005-11-06 14:52:31.00	4.00	abc123	ir_inter.wav	0x0008	285
9	2005-11-06 14:52:39.00	8.00	abc123	Profile.icm	0x0008	285
10	2005-11-06 14:52:50.00	11.00	abc123	sharedaccess.ini	0x0008	285
11	2006-02-28 06:59:59.00	9821229.00	abc123	digiras.chm	0x0008	320
12	2006-02-28 06:59:59.00	0.00	abc123	cyycoins.chm	0x0008	320
13	2006-02-28 06:59:59.00	0.00	abc123	cyzcoins.chm	0x0008	321
14	2006-02-28 06:59:59.00	0.00	abc123	kodak_dc.icm	0x0008	321
15	2006-02-28 06:59:59.00	0.00	abc123	cpqdap01.sys	0x0008	320
16	2006-02-28 06:59:59.00	0.00	abc123	nikedrv.sys	0x0008	257
17	2006-02-28 06:59:59.00	0.00	abc123	rio8drv.sys	0x0008	321
18	2006-02-28 06:59:59.00	0.00	abc123	riodrv.sys	0x0002	256
19	2006-02-28 06:59:59.00	0.00	abc123	rasirda.sys	0x0008	321
20	2006-02-28 06:59:59.00	0.00	abc123	cdaudio.sys	0x0008	259
21	2006-02-28 06:59:59.00	0.00	abc123	fsvga.sys	0x0008	320
22	2006-02-28 06:59:59.00	0.00	abc123	battc.sys	0x0008	320
23	2006-02-28 06:59:59.00	0.00	abc123	compbatt.sys	0x0008	258

Figure 17: Initial Wireshark timeline analysis UI.

An unanticipated side effect of using a host file to display the file name is that the host file is delimited using whitespace characters. This means that filenames containing whitespace characters are truncated in the display. This further detracts from the ability to perform timeline analysis using Wireshark without the benefit of a dissector.

3.2 Timeline Lua Dissector

It would be beneficial to create a protocol dissector to parse the embedded timeline data placed in the packet payload by the timeline parsing script. This would improve the analyst's ability to visualize and analyze timeline information within the Wireshark user interface. In addition, it would expose timeline fields for use as displayed columns and direct reference in Wireshark's filtering language.

The protocol dissector will be written in the Lua scripting language. The script will register the protocol, identify fields found in the payload and their type, then add the fields to the dissector tree. The general structure of the script includes variable declarations and three functions. The initialization function sets up the packet counter, the dissector function parses the actual protocol fields, and the *getStringLength* local helper function calculates the offset to the next string in the payload. The last line of the script registers the port used by the protocol in the dissector table. This last element must match the port used in the parsing script for automatic dissection to occur within Wireshark (Bjorlykke, 2009). The source for the timeline and super-timeline Lua scripts can be seen in Appendix C and D, respectively.

The completed dissector must be placed in either the Personal Plugins or Global Plugins folder previously identified in Figure 14. This will cause the dissector to be applied by default when the chosen port is seen in the loaded Pcap file. Once the dissectors are in place, Wireshark must be restarted to see the resulting effect.

The first indication can be seen in the packet details pane. Previously, below the Transmission Control Protocol entry was a single node identified as Data (Figure 18). This node is now replaced with either "Timeline Protocol" or "Super Timeline Protocol" based on the selected port and script that generated the Pcap file. Expanding this node

reveals all of the fields that make up the timeline entry found in the packet payload (Figure 19).

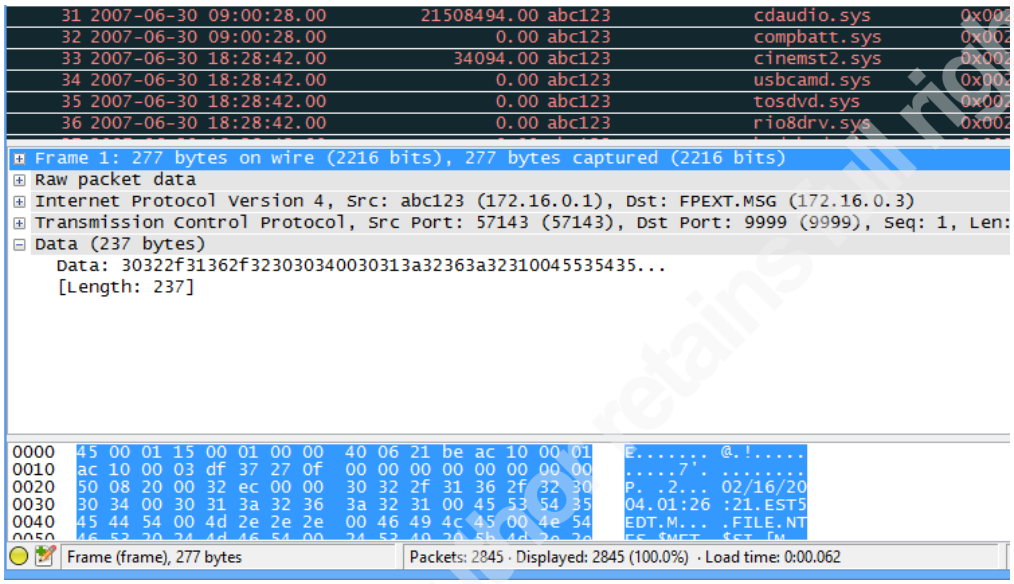


Figure 18: Packet details before dissector load.

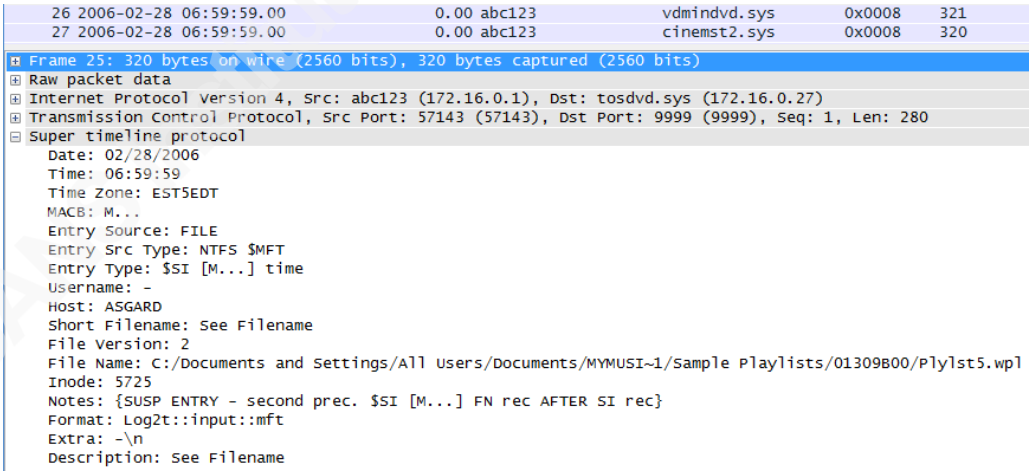


Figure 19: Packet details after dissector load.

Exploring the capabilities that the dissector exposes reveals that any of the elements can be added by right clicking and selecting *Apply as Column*. In addition, typing the protocol name in the Filter field reveals the list of protocol elements by name.

The list of super-timeline elements can be seen in Figure 20. This allows the analyst to more naturally filter based on fields that make sense in the context of timeline analysis.

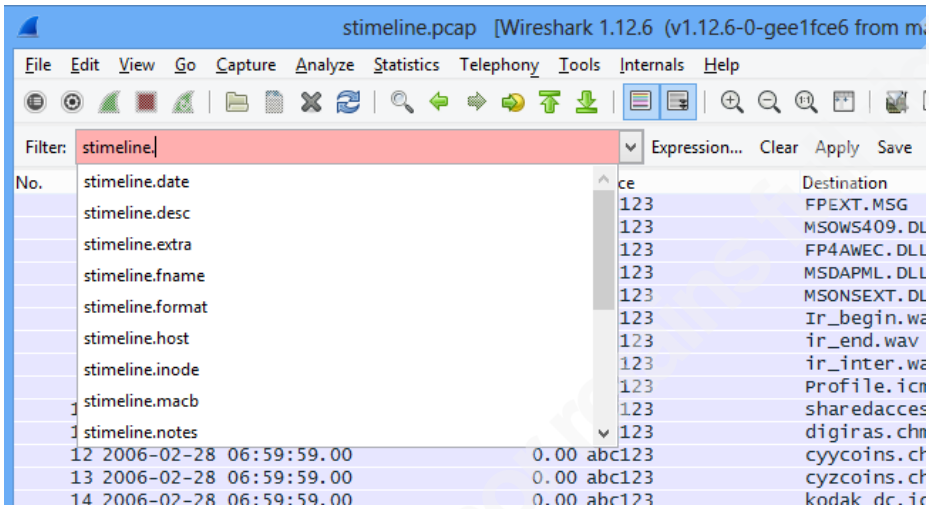


Figure 20: Super Timeline filter field values.

3.3 Wireshark Timeline Analysis Profiles

A final enabler for timeline analysis using the Wireshark tool is profile generation to provide context to the information presented in the Packet List pane. The most noticeable of these capabilities is Colorization Rules.

Prior to creation of analysis profiles, the interface will once again be rearranged to make better sense in the context of timeline analysis. The following changes will be applied to Wireshark with a super-timeline Pcap file loaded. The same process would apply equally to a standard timeline Pcap file with the missing fields omitted. Starting with the field list that ended Section 3.1, another round of column adds and removes will occur.

First, right click on the *Flags* field in the packet list pane and select *Remove Column*. A better representation of the MACB flags is now available through the parsed packet payload. To add this field, select any of the timeline packets, expand the *Super Timeline Protocol* entry in the packet details tree, right click on *MACB* and select *Apply as Column*. Do likewise with the *Short Filename*, *File Name*, *Entry Source*, and *Notes*

fields. This field configuration will serve as the baseline for colorization rule generation. A subset of this configuration can be seen in Figure 21.

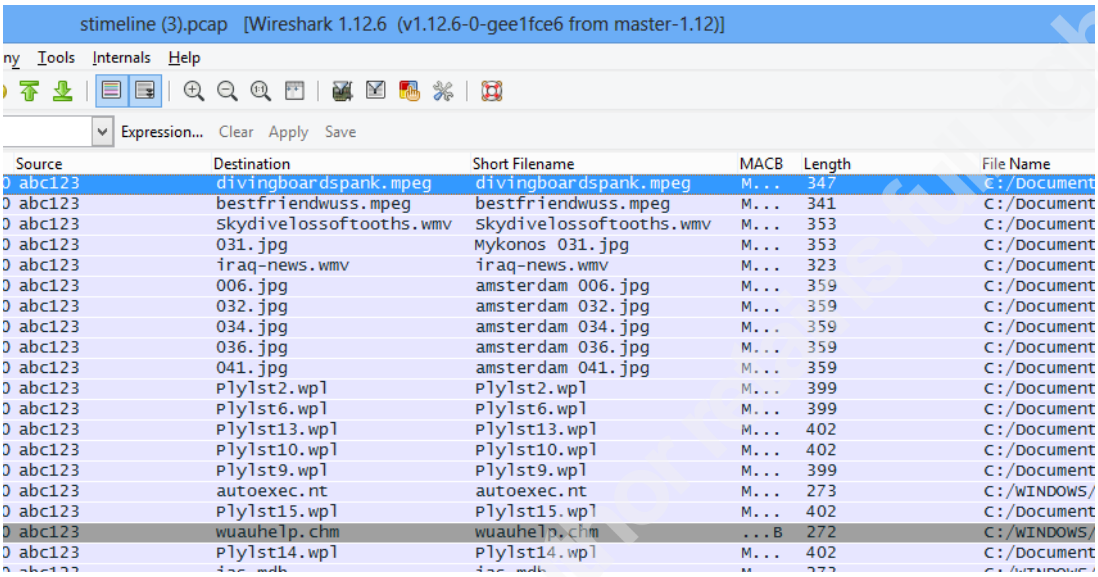


Figure 21: Super-timeline dissector field additions.

In order to generate a profile within Wireshark navigate to *Edit* > *Configuration Profiles*. This will activate the configuration profile dialogue seen in Figure 22. Click on the *new* button and enter a name for the profile. In this case, the profile will be used for identifying NTFS operations based on MACB flag combinations. Therefore, the profile will be named "Timeline – NTFS Operations – Lee."

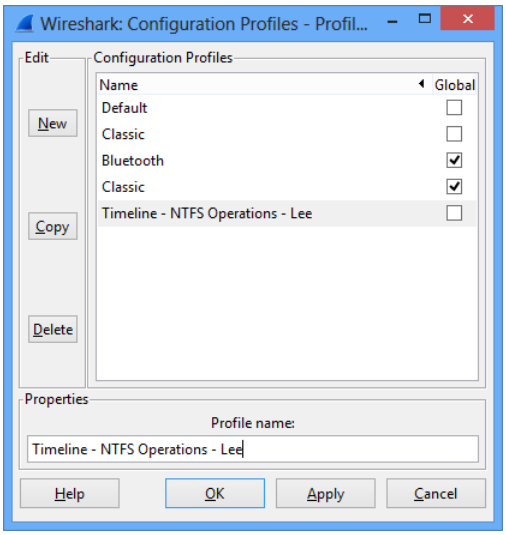


Figure 22: Wireshark Configuration Profile Dialogue.

Once the profile is added, it can be selected at any time using the profile selector found at the lower right corner of the display window. Profiles consist of a series of text files stored personal configuration directory identified in Figure 14. Each profile is stored in a sub-directory bearing the profile name. Any configuration changes applied through the user interface while the profile is active will be stored within this directory.

Initially, this profile will be extended to include colorization rules. The first action is to remove the default colorization rules. These rules apply to network traffic and are not applicable to forensic timeline analysis. Open the rules by navigating to *View > Coloring Rules*. Select the all of the existing rules and click the delete button. New coloring rules that pertain to forensic analysis can now be added.

Using the NTFS time rules for $STDINFO timestamps found in the FOR 508 course material simple colorization rules can be constructed. These rules all focus on the value of the MACB flag combination. The combinations shown in the course material cover eight different file system operations, seven of which are unique. Each rule can be added by clicking on the New button and specifying a color and filter rule. The colorization rules applicable to NTFS according to FOR 508 can be seen in Figure 23 (Lee, 2011).

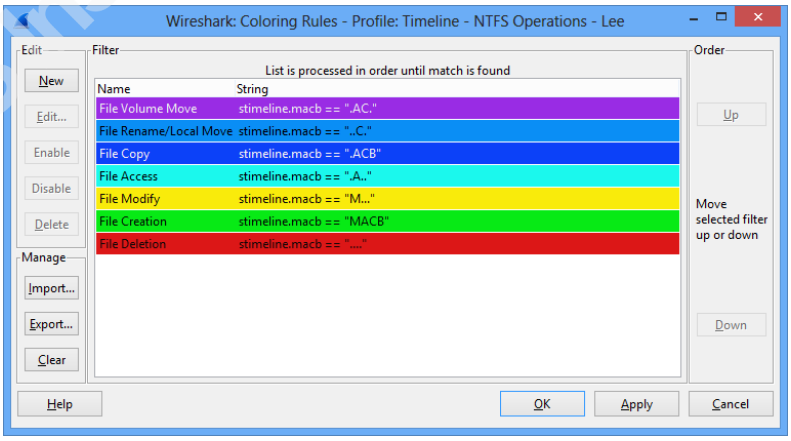


Figure 23: Timeline Colorization Rules for NTFS $STDINFO

Navigate to *View > Colorize Packet List* to ensure that packet colorization rules are applied to the current packet capture. Once this is complete, the packet list pane will

appear as seen in Figure 24. Note that there are still entries that are not assigned a color. This is due to incomplete understanding or ambiguities that exist in the way that NTFS timestamps are updated. As the meaning for these combinations become known, additional colorization rules can be added dynamically.

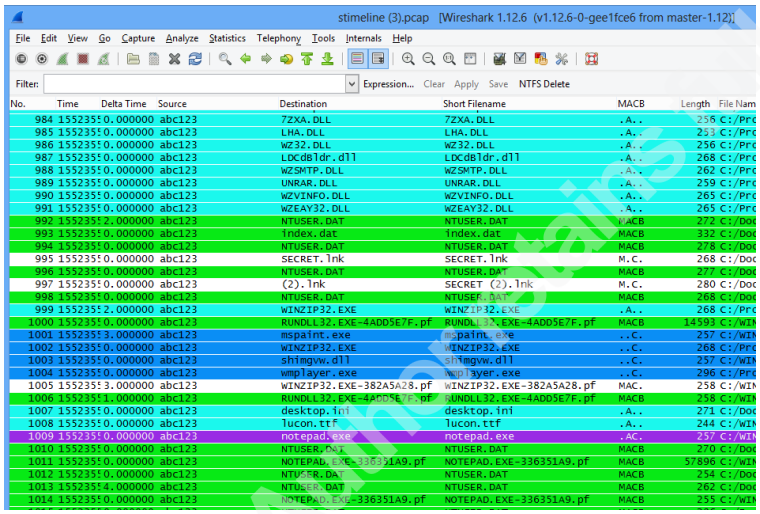


Figure 24: Colorized timeline packet list.

Additional colorization profiles can be found in Appendix E. These profiles implement each of the timestamp rule sets within the FOR 508 course materials in addition to Mr. Lee's Microsoft Excel colorization profile (Lee, 2012). The colorization rules for the Microsoft Excel profile include complex filters that have not been tested exhaustively. They represent a direct translation into Wireshark display filter language. The order of filtering has an impact on colorization and as a result, it may be necessary to break individual rules up to obtain the fully expected behavior.

The colorization rules described above provide the analyst context within the macro timeline view. In contrast, display filters allow the analyst to perform more focused analysis of timeline events. Using display filters, the analyst can zero in on dirty words or operations/behaviors of interest with respect to a particular line of investigation.

Display filters can be saved in Wireshark using one of two methods. The first method creates a reference to the stored filter on the filter bar as seen in Figure 25. The second method uses the *Analyze* > *Display Filters…* menu option seen in Figure 26.

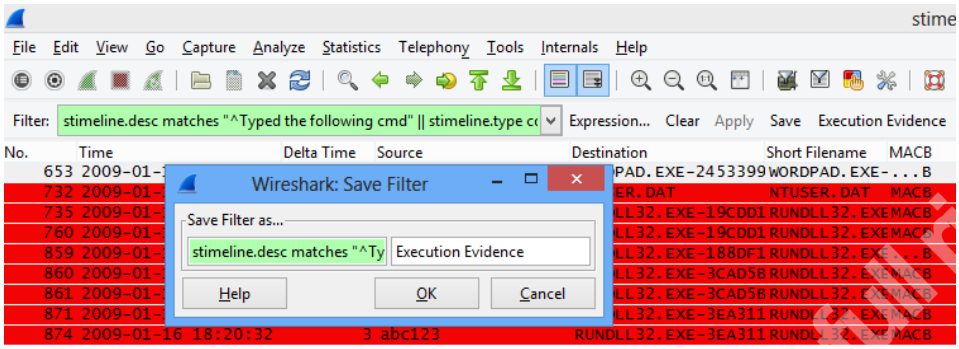


Figure 25: Display filter save using the filter bar.

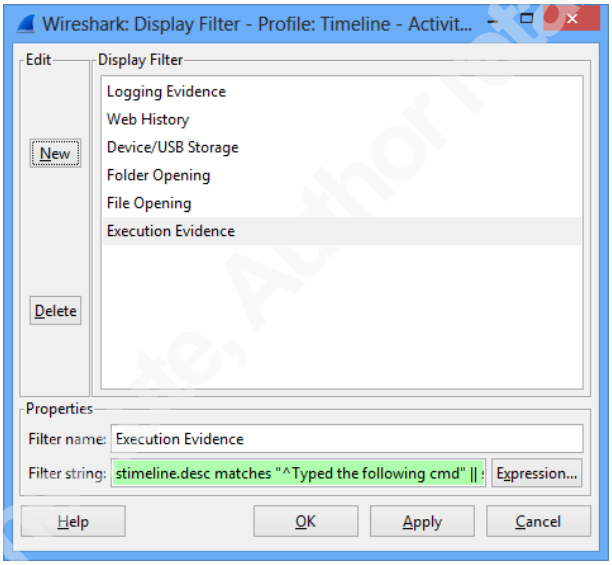


Figure 26: Display filter save using the Analyze > Display Filters… method.

The filter bar option makes stored display filters available through a series of buttons on the filter bar. To use this method, simply enter the display filter in the filter bar, click the save button, and provide a name. The drawback of this method is that the filter is not easily portable. No reference to the display filter exists within the profile directory.

In contrast, the *Analyze > Display Filters…* option creates entries in the "dfilters" text file located in the stored personal configuration directory identified in Figure 14. This allows an analyst to share display filters easily by copying the text file or the full profile directory to another computer.

David R. Fletcher Jr., david.fletcher.6@us.af.mil

A combination of both methods would likely be the best option for analysis. The filter bar filters would likely be a subset of the profile-stored filters. This gives ready access to filters used heavily during an investigation without limiting portability.

Applying the filters used for timeline colorization represents a good starting point for display filter development. Doing so allows the analyst to view file system operations and suspicious activity in context using colorization. As specific activity is identified as malicious, the analyst can switch to display filtering to identify additional occurrence of the same or similar activity. Baseline filters corresponding to NTFS, Linux, and Mr. Lee's Excel analysis profile can be found in Appendix F.

Colorization rules and display filters represent the most useful saved profile features in the context of timeline analysis. With a set of full profiles developed, an organization can simply archive the contents of the profile directories and make them available to their analysts. Analysts then simply need to extract the contents into their Wireshark personal profile directory for use.

4. Future Enhancements

While this script has shown that timeline analysis is possible using the capabilities of Wireshark, the user interface has room for improvement.

Investigations may involve multiple computers. The ability to correlate analysis across several computers would require some re-tooling of the timeline parser. Specifically, the IP range and host IP address would need to be a configurable option or command line parameter. Each Pcap file could then be generated separately using non-conflicting address spaces. Once all of the Pcap files have been created, the *mergecap* tool could then be used to combine the separate sources into a single file.

A comprehensive set of analysis profiles could be created and distributed as a single Zip file. This would allow the community to download the Zip file and apply the Wireshark analysis methodology more easily. The user would simply expand the profile

archive within their Wireshark profile directory to take advantage of colorization and filters.

This methodology could be applied to create another output format for either the *log2timeline* or *plaso* tools. The ability to directly create Pcap data from a timeline generation tool would remove an unnecessary intermediate step. Integration of this feature into the timeline generator would likely improve adoption as well.

Finally, a workaround for the Prefetch file metadata in the description field could be permanently solved. This would require an extension to the timeline parser to include generation of multi-packet TCP streams. While stream reassembly within Wireshark is not ideal, maintaining fidelity of the timeline contents is critical to a thorough investigation.

5. Conclusion

The Wireshark protocol analyzer is a flexible tool for network analysis with powerful statistics, highlighting, and filtering features. These features are valuable for more than just network analysis. Through conversion of timeline data from text to Pcap file an analyst gains access to the aforementioned features. In addition, the analyst is able to quickly create profiles that contain colorization rules and filters to expedite the analysis process.

The initial concept for this project involved correlating timeline data with common fields used in TCP/IP network analysis. During research, it became obvious that some of this correlation was unnecessary and actually caused difficulty.

Use of a host file failed to take into account filenames that contained whitespace characters; using this strategy proved a dead end. The options available to mitigate this issue included encoding the whitespace characters as non-whitespace strings or modifying the host file parser in the Wireshark application. Neither of these options were viable. The former would increase the burden on the analyst to interpret character representations on the fly. The latter would take a level of effort not practical given the timeframe for this paper.

David R. Fletcher Jr., david.fletcher.6@us.af.mil

The TCP flag manipulation made sense at a high level in the context of matching bit field operations. However, when applied, the Request for Comments became critical as combinations of TCP flags would cause unexpected behavior in Wireshark. For instance, initial choice of flags included use of the Reset flag to represent a MACB element. This caused the packet payload not to be displayed in Wireshark since reset packets do not have a payload. In addition, the filtering and display of the TCP flags field was not intuitive from a timeline analysis perspective. The analyst would have to mind map certain TCP flag bits to MACB fields on the fly.

All of the shortcomings outlined above were accommodated by use of the Wireshark dissector. This directly exposed timeline fields to the native Wireshark interface and allowed creation of columns, colorization rules, and filters using names that make sense to a forensic analyst.

6. Bibliography

Bishop, S. (2015, March 23). pytz - World Timezone Definitions for Python — pytz 2015.2 documentation. Retrieved from http://pytz.sourceforge.net/

Bjorlykke, S. (2009, June 17). *Lua Scripting in Wireshark* [PDF]. Retrieved from http://sharkfest.wireshark.org/sharkfest.09/DT06_Bjorlykke_Lua%20Scripting%20in%20Wireshark.pdf

Carbone, Richard. (2011). *Generating computer forensic super-timelines under Linux: A comprehensive guide for windows-based disk images*. Valcartier, Québec: Defence R&D Canada - Valcartier.

Carrier, B. (2005). *File system forensic analysis*. Boston, MA: Addison-Wesley.

Chappell, L. (2010). *Wireshark network analysis: The official Wireshark certified network analyst study guide*. San Jose, CA: Protocol Analysis Institute, Chappell University.

Kaplan, H. (2015, July 2). Lua/Dissectors - The Wireshark Wiki. Retrieved July 2, 2015, from https://wiki.wireshark.org/Lua/Dissectors

Lee, R. (2011). *File System Forensic Analysis, SANS Advanced Computer Forensic Analysis and Incident Response (V2011_0920)* (1st ed.). Bethesda, MD: SANS Institute.

Lee, R. (2011, December 7). SANS Digital Forensics and Incident Response Blog | Digital Forensic SIFTing: SUPER Timeline Creation using log2timeline | SANS Institute [Web log post]. Retrieved from http://digital-forensics.sans.org/blog/2011/12/07/digital-forensic-sifting-super-timeline-analysis-and-creation

Lee, R. (2012, January 25). Digital Forensic SIFTing: Colorized Super Timeline Template for Log2timeline Output Files. Retrieved from http://digital-forensics.sans.org/blog/2012/01/25/digital-forensic-sifting-colorized-super-timeline-template-for-log2timeline-output-files

Luttgens, J. T., Mandia, K., & Pepe, M. (2014). *Incident response & computer forensics*.

Metz, J. (2015, April 13). Home · log2timeline/plaso Wiki · GitHub. Retrieved May 10, 2015, from https://github.com/log2timeline/plaso/wiki

Python Software Foundation. (2015, May 27). Overview — Python 2.7.10 documentation. Retrieved from https://docs.python.org/2/

SecDev.org. (2007). Scapy. Retrieved from http://www.secdev.org/projects/scapy/

Stevens, W. R., & Fall, K. W. (2011). *TCP/IP illustrated: Volume 1*.

Trinh, T. (2012, June). Wireshark · Wireshark-users: Re: [Wireshark-users] Reading a zero-terminated string in Lua dissector. Retrieved from https://www.wireshark.org/lists/wireshark-users/201206/msg00010.html

Appendix A
Standard Timeline Conversion Script

```
#!/usr/bin/Python
#########################################################################
########
# This script will parse a timeline file to create a Pcap
representation
# of the timeline for analysis in wireshark using the built-in features
# such as profiles, colorization, filtering, and annotation.
#
# The basic process is to open the timeline file, parse each line, and
# output a host file and Pcap that can be used together.  Each entry in
# the timeline is processed to create an individual tcp packet.
Characteristics
# of the packet are mapped to the timeline entry information.  For
instance:
#
#     Source IP = Hostname
#     Destination IP = Target File
#     Source Port = Unused
#     Destination Port = Timeline Protocol Identifier
#     TCP Flags = MACB Flags
#
# This is just an example of the mapping power of using tcp.  Other
fields
# of the packet that may be useful are IPID, sequence numbers, options
field,
# and more.
#
# Limitations of this tool include:
#
# The host file is whitespace delimited so any whitespace characters
are
# interpreted as the end of the host name.  This causes file names with
white
```

```
# space to be displayed improperly.  This is a minor inconvenience as
it is
# remedied by importing the timeline parsing lua dissector which allows
the
# analyst to insert fields into the display to include the correct
filename.
#
# The maximum payload of a tcp packet is 65,535 bytes.  Timeline
entries that
# exceeded this maximum length were not expected but prefetch files
have a
# large amount of metadata included with them.  This could be resolved
by
# creating tcp sessions for this information but that paricular
solution
# defeats the purpose of use which is easy display, colorization and
marking
# of timeline data.
#
##########################################################################
######
# imports for required libraries
import argparse
import ntpath
import logging
logging.getLogger("Scapy").setLevel(1)

# import the Scapy module for packet generation
from Scapy.all import *
# import datetime and pytz to adjust for timezone offset
from datetime import datetime, timedelta
from pytz import timezone
import pytz

# Create a parser to read in command line arguments
parser = argparse.ArgumentParser(description='Convert a timeline file
into Pcap')
```

```
# Add argument for input timeline file
parser.add_argument('-f','--file', help='The timeline file to parse')
# Add argument for time zone
parser.add_argument('-z','--tzone', help='Timezone offset of the
timeline in pytz format')
# Add argument for output Pcap file
parser.add_argument('-o','--outfile',help='Output filename for the
resulting Pcap',default='./stimeline.Pcap')
# Add argument for hostname
parser.add_argument('-hn','--hostname',help='Name of the source host
for this evidence',default='computer')
# Add argument for hostfile
parser.add_argument('-hf','--hostfile',help='Output filename for the
resulting host file',default='./hosts')
# Parse arguments for use
args = parser.parse_args()

# Open the host file with write access
hostFile = open(args.hostfile, 'w')
# Set the location of the Pcap file
PcapFile = args.outfile

# Set the port to represent timeline protocol traffic
# this must match the port used in the target lua dissector
timelinePort = 7143

# Create a timezone object for use in setting packet timestamp
tz = timezone(args.tzone)

# Set the starting network address for the packet capture
# Currently, this packet capture will be able to process 56,634
# unique files
network = "172.16."
# Set the value of the third octet
three = 0
# Set the value of the fourth octet, reserving .1 for the source host
four = 2
```

```
# Set the source host information (IP and hostname)
srcIP = "172.16.0.1"
srcHost = args.hostname

# Create a packet list object to hold the crafted packets
pkts = Scapy.plist.PacketList()

# Write the first entry in the host file, identifying the source of
evidence
hostFile.write(srcIP + "\t" + srcHost + "\n")

# Dictionary to determine file/IP utilization
files = dict()

# Helper function for mapping months to month numbers
def getMonthNumberFromShortMonth(shortMonth):
   months = { "jan" : 1,
              "feb" : 2,
              "mar" : 3,
              "apr" : 4,
              "may" : 5,
              "jun" : 6,
              "jul" : 7,
              "aug" : 8,
              "sep" : 9,
              "oct" : 10,
              "nov" : 11,
              "dec" : 12
   }

   return months[shortMonth.lower()]

# Open the timeline file with read permissions for parsing
with open(args.file, 'r') as file:
   # For each line, parse the individual elements
   for line in file:
```

```
    elements = line.split(',')
    date =   elements[0]
    size =   elements[1]
    macb =   elements[2]
    mode =   elements[3]
    uid =    elements[4]
    gid =    elements[5]
    meta =   elements[6]
    fname =  elements[7]

    # if the date element is "Date" then we are reading the header,
skip
    if (date.lower() == "date"):
       continue

    # Create a short filename by splitting at the last forward slash
    short_fname_arr = fname.rsplit('/',1)

    # Parse the macb flags and set tcp flags accordingly
    # M=Push, A=Ack, C=Urg, B=Syn
    P = "P" if macb[0].upper() == "M" else "" # M
    A = "A" if macb[1].upper() == "A" else "" # A
    U = "U" if macb[2].upper() == "C" else "" # C
    S = "S" if macb[3].upper() == "B" else "" # B

    # Assemble the resulting tcp flags into a single field
    flags = S + A + P + U

    # Check the length of the short name array, if < 2 then
    # filename is first element, otherwise second
    if (len(short_fname_arr) < 2):
       short_fname = short_fname_arr[0][:-2]
    else:
       short_fname = short_fname_arr[1][:-2]

    # Check to see if we already have an entry for this file, since
    # multiple files may have the same name, we must use full name
```

```
      if fname in files:
       # If file already exists, reuse IP address
       ip = files[fname]
      else:
       # If not, then grab another IP, add it to the dictionary
         if (four == 254):
            if (three == 255):
               sys.exit('Timeline exceeded 56,654 entries...')
            else:
             four = 0
               three += 1
         else:
            four += 1
       ip = network + str(three) + "." + str(four)
         files[fname] = ip
       # Add a new entry to the host file
         hostFile.write(ip + "\t" + short_fname + "\n")

      # Parse the date from the entry for use in Scapy
      dateElements = date.split(' ')
      timeElements = dateElements[4].split(':')

      # If the timezone parameter is not null then use passed in
parameter
      # otherwise use None
      if (timezone != None):
         pktTime =
datetime(int(dateElements[3]),getMonthNumberFromShortMonth(dateElements
[1]),int(dateElements[2]),int(timeElements[0]),int(timeElements[1]),int
(timeElements[2]),0,tzinfo=tz)
      else:
         pktTime =
datetime(int(dateElements[3]),getMonthNumberFromShortMonth(dateElements
[1]),int(dateElements[2]),int(timeElements[0]),int(timeElements[1]),int
(timeElements[2]),0,None)

      # Assemble packet payload constructing null terminated string from
```

```
     # timeline file entry data
     data = date + "\x00" + size + "\x00" + macb + "\x00" + mode +
"\x00" + uid + "\x00" + gid + "\x00" + meta + "\x00" + short_fname +
"\x00" + fname + "\x00"     # Create an IP packet with TCP transport
and data payload

p=IP(src=srcIP,dst=ip)/TCP(sport=timelinePort,dport=9999,flags=flags)/R
aw(load=data)
     # Set packet timestamp to constructed value
     p.time = int(pktTime.strftime("%s"))

     # Append packet to packet list
     pkts.append(p)

# Write the Pcap file out to disk
wrPcap(PcapFile,pkts)

# Close open resources
hostFile.close()
file.close()
```

Appendix B
Super Timeline Conversion Script

```
#!/usr/bin/Python
#########################################################################
########
# This script will parse a timeline file to create a Pcap
representation
# of the timeline for analysis in wireshark using the built-in features
# such as profiles, colorization, filtering, and annotation.
#
# The basic process is to open the timeline file, parse each line, and
# output a host file and Pcap that can be used together.  Each entry in
# the timeline is processed to create an individual tcp packet.
Characteristics
# of the packet are mapped to the timeline entry information.  For
instance:
#
#     Source IP = Hostname
#     Destination IP = Target File
#     Source Port = Unused
#     Destination Port = Timeline Protocol Identifier
#     TCP Flags = MACB Flags
#
# This is just an example of the mapping power of using tcp.  Other
fields
# of the packet that may be useful are IPID, sequence numbers, options
field,
# and more.
#
# Limitations of this tool include:
#
# The host file is whitespace delimited so any whitespace characters
are
# interpreted as the end of the host name.  This causes file names with
white
# space to be displayed improperly.  This is a minor inconvenience as
it is
# remedied by importing the timeline parsing lua dissector which allows
the
# analyst to insert fields into the display to include the correct
filename.
#
# The maximum payload of a tcp packet is 65,535 bytes.  Timeline
entries that
# exceeded this maximum length were not expected but prefetch files
have a
# large amount of metadata included with them.  This could be resolved
by
# creating tcp sessions for this information but that paricular
solution
# defeats the purpose of use which is easy display, colorization and
marking
```

David R. Fletcher Jr., david.fletcher.6@us.af.mil

```
# of timeline data.
#
##########################################################################
######
# imports for required libraries
import argparse
import ntpath
import logging
logging.getLogger("Scapy").setLevel(1)

# import the Scapy module for packet generation
from Scapy.all import *
# import datetime and pytz to adjust for timezone offset
from datetime import datetime, timedelta
from pytz import timezone
import pytz

# Create a parser to read in command line arguments
parser = argparse.ArgumentParser(description='Convert a super-timeline
file into Pcap')
# Add argument for input super-timeline file
parser.add_argument('-f','--file', help='The super-timeline file to
parse')
# Add argument for timezone
parser.add_argument('-z','--tzone', help='Timezone offset of the
timeline pytz format')
# Add argument for output Pcap file
parser.add_argument('-o','--outfile',help='Output filename for the
resulting Pcap',default='./stimeline.Pcap')
# Add argument for hostname
parser.add_argument('-hn','--hostname',help='Name of the source host
for this evidence',default='computer')
# Add argument for hostfile
parser.add_argument('-hf','--hostfile',help='Output filename for the
resulting host file',default='./hosts')
# Parse arguments for use
args = parser.parse_args()

# Open the host file with write access add --hostfile argument
hostFile = open(args.hostfile, 'w')
# Set the location of the Pcap file add -o argument
PcapFile = args.outfile

# Set the port to represent timeline protocol traffic
# this must match the port used in the target lua dissector
stimelinePort = 57143

# Create a timzeon object for use in setting packet timestamp
tz = timezone(args.tzone)

# Set the starting network address for the packet capture
# Currently, this packet capture will be able to process 56,634
# unique files
network = "172.16."
# Set the value of the third octet
```

```
three = 0
# Set the value of the fourth octet, reserving .1 for the source host
four = 2

# Set the source host information (IP and hostname) add -src argument
srcIP = "172.16.0.1"
srcHost = args.hostname

# Create a packet list object to hold the crafted packets
pkts = Scapy.plist.PacketList()

# Write the first entry int the host file, identifying the source of
evidence
hostFile.write(srcIP + "\t" + srcHost + "\n")

# Dictionary to determin file/IP utilization
files = dict()

# Open the timeline file with read permissions for parsing
with open(args.file, 'r') as file:
   # For each line, parse the individual elements
   for line in file:
     elements = line.split(',')
     date =   elements[0]
     time =   elements[1]
     tz =     elements[2]
     macb =  elements[3]
     src =    elements[4]
     srctype = elements[5]
     type =   elements[6]
     user =   elements[7]
     host =   elements[8]
     short =  elements[9]
     desc =   elements[10]
     ver =    elements[11]
     fname =  elements[12]
     inode =  elements[13]
     notes =  elements[14]
     format = elements[15]
     extra = elements[16]

     # If the date element is "Date" then we are reading the header,
skip
     if (date.lower() == "date"):
        continue

     # Create a short filename by splitting at the last forward slash
     short_fname_arr = fname.rsplit('/',1)

     # Parse the macb flags and set tcp flags accordingly
     # M=Push, A=Ack, C=Urg, B=Syn
     P = "P" if macb[0].upper() == "M" else "" # M
     A = "A" if macb[1].upper() == "A" else "" # A
     U = "U" if macb[2].upper() == "C" else "" # C
     S = "S" if macb[3].upper() == "B" else "" # B
```

```
     # Assemble the resulting tcp flags into a single field
     flags = S + A + P + U

     # Check the length of the short name array, if < 2 then
     # filename is first element, otherwise second
     if (len(short_fname_arr) < 2):
        short_fname = short_fname_arr[0][:-2]
     else:
        short_fname = short_fname_arr[1][:-2]

     # Check to see if we already have an entry for this file, since
     # multiple files may have the same name, we must use full name
     if fname in files:
      # If file already exists, reuse IP address
        ip = files[fname]
     else:
      # If not, then grab another IP, add it to the dictionary
        if (four == 254):
           if (three == 255):
              sys.exit('Timeline exceeded 56,654 entries...')
           else:
              four = 0
              three += 1
        else:
           four += 1
        ip = network + str(three) + "." + str(four)
        files[fname] = ip
        # Add a new entry to the host file
        hostFile.write(ip + "\t" + short_fname + "\n")

     # Parse the date from the entry for use in Scapy
     dateElements = date.split('/')
     timeElements = time.split(':')

     # If timezone parameter is not null then use passed in paramter
     # otherwise use None
     if (timezone != None):
        pktTime =
datetime(int(dateElements[2]),int(dateElements[0]),int(dateElements[1])
,int(timeElements[0]),int(timeElements[1]),int(timeElements[2]),0,None)
     else:
        pktTime =
datetime(int(dateElements[2]),int(dateElements[0]),int(dateElements[1])
,int(timeElements[0]),int(timeElements[1]),int(timeElements[2]),0,None)

     # Assemble packet payload constructing null terminated strings
from
     # timeline entry data
     data = date + "\x00" + time + "\x00" + tz + "\x00" + macb + "\x00"
+ src + "\x00" + srctype + "\x00" + type + "\x00" + user + "\x00" +
host + "\x00" + short + "\x00" + ver + "\x00" + fname + "\x00" + inode
+ "\x00" + notes + "\x00" + format + "\x00" + extra + "\x00"
```

```
     # if the length of the data + description is shorter than the
packet
     # minus headers then append the desc to the payload
     if (len(data) + len(desc) <= 65494):
      data = data + desc + "\x00"
     # otherwise truncate description by appropriate amount to fit into
     # remaining payload space and append to the payload
     else:
        data = data + desc[:65494-len(data)] + "\x00"

     # Create an IP packet with TCP transport and data payload

p=IP(src=srcIP,dst=ip)/TCP(sport=stimelinePort,dport=9999,flags=flags)/
Raw(load=data)

     # Set packet timestamp to constructed value
     p.time = int(pktTime.strftime("%s"))

     # Append packet to packet list
     pkts.append(p)

# Write the Pcap file out to disk
wrPcap(PcapFile,pkts)

# Close open resources
hostFile.close()
file.close()
```

Appendix C
Standard Timeline Lua Dissector

```
-- Standard Timeline Protocol Dissector
-- This script is a dissector for standard timeline data embedded in
-- TCP packets with port 7143. The packet payload is a group of null
-- terminated strings which are used for forensic analysis.  This
-- dissector is typically used with a host file that translates the IP
-- addresses int he packet capture as file names.

-- Debug print for checking the dissector load in tshark
print ("Timeline.lua loaded")

-- Create a new dissector
FTIMELINE = Proto ("ftimeline", "Standard timeline protocol")

-- Create protocol fields
local f = FTIMELINE.fields

-- Add individual fields of the type stringz with appropriate labels
and comments
f.tstamp = ProtoField.stringz
("ftimeline.tstamp","Timestamp","Timestamp of timeline entry")
f.fsize = ProtoField.stringz ("ftimeline.fsize","File Size","File size
in bytes")
f.macb = ProtoField.stringz ("ftimeline.macb","MACB Flags","Modify,
Access, Change, Birth flags")
f.mode = ProtoField.stringz ("ftimeline.mode","Perm Mode","Unix Style
Permission Mode")
f.uid = ProtoField.stringz ("ftimeline.uid","UID","User Owner ID")
f.gid = ProtoField.stringz ("ftimeline.gid","GID","Group Owner ID")
f.meta = ProtoField.stringz ("ftimeline.meta","Meta","Mactime Meta
Field")
f.sfname = ProtoField.stringz ("ftimeline.sfname","Short Name","Short
File Name")
f.fname = ProtoField.stringz ("ftimeline.fname","File Name","File
Name")

-- Initialize packet counter
local packet_counter

-- Timeline initialization function
function FTIMELINE.init ()
   packet_counter = 0
end

-- Local function to determine string lengths for
-- calculating the offset of the next payload element
local function getStringLength(buffer, offset)
   return buffer(offset):stringz():len() + 1
end

-- Protocol dissector function
function FTIMELINE.dissector (buffer, pinfo, tree)
```

David R. Fletcher Jr., david.fletcher.6@us.af.mil

```
    -- Adding fields to the tree
    local subtree = tree:add (FTIMELINE, buffer())
    -- Set the initial offset to zero
    local offset = 0

    -- For each field represented in the protocol, read in a string
    -- calculate the new offset, and add the field to the tree
    local tstamp = buffer(offset):stringz()
    offset = offset + getStringLength(buffer, offset)
    subtree:add(f.tstamp, tstamp)

    local fsize = buffer(offset):stringz()
    offset = offset + getStringLength(buffer, offset)
    subtree:add(f.fsize, fsize)

    local macb = buffer(offset):stringz()
    offset = offset + getStringLength(buffer, offset)
    subtree:add(f.macb, macb)

    local mode = buffer(offset):stringz()
    offset = offset + getStringLength(buffer, offset)
    subtree:add(f.mode, mode)

    local uid = buffer(offset):stringz()
    offset = offset + getStringLength(buffer, offset)
    subtree:add(f.uid, uid)

    local gid = buffer(offset):stringz()
    offset = offset + getStringLength(buffer, offset)
    subtree:add(f.gid, gid)

    local meta = buffer(offset):stringz()
    offset = offset + getStringLength(buffer, offset)
    subtree:add(f.meta, meta)

    local sfname = buffer(offset):stringz()
    offset = offset + getStringLength(buffer, offset)
    subtree:add(f.sfname, sfname)

    local fname = buffer(offset):stringz()
    offset = offset + getStringLength(buffer, offset)
    subtree:add(f.fname, fname)

end

-- Register the port and protocol with the dissector table
DissectorTable.get("tcp.port"):add(7143, FTIMELINE)
```

Appendix D
Super Timeline Lua Dissector

```
-- Super Timeline Protocol Dissector
-- This script is a dissector for super-timeline data embedded in
-- TCP packets with port 57143.  The packet payload is a group of null
-- terminated strings which are used for forensic analysis.  This
-- dissector is typically used with a host file that translates the IP
-- addresses in the packet capture as file names.

-- Debug print for checking the dissector load in tshark
print ("STimeline.lua loaded")

-- Create a new dissector
STIMELINE = Proto ("stimeline", "Super-timeline protocol")

-- Create protocol fields
local s = STIMELINE.fields

-- Add individual fields of the type stringz with appropriate labels
and comments
s.date = ProtoField.stringz ("stimeline.date","Date","Date stamp of
timeline entry")
s.time = ProtoField.stringz ("stimeline.time","Time","Time stamp of
timeline event")
s.zone = ProtoField.stringz ("stimeline.zone","Time Zone","Timeline
time zone")
s.macb = ProtoField.stringz ("stimeline.macb","MACB","Modify, Access,
Change, Birth flags")
s.src = ProtoField.stringz ("stimeline.src","Entry Source","Timeline
entry source")
s.srctype = ProtoField.stringz ("stimeline.srctype","Entry Src
Type","Timeline entry source type")
s.type = ProtoField.stringz ("stimeline.type","Entry Type","Timeline
entry type")
s.user = ProtoField.stringz ("stimeline.user","Username","Timeline
entry user")
s.host = ProtoField.stringz ("stimeline.host","Host","Timeline entry
host")
s.sfname = ProtoField.stringz ("stimeline.sfname","Short
Filename","Short File Name")
s.ver = ProtoField.stringz ("stimeline.ver","File Version","File
Version")
s.fname = ProtoField.stringz ("stimeline.fname","File Name","Full File
Name")
s.inode = ProtoField.stringz ("stimeline.inode","Inode","File Inode")
s.notes = ProtoField.stringz ("stimeline.notes","Notes","Notes")
s.format = ProtoField.stringz ("stimeline.format","Format","Format")
s.extra = ProtoField.stringz ("stimeline.extra","Extra","Extra")
s.desc = ProtoField.stringz
("stimeline.desc","Description","Description")


-- Initialize packet counter
```

David R. Fletcher Jr., david.fletcher.6@us.af.mil

```
local packet_counter

-- Timeline initialization function
function STIMELINE.init ()
   packet_counter = 0
end

-- Local function to determine string lengths for
-- calculating the offset of the next payload element
local function getStringLength(buffer, offset)
   return buffer(offset):stringz():len() + 1
end

-- Protocol diessector function
function STIMELINE.dissector (buffer, pinfo, tree)

   -- Adding fields to the tree
   local subtree = tree:add (STIMELINE, buffer())
   -- Set the initial offset to zero
   local offset = 0

   -- For each field represented in the protocol, read in a string
   -- calculate the new offset, and add the field to the tree
   local date = buffer(offset):stringz()
   offset = offset + getStringLength(buffer, offset)
   subtree:add(s.date, date)

   local time = buffer(offset):stringz()
   offset = offset + getStringLength(buffer, offset)
   subtree:add(s.time, time)

   local zone = buffer(offset):stringz()
   offset = offset + getStringLength(buffer, offset)
   subtree:add(s.zone, zone)

   local macb = buffer(offset):stringz()
   offset = offset + getStringLength(buffer, offset)
   subtree:add(s.macb, macb)

   local src = buffer(offset):stringz()
   offset = offset + getStringLength(buffer, offset)
   subtree:add(s.src, src)

   local srctype = buffer(offset):stringz()
   offset = offset + getStringLength(buffer, offset)
   subtree:add(s.srctype, srctype)

   local type = buffer(offset):stringz()
   offset = offset + getStringLength(buffer, offset)
   subtree:add(s.type, type)

   local user = buffer(offset):stringz()
   offset = offset + getStringLength(buffer, offset)
   subtree:add(s.user, user)
```

```
    local host = buffer(offset):stringz()
    offset = offset + getStringLength(buffer, offset)
    subtree:add(s.host, host)

    local sfname = buffer(offset):stringz()
    offset = offset + getStringLength(buffer, offset)
    subtree:add(s.sfname, sfname)

    local ver = buffer(offset):stringz()
    offset = offset + getStringLength(buffer, offset)
    subtree:add(s.ver, ver)

    local fname = buffer(offset):stringz()
    offset = offset + getStringLength(buffer, offset)
    subtree:add(s.fname, fname)

    local inode = buffer(offset):stringz()
    offset = offset + getStringLength(buffer, offset)
    subtree:add(s.inode, inode)

    local notes = buffer(offset):stringz()
    offset = offset + getStringLength(buffer, offset)
    subtree:add(s.notes, notes)

    local format = buffer(offset):stringz()
    offset = offset + getStringLength(buffer, offset)
    subtree:add(s.format, format)

    local extra = buffer(offset):stringz()
    offset = offset + getStringLength(buffer, offset)
    subtree:add(s.extra, extra)

    local desc = buffer(offset):stringz()
    offset = offset + getStringLength(buffer, offset)
    subtree:add(s.desc, desc)

end

-- Register the port and protocol with the dissector table
DissectorTable.get("tcp.port"):add(57143, STIMELINE)
```

Appendix E
Timeline Colorization Rules

```
# SANS FOR 508 NTFS $STDINFO Timestamp Rules
# DO NOT EDIT THIS FILE!  It was created by Wireshark
@File Volume Move@stimeline.macb ==
".AC."@[39371,11281,58383][62139,61532,61532]
@File Rename/Local Move@stimeline.macb ==
"..C."@[2726,36555,62867][0,0,0]
@File Copy@stimeline.macb ==
".ACB"@[3214,16709,63538][64635,62207,62207]
@File Access@stimeline.macb == ".A.."@[6973,63545,60879][0,0,0]
@File Modify@stimeline.macb ==
"M..."@[63993,60242,2992][2514,2489,2155]
@File Creation@stimeline.macb ==
"MACB"@[1953,60103,5529][1404,3277,1508]
@File Deletion@stimeline.macb ==
"...."@[56364,6000,6000][3553,1732,1732]


# SANS FOR 508 Linux Timestamp Rules
# DO NOT EDIT THIS FILE!  It was created by Wireshark
@File Creation@stimeline.macb == "MAC."@[3201,63881,3034][0,0,0]
@File Modify@stimeline.macb == "M.C."@[58230,44528,5213][0,0,0]
@File Access@stimeline.macb == ".A.."@[63585,60199,5461][0,0,0]
@File Copy@stimeline.macb == "MAC."@[7422,58335,59580][0,0,0]
@File Move/Delete@stimeline.macb ==
"..C."@[9027,13069,58214][63724,63118,63118]


# Rob Lee Super Timeline Colorization Rules
# DO NOT EDIT THIS FILE!  It was created by Wireshark
@Log File@stimeline.srctype contains "Log" || stimeline.srctype
contains "XP Firewall Log"@[53778,55283,55516][0,0,0]
@Folder Opening@stimeline.short contains "ShellNoRoam/Bags" ||
stimeline.short contains "BagMRU"@[9626,31311,16693][62139,61532,61532]
@Device/USB Usage@stimeline.desc matches "drive mounted$" ||
stimeline.type contains "Drive Last Mounted" || stimeline.short matches
"drive mounted$" ||  stimeline.srctype contains "SetupAPI log" ||
stimeline.short contains "RemovableMedia" || stimeline.short contains
"STORAGE/RemovableMedia" || stimeline.short contains "USB" ||
stimeline.short contains "/USB/Vid_" || stimeline.short contains
"Enum/USBSTOR/Disk&Ven_" || stimeline.short contains "volume mounted"
|| stimeline.srctype == "MountPoints2 key"
@[8268,2592,64588][63724,63118,63118]
@Execution@stimeline.desc matches "^Typed the following cmd" ||
stimeline.type contains "CMD Typed" || stimeline.srctype contains
"RunMRU key" || stimeline.short contains "RunMRU" || stimeline.short
contains "UEME_RUNPIDL" || stimeline.desc contains ".pf" ||
stimeline.type contains "Last run" || stimeline.short contains
"MUICache" || stimeline.src == "PRE" || stimeline.srctype ==
"UserAssist key" || stimeline.type == "Time of Launch" ||
stimeline.short contains "UEME_" || stimeline.srctype contains
```

David R. Fletcher Jr., david.fletcher.6@us.af.mil

```
"PreFetch" || stimeline.srctype == "XP Prefetch" || stimeline.short
contains "UEME_RUNPATH" || stimeline.fname contains ".pf" ||
stimeline.short contains "was executed" || stimeline.short contains
".pf"  @[62354,1067,1067][0,0,0]
@Deleted Data@stimeline.desc matches "^.DELETED" || frame contains
"RECYCLE" || stimeline.short contains "DELETED" || stimeline.srctype ==
"Deleted Registry" || stimeline.srctype ==
"$Recycle.bin"@[2114,2114,2114][65006,65006,65006]
@File Opening@stimeline.desc contains "URL:file:///" || frame contains
"LNK" || stimeline.short contains "opened by" || stimeline.short
matches "^URL:file:///" || stimeline.short contains "CreateDate" ||
stimeline.short contains "visited file://" || frame contains ".lnk" ||
stimeline.short contains "recently opened file" || stimeline.desc
matches "^Recently opened file of extension" || stimeline.type contains
"File Opened" || stimeline.srctype == "RecentDocs key" ||
stimeline.type contains "Folder Opened"@[35232,63873,790][0,0,0]
@Web History@stimeline.srctype contains "Firefox 3 history" ||
stimeline.src == "WEBHIST" || stimeline.srctype == "Internet Explorer"
|| stimeline.desc matches "^URL" || stimeline.type contains "URL" ||
stimeline.desc contains "http://" || stimeline.desc contains "LSO" ||
stimeline.type contains "LSO" || stimeline.srctype contains "LSO" ||
stimeline.srctype contains "Flash cookie" || stimeline.src contains
"LSO" || stimeline.short contains "visited" || stimeline.short contains
"URL" || stimeline.short matches "^Flash Cookie"
@[61699,46758,3924][0,0,0]
```

Appendix F
Stored Timeline Analysis Filters

```
# Stored Display Filters FOR 508 $STDINFO Timestamp Rules
# Timeline - NTFS Operations - Lee
"File Volume Move" stimeline.macb == ".AC."
"File Rename/Local Move" stimeline.macb == "..C."
"File Copy" stimeline.macb == ".ACB"
"File Access" stimeline.macb == ".A.."
"File Modify" stimeline.macb == "M..."
"File Creation" stimeline.macb == "MACB"
"File Deletion" stimeline.macb == "...."


# Stored Display Filters FOR 508 Ext2/3/4 Timestamp Rules
# Timeline - Linux Operations - Lee
"File Creation" stimeline.macb == "MAC."
"File Modify" stimeline.macb == "M.C."
"File Access" stimeline.macb == ".A.."
"File Copy" stimeline.macb == "MAC."
"File Move/Delete" stimeline.macb == "..C."


# Stored Display Filters Rob Lee Excel Analysis
# Activity based analysis
"Logging Evidence" stimeline.srctype contains "Log" ||
stimeline.srctype contains "XP Firewall Log"
"Web History" stimeline.srctype contains "Firefox 3 history" ||
stimeline.src == "WEBHIST" || stimeline.srctype == "Internet Explorer"
|| stimeline.desc matches "^URL" || stimeline.type contains "URL" ||
stimeline.desc contains "http://" || stimeline.desc contains "LSO" ||
stimeline.type contains "LSO" || stimeline.srctype contains "LSO" ||
stimeline.srctype contains "Flash cookie" || stimeline.src contains
"LSO" || stimeline.short contains "visited" || stimeline.short contains
"URL" || stimeline.short matches "^Flash Cookie"
"Device/USB Storage" stimeline.desc matches "drive mounted$" ||
stimeline.type contains "Drive Last Mounted" || stimeline.short matches
"drive mounted$" ||  stimeline.srctype contains "SetupAPI log" ||
stimeline.short contains "RemovableMedia" || stimeline.short contains
"STORAGE/RemovableMedia" || stimeline.short contains "USB" ||
stimeline.short contains "/USB/Vid_" || stimeline.short contains
"Enum/USBSTOR/Disk&Ven_" || stimeline.short contains "volume mounted"
|| stimeline.srctype == "MountPoints2 key"
"Folder Opening" stimeline.short contains "ShellNoRoam/Bags" ||
stimeline.short contains "BagMRU"
"File Opening" stimeline.desc contains "URL:file:///" || frame contains
"LNK" || stimeline.short contains "opened by" || stimeline.short
matches "^URL:file:///" || stimeline.short contains "CreateDate" ||
stimeline.short contains "visited file://" || frame contains ".lnk" ||
stimeline.short contains "recently opened file" || stimeline.desc
matches "^Recently opened file of extension" || stimeline.type contains
```

```
"File Opened" || stimeline.srctype == "RecentDocs key" ||
stimeline.type contains "Folder Opened"
"Execution Evidence" stimeline.desc matches "^Typed the following cmd"
|| stimeline.type contains "CMD Typed" || stimeline.srctype contains
"RunMRU key" || stimeline.short contains "RunMRU" || stimeline.short
contains "UEME_RUNPIDL" || stimeline.desc contains ".pf" ||
stimeline.type contains "Last run" || stimeline.short contains
"MUICache" || stimeline.src == "PRE" || stimeline.srctype ==
"UserAssist key" || stimeline.type == "Time of Launch" ||
stimeline.short contains "UEME_" || stimeline.srctype contains
"PreFetch" || stimeline.srctype == "XP Prefetch" || stimeline.short
contains "UEME_RUNPATH" || stimeline.fname contains ".pf" ||
stimeline.short contains "was executed" || stimeline.short contains
".pf"
```