

Digital Forensics Research Workshop 2008 - Submission for Forensic Challenge.

M. I. Cohen, D. J. Collett, A. Walters

Introduction

The 2008 forensics challenge focuses on the merging and correlation of multiple sources of forensic evidence. In particular the analysis of network captures and memory images is performed to reconstruct the activities of the suspect.

This paper illustrates our solution to the challenge. We primarily used the PyFlag forensic package (found at <http://www.pyflag.net/>) and all source code is available there. Since this document references new features which may not be in the current stable release, we recommend readers get the latest development information as described here: <http://www.pyflag.net/cgi-bin/moin.cgi/Download>.

We start off by presenting our solutions to the primary questions of the case.

We proceed with illustrating how the PyFlag package can be used to analyse forensic evidence, such as required by the challenge. We illustrate a proposed walk-through of the application pointing out some of the interesting discoveries we made. The results are then summarised and the forensic reports produced by PyFlag are referenced. These results are in a suitable format to provide to further litigation processes.

We then proceed to discuss some of the challenges solved by the PyFlag package in relation to this particular analysis. We cover some of the techniques used such as memory analysis and network forensic analysis.

Note that many results we present involve timestamps. For the purpose of this report we will be presenting all times in the US/Eastern timezone.

Results of forensic analysis

Timeline

Below are some key times relating to the evidence:

1. kernel log boots at 2007-12-16 22:14:01 (As can be seen from the kernel message buffer - the statement **audit(1197861235.541:1): initialized** has the timestamp in unix time).
2. first connection in pcap started on 2007-12-16 22:32:16
3. last connection in pcap was at 2007-12-16 23:29:30
4. memory capture was obtained at (from internal data structures): 2007-12-16 23:33:42
5. The latest time in the zip files was 2007-12-17 14:18:05

The forensic analysis aimed to answer a number of specific questions:

What relevant user activity can be reconstructed from the data and what does it show?

- steve_vogon@hotmail.com received a confirmation email from gmail about the creation of a new gmail account (steve.vogon@gmail.com).
- Steve viewed a google spreadsheet confirming prices for the delivery of 4 files (referred to as Assets). These files were described as xls sheets containing usernames and passwords, a network diagram and an ftp pcap file. The spreadsheet had a list of prices and the phrase **OK -- ACCEPTABLE**.
- Steve communicates with faa tali - to arrange transfer of the documents.
- Steve enquires about opening a foreign bank account.
- Steve confirms with faa tali that he will deliver the items and mentions *"I plan to use your 219. location"*.
- A script is run on the machine exfiltrating a zip file within cookies. The script sends the file to the IP address 219.93.175.67 - an IP located in Kuala Lumpur Malaysia.
- The file is an archive containing 3 items - 2 spreadsheets and an network capture of an ftp connection which included credentials.
- The zip file was encrypted. We decrypted the file and examined its contents. It contained exfiltrated files with usernames and passwords and account numbers.

Is there evidence of inappropriate or suspicious activity on the system related to the user?

- The user booked a flight to costa rica for Dec 30th, returning Jan 30 2008. Travelling with a Catherine Lagrande.
- Searched for Extradition, arrests over seas and other suspicious terms.
- Searched for places to live such as the Maldives.
- Searched for a linux related vulnerabilities on hacker sites such as Metasploit.
- Used a specialised perl script to exfiltrate an encrypted zip file.
- Used the Gedit editor to wipe history files, edited a file called ELF_Exploit.sh.

Is there evidence of collaboration with an outside party? If so, what can be determined about the identity of the outside party? How was any collaboration conducted?

- Steve V was seen communicating with a person <faatali@hotmail.com> and negotiating with them about the sale of the files.
- The main way of negotiation was through sharing a google spreadsheet.

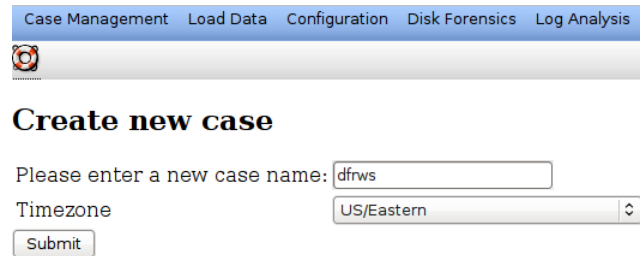
Is there evidence that sensitive data was copied? If so, what can be determined about that data and the manner of transfer?

- Potentially sensitive data was copied from the system and exfiltrated to a remote location in malaysia. The exfiltration occurred using a program which split the files into chunks and encoded each chunk inside a cookie. The cookies were sent to the remote address as a proxy connection.

DFRWS Challenge walk through

We used the PyFlag package (Version 0.87-pre1) downloaded from <http://www.pyflag.net/>. The package was installed on an Ubuntu Hardy (8.04) system according to the instructions provided on the web site (basically run *configure*, *make* and *make install*). We also included the urwid library to get interactive hex editor support (apt-get install python-urwid).

We begin by creating a new case. This will contain all relevant evidence for this investigation we call the case dfrws. We go to Case Management/Create new case:



The screenshot shows the PyFlag web interface. At the top is a navigation bar with links: Case Management, Load Data, Configuration, Disk Forensics, and Log Analysis. Below the navigation bar is a section titled "Create new case". It contains a form with the following elements: a text input field labeled "Please enter a new case name:" with the value "dfrws" entered; a dropdown menu labeled "Timezone" with "US/Eastern" selected; and a "Submit" button.

When creating the case, we must choose the case timezone. This is the timezone in which PyFlag will display all timestamps to the user. We chose "US/Eastern" as we wish to review the evidence in the local time in which the events occurred at the scene.

Next we load the DFRWS files. We may choose to load the zip file as a raw evidence, or alternatively unzip the zip file and load the partition itself. We chose to load the zip file as a raw image. For this image we chose the Standard IO Subsystem (other options include EWF (Eye Witness Format), or AFF (Advanced Forensic Format) files).

We also need to set the timezone of the image. This timezone is called the *evidence* timezone and must be set to the local timezone of where the image was acquired, it can be different to the *case* timezone we chose when creating the case. This allows us to load multiple pieces of evidence from different timezones and use a common timezone for the analysis phase allowing easy event correlation. It is especially important to choose the correct *evidence* timezone in this case because zip files store only localtime and not UTC. If we were to choose the wrong timezone here, we would not be able to compare zip timestamps with timestamps from the network capture or memory image. Time consideration issues are discussed further later in the report.

Since there are no partitions in a zip file, we set the partition offset to 0. We choose to name this source "z":

Case Management Load Data Configuration Disk Forensics Key

Load IO Data Source

Select IO Subsystem: Standard

Evidence Timezone: US/Eastern

Currently Selected files: dfrws2008-challenge.zip

Select Standard image:

Enter partition offset: 0

Unique Data Load ID: z

Submit

Next we are taken to a screen which allows us to load the filesystem:

Case Management Load Data Configuration Disk Forensics Keyword I

Load Filesystem image

Case: dfrws

Select IO Data Source: z

Magic identifies this file as: Zip archive data, at least v1.0 to extract

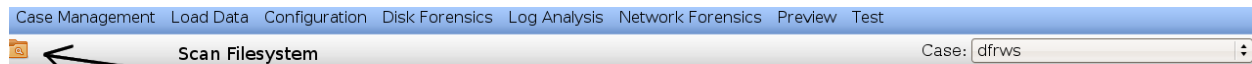
Enter Filesystem type: Raw

VFS Mount Point: /

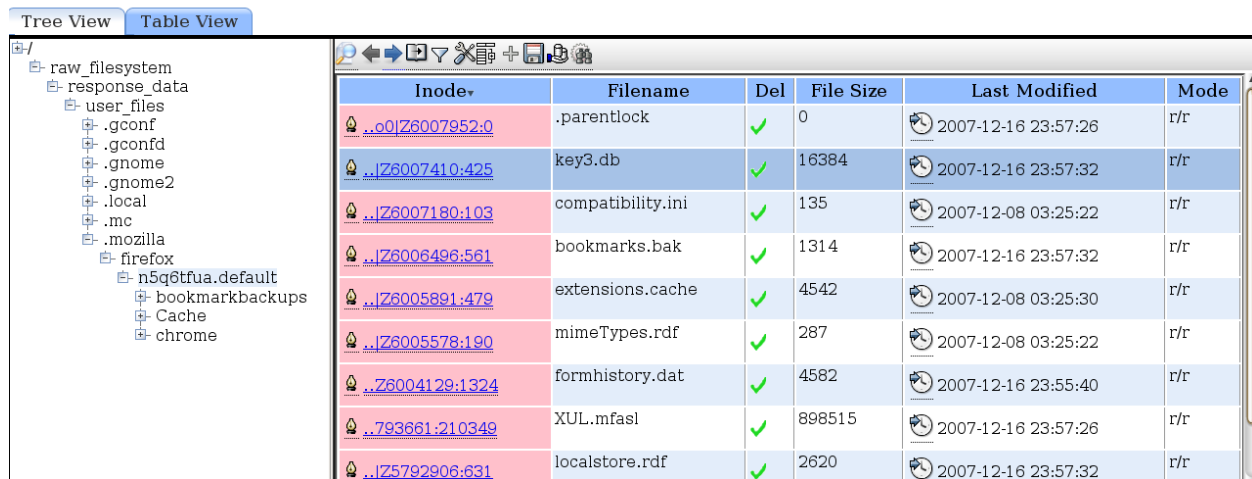
Submit

PyFlag currently has support for a number of filesystem drivers as shown above. Sleuthkit is the sleuthkit filesystem driver (which supports ntfs, ext, ffs among many more). PyFlag runs the magic command on the image and tells us that it was identified as a Zip file. We load the file as "Raw" in this case and select to mount it on the VFS at the root location ("/"). The Raw format tells pyflag to just insert the raw file into the virtual filesystem (VFS). No further analysis is performed.

We now want to scan the VFS to discover files contained within the VFS (and essentially to unzip the package file). We select the "Scan Filesystem" icon. This allows us to run scanners (Specialized pieces of software which discover files within the Virtual File System - VFS). Since we essentially want to unzip the distribution file we select "Compressed File Scanners". These option employs scanners which target compressed files, such as Zip, Gzip and Tar files. The VFS is scanned recursively and we are taken to the standard "Browse Filesystem View".



Browsing Virtual Filesystem



This view presents the VFS as a tree view. The VFS is the arena in which we do all our analysis. Objects are loaded directly into the VFS or discovered by scanners during the course of the analysis and are represented in the VFS as "Inodes". Like in a real filesystem, the VFS Inodes also have paths (and filenames), but internally PyFlag accesses all object via their Inodes. Unlike a real filesystem, PyFlag's Inodes are represented as strings. More on that later.

We have just finished loading the disk forensic component into PyFlag, but before we start analysing the case, we want to load the remaining components.

The next source of evidence is the pcap file. PyFlag treats each source of evidence as a unique "IO Source". We select the "Load IO Data Source" menu option. This time however, we select the "Load from VFS" option since the pcap file is already in the VFS. We see that the filename is of the form "vfs://dfrws/....". This is a url like reference for the pcap file (i.e. its taken from the dfrws case VFS). We name this IO source "n" to remind us its the network component.

Load IO Data Source

Select IO Subsystem:

Evidence Timezone:

Currently Selected files:

Select Standard image:

Enter partition offset:

Unique Data Load ID:

When we select "Submit" PyFlag recognises that this is a PCAP file, and therefore automatically offers to load it with the "PCAP Filesystem" driver. This driver perform the initial analysis of the pcap file and creates VFS entries for stream objects. We will load the network traffic under the "/net/" VFS mount point. This will ensure that all objects discovered during network analysis will be presented separately within the VFS directory structure. Note, however, that all over analysis will be performed on network objects, disk

objects and memory objects simultaneously. PyFlag does not see a distinction with the source of the VFS inodes.

Load Filesystem image

Case:

Select IO Data Source

Magic identifies this file as: PCAP tcpdump file
Selecting PCAP Virtual Filesystem automatically

VFS Mount Point:

The File system will then be loaded under the /net/ mount point. Note that all the PCAP loader did was reassemble the streams and place inodes for the forward and reverse streams in a directory structure:

Case:

Browsing Virtual Filesystem

Tree View Table View

- net
 - streams
 - 2007-12-17
 - 192.168.151.130-12.130.81.249
 - 49998:80
 - 192.168.151.130-12.187.16.75
 - 192.168.151.130-128.241.21.14
 - 192.168.151.130-151.193.207.59
 - 192.168.151.130-151.193.224.81
 - 192.168.151.130-151.193.224.82

Inode	Filename	Del	File Size	Last Modified	Mode
In[S2479]	reverse	✓	2304	2007-12-17 13:32:57	r/r
In[S2478]	forward	✓	2138	2007-12-17 13:32:57	r/r

The PCAP file driver VFS inodes reflect the network streams. For example the Last modified time contains the time of the first packet seen. We now run the network scanners on the PCAP file to discover the different protocols found within the streams. We select the "Scan FS" button again, except this time we only scan the VFS inodes under the "/net/" directory:

Scan Filesystem

Case:

Scan files (glob or path):

- ☒ Network Scanners
- ☒ File Type Related Scanners
- ☒ Compressed file support
- ☒ General Forensics
- ☒ File Carvers

- Collect information about HTTP Transactions.
- A Yahoo IM Protocol scanner
- Collect information about MSN Instant messenger traffic
- Detect Gmail web mail sessions
- A scanner for google docs related pages
- Detect YahooMail Sessions
- Detects Live.com/Hotmail web mail sessions
- Detect SquirrelMail Sessions

Clicking on the "configure" icon allows us to select which scanners to enable or disable. Usually those scanners which have a tendency to produce many false positives or take too long to run are disabled by default. If we disable the entire scan group, all scanners within that group are disabled. In this case we choose to enable all scanners, except for the carver group (Which will be applied more selectively).

Scanning path /net/

68 jobs pending (all cases)

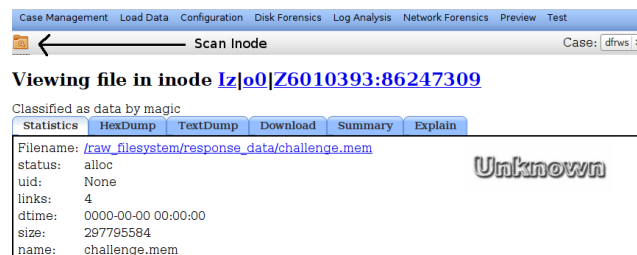
The following scanners are used: ['TypeScan', 'HTTPScanner', 'YahooScanner', 'MSNScanner', 'GmailScanner', 'GoogleDocs', 'YahooMailScan', 'HotmailScanner', 'SquirrelMailScan', 'RegistryScan', 'OLEScan', 'IIndex', 'RFC2822', 'MozCacheScan', 'MozHistScan', 'PstScan', 'DLLScan', 'MD5Scan', 'IndexScan']

System messages:

```
2008-07-06 23:11:28(7): Scanning file /net/HTTP/2007-12-17/_log?src=dis&a=5 (inode In|S2706/2707|H1839:1)
2008-07-06 23:11:28(7): Openning In|S2696/2697 for HTTP
2008-07-06 23:11:29(7): Openning In|S2694/2695 for HTTP
2008-07-06 23:11:29(7): Openning In|S2684/2685 for HTTP
2008-07-06 23:11:29(7): Openning In|S2758/2759 for HTTP
2008-07-06 23:11:29(7): Openning In|S2680/2681 for HTTP
2008-07-06 23:11:29(7): Openning In|S2756/2757 for HTTP
2008-07-06 23:11:29(7): Openning In|S2664/2665 for HTTP
```

While the scan is progressing, PyFlag keeps us informed of the progress. We see what scanners were invoked and how many jobs remain. PyFlag has a distributed architecture. This means that jobs (such as scanning) are assigned to worker processes which perform these jobs independently. This is a very scalable approach with processes being able to run on the same machine (utilising SMP) or on other machines in a cluster. We also see Scanners reporting which Inodes were selected by them (above we can see the HTTP Scanner reporting all the inodes of interest to it). All scanners see all inodes, but only operate on those inodes which are of interest to it (This allows multiple different scanners to operate on the same inode).

Next we want to carve the memory image. Since carving typically produces a lot of false positives we prefer to use it sparingly and be more targeted (Carvers are switched off by default). We therefore choose only to carve the memory image. We click on its inode in the tree view, and are presented in the "View Inode" screen. This screen has multiple tabs which show us information about the selected inode. Many of these features will be explored later. Click the Scan Inode icon to bring up the scanner selection window for only this inode.



From the scanner selection windows, enable the File Carvers scanner group and submit.

Finally, we can load the memory image directly from the vfs as a new iosource and use the 'Linux Memory' filesystem to mount the extracted data into the vfs. PyFlag integrates the excellent memory analysis package Volatility. As of Volatility 1.3 there is support for linux memory images through the use of *Profiles* and *Maps*. Profiles are templates representing kernel data structures specific to a version of the Linux kernel. While maps are symbol locations resulting from a specific compilation instance of the kernel. PyFlag allows users to select the symbols specific to their kernel version.

Load Filesystem image

Case:

Select IO Data Source

Magic identifies this file as: data

Enter Filesystem type

VFS Mount Point:

Profile

Symbol Map

In the above screenshot we see that the Linux Memory driver requires a Profile and Symbol Map which match the exactly version of our kernel. If we are using a standard kernel (i.e. one provided by a vendor) we can obtain the map from the kernel package.

The evidence is now loaded and analysed in PyFlag and the investigation process can begin by examining PyFlag's various reports found in the menus.

Webmail

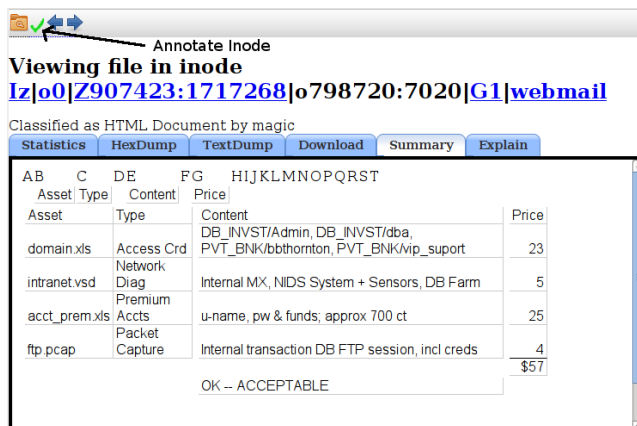
We will begin with the "Browse Webmail Messages" report.

Timestamp	Inode	From	To	CC	BCC	Subject
2007-12-17 14:36:47	006031b9e0maul	faa tali	None	None	None	Delivery information
2007-12-17 14:36:47	006031b9e0maul	faa tali	None	None	None	Re: Negotiate (Google Docs)

This table allows us to see, at a glance, all webmail messages found in the evidence by the scanners. These messages may have been recovered from http sessions found in the network capture, or browser cache found in the user files. We will see later how we can determine the exact source of the messages. The table is paged and can be navigated with the arrow icons in the toolbar. To work out the total number of rows in a table, use the Count Rows icon as shown.

We can examine the messages in more detail by clicking the Browse Inodes icon as shown. This will allow us to quickly page through all Inodes in the present table (considering any filtering etc) by presenting navigation icons on the toolbar. By default we are taken to the Summary tab which interprets the Inode and displays it in an easy to view format, for example it will display images or html directly in the browser. If we find any inodes that are of interest we can mark them as relevant by using the annotation icon (tick icon) in the

toolbar.



We will tick all of the webmails found except any which appear to be duplicates or drafts of more complete emails, or simply inbox displays which do not contain any data of interest. We will see later how we can export these interesting Inodes.

Gmail characteristics

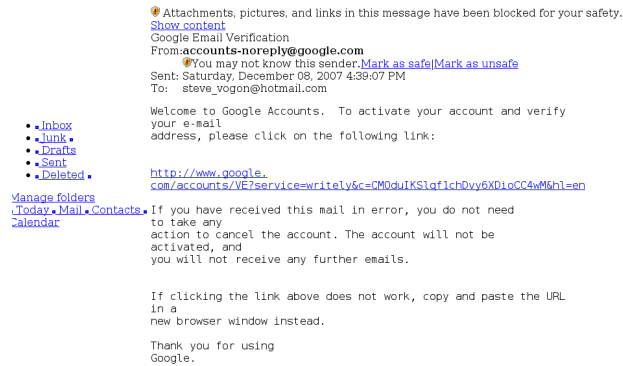
Gmail has some interesting characteristics which are worth discussing here. Consider the following snippet from the webmail table:

Case Management Load Data Configuration Disk Forensics Log Analysis Memory Forensics Network Forensics Preview Test					
Case dfnws					
2007-12-17 13:35:59	Minimum account opening balance	Hello, Can you please tell me what the minimum balance requirement is for opening an overseas account at your bank?	Edit Sent	Gmail	
2007-12-17 13:36:25	Minimum account opening balance	Hello, Can you please tell me what the minimum balance requirement is for opening an overseas account at your bank? Thank you, Steve K. Vogan	Edit Sent	Gmail	
2007-12-17 13:36:40	Minimum account opening balance	Hello, Can you please tell me what the minimum balance requirement is for opening an overseas account at your bank? Thank you, Steve K. Vogan	Read	Gmail	
2007-12-17 13:36:47	Delivery information	Hi, I will be delivering the items as we discussed. I plan to use your 219. location for the items. Please check for them within the next few hours and then arrange for a reciprocal transfer. There is one item I will keep in reserve subject to your follow-through. It's noted on the spreadsheet.	Read	Gmail	

We see that there are 3 requests to gmail, the first 2 are marked as Edit Sent (This means the browser has sent the information to the server while editing). The first request is a period "save draft" functionality which gmail performs while the user types their message. The second *Edit Sent* event is the user pressing submit. The final event is a *Read* event which happens on gmail after sending - the gmail application immediately reads the new message and displays it at the bottom of the current thread.

Hotmail

The first 2 messages in the web mail table are hotmail messages. Here is an example of how PyFlag renders them:



Attachments, pictures, and links in this message have been blocked for your safety.
[Show content](#)
Google Email Verification
From: accounts-noreply@google.com
You may not know this sender. [Mark as safe](#) | [Mark as unsafe](#)
Sent: Saturday, December 08, 2007 4:39:07 PM
To: steve_yogon@hotmail.com

Welcome to Google Accounts. To activate your account and verify your e-mail address, please click on the following link:
<http://www.google.com/accounts/VE?service=writely&c=CM0duIKS1qfchDvy68DioCC4wM6hl=en>

If you have received this mail in error, you do not need to take any action to cancel the account. The account will not be activated, and you will not receive any further emails.

If clicking the link above does not work, copy and paste the URL in a new browser window instead.

Thank you for using Google.

Navigation links on the left: [Inbox](#), [Junk](#), [Drafts](#), [Sent](#), [Deleted](#), [Manage folders](#), [Today](#), [Mail](#), [Contacts](#), [Calendar](#)

The page does not look anything like what a normal hotmail page looks like. This page was found in the mozilla cache and is quite old. The images and style sheets that are required to render it are no longer cached by mozilla. To aid in presenting the evidence as it appeared at the time, PyFlag has a concept of http sundry objects, described below.

HTTP Sundry

While displaying reconstructed web pages, you may notice that some pages do not render correctly. For example consider the msn page:



Viewing file in inode [In\[S1949/1950\]H1355:39253](#)
Classified as HTML Document by magic

Navigation tabs: [Statistics](#), [HexDump](#), [TextDump](#), [Download](#), [Summary](#), [Explain](#), [Show Packets](#), [Combined stream](#), [HTTP](#)

Sunday, December 16, 2007
[Get a free upgrade for Windows](#)

- [Español](#)
- [RSS](#)

- [Web](#)
- [MSN](#)
- [Images](#)
- [News](#)
- [Maps](#)
- [Shopping](#)
- [Video](#)

Search the Web
Search:

Popular Searches:

- [Chipmunks](#)
- [Nintendo rain check](#)
- [M. J. Morrison](#)

Clearly some images and style sheets are missing. We need to see whats missing and make a decision about the appropriateness of fetching those items directly from the msn servers. We can use the http_sundry table to list the missing objects. PyFlag has a helper to do this:

```
~/pyflag$ python utilities/http_sundry_loader.py --case dfrws -l "msn"
http://stc.msn.com/br/hp/en-us/css/44/ushpw.css
http://stc.msn.com/br/hp/en-us/css/44/ovrW.css  <---- Important style sheets
....
```

```

http://search.msn.com/partner/primedns.gif
http://stb.msn.com/i/B6/32E46DE281A68B9C33FC582D2569D.gif
http://stb.msn.com/i/48/86F1396496DFE1BAD68AB5F28409.gif
http://stb.msn.com/i/48/6CDE404B4BFEC334D023E5422081E0.gif <---- Important
Images
http://stb.msn.com/i/78/7CE57843948D6DF13E79A2DE4E15C.gif
http://stb.msn.com/i/62/60895A7A2C208D211512E82EAAC.jpg

```

We then need to fetch those objects from the internet (This should be pretty safe since the suspect does not own any of these servers). We prepare a download script which can be taken offline (Our analysis system is not connected to the internet):

```

~/pyflag$ python utilities/http_sundry_loader.py --case dfrws "msn" -o
Written download script to: msn.py

This can be done on another system -
~/pyflag$ python msn.py msn.zip
Downloading: http://stc.msn.com/br/hp/en-us/css/44/ushpw.css
Downloading: http://stc.msn.com/br/hp/en-us/css/44/ovrW.css
....
Downloading: http://www.msn.com/rss/msnentertainment.aspx
Downloading: http://www.msn.com/rss/msnmoney.aspx
Data saved into msn.zip, import using: http_sundry_loader.py --case casename --load
msn.zip

```

Note that the created script is purpose built for fetching those exact URLs and uses only modules in the standard python installation (i.e. no need for PyFlag to be installed on the internet machine). The internet machine can be running any OS supported by python (including windows). The zip file created contains all the URLs fetched (encoded in base64) and can be loaded into the sundry table using the command given.

Here is the same page using http sundry objects to supplement the evidence for rendering:



Unfortunately in some cases, such as the hotmail page shown previously, the images and css are no longer hosted on the server due to page changes. This is because we did not download the sundry objects soon enough after the incident. A lesson to learn from this is that a decision of whether we should use sundry objects in our rendering should be made as

soon as possible to minimize the possibility of objects being removed from their respective web sites.

HTTP Requests

We will now look at the "Browse HTTP Requests" report. This table shows all http requests found in the evidence. Again, these may have come from either the network capture or the browser cache. Clicking on any inode will launch the View File report to the Summary tab and display the reconstructed webpage, rendering all resources (images, css) if they can be found anywhere in the evidence or the sundry as loaded earlier.

We examine all the urls found in the case (either in cache or http traffic) by selecting Network Forensics/Browse HTTP requests:

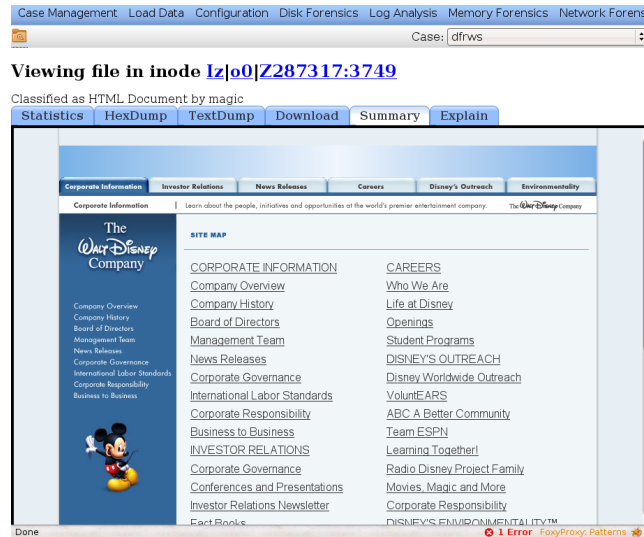
Requested URLs

Timestamp	Request Packet	Inode	Method	URL	Content Type
2007-12-17 13:52:50	6051	../1864 H1945:0	GET	http://www.amazon.com/Juicy-Fruit-Mtume/dp/B0000025UL	text/html; charset=ISO-8859-1
2007-12-17 13:54:05	6462	../1866 H2125:0	GET	http://www.facebook.com/	text/html; charset=utf-8
2007-12-17 13:54:28	6482	../68 H2239:7727	GET	http://www.live.com/	text/html; charset=utf-8
2007-12-17 13:54:59	6507	../0 H2562:42635	GET	http://search.live.com/results.aspx?q=hurricane	text/html; charset=utf-8
2007-12-17 13:55:43	None	../12287317:3749	GET	http://corporate.disney.go.com/corporate/sitemap.html	text/html
2007-12-17 13:56:07	6710	../2 H1380:19679	GET	http://corporate.disney.go.com/corporate/sitemap.html	text/html

There are a number of interesting observations we can make from the above screenshot:

- We can see the content Type of the document on the right most column. We are typically interested in documents of type "text/html" because usually images are embedded within those. In our case we can filter the table on the filter expression **"Content Type" contains html**
- As can be seen some entries have a valid *"Request Packet"* field, while some of requests have **None** in that field. This is because some of the requests come from the network traffic, while others come from cache. Note that other than this the two sources are treated exactly the same internally.
- The last two rows shown in the figure above are actually the same request - one comes from the network capture, while the other is cached in the Mozilla cache. The two rows are identical except for their time stamps, which are almost the same.

We can view any of the HTML pages. PyFlag tries to render the page in such as way that it looks as close as possible to how the page looked on the original system:

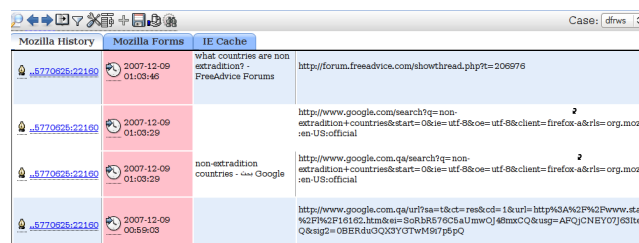


The page is parsed and any embedded images or style sheets are fetched from within the VFS or the http sundry objects if available. The above image was derived purely from the Mozilla cache.

Just as with the webmail example, we can mark any notable http sessions for inclusion in the case report.

Browser History

In addition to the Firefox cache which can be browsed and reconstructed natively in PyFlag, we can also examine Firefox's browser history and saved form fields using the Browser History report. Here we can see many of the same requests found in both the cache and network capture. The timestamps recorded in the history are approx 1-2 seconds earlier than those seen in the Cache.



We can also see the browsers saved form fields which Firefox stores in the same 'mork' file format as the history file. Here we can see the searchbar history and many form entries from a flight booking form. It appears that Steve is travelling with a companion, Catherine Lagrande.

Mozilla History	Mozilla Forms	IE Cache
..Z6004129-1324	leavingDate	12/30/2007
..Z6004129-1324	lastNameForFlight2	Lagrande
..Z6004129-1324	lastNameForFlight1	Vogon
..Z6004129-1324	LastName	V
..Z6004129-1324	IdentityAnswer	binky
..Z6004129-1324	goingTo	Costa Rica
..Z6004129-1324	firstNameForFlight2	Catherine
..Z6004129-1324	firstNameForFlight1	Steve

Internationalization

It is important for a forensic application to have good support for unicode and internationalization technologies. PyFlag is no different, with full unicode support. We see that the suspect was concerned about extradition and searched for information about this topic using the Arabic version of google:

http://127.0.0.1:8000/post

Viewing file in inode [lz|o0|Z907423:1717268|o1069056:7386|G1](#)

Classified as HTML Document by magic

Statistics HexDump TextDump Download Summary Explain

Steve.Vogon@gmail.com

Google

extradition costa rica

البحث

الوقت المستغرق: 0.23

النتائج من 1 إلى 10 من حوالي 124,000

ارتباطات داعية

[Law](#)
Ajman University of
Science &
Technology: Essence of
Empowerment
www.ajman.ac.ae

[Do you hate filing?](#)
Find anything in your
office
in 5 seconds or less - Free
Trial!
www.thepapertiger.com

[Quirns & Asociados](#)
Prestigious and well
known Central
American and Caribbean

[FBI Seeks Extradition?](#)
Do Not Want To Return? Fight FBI Charges
Anywhere in World
[Costa Rica Attorneys](#)
Full Service Law Firm.Harvard Law School Graduates.Leading Firm
in CR

U.S.-Costa Rica Extradition Treaty

[ترجم هذه الصفحة]
Title: Extradition Treaty Between the Government of the United States of America and
the Government of the Republic of **Costa Rica**
[الرئيسية](#) - [معلومات](#) - [معلومات](#) - [sarijose.usembassy.gov/exttreatyeng.html](#) - 48k

Moving to Costa Rica - Living & Working in Costa Rica

[ترجم هذه الصفحة]
Moving to & Living in **Costa Rica** . Moving to & Living in **Costa Rica** - Find out ... (Can
the country you go to **extradite** you back to the country you left? ...

Carved Memory Image

Although PyFlag is able to analyse memory images, often data is present in the memory image from processes which have exited. In that case standard memory analysis techniques are unable to find these processes, and we need to resort to carving. Previously we ran the carvers on the memory image and now we examine the found inodes:

able View

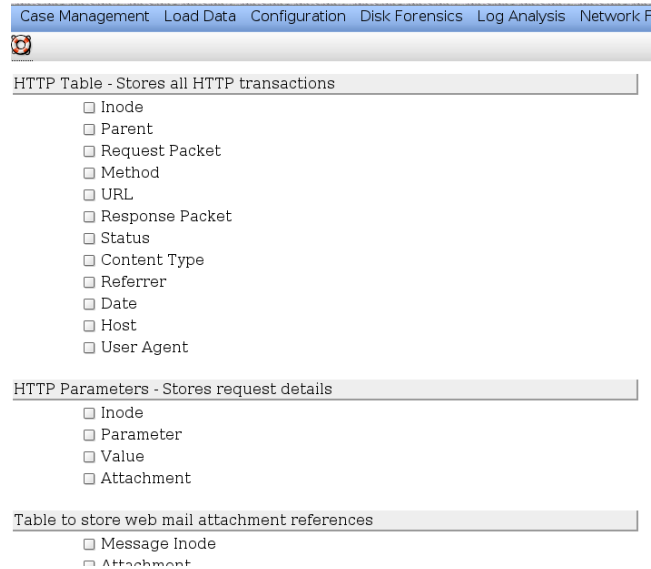
File Name	Offset	Size	Type
..258019328:3209	258019328.pl	3209	✓
..65904128:2620	265904128.sh	2620	✓
..2924672:11644	272924672.sh	11644	✓
..72936960:1154	272936960.sh	1154	✓
..278110208:662	278110208.sh	662	✓
..81477120:4407	281477120.sh	4407	✓
..283508736:735	283508736.sh	735	✓
..5220864:16299	285220864.sh	16299	✓
..87666176:1273	287666176.sh	1273	✓

As can be seen in the screen shot above, a perl script was discovered at offset 258019328. The script is included in the brief as `file:///dfrws/inodes/1516`. Brushing up on our PERL skills we realise that this is a script used to exfiltrate a file through the use of cookies:

- Lines 10 - 30 is a list of URLs to use for exfiltration
- The chunker subroutine splits a string into an array of chunks. The chunks are appended to parameters which can be "RMID", "Sessid" or "CVal".
- The ship_data routine shells out to wget using a proxy provided in the environment variable. The chunk is placed in the cookie field of the header. Note that there is a bug in the script in that it does not properly escape shell characters when passing to the system call - a common PERL vulnerability. The effect of this is that urls containing & character will be truncated.

PyFlag Generic Tables

[PyFlag](#) stores a great deal of information about the different files it discovers (In pyflag terminology they are called Inodes). The "Browse HTTP Requests" Menu option shows a table with a number of those properties together (namely Timestamp, Inode, URL and content type). Often we would like to display other information together so we can apply more effective filtering. This can be done using the "Generic Table" menu option.



We would like to see if the traffic in the capture exhibits the characteristics the script provides. The script uses a custom proxy to send requests to many different web sites through the same IP address.

We select the "Disk Forensics/Generic Report" option, which presents a list of all inode properties, distributed in various database tables. This report will automatically combine different fields into the same table.

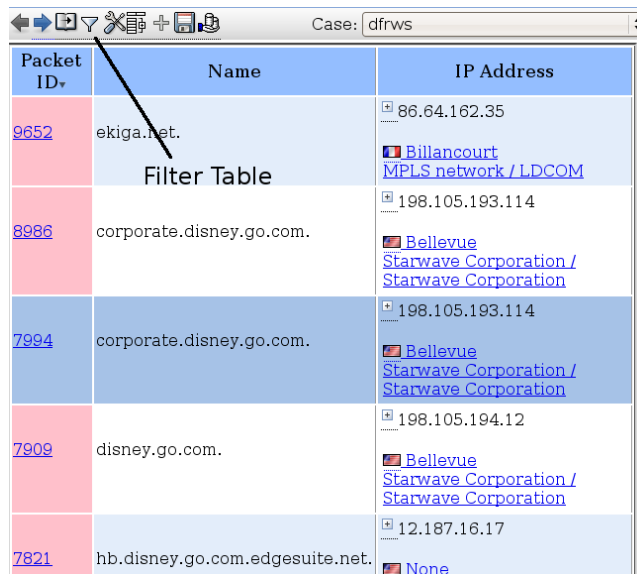
We select the Modified time, Destination IP and URL:

Modified	Destination IP	URL
2007-12-17 14:29:30	219.93.175.67 Kuala Lumpur TMNET / TMNET TELEKOM MALAYSIA	http://www.msn.com/
2007-12-17 14:25:31	219.93.175.67 Kuala Lumpur TMNET / TMNET TELEKOM MALAYSIA	http://en.wikipedia.org/wiki/Lee_Smith_%28baseball_player%29
2007-12-17 14:24:22	86.64.162.35 Bilancourt MPLS network / LDCOM	http://ekiga.net/ip/
2007-12-17 14:21:08	219.93.175.67 Kuala Lumpur TMNET / TMNET TELEKOM MALAYSIA	http://en.wikipedia.org/wiki/Lee_Smith_%28baseball_player%29
2007-12-17 14:18:48	219.93.175.67 Kuala Lumpur TMNET / TMNET TELEKOM MALAYSIA	http://en.wikipedia.org/wiki/Main_Page

Our copy of PyFlag is loaded with the GeoLocation database available from **maxmind.com**. PyFlag automatically resolves IP addresses using the database and indicates the location and ISP in charge of the IP address (In fact we can even search based on ISP or country). As can be seen many URLs are fetched from apparently the same IP address. Sites like msn, wikipedia and yahoo are all obtained from 219.93.175.67 - an IP address located in Kuala Lumpur, Malaysia. This seems inconsistent and suspicious. We deduct that the Malaysian IP address is the proxy used in the above script

DNS Analysis

Normally, when a user issues a request for a web page, the users browser will try to resolve the name to an address by a DNS resolution. Its is often instructive to see the DNS resolutions because that might hint to the originators intended server. PyFlag process all DNS packets in the capture into a database table which allows us to see what the IP addresses resolved to at the time the capture was taken (Of course DNS records may be changed quickly so there is no guarantee that new DNS lookups made during the time of the analysis were the same as when the capture was taken). We select "Network Forensics/ Browse DNS" report:



Packet ID	Name	IP Address
9652	ekiga.net.	86.64.162.35 Billancourt MPLS network / LDCOM
8986	corporate.disney.go.com.	198.105.193.114 Bellevue Starwave Corporation / Starwave Corporation
7994	corporate.disney.go.com.	198.105.193.114 Bellevue Starwave Corporation / Starwave Corporation
7909	disney.go.com.	198.105.194.12 Bellevue Starwave Corporation / Starwave Corporation
7821	hb.disney.go.com.edgesuite.net.	12.187.16.17 None

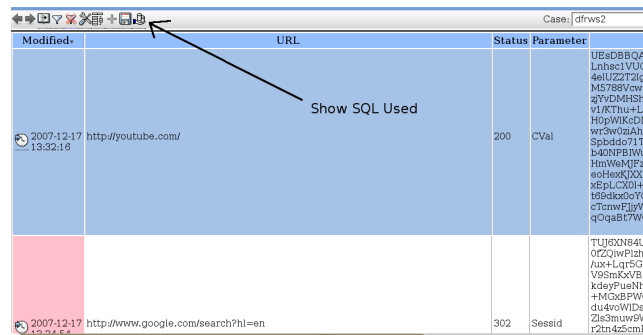
We want to see if the IP address in question was looked up by name - this might give us more information about it. We can filter the table by clicking on the "Filter Table" button. This requires we put in a filter expression. We type the expression **"IP Address" = 219.93.175.67** and see that there are no matches. This indicates that in all likelihood the connection was made without a DNS lookup, most likely by IP address.

Viewing Cookies

PyFlag actually decodes all cookies and HTTP parameters when parsing HTTP traffic. These are kept in the `http_parameters` table. Normally, however there is no pre-canned report presenting this information. We can use the "Generic Report" feature to make up our own report showing the important fields for our investigation.

In our case we select the following fields from the generic report - Modified, URL, Status, Parameter and Value. In our case we apply the following filter:

```
"Parameter" = RMID or "Parameter" = CVal or "Parameter" = Sessid
```



Case: dfrws2

Modified	URL	Status	Parameter
2007-12-17 13:32:18	http://youtube.com/	200	CVal
2007-12-17 13:32:18	http://www.google.com/search?hl=en	302	Sessid

In order to extract the exfiltrated attachment we need to write our own custom script. We need to obtain the SQL used to generate the table by clicking on the "SQL used" toolbar icon. We then write a simple python script to reverse the encoding:

```

1 import pyflag.DB as DB
2
3 sql = """
4     select `inode`.`mtime` as `Modified`,
5     `http`.`url` as `URL`, `http`.`status` as `Status`,
6     `http_parameters`.`key` as `Parameter`,
7     `http_parameters`.`value` as `Value`
8     from inode join `http` on `inode`.`inode_id` = `http`.`inode_id`
9     join `http_parameters` on `inode`.`inode_id` = `http_parameters`.`inode_id`
10    where ((1) and (`http_parameters`.`key` = 'RMID'
11        or `http_parameters`.`key` = 'CVal'
12        or `http_parameters`.`key` = 'Sessid'))
13    order by `inode`.`mtime` asc;
14 """
15 dbh = DB.DBO("dfrws")
16 dbh.execute(sql)
17 last = ''
18 result=''
19 for row in dbh:
20     if row['Value'] != last:
21         last = row['Value']
22         result += last
23
24 open("/tmp/output.zip", "w").write(result.decode("base64"))

```

The SQL statement was obtained from the GUI itself (by clicking the show SQL icon), we just execute it and iterate over all rows. We need to take into account that some rows are generated due to a redirect, causing the client to reissue the request to another URL, with the same chunk.

Then we just open "/tmp/output.zip" for writing and decode the result into it. The file is included in the attached PyFlag brief.

The result is a zip file which can be listed by unzip:

```
~/dfrws2008$ unzip -l output.zip
Archive:  output.zip
  Length      Date    Time    Name
  -----
  141824  12-08-07  23:19    mnt/hgfs/Admin_share/acct_prem.xls
  100864  12-08-07  23:09    mnt/hgfs/Admin_share/domain.xls
   2395   08-06-00  00:54    mnt/hgfs/Admin_share/ftp.pcap
  -----
  245083                      3 filesAna
```

Unfortunately the archive is encrypted. We use the fcrackzip (<http://www.goof.com/pcg/marc/fcrackzip.html>) program to crack the zip archive using a dictionary:

```
~/dfrws2008$ fcrackzip -D -p /usr/share/dict/words /tmp/output.zip
possible pw found: rhubarb ()
```

Analysis of the zip file

The collected files were analysed using standard forensic analysis. First we looked at the extracted metadata using PyFlag's command line OLE metadata extractor:

```
pyflag$ python src/FileFormats/OLE2.py acct_prem.xls
Author: Matthew Geiger
Revnumber: 1
Total_edittime: 1970/01/01 10:00:00
Lastprinted: Invalid Timestamp E8D60800:29
Created: 2007/12/08 23:12:33
Lastsaved: Invalid Timestamp E8D60800:29
```

It seems that the document was originally created by someone else named Matthew Geiger. We recommend he be investigated as a possible accomplice. The timestamps and fields present in the file are consistent with the file being created by Open Office. We then opened the file in OpenOffice and found a spreadsheet with the headings Username, PW, Balance. Although these suggest that they describe a account numbers and passwords, we were surprised to find that the account balance was filled with `RAND()*100000`, that is a function which generates a random number. Its obviously bogus and might indicate the Steve V was fabricating the items for sale, tricking the buyers.

The second xls file does not contain a name, but the creation time is similar to the other file:

```
pyflag$ python src/FileFormats/OLE2.py domain.xls
Revnumber: 0
Total_edittime: 1970/01/01 10:00:00
Lastprinted: Invalid Timestamp E8D60800:29
```

Created: 2007/12/08 23:09:41
Lastsaved: Invalid Timestamp E8D60800:29

The files were found in a directory **mnt/hgfs/Admin_share/** hinting that they possibly were retrieved from a remote share.

File analysis

This section discusses the different pieces of evidence which may be gathered from the user files.

The Gedit history file contains:

```
<?xml version="1.0"?>
<metadata>
  <document uri="file:///home/stevev/.recently-used.xbel"
atime="1197867496">
    <entry key="position" value="0"/>
  </document>
  <document uri="file:///home/stevev/.config/gtk-2.0/gtkfilechooser"
atime="1197867370">
    <entry key="position" value="0"/>
  </document>
  <document uri="file:///home/stevev/.lessht" atime="1197867444">
    <entry key="position" value="0"/>
  </document>
  <document uri="file:///home/stevev/.bashrc" atime="1197181011">
    <entry key="position" value="0"/>
  </document>
  <document uri="file:///home/stevev/temp/ELF%20exploit.sh"
atime="1197181503">
    <entry key="position" value="3522"/>
  </document>
  <document uri="file:///home/stevev/.bash_history"
atime="1197867429">
    <entry key="position" value="0"/>
  </document>
</metadata>
```

The a time entries are stored in unix epoch time, converted to US/Eastern below:

- The .bash_history file was edited on 2007-12-16 23:57:09
- A file called ELF exploit.sh was edited in steves temp directory on 2007-12-09 01:25:03
- The less history file was edited on 2007-12-16 23:57:24

The concerning this about this is that the history files were modified after the memory dump was obtained, and the user files were obtained quite a significant time later. This may indicate that evidence was modified - we recommend the acquisition process be reviewed.

The midnight commander history file is very interesting (.mc/history). Midnight commander is a text mode file manager application - the manager has two panes - a left pane and a

right pane, both can be navigated independently. The history file keeps valuable information:

- The last two directories created were DFRWS, retrieved_files - these seem to match the zip file we were given so were probably done by the incident response team.
- The [Dir Hist New Left Panel] history category shows the directories the left pane visited in the order they were visited. Interestingly, the last two directories are the temp directory and the home directory. Since the temp directory was not included with the evidence its possible that it has been deleted soon before the IR team has arrived or that the IR team has not copied it.

Note that (at least for us - our timezone is GMT +10) if we specified system timezone when loading the case, the times in the Zip file will be incorrect. The Zip file is a perfect example of why its important to specify the correct evidence timezone when loading it. (Alternatively if we actually unzipped the file to a local directory, the files would be saved locally on the disk in system time, and when viewing the case in US/Eastern time the timestamps would be wrong.)

Keyword searches

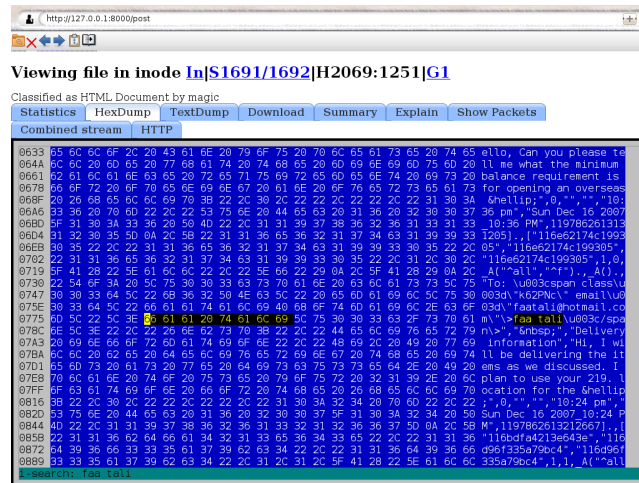
Previously we saw that the suspect made contact with the individual **faatali@hotmail.com**. We want to know what other contact was made with this individual. We want to perform a keyword search on this name. First we view all files by clicking "Disk forensics/view VFS" and selecting the table view tab.

PyFlag allows us to perform keyword indexing on a subset of the image - this is far more efficient since we already know that we are likely to see it in web messages and we can eliminate the very large files (the pcap and memory images). We can therefore filter on Size < 10000000. We click the "Show keyword hits" icon:

The screenshot shows the PyFlag application interface. At the top, there's a menu bar with 'Case Management', 'Load Data', and 'Configuration'. Below it is a toolbar with various icons. The main window is titled 'Browsing Virtual Filesystem' and has two tabs: 'Tree View' and 'Table View'. The 'Table View' is selected, showing a table with columns 'Inode', 'Mode', and 'File'. The table lists several files, including 'raw_filesystem', 'raw_filesystem/response_data/user_files/gconf/desktop/gnome/accessibility/keyboard/gconf.xml', and 'raw_filesystem/response_data/user_files/gconf/desktop/gnome/accessibility/gconf.xml'. A search bar at the top right contains the keyword 'faatali'. Below the search bar, a table shows the search results with columns 'Word', 'Class', and 'Type'. The results show 'faatali' as a word hit in the 'Class' column. A 'Submit' button is located below the search results table.

Word	Class	Type
faatali		word
faatali		word
google		word

The keyword list show us - This only shows a single hit per inode: We can view the inode and select "Hexview". The file is now displayed in PyFlag's interactive hexeditor. Pressing ctrl-s starts incremental search mode where we can locate all the hits in this file:



Automation

Forensics can be a very time consuming process. Although we have seen that the PyFlag GUI is quite capable in allowing investigators to drill into the evidence and discover important information, the loading process is fairly simple and might typically take a long time. For this end it is useful to be able to automate the loading and scanning of cases, so these can be done with little human interaction. PyFlag has a scripting shell which can be used for just this purpose. We presently describe how the DFRWS challenge can be loaded using a script.

```
## Clear the current case if it exists
delete_case dfrws

## Create a new case
create_case dfrws TZ=US/Eastern

## Start using the new case within the shell
load dfrws

## Load the zip file as a raw image
execute Load\ Data.Load\ IO\ Data\ Source case=dfrws iosource=z subsys=Standard
TZ=US/Eastern filename=dfrws2008-challenge.zip

## Scan this recursively
load_and_scan z / Raw MD5Scan MozCacheScan MozHistScan OLEScan TarScan VirScan
ZipScan GoogleDocs HotmailScanner HTTPScanner YahooMailScan SquirrelMailScan
GmailScanner

## Carve memory
scan_file /raw_filesystem/response_data/challenge.mem ScriptCarver IECarver

## Load the pcap file as a network forensic source
execute Load\ Data.Load\ IO\ Data\ Source case=dfrws iosource=n subsys=Standard
```

```
TZ=US/Eastern filename=vfs://dfrws/raw_filesystem/response_data/suspect.pcap
load_and_scan n /net/ PCAP\ Filesystem SquirrelMailScan YahooMailScan YahooScanner
POPScanner MSNScanner GmailScanner HTTPScanner HotmailScanner FTPScanner
GoogleDocs
```

```
## Load the memory image - First we create the iosource, then we load the image using
the profile and system map
execute Load\ Data.Load\ IO\ Data\ Source case=dfrws iosource=m subsys=Standard
TZ=US/Eastern filename=vfs://dfrws/raw_filesystem/response_data/challenge.mem
load_and_scan m /mem/ Linux\ Memory profile=2_6_18-8_1_15_el5
map=System.map-2.6.18-8.1.15.el5.map
```

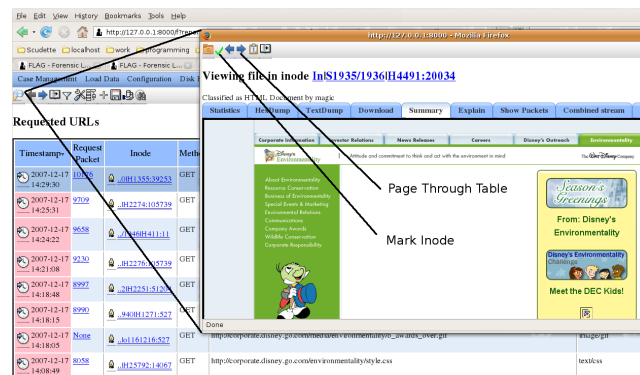
This script can run using the pyflash command: **pyflash -c dfrws.flash**

Preperation of Brief of Evidence

In a real investigation the results of our analysis would be used for prosecution in a court of law. Increasingly, digital evidence is provided in electronic form to aid in the management of large quantities of evidence. For this end PyFlag provides the ability to export reports into electronic form. The exported brief consist purely of self contained HTML pages with internal cross linking. These pages are packaged and delivered to the respective legal teams on some media such as a hard disk or cdrom. Third parties do not need to install PyFlag to view the evidence - it is purely self contained.

Presently we illustrate how this exporting process works. As part of our solution to the DFRWS challenge we provide such a brief of evidence which is referred to in this document.

The first step in the preperation of the brief is the marking or tagging of evidence. Since perusal of evidence is a time consuming process we need to be able to rapidly eliminate the unimportant inodes and quickly tag the important items. PyFlag presents a flexible table interface which allows powerful filtering. For every table an annotation icon is available which allows the user to rapidly iterate over all inodes in the table (after applying any filtering) and mark or tag the inodes:



Tables can be filtered using the operator annotated for example: **Inode annotated exfiltration** Will only show annotations with the word exfiltration in them. A convenience report called the *Case Report* presents the annotations in the case in one convenient table.

Any table within PyFlag can then be exported (including the annotation tables themselves of

course). This allows specific searches to be reported in their entirety. The result is a set of self contained HTML files which may be shared on whatever medium (cdrom, hdd, web server). Adding to that the ability to generate custom tables (by joining arbitrary columns together) and its possible to create very informative reports.

The exported pages also contain all the inodes as well as their explanations. The explanation page shows how to derive the inode from first principles - for network traffic it shows all the packets used, and their offsets.

Case: dfrws

Viewing file in inode In|S1923/1924|H2164:0

Classified as HTTP application/x-javascript by magic

Statistics HexDump TextDump Download Summary Explain Show Packets

Combined stream HTTP

IO Subsys n:

case	dfrws
iosource	n
subsys	Standard
TZ	US/Eastern
CWD	/
interactive	True
filename	vfs://dfrws/raw_filesystem/response_data/suspect.pcap

Stream In|S1923/1924

Packet ID	Offset	Length
7825	4060437	948
7827	4061471	286
8081	4180091	887
8083	4181064	259

HTTP Extract 0 bytes from In|S1923/1924 starting at byte 2164

Challenges

This section discusses some of the specific challenges encountered while undertaking the investigation.

Firefox Cache

Firefox profiles contain a wealth of forensic data for the investigator. Here we discuss the analysis techniques used to examine the contents of the .mozilla/firefox/ directory.

Although there are other cost-free firefox cache browsers available which are suitable for forensic analysis, the author could not find any for GNU/Linux, our analysis platform. Furthermore, none of the existing solutions were free software suitable for inclusion into a framework such as PyFlag. As such, this work was undertaken to complete the challenge.

Firefox stores its browser cache in the Cache subdirectory of a users profile directory. A number of files can be found here, they are explained below. The file structures were determined by examining Firefox source code. There does not appear to be any format documentation beyond the source code itself. A more detailed description of the formats can be found at http://www.pyflag.net/cgi-bin/moin.cgi/Mozilla_Cache_Format.

CACHE_MAP

This file is an index. It contains a header and a list of *records*, one for each cache entry.

_CACHE_001_

This file is split into 256 byte blocks. It contains cache metadata and data *entries*.

_CACHE_002_

As above, split into 1024 byte blocks.

_CACHE_003_

As above, split into 4096 byte blocks.

There are also many individual files with random looking filenames, these are cache data (or metadata) entries which were too large to fit in the block files and were placed in their own separate files.

A cache *record* consists of a hash and location pointers to a *metadata entry* and a *data entry*. The hash is simply the result of a one-way hash function applied to the URL. It is used by Firefox to quickly check if a given URL is contained in the cache. The location pointers are a structure which indicate where the metadata and data entries reside. They either point to extents inside the block files (_CACHE_00x_) or to separate files named after the record hash.

The *metadata entry* contains a fetch count, last fetched, last modified and expiration timestamps, the original url (called the 'key') and a *metadata block*. The *metadata block* is a series of named values separated by NULL bytes. Typical names seen are request-method (GET, POST) and response-head which is a full dump of the returned headers.

The *data entry* is simply the data returned by the server in response to the URL query. It does not contain any additional header. It is worth noting that the data may have content-encoding such as gzip or deflate applied to it. It is necessary to look at the content-encoding field in the response-head of the metadata entry to determine if any un-encoding is required.

In order to support Firefox cache in PyFlag, a file format parser was created using PyFlag's file format library, a scanner was then written using the parser.

The scanner performs the following actions:

- Triggers on the file _CACHE_MAP_ by *name*.
- Opens the other relevant files from the same directory (_CACHE_00x_)
- Adds a PyFlag VFS entry under _CACHE_MAP_ / for every *data entry*. If content-encoding is found, the VFS inode has either the deflate or gzip driver appended to it (|d1|/|G1) so that it is transparently handled by PyFlag.
- Adds an entry to the PyFlag HTTP table (as used by PyFlag Network Forensics) so that the Cache can be browsed by URL.
- Adds a new scan job for each new child VFS inode.

By leveraging the HTTP support present in PyFlag Network Forensics, it is now possible to do full web-page reconstruction of Mozilla Cache. Resource links can even be resolved from other evidence sources such as network captures or the http sundry table.

By representing each cache entry in a decoded form in the PyFlag VFS, we are also able to leverage all of pyflag's existing scanners including webmail scanning and keyword indexing. This would not be possible without native mozilla cache support.

Timezone Considerations

Timezones are important in forensics and become especially so when analysing multiple sources of evidence. These different sources may have come from different timezone locations, and/or may have different internal representations of time. Furthermore, some evidence sources may be embedded inside other evidence, but still require their own special timezone considerations.

This section explains how PyFlag handles timestamps to allow the investigator to analyse multiple sources seamlessly.

- All times in PyFlag are stored internally in UTC.
- When cases are created, a case timezone is associated with the case. The default value is the local timezone of the analysts workstation. This can be changed using the "Case Management/Configure Case" report.
- All times displayed in the PyFlag GUI appear in the case timezone. Note that the case timezone may be different from the evidence timezone (see below), and hence dates displayed may not reflect the local time of an event at the location in which it took place.
- Whenever a data **source** is added (e.g. a disk image or log file):
 - The analyst is prompted for an evidence timezone (the default is again the analysts local timezone). This describes the local timezone in which the evidence was collected. For example, for a disk image of a windows or unix workstation, it should be the same as the system timezone on that system.
 - The evidence timezone is stored in pyflag and associated with that datasource.
- As the VFS and other tables are populated by the Filesystem drivers and Scanners
 - The drivers/scanners convert dates found in evidence as necessary to store them internally as UTC.
 - Many types of data (e.g. most filesystems) use UTC timestamps already, so no conversion is required.
 - Some data sources (e.g. Zip files) store times in local (evidence) time. These must be converted from evidence timezone to UTC. The *evidence* timezone is inherited from the iosource in which the data was found. It is worth noting that if times in evidence are stored as localtime in a zone which employs daylight savings, the times may be ambiguous (i.e. occur more than once)!
 - It is up to the scanner to know if dates that it finds are UTC or evidence local (or some other zone) and make the appropriate conversion. Scanners and Filesystems do this using their own expert knowledge of the data being analysed.

Consider this example. In Australia, my default (system) timezone is "Australia/Sydney" (or GMT+10 ignoring daylight savings). I set my case timezone to system and load the challenge zipfile also using system time. When I review the zipfile entries in PyFlag, I am actually seeing "US/Eastern" localtime rather than Australia/Sydney.

This is because PyFlag knows that zip timestamps are stored in localtime so it converts them (evidence tz -> UTC, -10hrs) for internal storage and then converts again (UTC -> case timezone, +10) for viewing.

PyFlag was not told that the evidence was from US/Eastern, but because the conversions are symmetric, the display times are exactly as they are found in the zipfile.

In itself, this may not seem like a problem provided the investigator knows that zipfile times are local and interprets them as evidence localtime regardless of the case/evidence settings in PyFlag.

This however is dangerous. Consider the Mozilla/Firefox cache scanner which ran when the data was scanned. Firefox cache is an independant file format and contains its own internal timestamps. PyFlag knows that Firefox stores it's timestamps in *UTC* and adds them to PyFlag accordingly (no input conversion). When you review this data (e.g. in the HTTP Requests report), pyflag will perform an output conversion (*UTC -> case tz, +10*) for display. Therefore the investigator is seeing these timestamps in Australia/Sydney time. This is a mismatch with the US/Eastern which is being displayed for the zipfile contents, hence the investigator cannot use PyFlag filters etc to compare these times.

Consider another example where an NTFS filesystem is loaded. PyFlag knows that NTFS stores time in *UTC*, so it does not need to perform any input conversion for storage. You may think that this means the evidence timezone specified when the image was loaded is not relevant, but what happens when PyFlag scanners find a zipfile inside the NTFS image? PyFlag needs to know the evidence timezone so that it can record the zipfile timestamps correctly even though NTFS timestamps will always be correct regardless of the evidence tz.

In summary, we can see PyFlag provides a mechanism to allow multiple evidence sources to be analysed together in a single timezone. In order to support data sources where local times are used, it is critical to select the correct evidence timezone when loading data into PyFlag. It is also important to be mindful of the fact that display times are in your current case timezone which may be different to the local timezone in which the events took place. The case timezone can be changed at any time and the evidence reviewed in the new timezone by simply refreshing the reports.

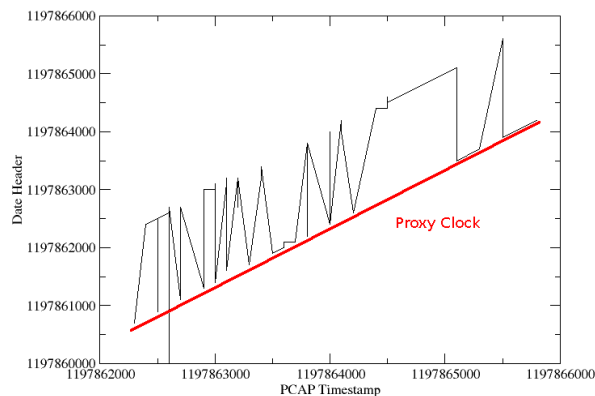
Timestamps in Network traffic

It is important to consider how timestamps are manifested in network traffic. This can be used to spot anomalies.

Within the HTTP response header we have a *Date* field. This field is the timestamp that the web server assigned to the request specified in GMT timezone. We also have a timestamp on each packet which we collected - this was assigned by the collecting computer. It is instructive to see the relationship between those entities. We can use the custom report to construct a PyFlag table which shows both the timestamp and the Date header (Of course PyFlag knows about the timezones, and so they would all be normalised to the case timezone):

Modified-	Destination IP	Request Packet	Date	Host
2007-12-16 23:29:30	219.93.175.67 Kuala Lumpur TMNET/TMNET, TELEKOM MALAYSIA	10176	2007-12-16 23:02:59	www.msn.com
2007-12-16 23:25:31	219.93.175.67 Kuala Lumpur TMNET/TMNET, TELEKOM MALAYSIA	9709	2007-12-16 22:58:00	en.wikipedia.org
2007-12-16 23:24:22	86.84.162.55 Bilalacourt HPLS network / LDCOM	9658	2007-12-16 23:26:43	ekiga.net
2007-12-16 23:21:08	219.93.175.67 Kuala Lumpur TMNET/TMNET, TELEKOM MALAYSIA	9230	2007-12-16 22:54:37	en.wikipedia.org
2007-12-16 23:18:48	219.93.175.67 Kuala Lumpur TMNET/TMNET, TELEKOM MALAYSIA	8997	2007-12-16 22:52:17	en.wikipedia.org
2007-12-16 23:18:15	198.105.193.114 Kuala Lumpur TMNET/TMNET, TELEKOM MALAYSIA	8990	2007-12-16 23:18:47	corporate.disney.go.com

We can plot the timestamp and date fields against each other:



We normally expect a single line with a bit of jitter (due to clock skew of the different sources), but interestingly in this case we get two distinct lines. The highlighted extrapolated line above has the Date field consistently 30 minutes less than what it should be. We can isolate those connections which lie on this line and see that all those connections are to the exfiltration IP address 219.93.175.67.

SSL Decryption

The challenge packet capture was found to contain several SSL connections. SSL is an encrypted communication so we cannot view the plaintext data without a decryption key. We hypothesised that the memory image may contain the symmetric keys used to encrypt the ssl connections found in the network capture.

To determine the feasibility of this hypothesis, a proof-of-concept experiment was conducted as follows:

1. Set up an ssl webserver using a certificate self-signed by our own demo CA. Configure the webserver to only negotiate RC4 as the symmetric cipher.
2. Proxy requests from the browser to our server using the free software tool 'ssldump'. If provided with the private key of the webserver, ssldump can decrypt the ssl connection passing through it and dump verbose protocol information including the negotiated session keys. ssldump only supports RC4, this is why the server was configured to force RC4.

3. Configure your environment to generate core-dumps for unhandled signals. This can be done using the 'ulimit' program in Linux. Also begin a network capture to obtain the ssl traffic.
4. Request a URL from the webserver using the firefox browser. Soon after the request has been completed, issue a signal to firefox causing it to dump core.
5. Search the core file for the keys which were printed by ssldump.

This experiment demonstrated that the symmetric keys were indeed present in the firefox core dump. We were able to prove that we could decrypt the SSL session extracted from the recorded pcap file using the keys printed by ssldump.

Next we build a simple brute-force script which takes as input a reassembled tcp stream and a memory image. It works as follows:

1. Examine the stream to determine the start of encrypted communications. This can be done by following the ssl records looking for a particular type of ssl command which indicates a switch to encrypted state. Our current method is rudimentary and may not work in all situations. Grab the first 100 bytes of the encrypted stream.
2. Starting at offset 0, build a key of the correct length (16 bytes for ssl rc4) from the dump data.
3. Initialise rc4 using this key and decrypt the start of the encrypted stream from step 1. Look for the string "HTTP" in the plaintext output. If the string is found, you may have a valid key.
4. Increment the offset by one byte and try a new key. Repeat until the end of the dump file is reached.

Using this technique we were able to recover the rc4 keys for the the network capture created in the first experiment using the associated core dump without using any of the data obtained by ssldump. This is validation that the technique is sound (at least against rc4 ssl connections).

The sample script used can be found here: http://www.pyflag.net/cgi-bin/moin.cgi/SSL_Decryption

We then tried this script against the rc4 ssl connection in the challenge network capture (which DNS indicates to be a connection to a mozilla addons repository). We used the challenge memory image as the key source. Unfortunately this did not yield a successful result.

Linux Memory Analysis

One of the major components of the DFRWS 2008 challenge was to improve the state of Linux memory forensics techniques and to develop tools that are applicable to a broad range of systems and forensic challenges that an investigator may face. In this section, we will discuss the efforts that we have made in order to address those objectives. Our goal was to make a variety of new tools and techniques available to investigators and demonstrate how they can be used to help investigate the memory sample provided as part of the challenge (challenge.mem). At the end of this section, we will also address how the information extracted from RAM is leveraged in the second major component of the challenge, the fusion of memory, hard disk, and network data.

Previous research has demonstrated that memory forensics is often an important component of the digital investigation process [[cite](#)]. Memory forensics offers the

investigator the ability to access the runtime state of the system and has a number of advantages over traditional live response techniques, typically used by forensic toolkits [cite]. While there has been some previous research into Linux memory forensics, the majority of the recent work has focused primarily on Windows memory analysis.

In 2004, Michael Ford demonstrated how an investigator could use many of the preexisting tools used for crash dump collection and analysis to help perform analysis in the wake of an incident[cite]. In particular, he described how the the "crash" utility can be used to investigate a crash dump collected from a compromised system. While "crash" proved a valuable tool for analyzing crash dumps, the author is forced to rely on "crude" techniques for analyzing memory samples that were not collected in a crash supported format (ie linear mapping of physical memory). Also in 2004, Mariusz Burdach describes collecting a sample of physical memory through from the /proc pseudo-filesystem and it's kcore file[cite]. He began by performing basic analysis (grep, strings and hex editors) to look for interesting strings and he then discussed advanced analysis that could be performed by painstakingly using gdb to analyze the system call table and list running processes. In 2005, Sam Stover and Matt Dickerson used a string searching method to find malware strings in the memory sample collected from /proc/kcore on a Linux system [cite]. Later in 2005, Burdach extended this research by releasing a the idetect tools for the 2.4 kernel, which aided in extracting file content from memory and listing user processes[cite]. In 2006, the FATKit project described generic architecture to effectively deal with memory forensics abstractions allowing support for both Linux and Windows analysis, as demonstrated in the example modules[cite]. In 2006, Urrea also described techniques for enumerating processes and manually rebuilding a file from memory[cite].

As we can see in each of these previous examples, debugging tools and their supporting information (ie Symbols) have played an important part of Linux memory forensics. As a result, we felt it was important to leverage as much of the previous work and experience with Linux kernel debugging as possible. Thus our first contribution with respect to this challenge was to create a patch for the [Red Hat crash utility](#), which is maintained by David Anderson. This is the same utility that was originally discussed by Ford, but now we have modified it so that it can analyze a linear sampling of physical memory, as in the case of the challenge.mem sample distributed with the challenge.

Red Hat Crash Utility

The Red Hat Crash Utility combines the kernel awareness of the UNIX crash utility with the source code debugging abilities of gdb. It is also has the ability to analyze over 14 different memory sample formats. Another advantage of crash is that it has support for a number of different architectures (x86, x86_64, ia64, ppc64, s390 and s390x) and versions of Linux (Red Hat 6.0 (Linux version 2.2.5-15), up to Red Hat Enterprise Linux 5 (Linux version 2.6.18+)). Thus it really does address the need to have a broad applicability. Our patch for crash can be found at the following following url:

http://www.4tphi.net/~awalters/dfrws2008/volcrash-4.0-6.3_patch

Once the patch has been applied (patch -p1 <volcrash-4.0-6.3_patch) and the source code built (make), you will also want to obtain the [mapfile and namelist](#) (a vmlinux kernel object file) for the DFRWS memory sample.

In order to process a linear sampling of memory, you will need to pass the --volatile command line option as seen in the following example:

```
./crash -f ../2.6.18-8.1.15.el5/System.map-2.6.18-8.1.15.el5 ../2.6.18-8.1.15.el5/vmlinux
```

```
../dfrws/response_data/challenge.mem --volatile
```

Crashing Challenge.mem

In this section, we will discuss how we can use the crash commands to help extract artifacts from the memory sample found in the challenge. Upon successful invocation, crash will present information about the system whose memory was sampled. For the image in the challenge, the output will look like [this](#).

From this information, we can see that the sample was taken on Sun Dec 16 23:33:42 2007 and the machine had been running for 00:56:51. It also gives us a lot of other interesting information from the image such as the amount of memory, the number of processors, etc. Our patch sets the current context to the Linux task with the PID of 0. As seen, in the output this is the PID for the "swapper" task. If necessary, this context can be changed using the "set" command. Information about available commands can be found through the "help" command. In the following sections we will demonstrate the type of information that can be extracted using crash. In particular, we will primarily focus on those things germane to the challenge.

Processes

Listing tasks is often one of the first things people want to do to see what is actually running on the system. By issuing this command, the investigator will receive information about process status similar to the Linux ps command:

```
crash> ps  
output
```

From this output we can extract information about the processes that were active on the box when the sample was collected. The ps command also has a number of useful command line options. For example, the investigator may want to display a processes parental hierarchy to determine how it was invoked (-p). As seen in the following output, the -t option can also be used to display the run times, start times, cumulative user and system times for the tasks. This information can be extremely useful as part of time line analysis and for determining the temporal relationships between events that occurred on the system.

```
crash> ps -t  
output
```

Using the -a option we are able to discern the command line arguments and environment strings for each of the user-mode tasks. This may be particularly useful when encountering an unknown process in memory or determining how an suspicious executable was invoked. This can also be helpful for mapping a process and its associated UID back to the user when the /etc/passwd file is not available. For example, by leveraging the environment strings we can determine that the bash process (PID: 2585) was started by user stevev.

```
crash> ps -a  
output
```

We are also able to extract the open files associated with the context of each task. Beyond presenting information associated with each of the open descriptors, it also prints current root directory and the working directory for each of those contexts. This can often provide valuable leads when dealing with the large volume of evidence associated with modern investigations.

```
crash> foreach files  
output
```

We can also extract information about each tasks open sockets. This can be useful to determine if there are any open connections with other systems that need to be investigated further. It will also show if the systems is listening on any ports which may have been points of entry or backdoors left behind. We can see that in the case of the challenge memory sample there aren't any open connections but the dhclient process (PID: 1565) has a socket with source port 68 and sendmail process (PID: 1872) has a socket with source port 25.

```
crash> foreach net  
output
```

Using crash we can also extract a lot of other information related to the state of the system:

Mounted file systems	crash> mount	output
Open files per file system	crash> mount -f	output
Kernel message buffer	crash> log	output
Swap information	crash> swap	output
Machine information	crash> mach	output
Loaded Kernel Modules	crash> mod	output
chrdevs and blkdevs arrays	crash> dev	output
PCI device data	crash> dev -p	output
I/O port/memory usage	crash> dev -i	output
Kernel memory usage	crash> kmem -i	output
Kernel vm_stat table	crash> kmem -V	output

There are a couple of things to note from the previous output information. First from the swap information we can see that the load on the system is not causing pages to be swapped out. Second by leveraging the data in the kernel message buffer we can get an indication of when the system was booted. For example, by looking at the **audit(1197861235.541:1): initialized** boot message which has a unix timestamp of 2007-12-16 22:14:01.

This was just a sample of the type of information that is available through the default command set that comes with crash. Another benefit associated with leveraging the Red Hat Crash Utility is that the command set can be extended through loading shared libraries. In the following section, we will discuss an extension module that will allow us to use Python scripts to interface with crash.

PyKdump Framework (Python scripting for crash)

[PyKdump](#), written by Alexandre Sidorenko, embeds a Python interpreter as a dynamically loadable 'crash' extension so you can create Python scripts to help perform analysis. In the following sections, we will show how PyKdump can help extract information from the challenge memory sample.

PyKdump includes a program called [xportshow](#) which can be used to extract a lot of useful network related information beyond what is available in the crash default command set. PyKdump and the xportshow program can also be used to extract important information from the challenge sample.

One of the first things we can do is extract detailed information about system's available interfaces. This allows us to extract information similar to that provided by the Linux command "ifconfig". This is useful for extracting the state of those interfaces including the times since they may have transmitted or received packets and whether the interface is in promiscuous mode or not. From this we can also confirm that the IP address of the eth0 interface is 192.168.151.130 which can help as we analyze the pcap data.

```
crash> xportshow -iv  
output
```

Using xportshow, we can also extract information from the internal ARP cache. This can be useful to determine other systems that may need to be investigated or to determine if the ARP cache has been manipulated in any way.

```
crash> xportshow --arp  
output
```

We can also extract the internal routing table to determine if the routes have been manipulated in an attempt to redirect traffic.

```
crash> xportshow -r  
output
```

While on the topic of layer 3 routing, we can also use xportshow to extract the route cache also known as the forwarding information base (FIB) on Linux. This stores recently used routing entries and is consulted before going to the routing table. Thus we can use this information to determine other machines the system was communicating with and look for signs manipulation. For example the route cache for the challenge image shows that our suspected system (192.168.151.130) previously communicated with the following addresses: 219.93.175.67, 86.64.162.35, 192.168.151.2, 192.168.151.254. The 219.93.175.67 address corresponds to the address where the zip files were being exfiltrated.

```
crash> xportshow --rtcache  
output
```

Now continuing to move up the stack we can also use xportshow to once again extract all the open sockets. As seen in the following results, xportshow presents this information in a format similar to netstat. This is extremely useful for determining both active network connections or listening services. It also provides a number of command line arguments for filtering the output.

```
crash> xportshow -a  
output
```

PyKdump also provides a crashinfo program that can print the system's runtime parameters (sysctl), file locks, and stack summaries.

As you can see, our patch now allows us to leverage both the Red Hat Crash Utility and PyKdump to extract a lot of valuable information from the memory sample in the challenge.

The goal of our further development efforts were to leverage the power of these tools while developing new tools and techniques that are applicable to an even broader range of systems and forensic challenges than just the debugging Linux systems. The following sections will describe how we addressed those goals using [Volatility](#), the open source volatile memory artifact extraction utility framework. We will also discuss how we are adding support to Volatility that will allow you to run your PyKdump commands transparently, even while working on a Windows host. By leveraging Volatility, our efforts for combining multiple data sources will not be limited to a particular operating system.

Volatility

Volatility is an open source modular framework written in Python for extracting digital artifacts from acquired samples of volatile system memory. From its inception it was designed to be a modular and extensible framework for analyzing samples of volatile memory taken from a variety of operating systems and hardware platforms. The Volatility Framework builds upon research we performed on both [VolaTools](#) and [FATKit](#). While previous versions of the framework focused on the analysis of Windows XP SP2 samples, as a part of this challenge we will demonstrate how it can be easily adapted to Linux as well. This challenge also allowed us to make use of the powerful new features which were added to Volatility 1.3.

The power of Volatility is derived from how it handles the abstractions of volatile memory analysis within its software architecture. This architecture is divided into three major component: Address Spaces, Objects and Profiles, and Data View Modules.

Address Spaces

Address spaces are intended to simulate random access to a linear set of data. Thus each address space must provide both a read function and a function to test whether a requested region is accessible. It is through the use of address spaces that Volatility is able to provide support for a variety of file formats and processor architectures. These address spaces are also designed to be stackable while maintaining the ability to have concurrent handles to the same data through different transformations. In order to analyze the challenge.mem sample, we make use of both the FileAddressSpace and the IA-32 paged virtual address space, IA32PagedMemory, that are also used for Windows memory analysis.

Objects and Profiles

Objects refer to any data that is found within an address space at a particular offset. The new object model included in 1.3, which was used in the software for this challenge, supports many of the semantics of the C programming language. Volatility uses profiles to define those object formats. When analyzing an Linux sample, the profile can be automatically generated from the source code or debugging information. For the challenge we will be using a profile generated for the 2.6.18-8.1.15.el5 kernel. We also include the System.map as a component of the profile as well.

Data View Modules

Data view modules provide algorithms to find where the data is located. These are the methods used to collect data or objects from the sample. For this challenge we created 11 new data view modules to facilitate analysis of Linux samples. The following sections will describe each of the new modules that was created. These new modules were also built for the new pluggable architecture included in Volatility 1.3. This allows new modules to be added without requiring any changes to source code.

Strings

As we mentioned previously, one of the most common forms of analysis performed on a sample of physical memory is to look for sequences of printable characters extracted using the "strings" command. Thus it is here that we will begin our discussion of analyzing memory using Volatility. One of the major limitations with relying on this method of analysis alone is that it is a context free search. Thus it simply treats the sample of memory as a big block of data. For example, while reviewing the strings from this image we are able to find strings related to bash [command history](#) resident in memory. From these commands we can see that someone on the system attempted to copy Excel spreadsheets and Pcap files from and admin share (/mnt/hgfs/Admin_share) to a temp file. At some later point they attempted to discover if a vulnerable version of the X windows system was running on the system. They then proceeded to download and execute a privelege escalation exploit from the metasploit project intended to gain root privileges.

In an attempt to add more context to these types of strings we created a module called linstrings which provided the equivalent functionality to Volatility's string command. This allows us to map the strings extracted from the memory sample back to the corresponding virtual address and associated process. This mapping is accomplished by walking the address translation tables and determining which processes has the ability to access the physical page where the string is located. In the Linux version we only consider the user land address space.

```
python volatility linstrings -f challenge.mem -p profiles/2_6_18-8_1_15_el5/
centos-2.6.18-8.1.15.el5.types.py -s profiles/2_6_18-8_1_15_el5
System.map-2.6.18-8.1.15.el5 -S challenge.strings > dfrws_strings_map
output
```

Examples of interesting strings

Physical Offset	[Pid:Virtual Address]	String
8393760	[2582:8fa1420]	http://219.93.175.67:80
10456534	[2585:8b59dd6]	tar -zxvf xmodulepath.tgz
197604536	[2585:8b4e4b8]	wget http://metasploit.com/users/hdm/tools/xmodulepath.tgz
107837393	[2582:92087d1]	[stevev@goldfinger ~]\$ cp /mnt/hgfs/software/xfer.pl .
207989168	[2585:8b4b9b0]	./xfer.pl archive.zip
212984368	[2585:8b4c230]	zip archive.zip /mnt/hgfs/Admin_share/acct_prem.xls /mnt/hgfs/Admin_share/domain.xls /mnt/hgfs/Admin_share/ftp.pcap
222017064	[2582:922f628]	[stevev@goldfinger ~]\$ rm xfer.pl
10456593	[2585:8b59e11]	./root.sh
197607328	[2585:8b4efa0]	export http_proxy="http://219.93.175.67:80"

The ability to map these strings back to their respective processes is extremely useful. We can see that all the strings in the previous table were addressable by processes with a UID of 501, which is the UID for user stevev, Steve Vagon.

Interesting files found in memory

/etc/passwd

[link](#)

/etc/group

[link](#)

linident/lindatetime

The linident module is used to provide valuable information about the system the memory sample was acquired from. This module provides similar information to the crash sys command but it has been augmented to include timezone information, which we have found useful during temporal reconstruction. As seen the the following output the local timezone for the system was GMT-5. It also provides the GMTDATE corresponding to when the sample was acquired. The current time and timezone information can also be obtained from the lindatetime module as well.

```
$python volatility linident -f challenge.mem -p profiles/2_6_18-8_1_15_el5/  
centos-2.6.18-8.1.15.el5.types.py -s profiles/2_6_18-8_1_15_el5/  
System.map-2.6.18-8.1.15.el5.map  
output
```

linps

We also provide a module that will extract what processes were running on the system when the sample was acquired. We have augmented this to also include the UID of the process owner. By combining this with the strings to process mapping provided by linstrings we are able to attribute those strings to a particular user. For example by correlating with the environment information previously discussed (or if /etc/passwd was available) we know any process with UID 501 can be attributed to user stevev. We also know that any strings mapping to those process are connected to that user as well.

```
python volatility linps -f challenge.mem -p profiles/2_6_18-8_1_15_el5/  
centos-2.6.18-8.1.15.el5.types.py -s profiles/2_6_18-8_1_15_el5/  
System.map-2.6.18-8.1.15.el5  
output
```

linpsscan

We have also included linpsscan which makes use of the Volatility scanning framework. Unlike linps which traverses the operating system data structures to find the processes that were running on the system, this modules performs a linear scan of the physical memory sample while searching for task_struct objects which it treats as a constrained data item. These constraints were automatically developed by sampling valid task_structs from the memory sample. The benefits associated with this technique can be seen in the fact that the previous module, linps, was only able to enumerate 89, while linpsscan found 10 more with the physical address space. We have also included the UID so each task_struct could be mapped back to a user.

```
python volatility linpsscan -f challenge.mem -p profiles/2_6_18-8_1_15_el5/
```

```
centos-2.6.18-8.1.15.el5.types.py -s profiles/2_6_18-8_1_15_el5/  
System.map-2.6.18-8.1.15.el5  
output
```

linmemdump

We also created a module called linmemdump. This module automatically rebuilds the address space for a specified process and dumps its entire addressable memory to a file for further analysis. This can be extremely useful if you are attempting a brute force encryption keys (ie SSL) or you want to add some context to your string searches. The process to be dumped can be specified by either a PID (-P) or task_struct physical memory offset (-o) depending on whether it was discovered with linps, or linpsscan, respectively..

```
python volatility linmemdump -f challenge.mem -p profiles/2_6_18-8_1_15_el5/  
centos-2.6.18-8.1.15.el5.types.py -s profiles/2_6_18-8_1_15_el5/  
System.map-2.6.18-8.1.15.el5 -P 3048
```

linpktscan

We also created a linpktscan module that performs a linear scan of the sample of physical memory looking for memory resident network packets. This module makes use of the Volatility generic scanning framework to describe network packets as constrained data items. The current implementation constrains the sought after data to either UDP or TCP packets with a header of minimum length that has a valid IP header checksum. Another nice feature of this module is that it also allows the investigator to extract those packets from memory and write them to a pcap file that can then be imported into their favorite packet analysis tool (ie Wireshark).

```
python volatility linpktscan -f challenge.mem -p profiles/2_6_18-8_1_15_el5/  
centos-2.6.18-8.1.15.el5.types.py -s profiles/2_6_18-8_1_15_el5/  
System.map-2.6.18-8.1.15.el5  
output
```

Using this module on the memory sample provided in the challenge, we were able to see that the system had recently communicated with the following IP addresses over http: 219.93.175.67, 198.105.193.114. Of particular interest are the packets being sent to the 219.93.175.67 address. This was the address where the zip file was exfiltrated using http cookies. By using linpktscan we are able to find and extract memory resident packets with cookies containing parts of the exfiltrated data. Thus we are able to connect the data in the pcap files back to the system.

On another interesting note, we are also able to extract FTP packets flowing between 10.2.0.2 and 10.2.0.1. These memory resident packets were part of the ftp.pcap file that was exfiltrated. Thus we know that at some point this file was loaded into memory on the system.

linvm

This module will display the virtual memory mappings for each process. This provides information analogous to that typically found by the maps file in the /proc entry for the process. This can be extremely useful for determining which files maybe memory mapped by a process and where they can be found within memory. This can be extremely helpful for determining how the address space is being used. We have also augmented the output to include information about the code, data, and stack regions of a processes virtual address

space in case an investigator wants to extract them from memory as well.

```
python volatility linvm -f challenge.mem -p profiles/2_6_18-8_1_15_el5/  
centos-2.6.18-8.1.15.el5.types.py -s profiles/2_6_18-8_1_15_el5/  
System.map-2.6.18-8.1.15.el5  
output
```

linsockets

We have also included a module linsockets which can be used to extract information about each task's open sockets. As previously mentioned this can be useful for determining if there are any open connections with other systems or if the system is listening on any unexpected ports and if so which process is responsible.

```
python volatility linsockets -f challenge.mem -p profiles/2_6_18-8_1_15_el5/  
centos-2.6.18-8.1.15.el5.types.py -s profiles/2_6_18-8_1_15_el5/  
System.map-2.6.18-8.1.15.el5  
output
```

linfiles

We are also able to extract the open files associated with the context of each task. As previously mentioned, this can often provide valuable leads to target files or directories of interest when dealing with large disk images.

```
python volatility linfiles -f challenge.mem -p profiles/2_6_18-8_1_15_el5/  
centos-2.6.18-8.1.15.el5.types.py -s profiles/2_6_18-8_1_15_el5/  
System.map-2.6.18-8.1.15.el5  
output
```

linmodules

The final module which we have included is linmodules. This will print basic information about the currently loaded kernel modules. This allows an investigator to determine if anyone may have attempted to load a kernel module to dynamically change the behavior of the kernel.

```
python volatility linmodules -f challenge.mem -p profiles/2_6_18-8_1_15_el5/  
centos-2.6.18-8.1.15.el5.types.py -s profiles/2_6_18-8_1_15_el5/  
System.map-2.6.18-8.1.15.el5  
output
```




As you can see, Volatility provides a powerful software architecture that allows it to be easily adapted to whatever type of hardware or operating system the investigator needs to analyze. It also provides extremely useful APIs and libraries that allow investigators quickly create new modules to support their investigations and to easily share those modules with colleagues. We are also finishing up code that will allow you to run PyKdump scripts transparently within both crash or Volatility. Another advantage of Volatility is that it allows the analyst perform their investigations on any operating system which supports Python. Thus we believe that Volatility allows us to achieve our goals of leveraging previous work in kernel debugging while being applicable to a broad range of systems. Finally, Volatility is currently integrated into a number of analysis frameworks including both PTK and PyFlag.

PyFlag Memory Analysis

PyFlag has officially supported memory forensics since its integration of Volatility in January of 2008. Thus allowing an investigator to correlate disk images, log files, network traffic, and memory samples all within an intuitive interface. It was also the first framework to support analysis of memory samples stored in either EWF or AFF formats. As a part of this challenge we have extended that integration so that it now has the ability to support the analysis of both Linux and Windows memory samples and can leverage the new functionality of Volatility 1.3. The rest of this section will discuss how you can perform memory analysis from within PyFlag. These techniques will be demonstrated on the memory sample included in the challenge, challenge.mem.

In order to analyze a memory sample with PyFlag, the sample must be loaded. This is accomplished by choosing the Load IO Data Source menu item found under the Load Data Tab at the top of the screen. At the Load IO Data Source page set the "Select IO Subsystem" to standard and leave the "Evidence Timezone" to SYSTEM. At the next "Load IO Data Source" page once again set the "Select IO Subsystem" to Standard and leave the "Evidence Timezone" to SYSTEM. Depending on whether your image can be found on disk or the Virtual File System, click either the finger pointing to the folder or the VFS files respectively. Having already loaded the evidence file, we will search for the memory sample within the VFS. Thus we click on the VFS folder. At this point we will be presented with a table listing all the files in the VFS. In order to find the file we are looking for, challenge.mem, we click the funnel in the upper left hand corner of the window which will allow us to filter the table. At the Filter Table pop-up screen type "Filename" contains challenge into the Search Query dialog box. Then, click the submit button at the bottom. At this point you should see our sample in the table. After choosing the sample you will be returned to the Load IO Data Source page. Fill in the "Enter partition offset:" box with a zero and the "Unique Data Load ID" box with "mem".

Load IO Data Source

Select IO Subsystem	<input type="text" value="Standard"/>
Evidence Timezone	<input type="text" value="SYSTEM"/>
Currently Selected files	 vfs://dfrws/raw_filesystem/response_data/challenge.mem
Select AFF image:	 <input type="text"/>
 Enter partition offset:	<input type="text" value="0"/>
Unique Data Load ID	<input type="text" value="mem"/>
<input type="button" value="Submit"/>	

Now click the Submit button in the lower left had corner of the windows. You will now be brought to the "Load Filesystem image" screen. Verify that the "Case" value is set to dfrws

and the "Select IO Data Source" matches the value you entered for "Unique Data Load ID" on the previous screen. Then set the "Enter Filesystem type" drop down to "Linux Memory" and choose a mount point (ie memmnt) for the "VFS Mount Point" entry box.

Load Filesystem image

Case:	<input type="text" value="dfrws"/>
<hr/>	
Select IO Data Source	<input type="text" value="mem"/>
<hr/>	
Magic identifies this file as: data	
Enter Filesystem type	<input type="text" value="Linux Memory"/>
VFS Mount Point:	<input type="text" value="memmnt"/>
<hr/>	
<input type="button" value="Submit"/>	

Next click the Submit button at the lower left hand of the screen. At this point you will be prompted to choose a Volatility profile. Select the 2_6_18-8_1_15_el5 "Profile" from the drop down menu and click the Submit button again. Next you will be presented with another drop down menu to select a "Symbol Map". Select System.map-2.6.18-8.1.15.el5.map from the drop downs.

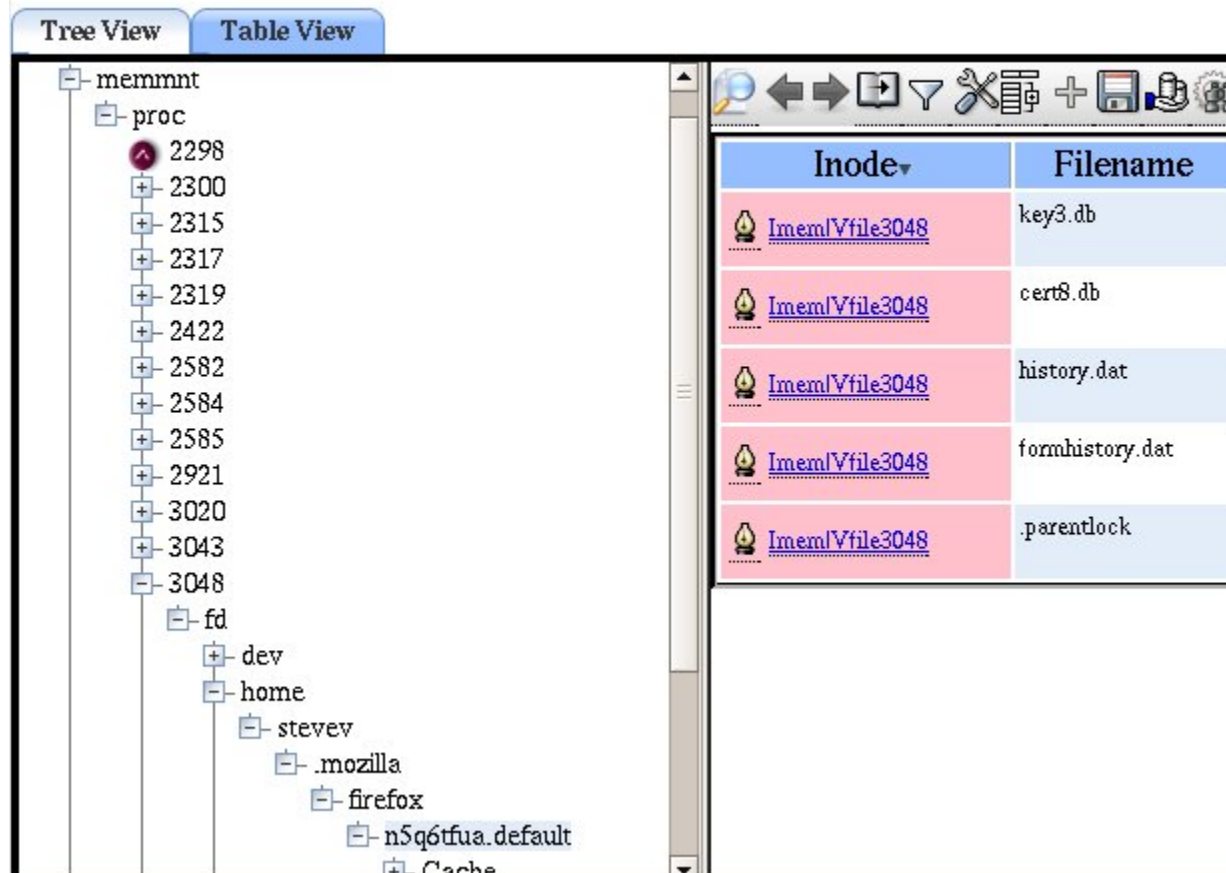
Load Filesystem image

Case:	<input type="text" value="dfrws"/>
<hr/>	
Select IO Data Source	<input type="text" value="mem"/>
<hr/>	
Magic identifies this file as: data	
Enter Filesystem type	<input type="text" value="Linux Memory"/>
VFS Mount Point:	<input type="text" value="memmnt"/>
<hr/>	
Profile	<input type="text" value="2_6_18-8_1_15_el5"/>
Symbol Map	<input type="text" value="System.map-2.6.18-8.1.15.el5.map"/>
<hr/>	
<input type="button" value="Submit"/>	

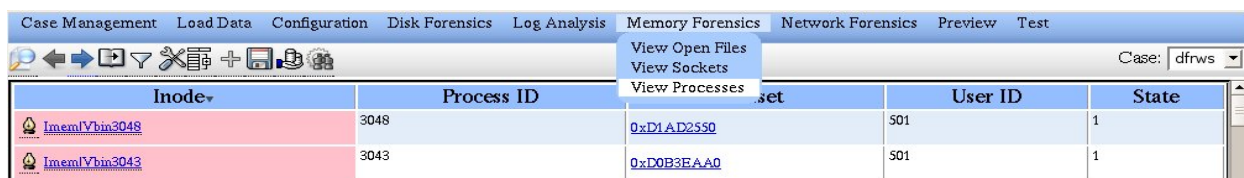
Once the System Map is selected, select the Submit button on the lower left had corner again. At this point it will begin to load the sample into the system. When it is finished loading you will return to the "Browsing Virtual Filesystem" windows and your sample will be

mounted at the specified "VFS Mount Point", which in our example is memmnt. Now that our memory sample has been loaded you can access the data through a browseable /proc interface or through the "Memory Forensics" menu item at the top of the screen.

Browsing Virtual Filesystem



On the other hand, you can also access the data through the "Memory Forensics" menu item at the top of the page as seen in the following image.



By clicking on a linked address it will automatically perform the address translation and take you to the correct offset within the physical address space. As we previously mentioned we can also run our carvers against the sample.

Conclusions

The 2008 DFRWS challenge is a very practical exercise with a scenario which is very relevant to a typical incident response investigation. The challenge deals with the processing of forensic information which to date has not been described thoroughly in the literature, namely linux memory forensics and network forensic analysis. To this end, many features have been developed in PyFlag and Volatility to address these emerging fields. We presented how tools such as PyFlag can integrate different forensic techniques in a way that is beneficial to the overall investigation - for example page rendering can occur from cache as well as from network data sources (or both).

We also saw how PyFlag can generate electronic briefs of evidence - a feature which is useful for sharing the forensic findings with legal teams and the like.