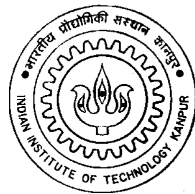
PickPacket: Design and Implementation of the HTTP postprocessor and MIME parser-decoder

A Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Bachelor of Technology

by

S. Prashanth Aditya



to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

January, 2003

Certificate

This is to certify that the work contained in this report entitled "*PickPacket: Design and Implementation of the HTTP postprocessor and MIME parser-decoder*", by *S. Prashanth Aditya*, has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

January, 2003

(Dr. Dheeraj Sanghi)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

(Dr. Deepak Gupta)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

Abstract

The proliferation of computers and the Internet has simplified global information exchange to such an extent that there is a scope for misuse or abuse of the Internet for malicious anti-social or anti-national purposes. An effective protection mechanism is needed to counter such designs if any. A good monitoring tool to scan network traffic for potentially harmful information will go a long way in achieving this goal. Monitoring tools can also be handy for network administrators to diagnose problematic network services and hardware. Companies can use such tools to safeguard their information repositories and research efforts, in addition to preventing abuse of network facilities by employees. The role of such monitoring tools in intelligence gathering is also profound, especially when information exchange is as easy as sending a mail which might look innocuous at first hand. Thus there is a pressing need to monitor, detect and analyze undesirable network traffic. However, this need may conflict with the need to maintain the privacy of individuals whose network communications are being monitored.

Reference [4] discusses PickPacket, a network monitoring tool that handles these twin issues effectively. PickPacket has four components - the *PickPacket Configuration File Generator*, the *PickPacket Filter* that captures packets, the *PickPacket PostProcessor* that analyzes captured packets, and the *PickPacket Data Viewer* that shows the captured data in a human-readable form. This report discusses an extension to the SMTP [5] packet filter that parses, filters and decodes multipart MIME [3] messages, and the postprocessing of captured HTTP [2] packets in PickPacket.

Acknowledgments

I take this opportunity to express my gratitude to my thesis supervisors Dr. Dheeraj Sanghi and Dr. Deepak Gupta. Their guidance ensured the smooth progress of the project and that my efforts were always in the right direction. I would also like to thank the other team members involved in the project - Neeraj, Brajeshji, Sanjayji, Abhay, Nitin and Ankit for their cooperation, support and the excellent rapport they shared with me. Abhay, Nitin and Ankit were instrumental in performing exhaustive tests on PickPacket and helped iron out all the bugs in the HTTP postprocessor. The help I received from Neeraj, Brajeshji and Sanjayji cannot be expressed in simple words. In spite of being seniors to me in age and experience, they never once let me feel so and were as friendly as any of my peers. Sanjayji in particular saw to it that I woke up regularly and in time for meetings and demonstrations after spending whole nights working and preparing things. I can never forget the time I spent on this project with them. I remember Neeraj telling me about his plans to open-source this project. It's sad that he's not around anymore. May his soul rest in peace. I am also grateful to my juniors Diwaker and Anurag for putting up with all my idiosyncrasies about the test machines, especially around the time of the mid-term demonstration.

This project would not have seen the light of day if it were not for the generous funding by the Department of Information and Technology of the Government of India. My thanks to them for having supported this project since its inception.

I also thank all my Professors here who have taught me valuable lessons not only in their courses but on life in general.

My friends in general and wingmates in particular were absolutely fantastic. They have all done their bit in making my stay here in IIT Kanpur a memorable one.

Last and definitely not the least, I thank my parents and grandparents. They were the people I could always turn to if nothing else would help. I can never do enough to return back to them all that they've given me.

Contents

1 Introduction **1**
1.1 Sniffers 2
1.2 PickPacket 3
1.3 Organization of the Report 4

2 PickPacket: Architecture and Design **5**
2.1 The PickPacket Configuration File Generator 6
2.2 The PickPacket Filter 7
2.2.1 PickPacket Filter: Output File Formats 9
2.2.2 PickPacket Filter: Text String Search 10
2.3 The PickPacket PostProcessor 10
2.4 The PickPacket Data Viewer 12

3 Design and Implementation of the HTTP postprocessor in PickPacket **13**
3.1 A brief description of HTTP 13
3.1.1 HTTP Resources: URLs and URIs 14
3.1.2 HTTP Transactions: Headers 14
3.1.3 HTTP 1.1: Additional features over 1.0 15
3.2 HTTP postprocessor: Goals 16
3.3 HTTP postprocessor: Design and Implementation 17
3.3.1 Parsing HTTP Packets 19
3.3.2 Request-Response pairing 21

4 The MIME parser-decoder: An extension to the SMTP filter in PickPacket **22**

4.1 The need for MIME 22

4.2 MIME headers and format of message bodies 23

4.3 MIME parser-decoder: Goals 27

4.4 MIME parser-decoder: Design and Implementation 28

5 Handling non-consecutive packets **31**

5.1 Messages without a boundary 32

5.2 Messages having a boundary 33

6 Testing and Results **35**

6.1 Testing the HTTP postprocessor 35

6.2 Testing the MIME parser-decoder 35

7 Conclusions **37**

7.1 Scope for further work 38

Bibliography **40**

A List of all HTTP packet parser states **41**

B List of all MIME filter parser states **43**

List of Figures

2.1 The Architecture of PickPacket 5
2.2 Filtering Levels 7
2.3 Demultiplexing Packets for Filtering 8
2.4 Some Components of a Filter 9
2.5 The Basic Design of the PickPacket Filter 10
2.6 Post-Processing Design [4] 12

Chapter 1

Introduction

The proliferation of computers and the Internet has simplified global information exchange to such an extent that there is a scope for misuse or abuse of the Internet for malicious anti-social or anti-national purposes. An effective protection mechanism is needed to counter such designs if any. A good monitoring tool to scan network traffic for potentially harmful information will go a long way in achieving this goal. Monitoring tools can also be handy for network administrators to diagnose problematic network services and hardware. Companies can use such tools to safeguard their information repositories and research efforts, in addition to preventing abuse of network facilities by employees. The role of such monitoring tools in intelligence gathering is also profound, especially when information exchange is as easy as sending a mail which might look innocuous at first hand. Thus there is a pressing need to monitor, detect and analyze undesirable network traffic.

However, this may conflict with the need to maintain the privacy of individuals whose network activities are being monitored. This report describes PickPacket, a network monitoring tool that can address the conflicting issues of network monitoring and privacy through judicious use, and two components of PickPacket - the HTTP [2] postprocessor and the MIME [3] parser-decoder.

Reference [4] discusses a framework for PickPacket.

1.1 Sniffers

Network monitoring tools are also called "sniffers". Several tools exist that can monitor network traffic. Usually such tools put the network card of the computer (running the tool) into "promiscuous mode". This enables the computer to "listen" to all the traffic on that section of the network. These packets can then be "filtered" based on the IP-related header data present in the packets. Usually such filtering involves the specification of simple criteria like the IP addresses and ports to look for in the packets. Filtered packets are "dumped" on to disk. The captured packets are analyzed to gather the required information.

A network adapter hosts a chip that rejects all packets whose destination MAC addresses are different from that of the adapter. Sniffers work by instructing the network adapter driver to disable this feature of the adapter. Once this is disabled, the adapter can receive all packets that come through the wire or segment on which it is present.

The disadvantages of such simple sniffing and filtering are many in number. Firstly, with simplistic filters, the amount of captured data on a very busy network segment would be too much. Secondly, no filtering is done on the basis of the content of the packet payload. Thirdly, as the entire data is dumped to the disk the privacy of innocent individuals who may have been using the network during the time of monitoring may be violated. These disadvantages of conventional sniffing motivate the design and implementation of PickPacket.

PickPacket uses in-kernel filters, derived from the BSD Packet Filter (BPF) [6]. The idea behind in-kernel filtering is that all packets first travel up the kernel's TCP/IP stack before getting delivered to a user-space application that understands the packet. If the packet filtering rules are applied before the packet reaches the user-space application (PickPacket, for instance), there will no more be a context switch for every packet that is received by the network interface. Further, a large number of packets will get discarded at the kernel level itself if they don't satisfy the IP-related criteria put down by the user. This makes the whole filtering process far more efficient than if the user-space application did everything by itself. The

higher levels of filtering, which are application-based filtering and sniffing application content, are done by the sniffing tool itself. Sometimes, the in-kernel filtering code might have to change dynamically. Typical examples of such issues involve monitoring FTP transactions where each file is transferred over a different data connection (the ports vary and so the in-kernel filter should change to monitor this connection and look for the new ports in the data packets), or RADIUS and DHCP transactions where the IP addresses of the hosts on the segment may change even during the monitoring/filtering period. In these cases, there's an overhead involved in dynamically generating and using the in-kernel filter.

Sniffers dump captured data onto disk directly without any processing of this data. As such, this dump is not human-readable. Sniffers therefore come bundled with their own post-capture analysis and processing tools which extract information from the dump and present it in a human-readable manner. In addition to just presenting the sniffed data, packet analyzers can be configured to provide different kinds of functionality like alerting network administrators if something has gone amiss.

1.2 PickPacket

The purpose of PickPacket is to monitor network traffic and to copy only selected packets for further analysis. It allows for the provision of a good number of filtering criteria. These can be specified for multiple layers of the protocol stack. There can be criteria for the Network Layer (IP address specification), Transport Layer (TCP and/or UDP and port numbers) and Application Layer (application dependent criteria such as filenames, emailids, URLs, text strings to be searched for etc.). The filtering component of this tool does not inject any packets into the network. Once the packets have been selected based on these criteria they are dumped onto disk.

The filter can be operated in any one of two modes called "PEN" or "FULL". The first mode is good enough to ascertain that a packet corresponding to a particular criterion specified by the user was encountered and minimal information is captured. In the second mode the data or content of such a packet is also captured. Using

these features with discretion can help protect the privacy of innocent users.

The packets dumped to the disk are analyzed offline. Separate files pertaining to the different connections monitored are output by the post-capture analyzer. PickPacket provides a summary of all the connections and also provides an interface to view the captured traffic in a human-readable manner. This interface uses existing software extensively to render the captured data. For instance, when rendering captured e-mail, Outlook Express may be used through the interface provided. A GUI for generating the rules that are input to the filter is also provided.

1.3 Organization of the Report

This report treats in detail the subject of analyzing captured Hypertext Transfer Protocol (HTTP) [2] packets and sniffing Multipurpose Internet Mail Extensions (MIME) [3] attachments in Simple Mail Transfer Protocol (SMTP) [5] packets. This was the scope of the work covered by this report. Chapter 2 describes the high level design and architecture of PickPacket. Chapter 3 discusses the post-capture analysis (postprocessing) of captured HTTP traffic and Chapter 4 elaborates on the design and implementation of MIME content that appears as attachments to email (SMTP). Chapter 5 deals with the testing of these components and results obtained. The final chapter concludes the report with suggestions on future work that can be done on this project. The three appendices A, B and C include a sample configuration file, details of the record files describing the postprocessed HTTP output and the base64 alphabet respectively.

Chapter 2

PickPacket: Architecture and Design

PickPacket can be viewed as an aggregate of four components - the *PickPacket Configuration File Generator*, the *PickPacket Filter*, the *PickPacket PostProcessor* and the *PickPacket Data Viewer*. A graphical representation of PickPacket's architecture is shown in Figure 2.1 where these components are shown in rectangles. In this scenario of usage, where each of the four components is given a separate

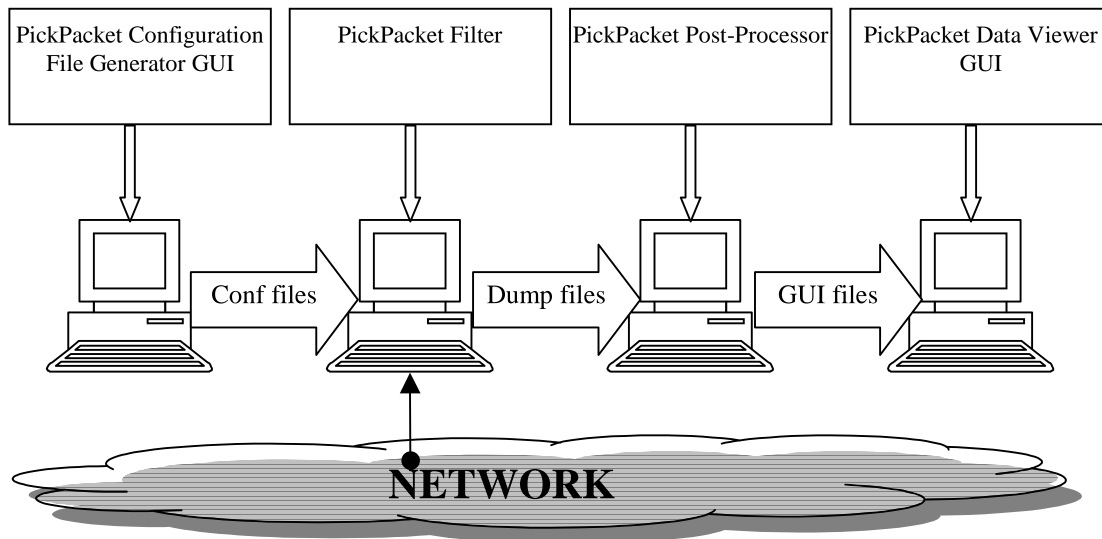


Figure 2.1: The Architecture of PickPacket
[7]

machine to execute on, the PickPacket Configuration File Generator would prepare a configuration file that would be transferred to the machine where the PickPacket Filter would run. The PickPacket Filter captures packets according to the criteria specified in the configuration file and stores them. The stored packets are transferred to the machine hosting the PostProcessor for postprocessing and analysis. The PickPacket PostProcessor would typically run on some machine other than the one on which the PickPacket Filter runs. The task of the PostProcessor is to break the dumped data into separate connections and retrieve that information from the captured packets which is necessary for showing the captured data through a user-friendly windows-based GUI. After postprocessing and analysis the PickPacket Data Viewer GUI shows the results.

2.1 The PickPacket Configuration File Generator

The PickPacket Configuration File Generator is a Java-based GUI. It is used to generate the configuration file that is input to the PickPacket Filter. This file is a text file with HTML like tags. While the Configuration File Generator is good enough for specifying Output File Manager criteria, Basic criteria and Application Level Protocol-specific criteria, advanced users might want to edit by hand the other criteria, namely the specification of the number of packets to store from each connection before discarding it (if a match doesn't occur) and the number of connections of each application level protocol to monitor.

The Output File Manager criteria specify the name of the dump file that should be the Filter's output and the size of that file. The Basic Criteria include IP address, port numbers and protocol specifications. The Application Level Criteria include specifics like email addresses (SMTP), usernames (FTP and Telnet), hostnames (HTTP) and text string specifications.

2.2 The PickPacket Filter

The basic functionality of the Filter lies in reading packets from the network and applying the criteria specified by the configuration file on these packets. If a match occurs, the connection in which the matching packet was found would be dumped. As mentioned in Chapter 1, sniffers operate at various levels along the protocol stack. At the first level, packets are filtered based on network parameters like IP addresses and port numbers. The next levels involves looking at application level protocol headers for matches against metadata-like criteria like usernames or hostnames. The third level of packet filtering looks at the content of application level protocol packets.

Again as mentioned in Chapter 1, the second and third levels are left to a user space application because no operating system has an application layer protocol built into its protocol stack, for the most part. But the first level, if done by the kernel internally, would be extremely effective. Figure 2.2 illustrates this organization of

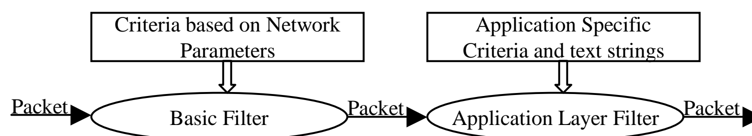


Figure 2.2: Filtering Levels
[7]

levels of filtering. The second and third levels of filtering are combined into the *Application Layer Filter.*

Different application layer protocols would need different application layer filters. In such a situation, the application layer filter in the above figure is split using a *Demultiplexer* into several filters. Each filter would then be handling its own application layer protocol. The Demultiplexer follows certain rules to identify the route each packet should take. This organization is illustrated by Figure 2.3.

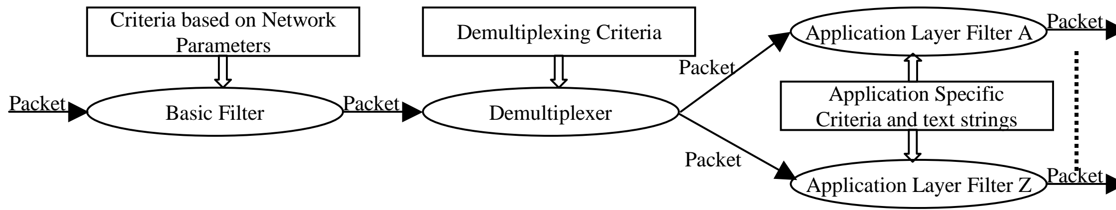


Figure 2.3: Demultiplexing Packets for Filtering
[7]

So we have at least the following components now in a filter equivalent to PickPacket's Filter - a Basic Filter, a Demultiplexer and several Application Layer Filters.

Packets over the network cannot always be assumed to be complete in the sense that the protocol data may be split across multiple packets which could be obtained out of sequence by the filter. Now considering the third level of filtering in the very first filter model elaborated in this section, it is obvious that a mechanism is required to ensure that whenever there is a search for text strings *within* the packet content, the filter should be on the watch for some packets arriving out of sequence. This mechanism is provided by what is called the *TCP Connection Manager* in PickPacket. So now there's one more component to PickPacket's Filter. This is illustrated in Figure 2.4.

An interesting design aspect is that the connection manager should not set about determining the sequence of packets for all connections. Whenever an application layer filter faces a problem, it can raise a signal which can be caught by the connection manager, following which it will determine the sequencing of packets for the connection in question.

The discussion so far prepares the ground for the basic design of the PickPacket Filter. Figure 2.5 illustrates the basic design of the Filter.

The *Initialize* component initializes the filter by reading the configuration file. The *Output File Manager* is another component which takes care of dumping the captured packets to disk. It has a set of criteria to work on too, namely the name of

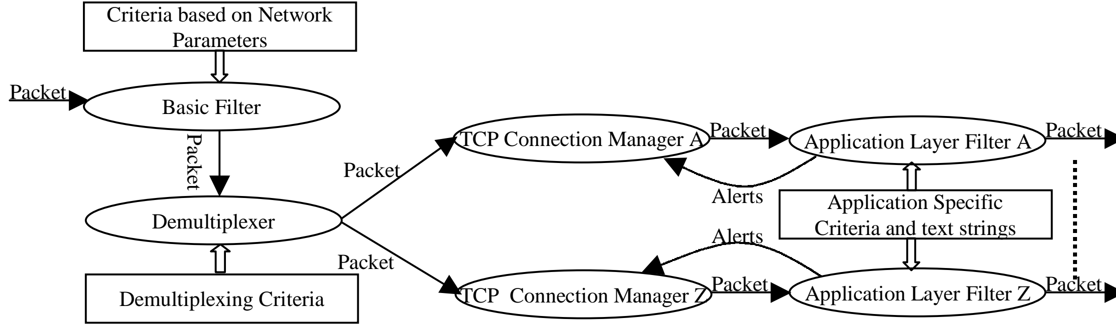


Figure 2.4: Some Components of a Filter
[7]

the dumpfile and its size. The in-kernel BPF code is generated by another module called the *Filter Generator*. In case no application layer criteria or text string search criteria are given in the configuration file, the *Demultiplexer* may call the *Output File Manager* directly and dump the packets without having to call the application layer protocol filters. The *Connection Manager* also avails of this feature. This feature is also required when all criteria have matched for a specific connection and the connection is still open. In this case, the remaining packets of the connection have to be dumped simply, as per design. Reference [4] discusses all these components in greater detail.

2.2.1 PickPacket Filter: Output File Formats

Conceptually, the *output file manager* can store files in any format. However, PickPacket stores output files in the *pcap* [9] file format. This file starts with a 24 byte pcap file header that contains information related to version of pcap and the network from which the file was captured. This is followed by zero or more chunks of data. Every chunk has a packet header followed by the packet data. The packet header has three fields - the length of the packet when it was read from the network, the length of the packet when it was saved and the time at which the packet was read from the network.

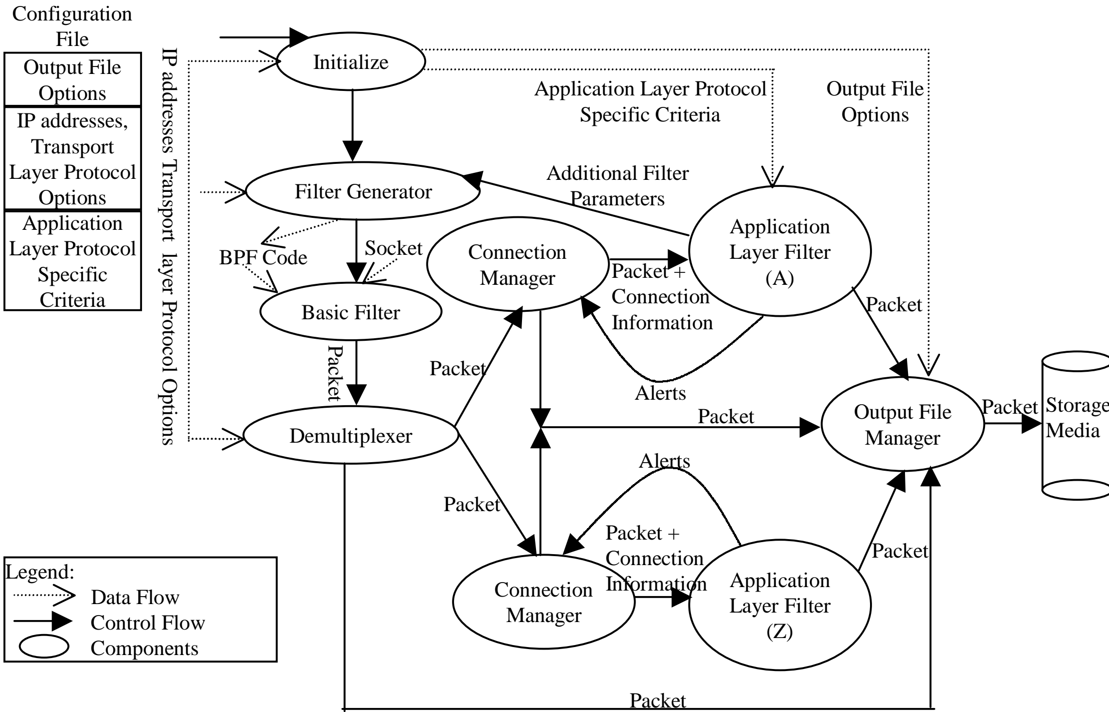


Figure 2.5: The Basic Design of the PickPacket Filter [7]

2.2.2 PickPacket Filter: Text String Search

The PickPacket Filter contains a text string search library. This library is extensively used by application layer filters in PickPacket. This library uses the *Boyer-Moore* [8] string-matching algorithm for searching text strings. This algorithm is used for both case sensitive and case insensitive search for text strings in packet data.

2.3 The PickPacket PostProcessor

The PostProcessor is an offline Linux-based analyzer that works on the dump generated by the Filter. This dump is in libpcap format. The PostProcessor needn't have anything to do with the configuration file, except place it along with all the

processed data in the directory associated with that session of monitoring. Therefore the job of the PostProcessor is to just look at all the dumped packets, separate them into various connections (multiple connections could have been monitored in one session) and process them after removing duplicate packets. The packet separation also involves separation based on the transport or application layer protocol. This part is done by the *Connection Breaker* and *Sorter* components of the PostProcessor. The *Sorter* is also responsible for rearranging packets which were received out-of-order from the network on the basis of the timestamp values corresponding to the time the packets were received. The above modules simulate a TCP state machine to separate and sort the packets.

After this stage, the PostProcessor extracts various pieces of information from the connection (or tuple)-wise sorted packets. This includes the TCP connection information as well as application layer protocol metadata like usernames (FTP and Telnet), hostnames (HTTP) along with the actual protocol content that was transferred. The *Information Retriever* component of the PostProcessor is responsible for this aspect of its functionality. The output generated by this module is the final output of the PostProcessor. An elaborate and precise directory structure is created in the working directory of the PostProcessor. Each monitoring session has its own metadata and protocol content dump directory. Within this directory, the PostProcessor fills up files with protocol content. These files also have the relevant extension (for instance, .eml for SMTP or mail content) which would enable them to be opened natively by a certain Windows application. For more details on this, please refer to the next section.

The PostProcessor also fills up other kinds of information in this directory, including the server-client dialogue files and the metadata and TCP connection record files.

The three components are shown in Figure 2.6.

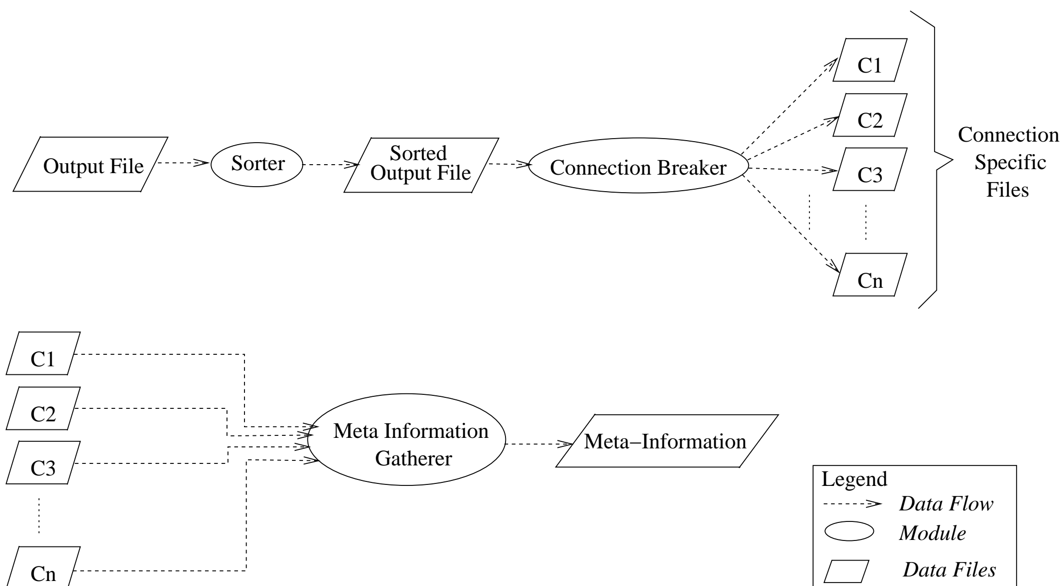


Figure 2.6: Post-Processing Design [4]

2.4 The PickPacket Data Viewer

The PickPacket Data Viewer works on the connection and metadata records and the the protocol content extracted by the PostProcessor. This component is a Visual Basic GUI which runs on Windows. The reason behind choosing Windows as the platform for this component was the coordination between the file types and application-handling programs that exists on this platform. The user simply has to launch the Data Viewer, select the connection of his choice and double-click it to launch the relevant application (say MS Outlook Express in case of an SMTP transaction) to view the contents of that transaction.

The rest of this report focuses on two specific application layer modules of PickPacket - the MIME parser-decoder, an extension to the SMTP filter and the HTTP postprocessor.

Chapter 3

Design and Implementation of the HTTP postprocessor in PickPacket

This chapter discusses the design and implementation of the HTTP postprocessor. First, the protocol is described with a focus on those features that are of interest in designing and implementing the postprocessor. The later part of the chapter deals with the design and implementation of the postprocessor, followed by a description of the HTTP packet parser that is used by the postprocessor to parse HTTP packets and retrieve the required information.

3.1 A brief description of HTTP

HTTP is the most widely used mechanism to deliver files and other data (called resources) on the World Wide Web. It makes use of TCP/IP sockets. The HTTP client is a resource "browser" that sends requests to an HTTP server and elicits a response in return. HTTP servers by default listen on to port 80, though they can use any other port. It is very common to see HTTP servers running on ports other than the standard port 80.

3.1.1 HTTP Resources: URLs and URIs

HTTP transmits resources. Files are also a kind of resource. A resource is some information that is identified by or pointed to by a Uniform Resource Locater (URL) [1]. The most common kind of a resource is a file, but a resource may be a dynamically generated query result, like the output of a server-side CGI script. When some data that is interpreted by a server is attached to the URL it is called a Universal Resource Identifier (URI) [1].

3.1.2 HTTP Transactions: Headers

HTTP transactions are called requests and responses. Requests are generated by an HTTP client and responses to requests are generated by an HTTP server. The format of the requests and responses is similar. Both consist of a line called the request line or the response line correspondingly, zero or more header lines, an empty line and a message body which may or may not be used.

All HTTP header lines adhere to RFC 822 specifications for internet message headers. Therefore header lines are less than 1000 characters long and end in a CRLF (even an LF will do). There can be whitespace between different parts of a header. The header string will be case-insensitive. The same is not the case with the header value.

The request line has three parts - a method name, the path to the requested resource, and the HTTP version in use. Each part is separated by a space. Method names and versions are in upper case. A typical request line is:

```
GET /pub/linux/gentoo/distfiles/index.html HTTP/1.1
```

There are several possible methods the most common of which are GET, POST, PUT and CONNECT.

The response line is also called the status line. This line also has three parts - the HTTP version, a response status code specifying the result of the request, and a reason phrase - separated by spaces. An example status line is:

```
HTTP/1.1 200 OK
```

Header lines provide information about the request or response, or about the content of the message body. HTTP 1.0 defines 16 headers and none are required to be present in the packet. HTTP 1.1 defines 46 headers, and the"Host:" header is required to be present in requests.

The item mentioned above as the optional message body contains the HTTP payload, which could be data the user is uploading (in case of HTTP PUT and POST requests) or the reply from the server (in case of responses to HTTP GET and CONNECT requests). If an HTTP message includes some content, there are header lines in the message that describe the content. The headers that are of interest to the HTTP postprocessor are the Content-Length and Transfer-Encoding headers. The Content-Type header is not of any importance to the HTTP postprocessor.

3.1.3 HTTP 1.1: Additional features over 1.0

HTTP 1.1 has also been defined and it brings quite some changes by way of improvement to HTTP 1.0. It is a superset of HTTP 1.0 in that all the functionality and syntax and headers supported by 1.0 are also available in 1.1. The suggested improvements come by way of "persistent connections", support for caching, "chunked encoding" of response data and the ability to serve multiple domains with one IP address. The first through third are all features to improve transfer performance. For instance, if data is sent chunk-by-chunk, the server can start sending data from a dynamically generated data set even before the set has loaded completely at the server's side. This breaks the complete response into smaller chunks and sends them in series. Such a response can be identified as it contains the "Transfer-Encoding: chunked" header. A chunked message body contains a series of chunks, followed by a line with 0, followed by optional footers (which are just like headers), and a blank line. Each chunk consists of two parts - a line which specifies the size of the chunk in hex followed by a semi-colon and some information which is not of much use and ends in a CRLF, and the chunked data again ending with a CRLF. An example of a chunked response is given below:

```
HTTP/1.1 200 OK
Content-Type: text/plain
```

```
Transfer-Encoding: chunked

1a; parameters - can be ignored e.g. charset information
The quick brown fox jumped
12
over the lazy dog.
0
footer1: value1
footer2: value2
[blank line]
```

A response equivalent to the above response, without using chunked encoding, is shown below:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 45
footer1: value1
footer2: value2
.
.
.
The quick brown fox jumped over the lazy dog.
```

This concludes a brief description of HTTP, which is enough to lay the groundwork towards elaborating on the design and implementation of the HTTP postprocessor and especially the packet parser. The rest of the chapter discusses the HTTP postprocessor.

3.2 HTTP postprocessor: Goals

The HTTP postprocessor works on the output of the HTTP filter. It works on the packets which are dumped by the filter onto disk. It needn't examine the configuration file for any criteria. All it has to do is extract information from the dump and

present it in a format which the Data Viewer can show in a user-friendly manner. The detail of the dump depends on the mode in which packets were captured by the filter. The output of the postprocessor should contain the TCP connection information (source and destination IP addresses and ports, and timestamps) and metadata information (Hostname, remote URI, local URI - the path to the processed content on disk).

Unlike the PickPacket Filter components, there isn't too much emphasis on speed and efficiency when it comes to the postprocessor because it will be analyzing the packets offline.

3.3 HTTP postprocessor: Design and Implementation

The HTTP postprocessor allocates a structure for each connection. This structure holds the information pertaining to that connection. Important members of this structure are the response and request structures. These structures have several parse states that are set by the HTTP parsers. There is a parser for parsing request packets and another parser for parsing response packets. The parsers and the state machines they use are explained in a detailed manner in the next section.

The HTTP packet parser steps through the packet data and returns after setting states for the request (or response) structure discussed above. Data may be left in the packet after parsing because of pipelining or errors. In the event of such an error, error recovery mechanisms ensure that all current states are set to none and the parser starts looking for a fresh request (or response) in the subsequent packet data. After the parser returns further processing might be necessary if partial content of a request or response has been received. The parser may be able to retrieve partial content in cases where the message body is split across packets.

Once all the packets in a connection have been parsed, the HTTP postprocessor appends TCP connection and HTTP metadata records to the corresponding record files in an output directory. The metadata record contains the local (ondisk) URI to the content of the request or response. The postprocessor also creates one directory

for each connection to store the content captured in that connection. The TCP connection records are stored in a file with the .http_cr extension while the HTTP metadata records are stored in a file with the .http_dr extension.

In the case of HTTP 1.1, there could be pipelined requests and responses. The postprocessor handles this by creating a directory (in the corresponding connection's directory) for each host involved the monitored transaction. Within this directory, the content that has been retrieved from the server (or sent from the client in the case of POST requests) is dumped, maintaining the exact directory structure that can be found in the server's root area. This is the output format used in the case of HTTP 1.0 also, to maintain uniformity.

The postprocessor also generates three conversation files for each connection. One of them contains all the commands sent by the client and is appropriately given a .C extension on disk. Similarly, there is a .S file which contains all the responses sent by the server during that connection. The third file, which has the extension .CONV contains the complete two-way dialog. These files contain only the HTTP requests and responses and not the TCP SYN/ACK-SYN/ACK/FIN packets.

Finally, the HTTP postprocessor writes a .pkt file into the output directory. This file contains the names of the configuration file used for that session (the configuration file is also copied to that directory) and the names of the TCP connection record and HTTP metadata record files. It is this file which the PickPacket Data Viewer looks for when a user wants to view the dumped and processed content.

Examples of TCP connection and metadata records, along with an explanation of the record format are provided in below:

```
Format of a record in the TCP connection record file (.http_cr):
----------------------------------------------------------------
ConnID;SrcMAC;DstMAC;SrcIP;DstIP;SrcPort;DstPort;ConnID.CONV;
ConnID.S;ConnID.C;hostname;date and time in human-readable form;
seconds component of start timestamp (seconds after Unix Epoch);
milliseconds component of start timestamp;
seconds component of end timestamp (seconds after Unix Epoch);
milliseconds component of ending timestamp;
```

```
Example:
--------
1;0:80:ad:1:d7:9b;0:0:e8:4a:8c:a5;172.31.19.7;203.200.95.130;
32917;3128;1.CONV;1.S;1.C;http://www.gnu.org;
Tue Jul 16 11:06:36 2002;1026797796;673285;1026797796;753285;

Format of a record in the HTTP metadata record file (.http_dr):
----------------------------------------------------------------
ConnID;SrcMAC;DstMAC;SrcIP;DstIP;SrcPort;DstPort;ConnID.CONV;
ConnID.S;ConnID.C;hostname;method;remoteURL;localURL;

Example:
--------
1;0:80:ad:1:d7:9b;0:0:e8:4a:8c:a5;172.31.19.7;203.200.95.130;
32917;3128;1.CONV;1.S;1.C;http://www.gnu.org;GET;
/graphics/gnu-head-sm.jpg;
httpdump_gui/1/www.gnu.org/graphics/gnu-head-sm.jpg;
```

3.3.1 Parsing HTTP Packets

The HTTP packet parser is one of the core components of the postprocessor. The structure of HTTP transactions has been discussed in Section 3.1.2. The parser was designed keeping in mind that in the case of HTTP 1.1, a packet can have more than one request or response, and also that responses are typically split across several packets because the size of the application layer protocol content is lesser than 1500 bytes (the standard ethernet frame length) and most HTTP content is definitely much bigger than this.

To take care of the two things mentioned above, as also to make parsing of a multitude of (mostly unwanted) headers, a state machine was developed for the parser. The states used by this machine were designed keeping in the mind the structure of HTTP headers. Each header consists of a header string followed by a delimiter,

followed by the header value and a CRLF. There could also be Linear White Space (LWS or LWSP) at the beginning of a header or between the header string and value. The parser must roughly take care of the BNF involved in defining header structures. At the top level, the states could be identified as (in the case of a request) - parsing the request line, parsing a header, parsing the protocol content, processed the request or no state. These five states are captured by the following enumerated values - NONE, PARSE_REQ_LINE, PARSE_HEADER, PARSE_MESSAGE, PROCESSED, and ERROR. Similarly the parser has the following top-level response parser states - NONE, PARSE_RES_LINE, PARSE_HEADER, PARSE_MESSAGE, PROCESSED, and ERROR. Corresponding to each state the parser has substates that define the amount of parsing of a particular packet by the parser at a slightly lower level. So if there was a PARSE_REQ_LINE, there could be substates which reflect whether the parser has already got the method or the URI. The parser has even lower level states, which will be called subsubstates here for lack of a better name. These are used to indicate the extent and status of parsing line delimiters at the end of each header in either a request or a response. *Appendix A* contains a list of HTTP packet parser states.

States are initialized once to start the parser. After that the parser examines each packet and sets appropriate states. Subsequent calls to the parser use the old state that has been set by the parser. In case an ERROR state is set the postprocessor tries to recover from this state by skipping to the first method or the first response in the packet depending on context. This strategy takes care of the first requirement placed on the parser. The second requirement is met by calling the parser in a loop till the entire data of the packet has been consumed by the parser. The parser, while in the state PARSE_MESSAGE also takes care of chunked encoding. Chunked-data or the content data as may be specified by HTTP headers is suitably unchunked and copied to a buffer. String searches for user specified strings are carried out within this buffer.

3.3.2 Request-Response pairing

In the case of HTTP 1.0 transactions, pairing requests and responses does not pose a problem because each connection sees only one request and one response. However, in the case of HTTP 1.1, with persistent connections and pipelining of requests and responses, it becomes important to come up with a robust technique to pair up (even approximately) in some cases requests and responses. Another difficulty faced in this context is that HTTP responses do not contain a copy of the URI or filename of the requested file/resource. Request-response pairing is important because the user might like to compare the locally saved copy and the remotely available copy of some resource while using the Data Viewer. The Data Viewer has to show locally saved URIs in terms of the original request (for details, refer to the format of a .http_dr record given above - note that the record contains both the local and remote URIs in its fields).

The strategy followed by the PickPacket PostProcessor is to pair up each request seen with the next immediate response seen. While this will definitely work over an uncongested, otherwise normal, network if the sniffing was started before the occurrence of any HTTP transactions, this might fail if the sniffing started in the middle of a pipelined, persistent transaction.

This completes the discussion on the HTTP postprocessor in PickPacket. In this chapter, the design and implementation of the HTTP postprocessor based on the structure of HTTP transactions was presented. Goals met by the HTTP postprocessor were also defined.

Chapter 4

The MIME parser-decoder: An extension to the SMTP filter in PickPacket

This chapter discusses the design and implementation of the MIME parser-decoder in PickPacket. In its first release, PickPacket did not incorporate the functionality of searching for text strings in MIME-encoded data. In this chapter, MIME is described, focussing on those aspects of the MIME specification that have a direct bearing on the design and implementation of the parser-decoder. Then the details of the design are presented, followed by implementation features.

4.1 The need for MIME

When email (SMTP) was first designed and implemented, multimedia wasn't very widespread. In fact, it was more or less non-existent. Messages bodies and headers were encoded in flat US-ASCII. RFC 822 offers more details on this early standard of message body formats. With the advent of character sets other than US-ASCII and the need to transfer content other than plain textual data, the specifications for MIME were laid down. The core features of MIME include allowing for textual message bodies and headers in character sets other than US-ASCII, a set of formats

for non-textual message bodies, a framework for the message body to be split into multiple parts. In fact, a mail containing an attachment or more is actually a multipart message, with different formats used to represent the headers and content of each part or attachment.

4.2 MIME headers and format of message bodies

MIME defines a number of new RFC 822 headers used to describe the content of a message part. The individual entities (headers and the content of a part constitute an entity) in a multipart message are separated by a boundary string, which is specified by the first "Content-Type" header in the transaction. This header occurs along with the initial SMTP and RFC 822 headers in the message, right after the "MIME-Version" header. The version header is of no consequence to PickPacket. Subsequent occurrences of the content type header occur in each entity in the message. The content type header in each entity just gives information on the type of the entity data, which could be text or image or application etc. This information is useful as an exclusion criterion. The reason for this is explained below in the discussion on the next header that is very important as far as the parser-decoder is concerned. Examples of the two headers discussed above are as follows:

```
MIME-Version: 1.0
Content-Type: multipart/mixed;
boundary="----Next_Part-0CD3DF56.A0J3DCD0"
```

The issue with transferring binary data is that it may contain certain characters that fall outside the range of 127 (7-bit) characters acceptable by SMTP (which is based on RFC 822). Further, RFC 822 also limits lines in message bodies to be no more than 1000 characters long, including the CRLF separator. Therefore such data should be encoded in such a manner that the encoded data satisfies these restrictions imposed by RFC 822. MIME allows for different encoding mechanisms. The mechanism is specified by the "Content-Transfer-Encoding" header field in the corresponding entity or message-body part. Data such as a Microsoft Word document

is essentially binary data in raw form. However, all the strings that are used in the document are also present in this raw data. Since this raw binary data cannot be transferred in native form as per the RFC 822 specification, it is encoded (typically in base64). The goal of the MIME parser-decoder is to decode this encoded data on the fly and search for the required text string in the decoded data. Therefore it's important to know the transfer encoding of each entity's data and this information is sufficient for the working of the parser-decoder. The content type headers that occur in entity header fields can be used as exclusion criteria. If the content is some type of image, for instance, there's no point in decoding the base64 content in that entity as there won't be any text strings in the decoded data. There are seven standard MIME content types defined by RFC 2046. Of these, only four, namely "text", "application", "ietf-token" and "x-token" are of importance to the MIME parser-decoder. The remaining, namely "image", "audio" and "video" are of no consequence as explained above. These are called "top-level content types" in MIME terminology. Additional top-level content types can be defined, given the extensibility of MIME headers. However, these will be non-standard and their names should start with "x-". Such content types, if encountered, should be handled by the parser-decoder as the decoded data could contain some text strings. Each top-level type has certain subtypes, which do not hold any importance in PickPacket's scheme of things. For instance, an entity could contain data of type "text/plain" or "audio/wav". "plain" and "wav" are subtypes of the respective top-level content types. The content type headers may contain additional information such as the charset used to represent (textual) content. Example of a transfer-encoding header are presented below:

```
Example 1
---------
Content-Type: application/MSWORD
Content-Transfer-Encoding: bAsE64

Example 2
---------
Content-Type: text/plain; charset="us-ascii"
```

```
Content-transfer-encoding: 7-bit
```

One may note that the headers as well as their values are case-insensitive.

A discussion on the various transfer encoding mechanisms possible in MIME is in order now. There are seven mechanisms viz. 7-bit, 8-bit, binary, quoted-printable, base64, ietf-token and x-token. The first three essentially mean that the entity data has not been encoded in any manner. It appears as it does in raw or native form. The next two are encoding transformations that have been applied to some arbitrary raw data (the details of which are not known). As far as the MIME parser-decoder is concerned, this means that a "Content-Transfer-Encoding" value of 7-bit, 8-bit or binary means that the content is filtered as is without any transformation, whereas a value of quoted-printable or base64 means that the data has to be decoded in accordance to certain rules that have been laid down by the MIME specifications. The base64 encoding alphabet has been provided below:

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w	(pad)	=
15	P	32	g	49	x		
16	Q	33	h	50	y		

Base64 encoding works by converting a group of 3 octets into 4 6-bit base64 characters. These characters are taken from the base64 alphabet shown above. The decimal "value" of the 6-bit sextet (which is actually in binary) decides which base64 character is used in the encoded data. Therefore, base64 can be decoded on the fly. One doesn't need to read the entire attachment to begin decoding it.

Quoted-printable, though a transformation, doesn't necessarilymodify characters like the alphabet or digits or special symbols (like parentheses, ampersands etc.). These are the characters whose ASCII values lie between 33 and 60 or 62 and 126. All other characters other than these and the space and horizontal tab characters (ASCII 32 and 9 respectively) have to be represented in the form "=XY" where X and Y are uppercase hexdigits (0-9 or A-F) such that XY corresponds to the ASCII hexvalue of the character. The horizontal tab and space characters may be represented as is except when they occur at the end of a line (as in RFC822 line). There are other rules and restrictions, such as soft line breaks to ensure that encoded "lines" are not more than 76 characters long and so on. Quoted-printable transformations are applied on data which can potentially be modified during transport. So spaces, line breaks, carriage returns etc. are converted to some other format to escape such harmful data corruption by transporting agents or servers. Thus, decoding quoted-printable data is more or less straightforward if the data has been encoded strictly in accordance with the RFC. However, the RFC is not so strict in certain cases and situations, which poses a lot of problems for a quoted-printable decoder, as a lot of cases then arise while parsing the decoded data. All of such cases are carefully handled by the PickPacket. MIME parser-decoder.

There is a multitude of other MIME headers. Some are standard (like the Content-Description header) and others are mostly user-defined (a mail client may generate and use its own headers to provide additional information). These are again of no consequence to the PickPacket MIME parser-decoder.

4.3 MIME parser-decoder: Goals

The MIME parser-decoder in PickPacket is supposed to parse and decode MIME content MIME-encoded multipart SMTP messages on the fly. The decoded data is then passed to the parent SMTP filter to be searched for the text strings provided by the SMTP criteria in the configuration file that is input to the PickPacket Filter. If a text string match occurs in the decoded MIME content, the parent SMTP filter dumps the connection to disk as per the mode of operation of the filter.

The MIME parser-decoder should be made as efficient as possible because there is an added overhead in decoding base64 and quoted-printable data now in the SMTP filter. Care should be taken to see that the filter doesn't start dropping packets while the MIME parser is decoding MIME content.

The requirements of the PickPacket PostProcessor and Data Viewer have absolutely no bearing on the design on the MIME parser-decoder. Even if a match occurs in the MIME content, no additional fields are set aside for later use by either of these two components. The packet is dumped by the PickPacket Filter and the PostProcessor analyzes the dumped packet as it is. It does not even have to know about the existence of the MIME parser-decoder. Therefore, it is essential that the MIME parser-decoder works on a copy of the "packet_data" each time it's called. This is because the parent SMTP filter function calls the packet dumping macros on the "packet_data" variable. The contents of this variable are what have been received from the network and should be preserved and returned exactly in the way they were obtained. Therefore a copy of this variable is made and passed to the MIME parser-decoder. This copy is decoded by the MIME parser and the decoded equivalent of "packet_data" is made available to the string search routines in the SMTP filter. The filter then works as it does with non-MIME SMTP messages.

4.4 MIME parser-decoder: Design and Implementation

The discussion on MIME headers and message body formats presented earlier in this chapter lays out the basics of the functionality to be provided by the PickPacket MIME parser-decoder.

The parser-decoder intercepts all packets being processed by the SMTP filter. Technically, MIME is an extension of RFC 822 and as such, only MIME messages should be processed by the parser-decoder, as an optimization concern. However, most common email clients send mails as MIME messages, fully compliant with RFCs 2045 through 2049. Hence, this decision to intercept all packets and parse and decode them. As far as the application domain of PickPacket is concerned, this does not pose much of a problem, because the parser-decoder is in any case called only when there are some text strings to be matched and there was no mismatch of email addresses, if they were specified in the filtering criteria. Following this, the MIME parser-decoder takes over completely. Each packet is parsed and if necessary, decoded. The results of the decoding process are passed on to the SMTP filter which has been modified to search for text strings in the decoded data. If a match occurs, the SMTP filter simply dumps the connection according to the mode of operation of the PickPacket Filter.

Each connection uses a "MIME_Packet" structure which stores, among other things, the current transfer encoding in use in the message, the current packet's contents, the parser state, substate and subsubstate information and packet boundary. The parser takes as input a copy of the packet data, this structure, the length of the packet and another buffer which initially is also a copy of the packet data, but will be updated with the decoded data as the parser does its job of decoding the packet. Whatever be the transfer encoding used, the decoded data will be of a smaller length than the encoded data (which is given by "packet_data") and hence the last few bytes after a section of decode data in the buffer which is supposed to hold the decoded data will be padded with null bytes. Hence, the offsets of the boundary, entity headers and entity content in the decoded data buffer will be

exactly the same as in the original packet data buffer.

The core of this component is the MIME entity parser, whose design goes much along the lines of the HTTP packet parser described in the previous chapter. The state machine used by the parser is based on the same design employed by the HTTP packet parser. Thus there are various parser states, substates and subsubstates. Parser states maintain information about the particular section of the MIME message being parsed. Hence, the parser knows whether it is parsing an entity header or its content or the boundary between entities. For example, a parser state of MIME_PARSE_HEADER means that the parser is currently parsing a set of headers or has just finished parsing a boundary. Parser substates maintain more specific information. For instance, a substate of MIME_GOT_HEADER_VALUE means that the header value has been parsed, whereas a substate of MIME_GOT_HEADER means that the header string has being parsed. Parser subsubstates maintain state information about parsing of CRLF delimiters between headers or between a header and the corresponding entity part or content. Thus, a subsubstate of MIME_READ_CR indicates that a carriage return, denoted CR, has been read, and that a line feed, denoted LF, is expected next. In addition to these states, there are some error states to indicate erroneous MIME bodies or mistakes in parsing. There is also a parser state of MIME_MAIL_ENDED, which is set after a CRLF.CRLF sequence is observed in the message. The state information is maintained at the end of each call to the MIME parser and the parser picks up from this state when next invocated by the SMTP filter for that connection. This takes care of cases where a header, a header value, boundary or content split across packets. *Appendix B* gives a list of all the MIME parser-decoder states.

The first job of the MIME parser routine is to find out the MIME part boundary which separates the different entities. Each multipart message has one unique boundary which is used throughout the message. After the boundary is retrieved, the parser looks for the next occurrence of the boundary while filtering the content in the current entity. Additional parser states and substates have been added to account for cases where the boundary itself is split across packets. To take care of such a situation, Boyer-Moore good shift and bad character tables for the boundary are

computed as soon as the boundary is obtained and are stored in the "MIME_Packet" structure corresponding to this connection. Please note that an implementation of the Boyer-Moore algorithm is used by PickPacket to perform any text string or emailid search in case of SMTP traffic (as is the case with hostname searches in HTTP traffic etc.).

The headers of the current entity are first parsed to retrieve the transfer encoding in use for this entity. In case of 7-bit, 8-bit or binary encoded content, the filter goes ahead without transforming the contents of the packet data or the decoded data buffer, which initially holds a copy of the packet data. In case of quoted-printable or base64 encoded data, the parser first completely decodes all the content in the current entity (or packet, whichever ends first) into the buffer supposed to hold decoded data. The base64 and quoted-printable data decoders are robust enough to account for cases such as splitting of base64 quartets and =XX groups, respectively, across packets. To account for such instances, the "MIME_Packet" structure provides a small 4-byte buffer to hold the remnant from the previous packet, depending on the conditions encountered by the decoders.

After a part is completely decoded, the boundary is skipped completely, the states are set to indicate parsing of headers and this cycle continues. After the packet data is exhausted, the parser-decoder returns the length of the decoded data to the parent SMTP filter. The decoded data buffer has already been filled at various stages in the parser-decoder. The parent SMTP filter now uses this new buffer and its length in its calls to the Boyer-Moore string search functions. If a match occurs, however, the SMTP filter calls its packet/connection dumping macros on the original "packet_data". Hence these macros are totally oblivious of the existence of the MIME parser-decoder. It is clear now that such an unobtrusive design greatly reduces the chances of error in the operation of the original filter.

This completes the discussion on the design and core implementation features of the MIME parser-decoder in the PickPacket Filter. The next chapter presents details on the handling of non-consecutive packets in a connection.

Chapter 5

Handling non-consecutive packets

Under normal circumstances, a network interface would not see non-consecutive TCP packets in a connection. However, congested networks may deliver non-consecutive packets. In such situations, the PickPacket Filter follows a simple policy of forgetting a prefix match of a string in a previous packet and starts looking for the required search strings afresh. This works well in the case of normal SMTP or HTTP connections where the entire payload consists of ASCII characters. When it comes to MIME, though, non-consecutiveness affects the decoding of MIME-encoded messages. Therefore, the MIME parser-decoder has to handle non-consecutive packets in a graceful manner instead of functioning erroneously. This chapter presents some details on the assumptions made in the design and the implementation of this design to handle non-consecutive MIME packets.

The most important thing about handling a MIME message is to get the entity boundary. The boundary decides the top-level state changes in the MIME parser's state machine. Even in case the parser receives non-consecutive MIME packets, it tries to parse them and decode them as per the boundary, if it is available. This gives rise to various cases, each of which is described below. An important thing to note here is that the TCP channel manager component of the PickPacket Filter doesn't allow previous (sequentially), missed packets of a connection further upstream to the application filters. So in no event would the MIME parser-decoder see a packet

that comes sequentially before a packet that has already been processed. Non-consecutiveness will only come in the form of a later packet arriving in place of the expected one, and all the packets between the expected one and the incoming one are lost for good as far as the MIME parser is concerned.

5.1 Messages without a boundary

Non-MIME (plain SMTP) messages and messages with a single attachment (sent using clients like metasend) do not have entity boundaries. The parser has to decide first whether this is the case with the current connection. Changes have been made in the parent SMTP filter to get the TCP sequence number of the DATA command packet sent by the client. The ending sequence number of this packet is determined and made available to the MIME parser-decoder. The parser is always on the lookout for the first packet after the DATA command packet. A flag is set when this packet is seen. The idea behind this is that the boundary, if it exists, would be specified in the first few packets, since the boundary specification occurs in the MIME headers before the beginning of any content. A limit of two packets (after the DATA command packet) has been imposed upon the parser to determine the boundary in case a non-consecutive packet was encountered at the very outset. This takes care of situations where the recipient list is so large that the headers extend into the second packet after the DATA command packet. At all stages, it is only the TCP sequence number of the current packet that determines the course of action to be taken by the parser. The starting sequence number of the incoming packet is compared against the sequence number of the packet expected to contain the boundary (this expected sequence number should be one more than the ending sequence of the DATA command packet or the first packet after the DATA command packet, as the case may be). If this condition matches, the parser looks for the boundary specification within the incoming packet, and sets the top-level state to MIME_PARSE_HEADER if the boundary is found. The MIME_Packet structure now includes a new member called boundary_status which is set to a certain value whenever the boundary is seen. By default, this member has the value

BOUNDARY_UNKNOWN. If at any stage, it is known that this message does not have a boundary, this value is set to BOUNDARY_ABSENT instead.

In case of non-consecutiveness, the parser therefore tries to search for the boundary in the first two (sequentially) packets after the DATA command packet. No special measures have been taken here to account for a split in the boundary specification across packets. The boundary is deemed to be absent if a "boundary=" or "BOUNDARY=" string is not found in these two packets. The parser then sets the top-level state to indicate that a part, and not the headers, are being parsed, and the rest of this entire connection is passed on without any decoding transformation whatsoever applied to the payload. This obviously leads to erroneous results in case of base64 encoded messages where there actually is a text string match, but in all other cases, the results should be as they would in case the packets arrived consecutively.

5.2 Messages having a boundary

This situation implies that the parser knows what the boundary is and some subsequent packet has arrived non-consecutively, or that it has discovered the boundary in the first two packets inspite of packets arriving non-consecutively at the outset.

An easy solution to this situation is to disregard the content until the occurrence of the next boundary in the message, whereupon the parsing and decoding can start afresh. In case the current entity contains 7-bit or quoted-printable content, this wouldn't cause any erroneous filtering (in most cases). However, the MIME parser-decoder attempts to continue decoding the current entity even if it is encoded in base64. The remnant from the previous packet is discarded, and the length of the first encoded line in the current packet is determined. The first few characters of this line, upto the remainder of this length when divided by 4, are also discarded. The basis for this is that clients normally send an integral number of quadruples in a line of encoded content. The base64 decoder then picks off and parsing continues as usual. Another attempt at a solution, which could be erroneous in certain situations, is to look at the difference in the ending sequence number of the previous packet

and the starting sequence number of this packet. Given the remnant from the previous packet and this difference, the number of quartets that have been missed could be calculated, with the remainder of the last quartet making up the first few characters in the payload of this packet, if the remainder is calculated to be non-zero. This remainder could then be skipped and the decoding could continue but this approach is naive if the appropriate count of CRLF line breaks in the missed packet(s) is miscalculated. The currently implemented approach, however, works in all situations provided the length of an encoded line is a multiple of 4, which is the case with any mail client.

The MIME parser-decoder has no information about a prefix match of a string in the previous packet since this functionality is part of the parent SMTP filter. Upon receiving the decoded (or otherwise) content from this packet, the parent SMTP filter would proceed to reject the prefix match and start looking afresh for the search strings from this packet onwards until another non-consecutive packet is encountered or the connection is exhausted completely.

Chapter 6

Testing and Results

6.1 Testing the HTTP postprocessor

The PickPacket filter was ran with all possible combinations of HTTP criteria. This included specification of no HTTP-specific criteria (all IPs monitored with ports set to 80 and 3128 to allow for the presence of IITK's HTTP proxy), single HTTP criterion (one hostname and one search string) and multiple HTTP criteria (multiple sets each with one hostname and one search string). These runs were done with only GET requests once, and only POST requests once. IITK's HTTP proxy doesn't support HTTP/1.1. An internal HTTP server was used to test HTTP/1.1 operations. The whole procedure was adopted with the filter operating once in PEN mode and once in FULL mode. A multitude of HTTP clients were used to check for inconsistencies in parsing the headers. The resultant dumps were all postprocessed successfully by the HTTP postprocessor without error.

6.2 Testing the MIME parser-decoder

The MIME parser-decoder was checked for both correctness of execution and performance under heavy loads. The first test consisted of running the filter on a set of mails drawn upon to exhaust all possible combinations of the various parameters

involved, namely, the mode of operation of the filter, the mail client used, the number and types of filtering criteria provided and the three different encodings (7bit, base64 and quoted-printable) encountered in practical situations. By types of filtering criteria we mean the composition of a set of criteria. Therefore, the parser was tested against criteria which specified only email addresses to match and no text strings, against criteria which specified only text strings and no email addresses and against criteria that specified both. The second part of this test consisted of testing the correctness of functioning of the parser-decoder in cases where the boundary was split across packets, or the text string to be searched for was split across packets, or base64 quartets or quoted-printable character groups were split across packets. The third part of the test consisted of checking the parser-decoder against packets arriving non-consecutively. In all cases, the observed results of the tests were exactly as expected according to the requirements.

The second test was meant to evaluate the performance of the parser-decoder under heavy loads. The objective is to ensure that the filter does not drop any packets while in the process of parsing and decoding the MIME content. This test was conducted by deploying two filters on a 100 Mbps segment consisting of five other nodes sending a series of mails to an SMTP server also on the same segment. The SMTP server was made to run on a 4-CPU (each an Intel Xeon 2.0 GHz) machine having 1 GB of RAM. The nodes generating the traffic and the nodes on which the filters were deployed were Intel Pentium 4 2.4B GHz workstations, each with 256 MB of RAM. One of the filters was used to simply read all packets (no application-level criteria specified) and direct the output to /dev/null. The other filter was given fifty-two sets of SMTP criteria to work on. At the end of the test, the number of packets sniffed by each filter was compared. Also, the average bandwidth achieved over the duration of the test was calculated using the information on the size of the traffic generated and the time to completion of the test. It was observed that under these conditions, the PickPacket Filter and the MIME parser-decoder worked without dropping any packets. The total size of the data transferred was roughly 1380 MB and the time in which this transfer took place was 170 seconds. Therefore the average bandwidth achieved was 64.9 Mbps.

Chapter 7

Conclusions

PickPacket is a network monitoring tool that can capture packets flowing across the network based on a highly flexible set of criteria. Judicious use of PickPacket can also help protect the privacy of individuals and dump only necessary data onto the disk. This is not something most sniffers are capable of doing. The captured data is stored in standard tcpdump/libpcap format which makes it easy to analyze. However, it comes bundled with its own suite of application-level postprocessors and an easy to use information viewer.

PickPacket is architecturally divided into four components the PickPacket Configuration File Generator, the PickPacket Filter, the PickPacket Post Processor, and the PickPacket Data Viewer. Each of these components was briefly discussed. PickPacket uses in-kernel BPF to capture packets. The packets filtered by the in-kernel filter are passed to the application level filter for further processing.

This report has discussed two components of PickPacket. One of them is the HTTP postprocessor which takes as input the packets captured by the PickPacket Filter based on the filtering criteria and dumped on to the disk. The postprocessor analyzes these packets and retrieves various pieces of information from them and arranges them on disk in a manner that allows the Data Viewer to show that information in a human-readable form. The second component is the MIME parser-decoder extension to the existing SMTP filter component in the PickPacket Filter. This works on multipart SMTP messages, parsing MIME headers and content and

decoding the content if necessary to perform text string searches on the various attachments.

7.1 Scope for further work

PickPacket currently works explicitly on SMTP, FTP, HTTP and Telnet. All other protocols, if captured (on the basis of IP and port-level criteria) are classified as OTHER protocols. There is scope for extending PickPacket to support other application level protocols like POP and IMAP. Currently, there is no support for searching text strings in encoded HTTP data (either MIME or some other encoding). Encryption of dumped packets and using digital signatures can make PickPacket more useful to law enforcement agencies. This can make packets captured admissible as unconditional evidence. One major limitation of PickPacket is that it currently does not support dynamic address allocation based networks. This would be required of PickPacket to make it useful in scenarios involving Internet Service Providers. PickPacket should be extended to include protocols like RADIUS and DHCP to achieve this.

Bibliography

[1] T. Berners-Lee, R. Fielding, U. C. Irvine, and L. Masinter. "Uniform Resource Identifiers (URI): Generic Syntax". Technical report, 1998. http://www.ietf.org/rfc/rfc2396.txt.

[2] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. "Hypertext Transfer Protocol". Technical report, 1997. http://www.ietf.org/rfc/rfc2068.txt.

[3] N. Freed and N. Borenstein. "Multipurpose Internet Mail Extensions". Technical report, 1996. http://www.ietf.org/rfc/rfc2045.txt.

[4] Neeraj Kapoor. "Design and Implementation of a Network Monitoring Tool". Technical report, Department of Computer Science and Engineering, IIT Kanpur, Apr 2001. http://www.cse.iitk.ac.in/research/mtech2000/Y011111.html.

[5] J. Klensin. "Simple Mail Transfer Protocol". Technical report, 2001. http://www.ietf.org/rfc/rfc2821.txt.

[6] Steve McCanne and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture". In *Proceedings of USENIX Winter Conference*, pages 259–269, San Diego, California, Jan 1993.

[7] Brajesh Pande. "The Network Monitoring Tool - Pickpacket: Filtering Ftp and Http Packets". Technical report, Department of Computer Science and Engineering, IIT Kanpur, Sep 2002. http://www.cse.iitk.ac.in/research/mtech2000/Y011104.html.

[8] Boyer R. and J Moore. "A fast string searching algorithm". In *Comm. ACM 20*, pages 762–772, 1977.

[9] Jacobson V., Leres C., and McCanne S. *"pcap - Packet Capture Library"*, 2001. Unix man page.

Appendix A

List of all HTTP packet parser states

```
typedef enum Parser_State {
    HTTP_STATE_NONE,
    HTTP_PARSE_REQUEST_LINE,
    HTTP_PARSE_RESPONSE_LINE,
    HTTP_PARSE_HEADER,
    HTTP_PARSE_MESSAGE,
    HTTP_PROCESSED_RESPONSE,
    HTTP_PROCESSED_REQUEST,
    HTTP_ERROR
} Parser_State;

typedef enum Parser_Sub_State {
    HTTP_SUB_STATE_NONE,
    HTTP_SKIPPING_CRLF,
    HTTP_GETTING_METHOD,
    HTTP_GOT_METHOD,
    HTTP_GETTING_URI,
    HTTP_GOT_URI,
    HTTP_GETTING_VERSION,
    HTTP_GOT_VERSION,
```

```
    HTTP_SKIPPING_TO_CR,
    HTTP_SKIPPED_TO_CR,
    HTTP_GETTING_CRLF,
    HTTP_GOT_CRLF,
    HTTP_GETTING_HEADER,
    HTTP_GOT_HEADER,
    HTTP_GETTING_HEADER_VALUE,
    HTTP_GOT_HEADER_VALUE,
    HTTP_SKIPPING_LWS,
    HTTP_SKIPPED_LWS,
    HTTP_SEEN_ALL_HEADERS,
    HTTP_READING_CHUNK_LENGTH,
    HTTP_READING_CONTENT,
    HTTP_GOT_TRAILER,
    HTTP_SKIPPING_VERSION,
    HTTP_GETTING_STATUS_CODE,
    HTTP_SKIPPING_TRAILERS,
    HTTP_SUB_ERROR
} Parser_Sub_State;

typedef enum Parser_Sub_Sub_State {
    HTTP_SUB_SUB_STATE_NONE,
    HTTP_READ_CR,
    HTTP_READ_LF,
    HTTP_SUB_SUB_ERROR
} Parser_Sub_Sub_State;
```

Appendix B

List of all MIME filter parser states

```
typedef enum Parser_State {
    MIME_STATE_NONE,
    MIME_PARSE_HEADER,
    MIME_PARSE_PART,
    MIME_PROCESSED_PART,
    MIME_MAIL_ENDED,
    MIME_PARSE_BOUNDARY,
    MIME_SEARCH_BOUNDARY,
    MIME_ERROR
} Parser_State;

typedef enum Parser_Sub_State {
    MIME_SUB_STATE_NONE,
    MIME_SKIPPING_CRLF,
    MIME_SKIPPING_TO_CR,
    MIME_SKIPPED_TO_CR,
    MIME_GETTING_CRLF,
    MIME_GOT_CRLF,
    MIME_SKIPPING_LWS,
    MIME_SKIPPED_LWS,
```

```
    MIME_GETTING_HEADER,
    MIME_GOT_HEADER,
    MIME_GETTING_HEADER_VALUE,
    MIME_GOT_HEADER_VALUE,
    MIME_SEEN_ALL_HEADERS,
    MIME_READING_PART,
    MIME_READING_BOUNDARY,
    MIME_SUB_ERROR
} Parser_Sub_State;

typedef enum Parser_Sub_Sub_State {
    MIME_SUB_SUB_STATE_NONE,
    MIME_READ_CR,
    MIME_READ_LF,
    MIME_SUB_SUB_ERROR
} Parser_Sub_Sub_State;
```