FACE: Automated Digital Evidence Discovery and Correlation

Andrew Case, Andrew Cristina, Lodovico Marziale, Golden G. Richard III, Vassil Roussev
Department of Computer Science
University of New Orleans
New Orleans, LA 70148
Email: {acase, acristin, vico, golden, vassil}@cs.uno.edu

***Abstract*—Digital forensic tools are being developed at a brisk pace in response to the ever increasing variety of forensic targets. Most tools are created for specific tasks–file system analysis, memory analysis, network analysis, etc.–and make little effort to interoperate with one another. This makes it difficult and extremely time-consuming for an investigator to build a wider view of the state of the system under investigation. In this work, we present FACE, a framework for automatic evidence discovery and correlation from a variety of forensic targets. Our prototype implementation demonstrates the integrated analysis and correlation of a disk image, memory image, network capture, and configuration log files. The results of this analysis are presented as a coherent view of the state of a target system, allowing investigators to quickly understand it. We also present an advanced open source memory analysis tool, *ramparser*, for the automated analysis of Linux systems.**

I. Introduction

The leading challenge for digital forensic investigations is that of scale. According to FBI statistics, the average case size has tripled in three years-from 80GB in FY 2003 to 250GB in FY 2006 [10]. This trend has had a significant impact on the field operations at digital forensics labs. Indeed, the mere acquisition, extraction and preprocessing of the data sources creates a long list of technical problems and adds to the already long turnaround times. However, a deeper and more significant implication of recent trends is the escalating complexity of scenarios of digital investigations. As the capabilities of individual applications, the size of forensic targets, and the number of networked systems all increase, the number of possible interactions and possible outcomes of forensic interest grows exponentially. The vast majority of current forensic tools focus on extracting first-order information about individual artifacts–name, size, location, timestamps, keywords, etc–and presenting it to the investigator. This leads to a browse-and-search approach to the investigation that leaves all the tedious work of "connecting the dots" to the investigator. As the complexity of systems grows rapidly, it becomes ever more difficult for an investigator to perform thorough, reliable, and timely investigations.

In our view, developers of digital forensics tools must adopt a broader, scenario-driven philosophy to tool development. We must recognize that merely swamping the user with all available data, or presenting basic search and filtering techniques, falls well short of what is actually needed. In general, most investigations can be reduced to four basic questions: a) what happened? b) when did it happen? c) how did it happen? and d) who did it? Along those lines, extracting the file system MAC times, for example, is usually a necessary step in establishing the sequence of events that led to the observed state of the system. However, merely providing tens of millions of timestamps to the investigator is not, by itself, very useful. The investigator would likely correlate MAC data with timing data from other available sources, such as MAC times from other file systems and timestamps from available network logs/traces. From an engineering perspective, it is clear that the task of extracting timing data from different sources should be split among different functional modules, and the correlation analysis aspect should be handled separately. The current state-of-the-art, however, is dominated by monolithic proprietary tools with very limited analytical abilities and individual, open-source tools that focus on performing specific extraction tasks but have virtually no analytical facilities.

The presented work is inspired by the 2008 DFRWS Challenge [7] and seeks to make two distinct contributions. First, we present a set of new tools that perform deep analysis of Linux physical memory images. Second, we present a proof-of-concept environment, called FACE, which illustrates our vision of an integrated approach to the forensic analysis of computer systems by bringing together the information generated by the different tools. Specifically, we demonstrate the ability to automatically correlate events and objects among a memory image, file system image, and network capture. As the resulting discussion shows, this approach can significantly speed up the investigative process by providing a higher-level, logical view of related events and objects. This also results in an intuitive interface that allows the investigator to follow leads and filter out irrelevant data in a natural way.

II. Related Work

A number of tools have been proposed for forensics analysis, including commercial offerings and a variety of open source tools. Most of these tools operate on only a single type of digital evidence (e.g., a memory dump, a disk image, or network traces). A selection of the most relevant tools is

This work was supported in part by the National Science Foundation under grant # CNS-0627226.

discussed below, categorized by the primary type of digital evidence that they process.

A. Disk Analysis Tools

FTK [12] and Encase [9] are commercial digital forensics suites that offer a point–and–click interface for analyzing captured disk images. The Sleuthkit and Autopsy [3] provide an open source alternative to commercial suites, allowing analysis of disk images at multiple levels, from individual data units through the filesystem layer. [5] provides an excellent reference for filesystem forensics. When filesystem metadata is missing or damaged, file carving tools such as Scalpel [20] or Foremost [11] can be used to carve sequences of bytes into recovered files.

B. Live Forensics Tools

Encase Enterprise edition [9] adds live forensics capabilities for enterprise networks by deploying software agents on machines to be monitored. These agents can capture memory and perform other monitoring activities under the supervision of a forensic analyst. The Mobile Forensics Platform [1], now called the OnlineDFS in its commercial incarnation, allows remote, live investigation of forensics targets without the need to install software agents on the machines under investigation. Administrative credentials are used to retrieve a variety of information about the running system, including process lists, open files, and networking statistics.

C. Off-line Memory and Log Analysis Tools

There is increasing interest in performing deep memory analysis as a standard part of digital forensics investigation, because a substantial amount of potential evidence is lost if this source is ignored. Recent students have illustrated that data persists for a long time in volatile memory [6][27]. Unlike tools that analyze running machines, such as Encase Enterprise and OnlineDFS, off-line memory analysis tools extract digital evidence directly from physical memory dumps. These memory dumps may be acquired using a number of different mechanisms (dependent on OS type and version), from hardware–based approaches such as Tribble [4] and via Firewire [15] to software–only approaches, such as using *dd* to access the physical memory device or via insertion of custom kernel modules. These memory dumping mechanisms are not infallible and some high–tech approaches to subverting memory acquisition have been proposed [21]. Fortunately, unless the subversion mechanism is very deeply embedded in the OS, a substantial amount of overhead may be incurred to prevent acquisition, potentially revealing the presence of a malicious agent [13]. A recently released tool provides another alternative for memory acquisition, by converting Windows hibernation files to usable memory dumps [22]. Finally, a novel approach to memory acquisition called BodySnatcher, involving injection of a small, forensic OS that subverts the running OS, was presented at DFRWS 2007 [23].

Surprisingly, there has been little work in deep parsing of Linux memory dumps. idetect [16] is a proof of concept tool that parses 2.4–series memory dumps and enumerates page frames, discovers user mode processes, and provides detailed information about process descriptors. [28] discusses many of the relevant OS structures that must be parsed to extract digital evidence from Linux memory dumps.

Recently, a number of utilities for parsing Windows memory dumps have been developed, with a primary catalyst being the 2005 DFRWS memory analysis challenge (http://dfrws.org/2005/challenge/). The Volatility framework [30] extracts information from Windows XP SP2 memory dumps, including a list of running processes, open network connections, loaded DLLs, and Virtual Address Descriptor (VAD) information. The knttools [17] dump information about processes, threads, access tokens, the handle table, and other OS structures from a Windows memory dump. [18] is capable of outputting similar information. Cross-referencing is used to detect hidden objects. Schuster's ptfinder tools [25] take a different approach and instead of walking OS structures, attempt to carve objects that represent threads and processes directly from the memory dump. This allows hidden processes to be more easily discovered and can also reveal information about recently terminated processes. [29] provides an overview of several memory acquisition tools for Microsoft Windows. The system most closely related to ours is PyFlag [19], which allows viewing of log files, network traces, windows memory dumps (by incorporating Volatility), and other data within a common framework.

Our system is differentiated primarily by its correlation capabilities. Unlike the tools discussed above (with the exception of PyFlag), our framework allows an investigator to quickly organize and correlate evidence from a number of sources, including memory dumps, network traces, and file system images. While PyFlag does handle multiple sources, our tool goes one step further by providing views of related data across multiple sources. For example the "user" view displays the name and user id of the user, running processes owned by the user, and network sockets and files owned by the processes. Similar views exist for processes, files, etc. This enables the investigator to quickly target interesting evidence and follow sequences of events that provide a "big picture". In addition, our RAM parsing component for Linux performs deep analysis, revealing not only common information such as a list of running processes and open file handles, but also deeper information, such as network packets pending on open network connections.

III. *ramparser*: A Linux Memory Analysis Tool

Due to the lack of available memory parsing tools for Linux, we developed a new tool called *ramparser* that performs deep analysis of Linux memory dumps. Specifically, the current version is able to handle a range of 2.6 kernel variants. The information provided by *ramparser* from the memory dump includes running processes, open network connections, in-kernel socket buffers, loaded kernel modules, and a specific process's memory-mapped and open files, code, and data. For experienced UNIX users, the tool is capable of simulating

commands such as ps, lsmod, and netstat. It is also capable of writing out process-specific information and data to files for later investigation. The main point of *ramparser* is to provide detailed output about all running processes that can be used by FACE's correlation engine.

Initialization. All process–related operations rely on having the valid kernel virtual address of init's $task\_struct$. Because of this the first thing that *ramparser* does is scan the memory dump and locate the init process' address by carving $task\_struct$'s from memory until one is found with a pid of 1. After this, *ramparser* will be able to walk the entire list of active processes. The address of $init\_mm$ is set next so that we can find paged data in the kernel. The kernel convention is to use $init\_mm$ as the page directory pointer for any in–kernel data which requires paging instead of being identity mapped. After these values are set, the program parses the user supplied arguments and performs the desired analysis.

Generic validation. In order to reliably walk large amounts of memory and find valid structures, common routines were created to validate specific data structures used often in the kernel. For example, the $list\_head$ data structure, which implements circularly linked lists in the kernel, contains two members, next and prev. In order to help debugging use after free(), these members are set to poison values after being freed. This creates only two possible values for the members, either a valid kernel pointer or their respective poison values. Incorporating these restrictive ranges and others when searching for various data types allows the program to run quickly and with few false positives. The most reliable and useful types to validate are lists, enums, stack based character buffers, and kernel pointers which cannot be null. Enums are very useful for eliminating false positives because they generally have a small range of valid values. Similarly, the kernel uses many integers as enums, only allowing values such as -1, 0, or 1 to be assigned. Stack based character buffers are excellent for debugging since they can be easily printed, and validating the strings helps reduce false positives. Kernel pointers that cannot be null also have a relatively small range, i.e., PAGE_OFFSET to 0xffffffff on 32–bit systems. The possibility of pointers being null is not useful since RAM generally has many zero filled areas and accounting for them slows down searching. Less useful for searching, but still accounted for, are some integers and shorts which should never have negative values.

Finding $task\_struct$'s. Finding $task\_struct$'s is very reliable since the structure contains an enum for the sleep type, a stack–based character buffer for the process name, many non-null kernel pointers, unsigned integers, and lists. By validating the numerous members of the structure as it walks memory, the program rarely produces false positives. The -d option to *ramparser* will parse the memory image for $task\_struct$'s and print the revelant members.

Retrieving Process Information. After finding init it is then possible to walk the 'tasks' member of init's $task\_struct$ which holds the linked list of all active tasks. Partially simulating the ps(1) command is a simple operation, involving walking the process list and printing out detailed information for each process. More useful operations such as walking a process' memory maps, open files, and network sockets are possible by using similar constructs already used in the kernel. The *ramparser* -x option performs the simple ps(1) operation. See Figure 1 for a sample process listing.

Finding mapped files. Under Linux, virtually contiguous mapped regions with the same permissions are represented by a vm_area_struct. These represent a process' stack, heap, code section, data section, the data and code section of shared libraries, shared memory, and anonymously mapped memory. These structures can be viewed on a running machine by executing 'cat /proc/<pid>/maps' for the process of interest. *ramparser*'s -p options simulates the maps file for a process by walking the list of vm_area_struct structures that are contained within the process' mm_struct and printing the starting and ending address, permissions, and the mapped file's name, if available. The -v option of *ramparser* will write the memory pages covered by a processes vm_area_struct's to disk. Since these areas are paged, each memory region is handled 4096 bytes a time when determining the offsets of the data. Using these offsets, an investigator can completely recreate the running process at the time the memory image was taken. See Figure 2 for sample output for an FTP process.

Finding open files. The process descriptor contains a $files\_struct$ structure which includes a $struct\ fdtable$ which holds an array of $struct\ file$ structures. By following these links, it becomes easy to traverse all the file descriptors of a process. Each file descriptor is represented by a $struct\ file$ which *ramparser* uses to extract the open files, their permissions, and file descriptor number. Files with forensics interest include open files on disk, pipes, and sockets. This information can be viewed on a running system by executing 'ls -l /proc/<pid>/fd'. *ramparser* simulates this functionally in the -o option by walking the file descriptor array and printing the file descriptor number and name of the file opened. Filenames for files on disk are simply their full pathnames, while names for sockets and pipes are formed by joining socket[<inode number>] and pipe[<inode number>] respectively. See Figure 3 for sample output for an FTP process.

Finding sockets/netstat information. Since UNIX systems treat sockets as file descriptors, information about open network connections can be gathered by analyzing a process' socket file descriptors. Each socket is represented by a $struct\ socket$ which contains a pointer to the $struct\ sock$ for the socket. The socket structure contains the socket's family, protocol, receive and send queues, and state. The $inet\_sock$ structure representation of $struct\_sock$ gives the source and destination addresses and ports. Using this information it is possible to implement a netstat(8) like functionality. *ramparser* simulates netstat for all sockets when run with the -N option or gives information about a single process when run with -n. See Figure 4 for sample netstat output.

Finding network buffers. Most network investigations involve packet captures taken from the hostile network after suspicious activity is detected. By using the internal kernel

network structures it is possible to get even more information related to the network activity at the time of the memory dump. A network socket buffer is represented by $struct\ sk\_buff$ which contains protocol–specific information and points to the beginning and end of a complete packet. Inside each $struct\ sock$ structure is a receive queue and a send queue of socket buffers which hold data yet to be processed. By collecting these queues from open sockets, data that is yet to be sent out or data that is yet to be processed by a userland application can be gathered and associated with a specific process. Our experiments revealed that the receive queue was usually empty since userland servers process data very quickly, which removes the buffers from the queues. Unlike the receive queues, however, the send queues were generally full during large file transfers. Tests were run uploading files through FTP to outside networks, and *ramparser* was able to recover large parts of the files being transferred. *ramparser*'s -k and -q options can be used to dump the send and receive queues of a process to files.

Finding loaded modules. Loadable modules allow users to insert code dynamically into a running kernel. While this has obvious advantages, it is also a very common entry point for rootkits and other malware to run kernel level code. Modules are represented in the kernel by a $struct\ module$ which is defined in include/linux/module.h. *ramparser* is able to find loadable modules in memory with rare false positives due to the module structure containing an enum, list, stack based character buffer, and many kernel pointers. Searching for modules is the slowest part of the code since many of the pointers can be null. This decreases the performance of address validation. After locating a valid struct it is possible to recreate output obtained from the lsmod(8) command as viewed on a running Linux machine.

IV. FACE: Forensics Automated Correlation Engine

Memory analysis is just one component of a forensic investigation and most of the answers to the four questions we posed (e.g., "what happened?", "when did it happen?", etc.) require a more complete view of the system. Such a view is possible if all available information is considered, including the memory dump, filesystem, log files, and network traces. The memory dump allows for reconstruction of all processes that were running on the system, and for per–process analysis, such as collecting open files, active sockets, and memory mappings. The packet capture contains timestamps and streams which can be used to easily match network traces to active sockets on the system. The wtmp and utmp files store information about when users logged in, their login location, and the time of login. Timelines and the correct order of events are critical during investigations, and by using the timestamps from both the login files and network traces the engine can accurately frame what actions a user performed during specific times. The passwd and group files map users to their user id, group id, home directory, and login shell. On–disk metadata and kernel structures only refer to user ids and group ids which are not friendly to a user, but by incorporating information from the passwd and group files, the UI can present the user with the names representing these groups and users. This also speeds up investigations since a user can be quickly identified based on their username.

We propose a framework which provides automated parsing of multiple forensically interesting objects and correlation of the results between them. The five main objects used for data are memory dumps, network traces, disk images, log files, and user accounting and configuration files. A fairly complete reconstruction of the state of the running machine and its network activity becomes possible by fully parsing and analyzing these sources. By correlating the information, we are able to link data in a network trace back to the user who started the process which caused the traffic. This is possible by first matching an open socket in the memory dump to packets in the network trace. Since we know what process owns the open socket, we can then determine what user started the process and from where they logged in. It is also possible to determine what file on disk was being transferred by observing the open files of the process. Our system can also correlate data currently queued in the kernel's network stack with data of the network trace. Partially transferred files can be fully reconstructed by joining these two sets of data together. Analysis of malicious or unknown binaries or processes is made easier as the framework allows the user to download all or part of a process' memory or a file's data. Similarly, sections of memory can have more traditional forensics procedures applied to them such as file carving and hashing. Our framework also allows an investigator to get categorized views of disk and user activity at the time of the memory dump. Using FACE, it is possible to display all activity from a single user such as open files, active network connections, and running processes.

The rest of this section describes the various components in our framework, including parsers for network traces, configuration and log files, and the correlation engine.

A. Parsers

In addition to *ramparser*, described in the previous section, we implemented additional parsing tools, discussed below.

1) Pcap Parser: To avoid duplicating the effort of excellent network capture parsing tools like Wireshark, we opted to implement a much simpler module for parsing captures into a format for the correlation engine to work with. Our module reads in a pcap-format capture file and breaks it down into streams. Here we define streams as a collection of packets having the same source and destination ip address and port. This allows the correlation engine to display inflows (collections of packets originating from the host) and outflows (those packets originating from elsewhere but destined for the host). For each of these streams the module outputs the type: TCP or UDP (other protocols are not currently implemented), and a list of the packets in the stream. For each packet in the list, we output the timestamp from the pcap header for the packet and the beginning and ending offsets of the packet in the capture

file. Output is to a plain text file in a LISP-like s-expression format.

2) Configuration / Log File Parsers: In order to glean more information from the target system, we wrote three simple modules to parse a selection of files from the target filesystem. First, we parse /etc/passwd for information about users on the system (home directory, default shell, the contents of the comment field, etc) and user id to username mappings. The next module parses /etc/group for group membership information and group id to group name mappings. The third module parses /var/log/wtmp. This binary format file contains information on user logins to the system - namely the username, time and date of login, and where the login was from (local or some remote host). Each of these modules outputs a text file in a format similar to the one used for the pcap parser. Note that the files chosen are only a small subset of the forensically interesting files in the filesystem. In the future we intend to implement similar modules for many other interesting files (e.g., /var/log/messages).

B. The FACE Correlation Engine

The overall vision behind FACE is to be an extensible platform for correlating and visualizing the logical connections among objects and events discovered by various evidence-collection tools. Clearly, both the correlation and visualization aspects are open-ended problems and our goal is to build the essential framework around which ever more sophisticated correlation and visualization components can be attached. This approach will also open up the opportunity to adapt established solutions from other areas to the forensics domain.

In its first incarnation, FACE is web-based and pulls together results from a number of standalone forensics and system administration utilities to present a hyperlinked interactive report. The tool is written in Common Lisp, and uses an internal database to store and relate the various conceptual units (such as user identities, filenames, etc.) together. Currently, the tool knows how to read output from our *ramparser*, most Linux filesystems, the login log file (wtmp), the /etc/passwd file, the /etc/group file, and pre-processed pcap files. Adding support for new sources of data involves writing an input function, a display function, and correlation functions. The implementation of input and display functions can be greatly simplified by using our standard output format. Activating the new additional capabilities is done by updating the tool's configuration file; no further software integration effort is necessary.

The current version of FACE presents users with five main data views: users, groups, processes, filesystem, and network captures. Each of these main views can display a listing of all entries in that category, or a particular entry in more detail. Most fields in the detailed view are displayed as hyperlinks to related information (possibly viewed in a different tool).

The user list displays the user's name, their UID, GID (linked to the group entry), the comment field from /etc/passwd, their home directory and shell (linked to their entries in the filesystem), and a count of currently running processes for that user.

In the detailed user view, some additional information is presented, such as last login time, and a listing of all processes run by that user, and a list of all files currently opened by that user. This view is composed of information obtained by correlating data from *ramparser*, wtmp, /etc/passwd, and /etc/group. The files and processes are linked to their respective detail screens.

The groups view displays a listing of all groups, detailing the name of the group (as a link to the group's detailed entry), the GID of the group, and the primary and supplementary members of the group (as links to those member's detailed user views).

The process listing shows every process running on the system (as per *ramparser* memory analysis), giving the name, the PID of the process, the user running the process, and a count of open files, open TCP/IP sockets, and memory mappings. The detailed process view also allows the investigator to view the code segment, data segment, stack or heap of the process in a hex dump like format or as raw bytes, suitable for saving to a file for further analysis. This detailed view also shows opened files and their file descriptor ids. If the open file is a real file, it is displayed as a link to the filesystem view of that file. If it is a TCP/IP socket, it is a link to the display page of that socket. The next piece of displayed data is a listing of all TCP/IP sockets for this process. Each entry gives the inode of the socket, the source IP:port and destination IP:port pair, and a count of the number of entries in the send and receive buffers. Clicking on the inode number will present the user with a detailed view of that socket. Finally, the detailed process view shows memory mappings for the process. Mappings of actual files, opened by mmap, or as a shared library, will be displayed as links to the filesystem view of those files. In addition, hexadecimal or raw displays of code and data segments for libraries or data for regular files are available. Similarly, anonymous areas of mapped memory are identified as such and can be viewed.

The filesystem view allows the investigator to browse a filesystem associated with the investigation. Each entry in the filesystem view is addressed by the URI /files/<path to file>, with /files/ representing the root directory. An entry in the filesystem view is treated as either a directory, or a regular file. In either case, the entry will display the owner's username and group (with appropriate links), the permissions flags, the size, the modification, access and change (MAC) times, as well as a listing of all processes which currently have the file opened or mapped. If the file is a directory, the entry will also list all directory entries as links to their specific filesystem pages. If the file is anything other than a directory, the name of the file will be displayed as a link to the actual file as it exists on the filesystem. The investigator may use this link to save a copy of the file for further analysis.

The main packet view simply lists all packets found in the pcap network trace. The packets are assigned a unique id, and displayed as links to the individual packet's detailed view. The detailed view of a single packet lists the source IP and port,

the destination IP and port, the time at which the packet was captured, and allows the investigator to view the packet in both hexadecimal view and as raw bytes.

Clicking on a socket's inode will present the investigator with a detailed view of that socket. This view gives the bound port and IP (if any), the connected peer's IP and port (if any), and will give a listing of all incoming and outgoing packets associated with this socket. The investigator may view packets individually, or they may view the entire stream in hex or as raw bytes.

V. An Experiment: FACE in a Typical Investigative Scenario

A. Overview

It is not uncommon in corporate environments for malicious users or for malware to transfer sensitive documents outside of a company's perimeter. Using current forensics tools, the methods to determine who transferred a file and when rely on network traces or disk images that contain parts of the file or substantial fragments. This leaves significant doubt about who really transferred the file and the file's origin. Relying solely on network traffic is inefficient because MAC addresses can easily be cloned or modified. Similarly, simply finding a sensitive file on an employee's hard drive does not prove that the file was transferred from that machine.

We developed an experiment to test our system's ability to correlate data from multiple sources, to provide a clear investigative picture, and to give more detailed information about the actions taken by a user. Our experiment involved transferring a large file over FTP while taking a memory dump and recording network traffic. By combining memory, disk, and network information, we hoped to map the data in the network stream back to the user who created the process on the originating machine.

Our scenerio involves Ryan Acer, a network administrator for a small company who is suspected of selling trade secrets to a competitor. An outside investigative agency was secretly hired to determine if any trade secrets were being transferred outside of the company network, and if so, by whom. The investigators recorded all network traffic for a day and when they noticed an unusual spike in outbound traffic, they initiated a memory dump of Ryan Acer's workstation. Later that day, they retrieved a copy of the Ryan's Linux partition using dd, and used these three sources of data as input to FACE.

B. Experimental Setup

Two Debian 40r3-i386 VMWare images were used, one as a user workstation and the other as the server. The services of interest to us were the Apache 2.2.3 HTTP server and the vsFTPd 2.0.5 FTP daemon. We transferred a large file over FTP from the client machine to the FTP server while simultaneously downloading content from the web server. This mimics the activity of a user who is casually browsing the Internet while uploading sensitive documents to an outside location. The sensitive document for the experiment is named 'file2' and contains content that is easily recognizable as proprietary. For the experiment, after the first HTTP download is complete, the client VMWare image was suspended. Performing the experiment in this manner allowed us to have a large amount of data in both the network trace and in the FTP processes' network packet queues. This also suspends the process while the FTP file connection was still open, which allows us to look at the list of open files and match data on disk with pieces of data in the socket queues and in the network stream. Running Xorg and other non-essential graphical applications while downloading the file made the experiment more realistic and produced many more processes for our tools to analyze. Wireshark recorded all traffic up to the time of the memory dump and its output was saved to a pcap file. The client file system was copied using dd and netcat. Together this data represents the complete state of the machine at the time and its network traffic.

After the scenario was enacted, we ran *ramparser* and our pre-processing scripts on the Debian memory image, the pcap network trace and the dd image of the filesystem and then the web interface/correlation engine was started. The first step of the investigation is to discover what happened. In this scenario, the owners of the corporation believe the suspect was transferring proprietary files to an outside entity, so the first step is to look at what the user was doing. To do this, the investigator retrieves the process list using FACE. Eight processes have open network connections. Six of these are system processes and daemons, which leaves two user land processes as initial targets of investigation. The first process to be investigated is the Firefox web browser. Clicking on Firefox's entry shows that there is only one open network connection, so the investigator would look at that more closely. Upon viewing that connection's details, the investigator sees that there is relatively little network traffic, so he can quickly view the streams to see that no proprietary information was being uploaded through Firefox. Further, none of the open files appeared to be proprietary data.

After noting that the user was not likely uploading proprietary data via Firefox, the investigator turns to the next likely culprit, the FTP process. Viewing this process shows two active network connections and one of them has a large amount of associated traffic and a significant amount of data left in the send queue (see Figure 5 for a screen shot; space limitations prevent screen shots of the other information). Further, the investigator notes that FTP has as one of its open files /root/file2, the proprietary file mentioned in the scenario setup. The investigator first views the FTP command stream, and locates the command to upload file2 (STOR file2) in the outgoing network stream. The investigator then views the contents of the data stream seeing that the proprietary information is in fact being transferred over the network as file2. Finally, the investigator clicks on the entry for open file /root/file2, and verifies the access time and the contents of the file, that it is indeed the proprietary file that he should not have uploaded.

Now that the investigator knows which process was used to upload the file, he can attempt to figure out when the

file was uploaded. This can be done by simply noting the times associated with the first and last packet in the FTP data connection which was used to upload the file (2008-3-12T10:10:47 and 2008-3-12T10:11:0), and can then note that this time is about the same as the last access time of the file /root/file2 (2008-3-12T10:11:0). The final question that the investigator needs to answer is whether or not the suspect was the one who uploaded the file. The FTP process was run as the root user, whose last login was 2008-3-12T9:34:57, about 35 minutes before the file was transferred. The login was noted to be local, so the user had to have physical access to the machine; this rules out an outside attacker. The only other clue we have to the perpetrator's actual identity is the username and password pair used in the FTP process: racerx/racerx123. Because the suspect's name is Ryan Acer, this is indicative but not conclusive evidence that he may have been the culprit.

VI. Conclusion

This paper makes two main contributions–we presented a new tool for Linux memory forensics, called *ramparser*, and an integrated forensic framework called FACE. While not the first tool of its kind, *ramparser* is the most advanced to date and provides several key features many of which have not been available on Linux up to this point:

- *Process functions*: identification of running processes, loaded modules, processes' code, stack, head, code, and data segments.
- *File functions*: identification of open files, shared libraries, shared memory, anonymously mapped memory.
- *Network functions*: identification of open sockets, protocol information, as well as send and receive queues.

In standalone mode, most functions are accessed via a familiar interface–a simulation of standard Linux commands used on live systems, such as *ps*, *lsmod*, and *netstat*. The presented digital forensic framework called FACE (Forensic Automated Correlation Environment) advances the state of the art by demonstrating fully automatic correlation of disparate evidence sources. This is an effort to help the investigator by following an integrative approach to forensics which links together the different objects and events of interest and presents a higher-level view of the forensic target as a whole. This approach significantly improves the investigative process in at least two ways:

- *Automatically performs routine correlation tasks*. We demonstrated the automatic correlation of forensic data from a file system, Linux memory image, and network capture. This saves significant amounts of manual work and helps manage the growth in complexity of investigative targets.
- *Presents a logical view of the entire target computer system*. The logical, hyperlinked view presented by the system provides an intuitive means to explore relevant data and to ignore noise. Further, this system-level view provides investigators with the appropriate level of abstraction and allows them to focus on real detective work as opposed to understanding a multitude of separate tools. This also facilitates the generation of a final report and, ultimately, the presentation of evidence.

VII. Future Work

We view the presented work as a first step towards a larger contribution to the digital forensics community. The overall goal is to create a platform that both provides new capabilities (such as deep Linux memory dump analysis) and incorporates existing tools. We view three major directions in which to expand our work:

- *Improved correlation*. We will seek to generalize and expand our correlation capabilities by leveraging existing methods, including statistical ones. We recognize that it is often impossible to give a precise answer so some queries, however we are also convinced that we can apply sound scientific methods to help solve difficult forensic problems.
- *Improved visualization*. We plan to leverage decades of visualization research to help us build better visual aids for the investigator. It is our belief that the importance of forensic data visualization will increase in lockstep with the continuing surge of forensic data.
- *Improved interoperation*. We will provide standard means of data exchange and an API, which will allow others to build better tools, and will allow their transparent incorporation into FACE. This will also help utilize the multitude of tools that can provide parsing of various data sources.

References

[1] F. Adelstein, "MFP: The Mobile Forensic Platform," International Journal of Digital Evidence, Vol 1, No 1, 2003.

[2] A. Arasteh, "Forensic Memory Analysis: From Stack and Code Execution History," Proceedings of the 2007 Digital Forensic Research Workshop (DFRWS), 2007.

[3] B. Carrier, The Sleuthkit and Autopsy, http://www.sleuthkit.org/.

[4] B. Carrier and J. Grand, "Hardware-Based Memory Aquisition Procedure for Digital Investigations," Journal of Digital Investigation, Vol 1, No 1, 2004.

[5] B. Carrier, File System Forensic Analysis, Addison-Wesley, Reading, PA., 2005.

[6] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding Data Lifetime via Whole System Simulation," Proceedings of the 13th USENIX Security Symposium, August 2004.

[7] http://dfrws.org/2008/challenge/.

[8] B. Dolan-Gavitt, "The VAD Tree: A Process-Eye View of Physical Memory," Proceedings of the 2007 Digital Forensic Research Workshop (DFRWS), 2007.

[9] Encase, http://www.guidancesoftware.com.

[10] FBI RCFL Program Annual Report for Fiscal Year 2006, http://www.rcfl.gov/Downloads/Documents/RCFL_Nat_Annual06.pdf, p. 7.

[11] The Foremost File Carver, http://foremost.sourceforge.net.

[12] The Forensic Toolkit (FTK), http://www.accessdata.com.

[13] J. Kornblum, "Exploiting the Rootkit Paradox with Windows Memory Analysis," International Journal of Digital Evidence, 5(1), Fall 2006.

[14] J. Kornblum, "Using Every Part of the Buffalo in Windows Memory Analysis," Digital Investigation, January 2007.

[15] A. Boileau, "Firewire, DMA, and Windows," http://www.storm.net.nz/projects/16.

[16] M. Burdach, idetect, http://forensic.seccure.net/tools/idetect.tar.gz.

[17] knttools, http://www.gmgsystemsinc.com/knttools/.

[18] memparser, http://sourceforge.net/projects/memparser.

[19] Pyflag, http://www.pyflag.net.
[20] G. G. Richard III, V. Roussev, "Scalpel: A Frugal, High-Performance File Carver," Proceedings of the 2005 Digital Forensics Research Workshop (DFRWS 2005).
[21] J. Rutkowska,"Beyond The CPU: Defeating Hardware Based RAM Acquisition Tools (Part I: AMD case)", BlackHat DC 2007 presentation.
[22] N. Ruff, M. Suiche, "Enter Sandman (Why You Should Never Go To Sleep)", PacSec Applied Security Conference, 2007, Tokyo, Japan.
[23] B. Schatz, "BodySnatcher: Towards Reliable Volatile Memory Acquisition by Software," Proceedings of the 2007 Digital Forensic Research Workshop (DFRWS), 2007.
[24] S. Schultz, "Offline Forensic Analysis Of Microsoft Windows XP Physical Memory," Naval Post Graduate School Thesis, September 2006.
[25] A. Schuster, "Searching for Processes and Threads in Microsoft Windows Memory Dumps", Proceedings of the 2006 Digital Forensic Research Workshop (DFRWS), 2006.
[26] A. Schuster, "Pool Allocations as an Information Source in Windows Memory Forensics," International Conference on IT-Incident Management and IT-Forensics, October 2006.
[27] J. Solomon, E. Huebner, D. Bem, M. Szezynska, "User Data Persistence in Physical Memory," Digital Investigation, Vol 4, No 2, pp. 68–72, June 2007.
[28] J. M. Urrea, "An Analysis of Linux RAM Forensics," Naval Post Graduate School Thesis, March 2006.
[29] T. Vidas, "The Acquisition and Analysis of Random Access Memory," Journal of Digital Forensic Practice, Vol 1, No 4, pp. 315–323, December 2006.
[30] Volatility Framework, https://www.volatilesystems.com/default/volatility.

```
root@nssal-gpu-01:~/ramparser# ./parser debian-
rws.vmem -x
PID       UID   GID    NAME
    1     0     0      init
    2     0     0      migration/0
    3     0     0      ksoftirqd/0
    4     0     0      watchdog/0
    5     0     0      migration/1
    6     0     0      ksoftirqd/1
    7     0     0      watchdog/1
    8     0     0      events/0
    9     0     0      events/1
   10     0     0      khelper
   11     0     0      kthread
   15     0     0      kblockd/0
   16     0     0      kblockd/1
   17     0     0      kacpid
   61     0     0      kseriod
                  ...

 2395     0     0      bash
 2400     0     0      startx
 2416     0     0      xinit
 2417     0     0      Xorg
 2421     0     0      fluxbox
 2425     0    43      xterm
 2426     0     0      bash
 2515     0     0      pdflush
 2521     0     0      firefox-bin
 2548     0     0      ftp
root@nssal-gpu-01:~/ramparser#
```

Fig. 1. *ramparser* ps output.

```
root@nssal-gpu-01:~/ramparser# ./parser debian-dfrws.vmem -p 2548
08048000-08058000 r-xp /usr/bin/netkit-ftp
08058000-0805a000 rw-p /usr/bin/netkit-ftp
0805a000-08088000 rw-p [heap]
b7cd6000-b7cd7000 rw-p
b7cd7000-b7cdd000 r--s /usr/lib/gconv/gconv-modules.cache
b7cdd000-b7e05000 r--p /usr/lib/locale/locale-archive
b7e05000-b7e0d000 r-xp /lib/tls/i686/cmov/libnss_nis-2.3.6.so
b7e0d000-b7e0f000 rw-p /lib/tls/i686/cmov/libnss_nis-2.3.6.so
b7e0f000-b7e21000 r-xp /lib/tls/i686/cmov/libnsl-2.3.6.so
b7e21000-b7e23000 rw-p /lib/tls/i686/cmov/libnsl-2.3.6.so
b7e23000-b7e25000 rw-p
b7e25000-b7e2c000 r-xp /lib/tls/i686/cmov/libnss_compat-2.3.6.so
b7e2c000-b7e2e000 rw-p /lib/tls/i686/cmov/libnss_compat-2.3.6.so
b7e2e000-b7e37000 r-xp /lib/tls/i686/cmov/libnss_files-2.3.6.so
b7e37000-b7e39000 rw-p /lib/tls/i686/cmov/libnss_files-2.3.6.so
b7e39000-b7e3a000 rw-p
b7e3a000-b7e3c000 r-xp /lib/tls/i686/cmov/libdl-2.3.6.so
b7e3c000-b7e3e000 rw-p /lib/tls/i686/cmov/libdl-2.3.6.so
b7e3e000-b7f65000 r-xp /lib/tls/i686/cmov/libc-2.3.6.so
b7f65000-b7f6a000 r--p /lib/tls/i686/cmov/libc-2.3.6.so
b7f6a000-b7f6c000 rw-p /lib/tls/i686/cmov/libc-2.3.6.so
b7f6c000-b7f70000 rw-p
b7f70000-b7fa8000 r-xp /lib/libncurses.so.5.5
b7fa8000-b7fb0000 rw-p /lib/libncurses.so.5.5
b7fb0000-b7fb1000 rw-p
b7fb1000-b7fdc000 r-xp /lib/libreadline.so.5.2
b7fdc000-b7fe0000 rw-p /lib/libreadline.so.5.2
b7fe0000-b7fe1000 rw-p
b7fe2000-b7fe8000 rw-p
b7fe8000-b7fe9000 r-xp [vsdo]
b7fe9000-b7ffe000 r-xp /lib/ld-2.3.6.so
b7ffe000-b8000000 rw-p /lib/ld-2.3.6.so
bf823000-bf838000 rw-p [stack]
root@nssal-gpu-01:~/ramparser#
```

Fig. 2. FTP process mappings.

```
root@nssal-gpu-01:~/ramparser# ./parser debian-dfrws.vmem -o 2548
0 -> /dev/pts/0
1 -> /dev/pts/0
2 -> /dev/pts/0
3 -> socket:[6513]
4 -> socket:[6513]
5 -> socket:[6513]
6 -> /root/file2
8 -> socket:[6515]
root@nssal-gpu-01:~/ramparser#
```

Fig. 3. Listing of open files for FTP process.

```
root@nssal-gpu-01:~/ramparser# ./parser debian-dfrws.vmem -N | sort
Proto     Local Address           Foreign Address         State           PID   Program name
TCP       0.0.0.0:111             0.0.0.0:0               LISTEN          1959  portmap
TCP       0.0.0.0:113             0.0.0.0:0               LISTEN          2281  inetd
TCP       0.0.0.0:22              0.0.0.0:0               LISTEN          2311  sshd
TCP       0.0.0.0:60126           0.0.0.0:0               LISTEN          2332  rpc.statd
TCP       127.0.0.1:25            0.0.0.0:0               LISTEN          2270  exim4
TCP       192.168.20.128:45351    192.168.20.129:20       ESTABLISHED     2548  ftp
TCP       192.168.20.128:55071    192.168.20.129:80       ESTABLISHED     2521  firefox-bin
TCP       192.168.20.128:59447    192.168.20.129:21       ESTABLISHED     2548  ftp
TCP       192.168.20.128:59447    192.168.20.129:21       ESTABLISHED     2548  ftp
TCP       192.168.20.128:59447    192.168.20.129:21       ESTABLISHED     2548  ftp
UDP       0.0.0.0:111             0.0.0.0:0                               1959  portmap
UDP       0.0.0.0:32768           0.0.0.0:0                               2332  rpc.statd
UDP       0.0.0.0:68              0.0.0.0:0                               2301  dhclient3
UDP       0.0.0.0:812             0.0.0.0:0                               2332  rpc.statd
UNIX                                                                      2195  klogd
UNIX                                                                      2301  dhclient3
UNIX                                                                      2301  dhclient3
UNIX                                                                      2332  rpc.statd
UNIX                                                                      2375  login
UNIX                                                                      2416  xinit
UNIX                                                                      2417  Xorg
UNIX                                                                      2421  fluxbox
UNIX                                                                      2425  xterm
UNIX                                                                      2521  firefox-bin
UNIX                                                                      938   udevd
UNIX      /dev/log                                                        2189  syslogd
UNIX      /tmp/.X11-unix/X0                                               2417  Xorg
UNIX      /tmp/.X11-unix/X0                                               2417  Xorg
UNIX      /tmp/.X11-unix/X0                                               2417  Xorg
UNIX      /tmp/.X11-unix/X0                                               2417  Xorg
UNIX      /tmp/.X11-unix/X0                                               2417  Xorg
UNIX      /var/run/acpid.socket                                           2226  acpid
UNIX      /var/run/acpid.socket                                           2226  acpid
root@nssal-gpu-01:~/ramparser#
```

Fig. 4. *ramparser* netstat output.

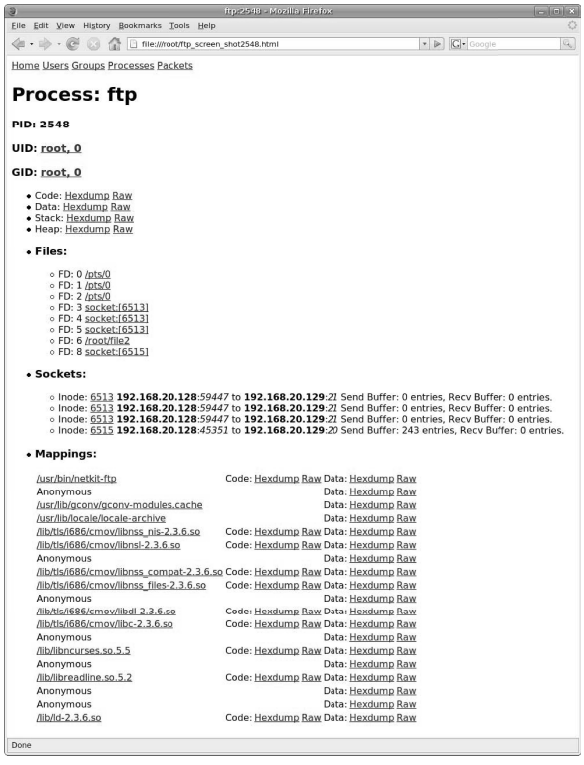

Fig. 5. FTP process information.