available at www.sciencedirect.com

journal homepage: www.elsevier.com/locate/diin

Digital Investigation

PyFlag – An advanced network forensic framework

M.I. Cohen

Australian Federal Police, Brisbane, Australia

Keywords:
Network forensics
Protocol analysis
Telecommunication intercept analysis
Digital forensics
Web application forensics
Python (programming language)
HTML analysis

ABSTRACT

Network forensics is an investigation technique looking at the network traffic generated by a system. PyFlag is a general purpose, open source, forensic package which merges disk forensics, memory forensics and network forensics.
This paper describes the PyFlag architecture and in particular how that is used in the network forensics context. The novel processing of HTML pages is described and the PyFlag page rendering is demonstrated. PyFlag's novel processing of complex web applications such as Gmail and other web applications is described. Finally PyFlag's report generation capabilities are demonstrated.
© 2008 Digital Forensic Research Workshop. Published by Elsevier Ltd. All rights reserved.

1. What is network forensics?

Network forensics refers to the forensic analysis of captured network traffic. In the simplest case the traffic represents network communication exchanged by a party during the course of some activity. Network forensics allows us to make forensic determinations based on the observed traffic, which may be relevant in the course of an investigation.

This paper deals with the technical capabilities of an advanced network forensic system. We deliberately do not deal with the legal aspects of obtaining the network capture in the first place.[1] We assume that a full network dump is available for analysis as obtained by the *tcpdump* program (Tcpdump, 2007). Network capture files are typically named PCAP files after the library used to read them.

Considering the analysis of network dumps in the forensic context, as with other forensic disciplines, the network forensic analyst attempts to reconstruct or understand events from the information observed in the network capture. The aim of the analysis is commonly to establish high level facts such as attribution, intent, identity, timelines and other information which may be relevant to the case.

From a forensic point of view we are often more interested in high level deductions, than in low level network protocol information. For example, in a typical fraud case we might be interested in a file of interest which was sent from a computer over the network. We may be less interested, in the first instance, in how the file was sent (e.g., using email, web pages or instant messenger), then in the contents of the file itself.

Furthermore, we want to apply the same well tested tools and techniques from the mature disk forensic field on files discovered in the network capture. For example, compare hashes on transmitted images with well known image sets, or perform a keyword search on network traffic. This top down approach is typical in digital forensics as a way to control and regulate the vast quantities of data present.

This paper introduces PyFlag as an innovative network forensic platform. Initially PyFlag is compared to current open source tools. The general PyFlag architecture is introduced, and the major architectural components are outlined. The paper then examines some important examples of the network forensic components in detail illustrating the trade offs made in their design. In particular we look at the components required to facilitate decoding of web mail – from the stream reassembler, the HTTP dissector and finally HTML parsing. The rendering of HTML pages and web mails is further explored.

E-mail addresses: scudette@gmail.com, Michael.Cohen@afp.gov.au

[1] There are numerous jurisdictional rules and regulation with regard to obtaining lawful intercepts of network communications, which fall outside the scope of this work. Readers are encouraged to obtain relevant legal advice.

1742-2876/$ – see front matter © 2008 Digital Forensic Research Workshop. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2008.05.016

2. Related tools

There are a number of tools available for processing PCAP files, most notably *wireshark* (Wireshark, 2008), *snort* (Snort, 2008) and *tcpdump* itself. These standard tools implement some very sophisticated protocol dissectors,[2] which allow the analyst to break down each protocol in detail. Unfortunately these tools were designed to assist the network administrator with debugging network problems, and are not specifically aimed towards forensic analysis. Most of these tools are not designed to deal with the large volumes of information typical of a forensic investigation,[3] and require low level understanding of the underlying network protocols by the analyst.

For example, wireshark adopts a bottom-up approach showing the raw packets first, and then reassembling each stream on demand by the user. It is also unable to scale to network captures larger than a few megabytes due to the fact it keeps all data structures in memory at the same time. The author is unaware of open source tools which have a forensic focus.

Other commercial offerings include Eeye's Iris product (Eeye Iris, 2008), as well as Verint's data interception tools (Communications interception, analysis, and service provider compliance, 2008). These tools specialise in analysing network activity mainly for intrusion detection purposes. Although these tools claim the ability to extract documents contained within emails sent over the network, their integration with standard forensic techniques is limited (e.g., hash comparison, keyword indexing and searching). These tools also do not offer the ability to integrate different sources of data into the same case, such as disk images and memory images.

Ideally a network forensic tool should have the following properties:

- Efficiently process very large capture files. The system should ideally be scalable (i.e., more hardware leads to faster processing).
- Extract high level information, for example, files transferred, social networks (who talked to whom) or keyword indexing.
- Be able to substantiate each deduction, clearly demonstrating how the deduction can be cross checked using first principles.

PyFlag's novel approach is the integration of all aspects of forensics into the same package. This allows users to use powerful disk forensic techniques, such as keyword indexing and hashing to process network bourne data. In addition, evidence from disks and memory images can be used to complement the network evidence. For example, the inclusion of items from the suspects' Internet cache can be used to augment the data obtained from network traffic in producing more accurate rendering of web pages.

PyFlag provides for specialised scanners (see section 3.4) which are capable of dissecting higher level application specific data such as that found in web mails. PyFlag currently supports Hotmail, Gmail and Yahoo mail among others and can extract messages from HTML and Javascript pages. The integration of network forensics with disk forensics allows these same scanners to operate on pages from the disk cache.

[2] Wireshark claims to support over 500 protocols.

[3] The typical network forensic capture would be several gigabytes in size.

3. PyFlag architectural overview

PyFlag (Cohen and Collett, 2005) was originally designed by the Australian Department of Defence, and was later released under the GPL (Free Software Foundation, 2007) license. It was originally designed as a database driven analysis tool for digital forensics, but later included an advanced network forensic capability.

The design of PyFlag is described in detail elsewhere (Cohen and Collett, 2005), but here we provide a brief overview.

The main concepts are shown in Fig. 1.

3.1. The IO source

PyFlag is able to handle images in a variety of formats. The actual image format is abstracted through the IO subsystem, which provides for a standard interface by which an image is presented to PyFlag. Some of the commonly used IO source drivers include EWF (Kloet et al., 2007) which adds support for

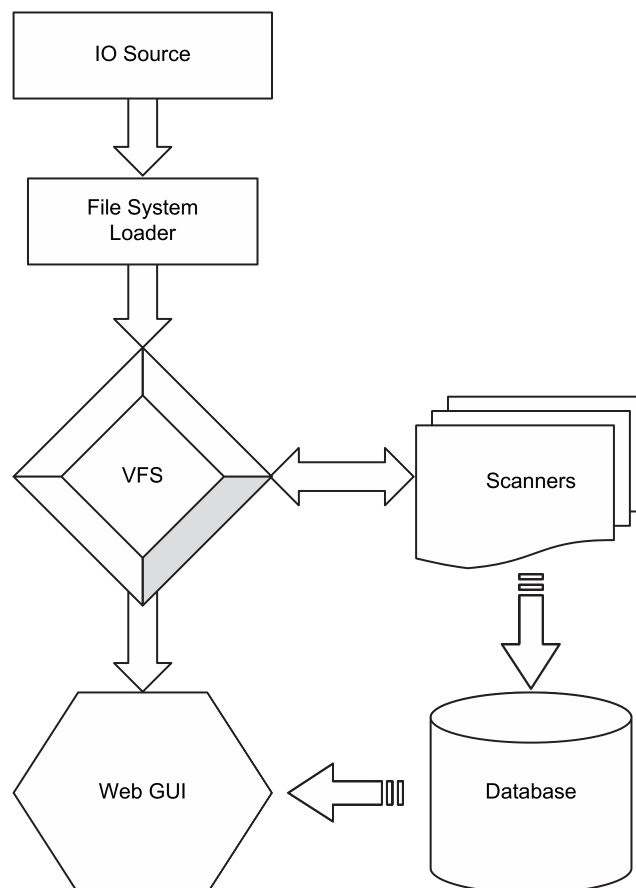

Fig. 1 – An overview of the PyFlag architecture.

the industry standard eye witness evidence file format, and the standard IO source which adds support for raw images as taken by the dd program.

This abstraction allows very large PCAP files to be efficiently stored using the EWF format which provides benefits such as compression, hash verification and case metadata.[4]

IO sources are named by the user and referred to within PyFlag by their name.

3.2. The virtual file system

The virtual file system (VFS) is a central concept in PyFlag. The VFS is essentially a tree like structure which forms an arena for representing all objects within PyFlag. The VFS is modeled after a real filesystem, and VFS objects are called inodes.

The VFS inode is a string which describes how the object can be derived, and has an associated path and filename. This path arranges inodes into a directory structure within the VFS.

Internally all objects are represented as inode id's (an internal integer reference), which is common in most tools. The novel approach taken by PyFlag is that the inode string represents the path taken to arrive at this data. This allows users at a glance to see where each deduction was derived from.

3.3. The filesystem loader

Once images are accessed in a standard way via the IO source abstraction, the VFS must be populated with inodes. The filesystem loader is a specific module which is responsible for loading the VFS with inodes. Traditional filesystem loaders read inodes from a real filesystem (such as NTFS or Ext2) and populate the VFS with these. More sophisticated filesystem loaders create virtual inodes to represent abstract objects. For example the PCAP filesystem loader represents TCP streams with VFS inodes, while the memory forensic loader represents process IDs and other memory structures using VFS inodes.

Each filesystem loader loads its inodes into the VFS at a particular *Mount point*. This allows many different IO sources and filesystems to be represented within the same VFS. In this way evidence from different sources can be correlated together. For example, it is possible to load a related PCAP file, a disk image and a memory image into the same case. Since all the sources are treated the same, and they all produce VFS inodes, these are all shown together throughout the application.

3.4. Scanners

Scanners are modules which operate on VFS inodes and gather specific information about them. Sometimes a scanner may create new VFS inodes and scan those recursively. Other times information will be stored in the database about the VFS inodes in specialised tables.

For example, the zip scanner targets zip files. If a VFS inode is identified as a zip file, the zip scanner will list the archive and create VFS inodes for each member of the archive. All other scanners are then run on the VFS inodes produced.

The type scanner looks at the file header (sometimes called magic) and determines the file type from the header. This is stored in a database table. Combining the two scanners, the type scanner will be launched on each file within the zip archive (since they are all simply VFS inodes). Due to this recursive operation, scanners allow for the recursive discovery of encapsulated files.

The inode itself indicates how the object is to be derived. For example the inode notation `Inet|S1/2|o423:40000|m1|Z100:2000` can be broken down:

- The IO source named *net* is to be used.
- Stream 1 needs to be combined with stream 2 chronologically through the stream reassembler.
- At offset 423 (within that stream) we extract 40,000 bytes.
- The extracted data is to be parsed as a mime message (RFC2822). The first attachment is to be extracted.
- At offset 100 of this attachment we find a zip file header of length 2000. We extract this zip file data.

As can be seen the inode string itself is sufficient to describe how the data may be obtained from first principles. This satisfies our requirement that all data derived by PyFlag must be directly verifiable using first principles.

4. PyFlag network forensics

As discussed in previous sections, one of the strengths of the PyFlag architecture is the ability to recursively examine data at multiple levels, and discover files encapsulated within other files. This approach is ideal for network protocols which are typically layered, with higher level protocols being carried over lower level protocols.

Typically the following steps are taken in dissecting network traffic:

- PCAP files are parsed and individual packets are extracted from them.
- These packets are then dissected at the different low level protocols, such as Ethernet, IP, TCP or UDP.
- Related TCP packets are collected into streams by a TCP stream reassembler. The reassembler takes into account packet retransmission and out of sequence ordering.
- The resulting streams are then dissected with higher level protocol dissectors such as HTTP, IRC, MSN Chat, etc.

This process is implemented in PyFlag via the PCAP filesystem loader. The PCAP filesystem loader uses a fast TCP stream reassembler written in C to collate packets into streams, and these streams are represented in the VFS during load time. The filesystem also allows for packet dissectors to operate on those packets which do not belong in TCP streams, for example, DNS and SIP.

Once the VFS is completely populated by the loader, it is possible to scan the VFS inodes using the full suite of network scanners. The network scanners operate on the VFS inodes and attempt to discover those streams of the specific protocols

[4] EWF files may be created using the ewfacquire utility from libewf which is included with PyFlag itself.

handled. For example, the HTTP scanner identifies HTTP like transactions within the streams and creates VFS inodes for each HTTP object transmitted.

It is important to note that when scanning, the network scanners typically scan the inodes they create using the full suite of scanners. This includes the standard disk forensic scanners, such as virus scanning or the type scanner. Once an inode is created within the VFS there is no difference between network derived inodes, and disk derived inodes – all standard forensic analysis techniques can be applied to network traffic.

5. Network forensic components

PyFlag's network forensics modules comprise of a number of different functional components which operate together to achieve their goal. The following section discusses in detail some of the more important components and the design criteria used in their implementation.

5.1. *The stream reassembler*

TCP stream reassembly is a critical component of any system which needs to examine the contents of TCP streams. The most common such device is the network intrusion detection system (NIDS), where the stream reassembler is considered as a critical component (Wojtczuk, 2008).

The reassembly problem is a difficult one to solve, since there are many TCP/IP packet sequences with non-deterministic interpretation (Ptacek and Newsham, 1998). Recent research has shown that it is impossible to build a TCP reassembler capable of correct reassembly without knowledge of the host OS and configuration (Ptacek and Newsham, 1998). There are a number of tools which are able to generate packet sequences which lead the stream reassembler into producing completely different TCP streams than those which were actually received by the hosts in question (Song).

Failure to reassemble the TCP stream correctly may result in *insertion attacks* or *evasion attacks* (Ptacek and Newsham, 1998), both resulting in a different stream produced by the reassembler than that actually sent.

Other issues encountered by the reassembler include:

- Termination of streams. When should we consider the stream terminated? Should we include packets which appear to belong to the stream after it is terminated?
- Out of sequence packets. A major part of the stream reassembler is to reorder packets which are out of sequence.
- Missed packets. Sometimes we are unable to find a packet within the stream. This creates a hole in the data which we need to pad out. When should we give up on seeing a retransmission? How do we pad the data?
- Speed. Although in a forensic context we do not typically need to process packets in real time, we would still like to have fast processing. From the NIDS world we know that the more thorough the stream inspection is, the slower the reassembler (because it needs to do more checks and better emulation of TCP stacks).

The PyFlag reassembler is therefore a compromise between many competing requirements. Our basic assumption is that the traffic will be found in a natural form – that is to say that the communicating parties are not aware of the forensic network capture and will not be deliberately attempting to subvert the reassembler by breaking streams into pedagogically ambiguous cases. As described previously, such subversion is always possible for any type of stream reassembler and if the parties are aware of the interception they can always produce traffic which is undecodable.[5] This assumption may not hold in some cases, and the forensic analysis might need to examine whether the stream reassembler was subverted.

5.1.1. *Combined streams*

The reassembler also provides for *combined streams* – a term which refers to the stream formed by combining a forward and reverse stream in chronological order. This is important for many protocols which use bidirectional, request response messages. These protocols require one side of the communication to be fully received before the other side begins transmission. In this case, retransmissions and out of order packets do not affect the combined stream (because the communication from each side must be finalised before the order side begins transmission).

On the other hand, for protocols where both sides are allowed to transmit asynchronously (e.g., MSN chat protocol), retransmissions can affect the combined stream.

5.2. *Packet handlers – e.g., DNS*

For TCP/IP streams, we are usually only interested in the data stream which we get from the reassembler. However, for non-TCP packets, we may need to write specific handlers.

PyFlag allows us to write specific packet handlers which will be called with all packets which do not belong in a TCP stream. An example is the DNS handler which stores all the DNS name resolutions as obtained from the traffic.

This is important since DNS records may have changed from the time the capture is taken to the time it is analysed. We would most often see a DNS request for an A record, an answer will be returned of a certain IP address, and a TCP connection to that IP address take place. By recording the DNS lookup we are able to establish which host name the system was attempting to contact at the time the capture was taken.

5.3. *Stream dissectors – e.g., HTTP*

Most higher level protocols utilising the TCP/IP layer do not use specific packet level information, and therefore require only the reassembled TCP stream for complete decoding. PyFlag's stream dissectors are specialised scanners which deal only with reassembled TCP streams (i.e., they only deal with *inodes* created by the stream reassembler). The scanners then parse the content of the stream and potentially the reverse stream to decode the protocol in question.

[5] For example the simplest way to avoid analysis is to use encryption on network communications.

```
GET /search?q=pyflag HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0
Accept-Encoding: gzip,deflate
Keep-Alive: 300
Connection: keep-alive

HTTP/1.1 200 OK
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Server: gws
Transfer-Encoding: chunked
Content-Encoding: gzip
Date: Mon, 17 Dec 2007 03:42:15 GMT

b57
.... [Binary data] ...
```

Fig. 2 – A sample of an HTTP transaction. In this example the response (an HTML page) is both chunked and compressed using gzip.

Although PyFlag supports a number of protocols for dissection, we describe the HTTP protocol dissector in detail here since it is one of the most useful protocols from a forensic point of view.

The HTTP protocol is described in RFC 2616 (Network Working Group, 1999), and appears at first to be remarkably simple. The protocol contains a request and reply section. The request consists of a method, a URL, and a protocol version followed by a sequence of headers. The request may contain a body section. The response consists of the protocol version, a response code and a sequence of headers. The response may also contain a body. Both the request and response body section may contain arbitrary data. The length and content type of this data must be specified in the respective headers, but the content is unrestricted.

We term the *HTTP Object* as the bundle of data returned by the server in response to the HTTP request. This object may or may not be exactly the same as the data in the response's body (due to transfer encoding). It also may not correspond to an actual file on the server, as in the case of dynamically generated content.

An example of an HTTP session is shown in Fig. 2. As can be seen in this example, in order to correctly decode the response special support for compressed and chunked data must be implemented. If no compression or chunking was used, the HTML page would be visible within the traffic.

The HTTP dissector parses the combined stream. For each HTTP transaction the following actions are taken:

- Basic information about the transaction is recorded, such as URL, content type, length, method and timestamp as reported by the HTTP *date* headers.
- Parameters exchanged via the GET and POST method are properly decoded using a standard CGI parser and stored in a separate table for each transaction. There can be a large number of different parameters passed for a single transaction, and their size can be very large (e.g., file uploads). These parameters are examined by subsequent dissectors.
- The dissector creates VFS inodes to access the actual HTTP object received. This may be a simple offset driver, which simply extracts the file from the stream using an offset and a length. Depending on the HTTP encoding, specialised drivers may be used for handling *chunked* transfer encodings (Network Working Group, 1999) and *gzip* or *deflate* content encoding.
- The HTTP scanner invokes further scanners on the newly created inodes. This employs standard forensic techniques (such as hashing, virus scanning, etc.) on the contents of files transferred over HTTP.

6. HTML rendering

As argued in Section 1, the forensic examiner is more interested in high level information obtained from the traffic rather than low level protocol information. Previously we have seen how PyFlag is able to recover HTML files transferred over HTTP. When a user typically navigates to a web site, their browser may generate several different HTTP connections,

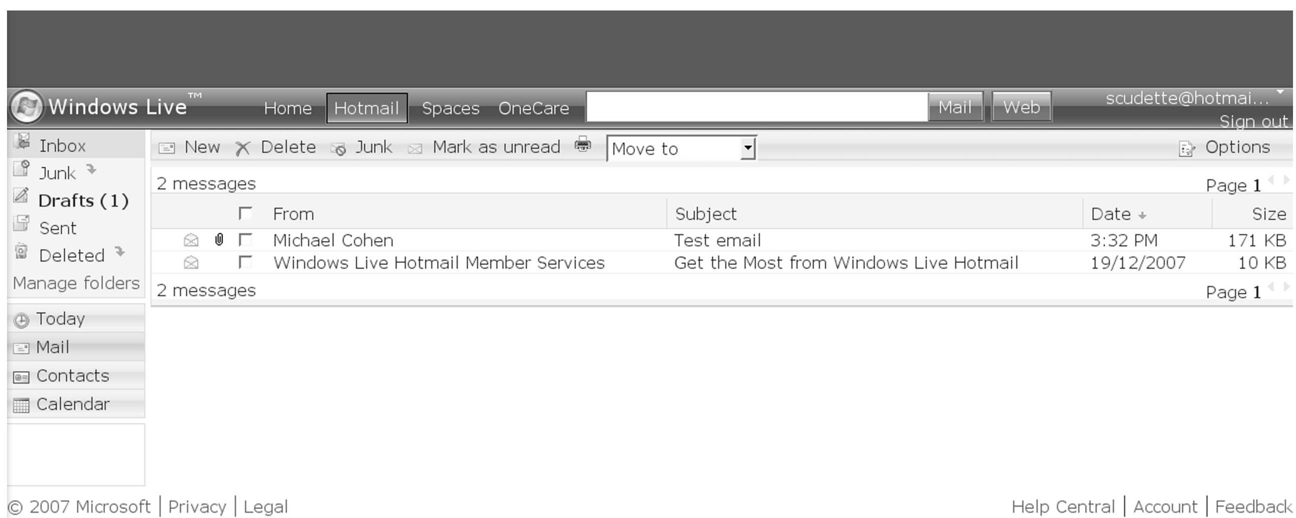

Fig. 3 – A rendering of a page visited from Hotmail. All elements of this page including style sheets and images were automatically recovered from the network capture and the HTML markup was sanitized.

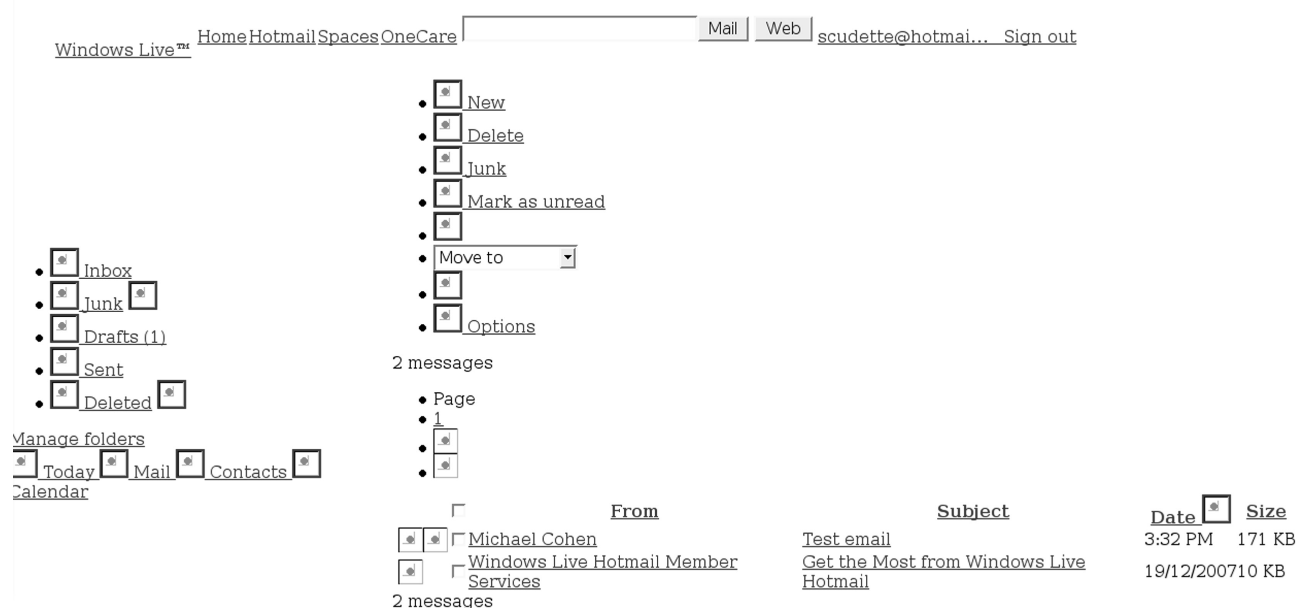

Fig. 4 – The same page as shown in Fig. 3, with the browser cache set to not expire before the capture was started. All images and style sheets are found in the cache, and therefore not transferred over the network. PyFlag is unable to resolve these references, and in particular without relevant style information, the rendering layout is very different than the real site presents.

and download many different objects, all related to the same logical page. For example, images, style sheets and javascript pages are all separate objects which serve to make up the same page.

The forensic examiner is interested in viewing the page as closely as possible to how the suspect viewed it at the time. We use the following definitions:

Page rendering: Page rendering is defined as a representation of the page as derived from the traffic which is as close as possible to the page viewed by the suspect at the time the traffic was captured.

Note that the requirement for page rendering is that the page visually appears as close to what it would have appeared at the time it was viewed. It is simply a way of visualizing the page, and does not replace the actual HTML received (which is still available to the analyst). The rendering process may alter some of the HTML before rendering it with a browser in order to make accurate rendering possible (for example, adjusting references to images and style sheets).

The need to produce accurate page rendering is balanced by the need for HTML sanitization:

HTML sanitization: HTML sanitization is the process by which the HTML page is modified so that when opened in a browser or rendered, no external links remain, and the browser does not make requests to other web sites on the Internet.

When a page is rendered, the HTML is opened in a browser. The browser will attempt to retrieve all links within the HTML page in order to render it (for example, images, style sheets). The browser will also run any javascript embedded in the page. Ideally the forensic analyst system should not be connected to the Internet – so the risk of fetching external links is minimal. However, allowing javascript is unlikely to work as many javascript pages synthesize web requests to dynamic web services.

Since we do not want the browser to re-fetch images from the original servers, in case these requests alert the server's administrator,[6] we want the images to be taken from the traffic itself. PyFlag is able to re-write URLs within HTML pages and redirect them back into retrieving the images from the VFS itself. Although PyFlag removes external links from the rendered page, it still reports on them to allow the analyst to see which links were removed.

The issue of HTML sanitization is a complex one (Pilgrim, 2006). HTML is a very flexible language and is closely integrated with javascript. PyFlag's approach is to parse the HTML page and produce a new page with a subset of tags and attributes which are considered safe. Any links are then resolved internally into the VFS if possible, or removed altogether. Note that the original link is always available to the analyst through a popup window which activates when the element is clicked on.

Javascript is not allowed to run, since it is difficult to constrain its actions.[7]

The result of this sanitization depends on the complexity of the page and the role javascript plays in it. For simple sites this is sufficient and the page is rendered very accurately. An example of this is shown in Fig. 3.

[6] Even if we did allow this, the images on the server may have changed since the capture was taken.

[7] Many javascript programs are able to synthesize a web request programmatically in a way which is difficult to predict using only static analysis.

```
while(1);

[[["v","8cfdg5ud8268","5","1a208671aae34cd5"]
,["ub",[["^i",1198934450019]
,["^f",1198934450019]
,["^k",1198934450019]
,["^all",1198934450019]
,["^t",1198934450019]
,["^r",1198934450019]
,["^s",1198934450019]
,["^b",1198934450019]
]
,1198934479193]
,["ugn","test test"]
,["cfs",[]
,[]
]
,["uiv",50]
,["st",1198934483]
,["cs","117260e1b9cdd085","117260f8b2aeba91",1,[]
,[]
,1198934344101859,"117260e1b9cdd085",_A("117260e1b9
cdd085")
,_A()
,[]
,[["117260e1b9cdd085",_A("^all","^f","^i")
]
]
]
,["ms","117260e1b9cdd085","",4,"test test
\u003ctest43421@gmail.com\>","test test","test43421
@gmail.com",1198934334000,"Hi there, This is a test
 email. Signed: Test",_A("^all","^f","^i") ,1,1,"This
is a test email",["117260e1b9cdd085",_A("test test
\u003ctest43421@gmail.com\>")
,_A()
,_A()
,_A()
,"This is a test email","Hi there,\u003cbr\>
 This is a test email.\u003cbr\>\u003cbr\>
Signed: Test\u003cbr\>\n",[[["0.1","beach.jpg",
"image/jpeg",126276,0,"f_fas6ipsc0","/mail/images/
graphic.gif","image/jpeg","?ui\u003d2&ik\u003db1ca
31c59b&realattid\u003df_fas6ipsc0&attid\u003d0.1&
disp\u003dthd&view\u003datt&th\u003d117260e1b9cdd085"
,"?ui\u003d2&ik\u003db1ca31c59b&realattid\u003df_
fas6ipsc0&attid\u003d0.1&disp\u003dattd&view\u003da
tt&th\u003d117260e1b9cdd085","?ui\u003d2&ik\u003db1
ca31c59b&realattid\u003df_fas6ipsc0&attid\u003d0.1
&disp\u003dinline&view\u003datt&th\u003d117260e1b9
cdd085",,,,,,]
```


Fig. 5 – A sample of a Gmail AJAX request. As can be seen the data passed from the server consists of a complex nested data structure. The exact format of this data structure is undocumented. Within the data structure the contents of the message can be seen. PyFlag extracts this information from the relevant array members, but no page rendering can be constructed.

6.1. *Browser cache compensation*

Sometimes the suspect's browser will contain some files required to render the page in its browser cache, and will not request those from the server. The browser cache may retain objects for very long periods of time – during which the browser will not request the objects from the web servers – but will use the cache in rendering the page. This presents a problem for the network forensic analyst because some of the cached images and style sheets are essential for the correct rendering of the page – but those are never transmitted over the network. This problem is exasperated when the duration of the capture is short relative to the cache expiry lifetime.

Fig. 4 shows the same page as in Fig. 3, but with the browser cache set to aggressively cache objects. As can be seen, the rendering is very poor because critical style sheets and images are missing.

Because PyFlag is able to integrate disk forensics and network forensic technologies together, a good option at this point is to obtain the target's web cache directory through standard forensic acquisition. PyFlag will parse the web cache into the VFS – at which point the images and style sheets from the network can be resolved.

The need for HTML sanitization stems from the desire to not alert suspects' servers with extra HTTP requests. However, in this case, clearly the missing objects can safely be fetched from the Hotmail site without the threat of alerting the suspect – leading to an improved page rendering.

PyFlag allows the user to selectively populate such *HTTP sundry* objects. These objects are selectively downloaded directly from their site[8] and are used to assist in rendering pages. The decision of fetching HTTP sundry objects is made by the analyst as a trade off between the increased risk of alerting the site administrator and the improved rendering. The HTTP sundry objects are added to the VFS as normal objects, which are identified using an inode prefix of *xHTTP*. Typically users would only download small images such as gifs and pngs or css files as HTTP sundries.

7. Web applications – web mail

In the previous section we saw how PyFlag is able to render HTML pages to emulate how the user viewed those pages during the time the capture was taken. This works well for simple web pages.

Recently, however, the wide proliferation of advanced web applications employing technologies such as AJAX (Ajax, 2008) to create highly interactive applications poses a problem for traditional rendering. Some of the most sophisticated applications such as *Gmail, google maps, Google Docs, Google Spreadsheet* are also of great forensic interest. Most web mail applications use some form of javascript with both *Yahoo!* and *Microsoft* offering full AJAX versions of their web mail portals.

7.1. *Example – Gmail*

The complexity of web applications makes accurate rendering difficult. For example, Gmail (Google Inc, 2008) has redefined the concept of a web application, traditionally consisting of a sequence of server generated HTML pages. Gmail is best thought of as a desktop application written in javascript and running inside the browser, which uses HTTP requests to communicate with its server.

When a user logs into Gmail, their browser loads an obfuscated javascript program. Once the application is loaded it runs, and makes requests to the server to retrieve the data it requires. The data is delivered using a proprietary and undocumented format carried over HTTP. An example is shown in Fig. 5.

[8] The script allows for generating an offline download script to fetch the files on a different system, and then import them into the case.

Case Management Load Data Configuration Disk Forensics Keyword Indexing Log Analysis Network Forensics Preview Test

Case: PyFlagTestCase1

2007-12-29 23:19:02	..56:1263|G1|l7	test test	None	None	None	Gmail is different. Here's what you need to know.	Messages that are easy to find, an inbox that organizes itself, gre spam-fighting tools and built-in chat. Sound cool? Welcome to Gmail To get started, you may want to: Learn about some of Gmail?s unique features on the Getting Started page. Follow our Switching Guide to learn how to announce your new Gmail address contacts, and forward your email from Yahoo! Mail, Outlook, Hotmail Set up your mobile phone to get super-fast access to Gmail. Visit our Help Center to find specific answers to all your questions. Users have often told us that the more they use Gmail, the more the its benefits. So go ahead and give it a try. We'll keep working on the best email service around, and we appreciate your joining us fo Thanks, The Gmail Team
2007-12-29 23:18:55	..923:559|G1|l5	None	"test test" ,			This is a test email	Hi there, This is a test email. Signed: Test

Fig. 6 – Gmail messages are listed in a table. PyFlag is unable to render the page since it is built dynamically on the client's browser – but this view provides the important information in one simple table.

In the case of Gmail there is no HTML carried over the HTTP requests. Forensic page rendering of Gmail sessions is a difficult task using static parsing alone since the pages are built by the javascript application.

Practically, however, the forensic examiner is less interested in the way the application looks, than in the high level information such as the messages sent, received and edited. The analogy is that when analysing SMTP and POP traffic, the analyst does not care what mail agent was used to send and receive the messages, and how they were presented on the user's screen as much as they care about the content of the message itself.

PyFlag's approach is to treat Gmail as a proprietary protocol layered over HTTP. PyFlag's Gmail scanner examines HTTP objects attempting to identify the Gmail data exchange protocol heuristically. The scanner then extracts metadata

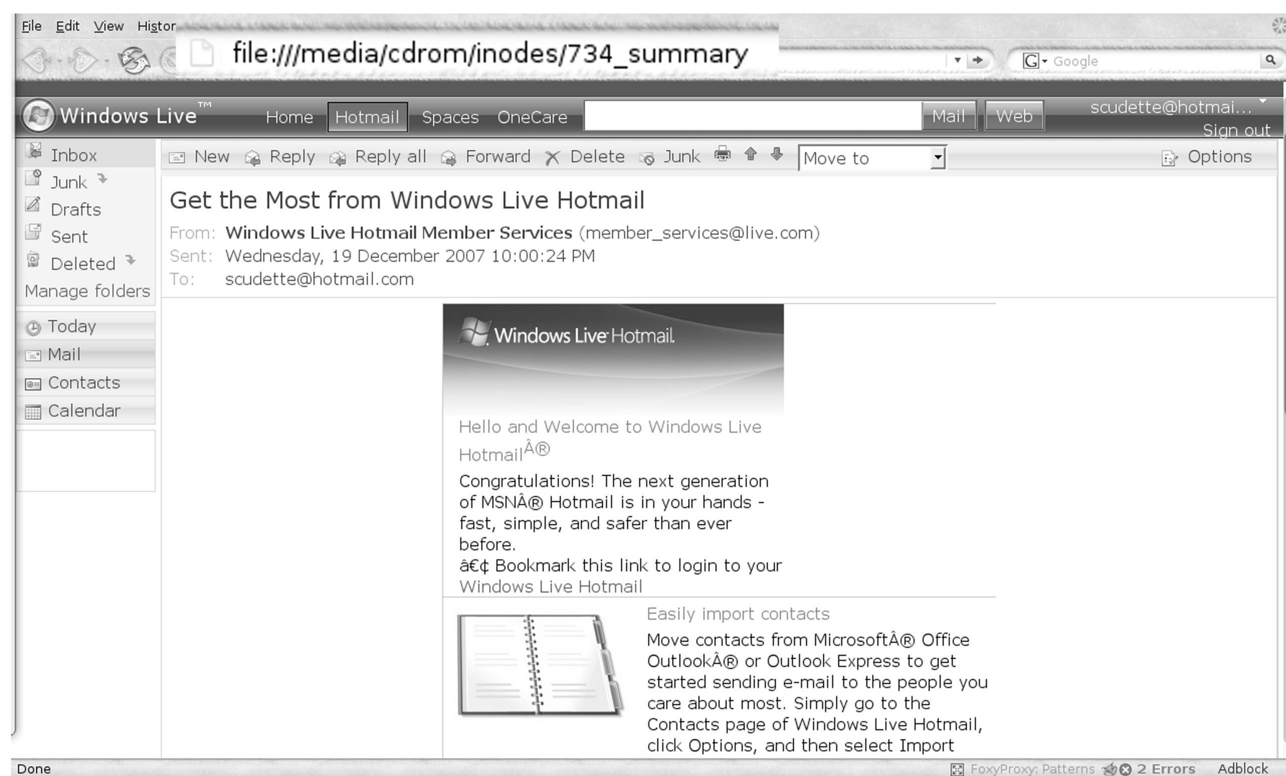

Fig. 7 – An inode from the report produced by PyFlag. Note that the URL to the page is referring to a local file. All images and style sheets used by the page are also stored locally as files. This allows the pages to be reviewed and distributed as a stand alone report.

from the protocol based on reverse engineering of the Gmail protocol.[9]

The messages are presented in a single table as shown in Fig. 6.

8. Presentation and reporting

Although the PyFlag GUI is suitable for interactively examining evidence, there are a number of cases where it is desirable to produce stand alone reports from PyFlag which are suitable for further distribution. For example, briefs of evidence are increasingly distributed in electronic form rather than paper form. In these situations it is desirable to produce a self-contained CDROM with the content of the analysis including links to the relevant inodes.

Fig. 7 shows an example of such a brief. Note that the URL in the example is file:///media/cdrom/inodes/300. The pages are all fetched from the CDROM with no external references. All internal links such as images and style sheets are also converted to local files and stored on the CDROM.

The CDROM may also contain any PyFlag table with suitable filters applied. Each inode stored on the CDROM is also explained (i.e., a description of how the data was arrived at is included – this allows third party verification of all results).

9. Conclusion

Network forensics is an increasingly useful tool for many investigations. Network forensics processing presents many challenges including the large number of protocols present on the Internet and the need for adapting to new protocols flexibly and quickly. The use of complex web applications such as web mail requires further processing beyond the HTTP protocol to support further decoding specific to the applications themselves.

PyFlag is emerging as a capable platform for network forensic analysis featuring advanced reconstruction of web pages. PyFlag provides for further application specific analysis for popular web mail sites.

REFERENCES

Ajax (programming), http://en.wikipedia.org/wiki/AJAX; March 2008.

Cohen M, Collett D. Python forensic log analysis GUI (PyFlag), http://www.pyflag.net; 2005.

Eeye Iris network security traffic analyser, http://www.eeye.com/html/Products/Iris/index.html; May 2008.

Free Software Foundation. Gnu general public license, http://www.gnu.org/copyleft/gpl.html; June 2007.

Google Inc. Gmail, http://mail.google.com; 2008.

Kloet B, Metz J, Mora R-J. Libewf – a library for support of the expert witness compression format, https://www.uitwisselplatform.nl/projects/libewf; May 2007.

Network Working Group. Hypertext transfer protocol – HTTP/1.1, http://www.w3.org/Protocols/rfc2616/rfc2616.html; 1999.

Pilgrim M. HTML sanitization [universal feed parser], http://www.feedparser.org/docs/html-sanitization.html; 2006.

Ptacek TH, Newsham TN. Insertion, evasion and denial of service: eluding network intrusion detection system, http://www.snort.org/docs/idspaper; Jan 1998.

Snort – the de facto standard for intrusion detection/prevention, http://www.snort.org; March 2008.

Song, D. http://www.monkey.org/dugsong/fragroute/Fragroute; 2001.

Tcpdump, http://www.tcpdump.org/Sep; 2007.

Communications interception, analysis, and service provider compliance, http://verint.com/communications_interception; May 2008.

Wireshark, http://www.wireshark.org/Feb; 2008.

Wojtczuk R. Libnids. Version 1.23, http://libnids.sourceforge.net; Feb 23, 2008.

Dr. Michael Cohen is currently a Data Specialist within the Australian Federal Police. He previously worked for the Information Security Group in the Defence Signals Directorate as a Senior Technical Adviser. Dr. Cohen specialises in Digital Forensics and Telecommunication Intercept analysis. His hobbies include Python, C, and riding his bike.

[9] Since the Gmail application is downloaded each time the user logs on, Google is able to change the protocol at a whim without the need to maintain backwards compatibility. This implies that PyFlag's parser needs to be highly adaptable to new versions.