Global Information Assurance Certification Paper

What's running on your network?
Analyzing pcap data with tshark

GIAC (GCIA) Gold Certification

Author: François Bégin, francois.begin@telus.com
Advisor: Dominicus Adriyanto

Accepted: December 22nd, 2012

Abstract

Network packet capture files are extremely useful to security analysts who try to determine issues and possible compromise in an environment. A complete packet capture contains a wealth of information but extracting something useful from that data can be both challenging and time-consuming. This paper showcases the capability of *tshark* and other open-source tools for extracting such information. By demonstrating how simple outputs from *tshark* can be correlated with external resources, this paper makes a strong case that security analysts should consider adding *tshark* to their toolbox. Emphasis is placed not just on getting familiar with *tshark* but also on concrete examples such as IP geo-location, malware threat correlations, email message extractions, etc.

1. Introduction

Now more than ever, IT infrastructures are targeted by malicious outsiders, ranging from ideologically motivated groups such as Anonymous (Norton, 2012) to corporations and governments utilizing highly sophisticated Advanced Persistent Threats (Juels & Yen, 2012). As a result, data theft has become an area of concern (Websense, 2012). With attackers relying on a wide array of data exfiltration techniques (botnets, email lures, web re-directs, etc.), protecting a network means looking at data coming in and out of that network.

A decade ago, Kaufman, Perlman & Speciner wrote: "As long as information can be passed one bit at a time, anything can be transmitted, given enough time" (2002). This sentence illustrates well what security analysts are facing nowadays. Analyzing network traffic is therefore an essential skill for security analysts as it can often provide evidence of clues to system compromises. Even without raising the specter of security breaches, knowing what is happening on a given network is critical to help establish a baseline. This baseline is essential when trying to identify abnormal behavior. This paper therefore intends to introduce some open-source tools to help analyze a network capture and extract useful data from it.

While it is understood that there are many tools available to security analysts as well as commercial products that can perform this analysis, this paper will focus mainly on *tshark*, which is part of the *Wireshark* toolset (wireshark.org, 2012). This powerful tool produces output that can easily be customized with command-line utilities such as *awk*, *sed*, *sort*, *cut*, etc. This in turn facilitates the integration of these outputs with outside databases containing rich data related to security threats. As such, this paper should be viewed as another approach to data analysis, one that a security analyst can further customize to his specific needs.

François Bégin, francois.begin@telus.com

2. Preparation & tools

2.1. Setting up for the capture

Capturing packets, even in a small-to-mid-size network, can be complicated. Is a network diagram available? Is it up to date? Does it capture all the relevant elements of the environment? When investigating an incident, this diagram is literally your crime scene map (Fishburne, 2012). Assuming one such diagram is available, and assuming that internet services are hosted in that network, it is likely segmented as per figure 1:

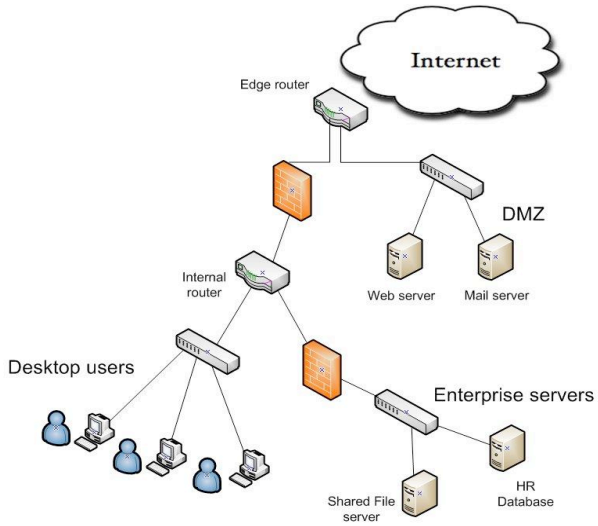

Figure 1 Simple network with internet services

As seen above, even this simple network is made up of three distinct segments: the DMZ, the internal network of desktop users and the segment for enterprise servers, each of which are separated by routers and firewalls. The next question to take into account is therefore: "From which segment should data be captured?" As many other things in life, there is no one correct answer. It depends for instance on what is being investigated (data exfiltration, malware activity, etc.) as well as the technique available to perform the capture (inline, using mirrored ports, port aggregation, etc.).

Discussing the intricacies of setting up for the capture itself is outside the scope of this paper but interested readers can refer to the *Ethernet capture setup* wiki of the Wireshark project (wireshark.org, 2012) for more information.

For the remainder of this paper, the assumption is that a sniffing host is attached to a mirrored port of the internal network switch. In other words, all desktop users' traffic (both inbound and outbound) will be captured and stored in *pcap* format.

2.2. Introducing the tools

The following command-line tools were selected to conduct the capture and analysis:

tcpdump: the well-known command-line protocol analyzer

tshark: the command line counterpart to the Wireshark GUI program, also part of the Wireshark suite. For the purpose of this paper, a version > 1.8 is required.

curl: a command-line tool for transferring data using common network protocols such as HTTP.

gnuplot: A plotting program to create graphs

tcpflow: a tool to extract data streams

2.3. Packet capture performance considerations

The next step prior to the actual analysis is to select the tool to use to capture the data. Initially, the decision was made to use *dumpcap* (also part of Wireshark). The thinking behind this choice was simple: *dumpcap* being a raw packet dumper rather than a protocol analyzer (such as the more well-known *tcpdump*), it was expected that it would perform better. Another reason was that this would decrease the chance of the capture tool being affected by vulnerabilities. For instance, the Common Vulnerabilities and Exposures site (CVE, 2012) lists numerous vulnerabilities in *tcpdump* such as CVE-2007-3798 but none for *dumpcap*.

As it turned out though, *dumpcap*'s performance lagged behind *tcpdump*'s and it was the latter that was used for capturing *pcap* data. The interested reader can consult Appendix A to see the tests that were done to reach this conclusion. The tests are far from exhaustive and are only presented to highlight certain aspects to take into consideration when capturing network traffic. A detailed analysis of packet capture performance is outside the scope of this paper, as more complex tweaks should also be discussed such as

memory-mapping, multi-processors/cores/threads, customized drivers, the OS used for the capture, etc. Ample literature on the subject is readily available, with Braun, Didebulidze, Kammenhuber & Carle (2010) providing a good survey. Suffice it to say that ideally the sniffing station should be able to handle the network load and capture full packets without any drops.

Having established this, and having determined that *tcpdump* would provide better performance in the target environment, the sniffing station was deployed as per Figure 2:

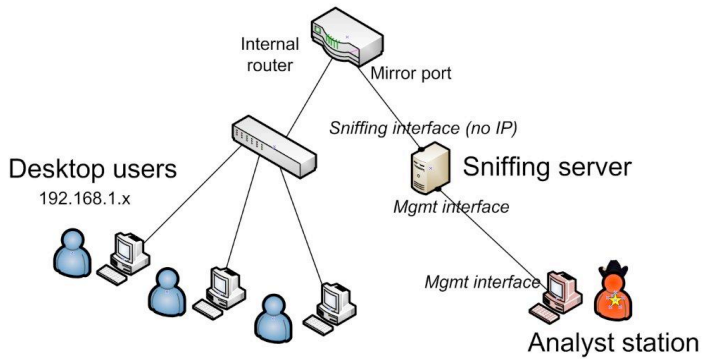

Figure 2 Deployment of a sniffing station

The interface on which the capture is taking place was brought up without an actual IP, making the sniffing box act as a passive listener in the network. A separate, dedicated management interface from which an analyst would retrieve the packet capture was also turned on, allowing for out-of-band communication to the sniffing server.

3. Analysis

3.1. Physical layer analysis

If the packet capture takes place on a layer 3 device such as an aggregation point on a router, physical layer analysis is not going to yield many results. As part of the IP routing algorithm (Comer, 2000), IP datagrams are encapsulated inside Ethernet frames and the MAC addresses in the frame header will change as the frame hops from router to router. Looking at the physical layer of such a capture file would only reveal the physical addresses of the routers.

On the other hand, if the capture takes place on a layer 2 device, such as capturing data on a segment serviced by a switch by using a mirrored port, the MAC addresses of the actual devices chatting on the segment will be revealed. Although this information will prove too 'low-level' to be useful in most circumstances, it will serve to introduce the main tool this paper relies upon: *tshark*. As already mentioned, *tshark* is part of the Wireshark project. It actually is the command line equivalent of *wireshark*. Like its GUI counterpart, it can filter and extract protocol fields:

$ tshark -r tap_capture_Sunday.pcap -n -T fields -e eth.addr | awk -F, '{print $1"\n"$2}' | sort | uniq | tee mac_addresses

00:03:47:71:98:b2

00:04:4b:03:4e:0f

00:11:11:1e:8b:a0

[...]

The above *tshark* command reads a capture file (*-r tap_capture_Sunday.pcap*) without attempting to resolve host/port names (*-n*), and extracts the Ethernet addresses field (*-T fields -e eth.addr*). Both the source MAC address and the destination MAC address are extracted by *-e eth.addr* and separated by a comma as such: 00:11:11:1e:8b:a0,00:26:88:03:67:08. Using *awk*, the two addresses are separated by a line feed (*awk –F, '{print $1"\n"$2}'*), sorted (*sort*), and only unique MAC addresses are kept (*uniq*). Finally, the *tee* command shows the result to the screen and saves a copy to a file called *mac_addresses*.

This list can be used as a baseline of devices that live on a given segment. Having a list of all MAC addresses visible in a given network segment can prove valuable when trying to identify issues: maybe a device has died unexpectedly or a new device has been introduced in the environment without proper notification. Many companies now rely on management systems that cover assets, applications and services in the form of Configuration Management Databases or Configuration Management Systems (O'Donnell & Casanova, 2009). If you have such a system in place, the MAC address might already be part of the primary key of your Configuration Items (Evergreensys.com, 2007) and this in turn can provide some further asset correlation. This list can also be used when applying port security (Donahue, 2011).

If there is no formal asset management in the environment, a MAC address vendor lookup site (MACVendorLookup.com, 2012) combined with the tool *curl* can provide hardware vendor data on the extracted addresses. Note that an API key is required to get this information (registration is required):

$ for MAC in `cat mac_addresses` ; do echo $MAC" : `curl -s http://www.macvendorlookup.com/api/<API_KEY>/${MAC} | awk -F \| '{print $1}'`" ; done

00:03:47:71:98:b2 : Intel Corporation

00:04:4b:03:4e:0f : Nvidia

01:00:5e:00:00:01 : none

[...]

The *for-loop* is just a generic Unix shell loop of the form *for <data list>; do <cmd>; done*. In this particular case, the data list is made of MAC addresses contained in file *mac_addresses*. For each of these addresses, the MACVendorLookup.com site is accessed using *curl* via a simple HTTP GET:

$ curl http://www.macvendorlookup.com/api/<API_KEY>/00:03:47:71:98:b2

Intel Corporation|Ms: Lf3-420|2111 N.e. 25th Avenue|Hillsboro Or 97124|United States

The site replies with details about the hardware vendor. It should be pointed out that *tshark* can also provide additional packet data by using the *–V* option. This option prints packet details rather than the typical one-line summary:

$ tshark -V -r tap_capture_Sunday.pcap
Frame 1: 130 bytes on wire (1040 bits), 130 bytes captured (1040 bits)
Arrival Time: Sep 9, 2012 10:47:26.796081000 MDT
Epoch Time: 1347209246.796081000 seconds
[Time delta from previous captured frame: 0.000000000 seconds]
[Time delta from previous displayed frame: 0.000000000 seconds]
[Time since reference or first frame: 0.000000000 seconds]
Frame Number: 1
Frame Length: 130 bytes (1040 bits)
Capture Length: 130 bytes (1040 bits)
[Frame is marked: False]

[Frame is ignored: False]

[Protocols in frame: eth:ip:tcp:ssh]

Ethernet II, Src: ***AsustekC****_c3:a8:08 (20:cf:30:c3:a8:08), Dst:* ***Giga-Byt****_c9:6a:50 (50:e5:49:c9:6a:50)*

[…]

With *–V*, the hardware vendor is shown but the point here is to be able to extract precise data and correlate that data with external data sources. This approach will be used again in section 3.3.1 (DNS – Ad & Malware threats correlation).

A list of all protocols and fields available to *tshark* can be found by running *tshark –G* but the output of this command is overwhelming. A better approach is to consult the *Wireshark Display Filter Reference* (Wireshark, 2012). For instance, following the link to **ethernet** on the main page reveals all the fields that *tshark* can extract from an Ethernet frame such as *eth.addr*, *eth.src*, *eth.dst*, *eth.padding*, *frame.len*, etc. as shown in figure 3

Display Filter Reference: Ethernet
Protocol field name: eth
Versions: 1.0.0 to 1.8.4
Back to Display Filter Reference

Field name	Type	Description	Versions
eth.addr	Ethernet or other MAC address	Address	1.0.0 to 1.8.4
eth.dst	Ethernet or other MAC address	Destination	1.0.0 to 1.8.4
eth.fcs	Unsigned integer, 2 bytes	Frame check sequence	1.8.0 to 1.8.4
eth.fcs_bad	Boolean	FCS Bad	1.8.0 to 1.8.4
eth.fcs_good	Boolean	FCS Good	1.8.0 to 1.8.4
eth.ig	Boolean	IG bit	1.0.0 to 1.8.4
eth.invalid_lentype	Unsigned integer, 2 bytes	Invalid length/type	1.8.0 to 1.8.4
eth.len	Unsigned integer, 2 bytes	Length	1.0.0 to 1.8.4
eth.lg	Boolean	LG bit	1.0.0 to 1.8.4
eth.padding	Sequence of bytes	Padding	1.8.0 to 1.8.4
eth.src	Ethernet or other MAC address	Source	1.0.0 to 1.8.4
eth.trailer	Sequence of bytes	Trailer	1.0.0 to 1.8.4
eth.type	Unsigned integer, 2 bytes	Type	1.0.0 to 1.8.4
eth.vlan.cfi	Unsigned integer, 2 bytes	CFI	1.6.0 to 1.6.2
eth.vlan.id	Unsigned integer, 2 bytes	VLAN	1.6.0 to 1.6.2
eth.vlan.pri	Unsigned integer, 2 bytes	Priority	1.6.0 to 1.6.2
eth.vlan.tpid	Unsigned integer, 2 bytes	Identifier	1.6.0 to 1.6.2

Figure 3 Ethernet protocol display filters

Extracting these fields then becomes just a matter of adding them to the *tshark* command as required:

```
$ tshark  -r tap_capture_Sunday.pcap -n -T fields -e eth.addr -e frame.len –c 3
50:e5:49:c9:6a:50,20:cf:30:c3:a8:08	130
20:cf:30:c3:a8:08,50:e5:49:c9:6a:50	66
50:e5:49:c9:6a:50,20:cf:30:c3:a8:08	130
```

The above *tshark* command displays the source and destination MAC addresses (*-e eth.addr*) as well as the corresponding frame lengths (*-e frame.len*) for the first 3 packets in the capture (*-c 3*). This data in turn could be tallied to find the most 'talkative' hosts at the physical layer. At such a low level of the OSI model, this information lacks interest but at the network layer, this will prove more interesting as the next section will demonstrate.

3.2. Network layer analysis

3.2.1. IPv4 - Datagram sizes distribution

Although the demise of IPv4 has been predicted for years, this protocol remains prevalent. IPv6 traffic does increase year after year but it still only amounts to a fraction of all internet traffic and faces deployment challenges (Karpilovsky, Gerber, Pei, Rexford, & Shaikh, 2009). This section therefore focuses on IPv4 traffic and a simple exercise: to create a graph showing the distribution of the packet sizes in our packet capture. Getting a list of IP datagram lengths as well as a tally of the number of packets of each size is simple enough with *tshark*:

```
$ tshark -r tap_capture_Sunday.pcap -n -T fields -e ip.len | sort –n | uniq -c  | awk '{print $2" "$1}' | sort –n | tee /tmp/ipLen.dat
32	391
40	90387
41	11
[...]
```

The above *tshark* command extracts the IP datagram length (*-e ip.len*) for each packet. The following *sort –n | uniq -c | awk '{print $2" "$1}' | sort -n* command sorts the output of *tshark* (*sort –n*), creates a tally of the number of packets for each packet size (*uniq -c*) and lists the packet length followed by the number of packets of that length (*awk '{print $2" "$1}'*). The final *sort* command lists the results in increasing order of

packet length. Referring back to the output, it can be seen that 391 packets were 32 bytes long, 90,387 packets were 40 bytes long, etc. The results are displayed to the screen and saved to a file called */tmp/ipLen.dat.*

Turning this into a histogram is not very complicated using a tool such as *gnuplot* (gnuplot.info, 2012). Appendix C shows the content of a file called *IPdistrib.gnuplot* which will take the data in file */tmp/IPlen.dat* and turn it into a histogram. To create this histogram, simply run

$ gnuplot < ./IPdistrib.gnuplot

The actual picture is saved to file *./reports/graphs/IPdistrib.png* and is shown in Figure 4. Note that the y scale is logarithmic for a better fit.

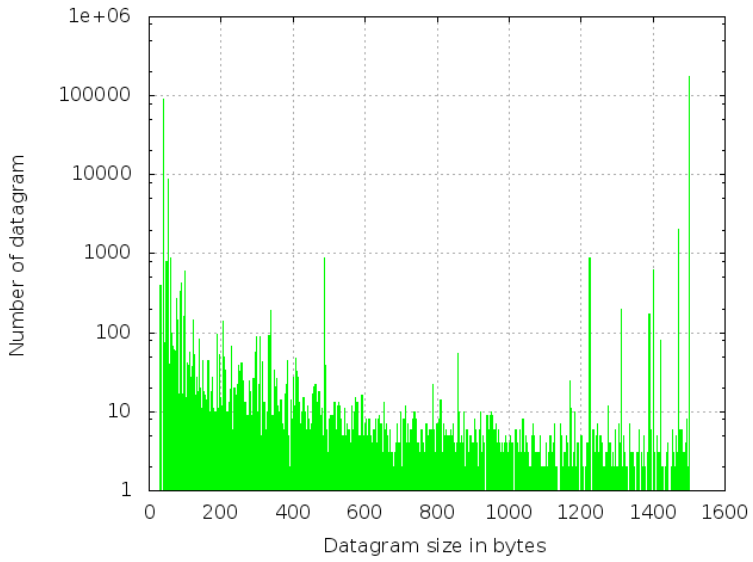

Figure 4 Distribution of the size of IP datagrams

If the goal is to find the top 3 datagram sizes, the following command can be used:

```
$ cat /tmp/ipLen.dat | sort -k2 -nr | head -3
1500   173838
40      90387
52      8774
```

In the above example, the *sort* command orders the data on the second field (*-k2*) in reverse numerical order (*-nr*) and the *head* command returns the top 3 results. One might be interested to take a closer look at those 90,387 datagrams of size 40 bytes. *Tshark* can shed some light on those packets. One useful piece of information currently missing is what IP addresses are generating these packets.

```
$ tshark -r tap_capture_Sunday.pcap -n -T fields -e ip.src -e ip.dst -e ip.proto -R "ip.len eq 40" | sort | uniq -c | sort –rn
47691   192.168.1.121   80.190.148.74   6
18529   192.168.1.64    206.108.207.168 6
6297    192.168.1.121   206.108.207.138 6
3885    192.168.1.64    69.192.83.235   6
[...]
```

The above *tshark* command extracts the source (*-e ip.src*) and destination (*-e ip.dst*) IP addresses as well as the underlying IP protocol (*-e ip.protocol*). A filter is applied to limit the output to datagrams that are 40 bytes in size (*-R "ip.len eq 40"*). The final *sort | uniq –c | sort –rn* command tallies the results and presents them in reverse order, from most frequent to least frequent.

One pair of hosts stands out from this list with 47,691 packets: 192.168.1.121 and 80.190.148.74. This pair generated more than twice the number of packets than the next highest pair. Protocol 6 translates to TCP. Let's see what this particular conversation was all about:

```
$ tshark -r tap_capture_Sunday.pcap -n -T fields -e tcp.dstport -R "ip.len eq 40 && ip.src eq 192.168.1.121 && ip.dst eq 80.190.148.74" | sort -n | uniq –c
47691   80
```

The above *tshark* command applies a tighter filter (*-R "ip.len eq 40 && ip.src eq 192.168.1.121 && ip.dst eq 80.190.148.74"*) and extracts the destination port of the TCP traffic (*-e tcp.dstport*). The *sort –n | uniq –c* command does the usual tally per TCP destination port.

Interestingly, every datagram was sent to TCP destination port 80, so this is likely HTTP traffic. At this point, an astute reader might have noticed the small size of the

datagrams. Do these datagrams contain anything? Actually, they do not contain any data. The minimum length of an IP header is 20 bytes (without any options). The minimum length of a TCP header is also 20 bytes (without any options). The sum of these headers is 40, which is equal to the packet size that was found. In other words, a packet size of 40 for an IP packet containing a TCP segment has no room for anything but the IP & TCP headers. What does this reveal? Perhaps looking at the TCP flags with *–e tcp.flags* will help:

$ tshark -r tap_capture_Sunday.pcap -n -T fields -e tcp.flags -R "ip.len eq 40 && ip.src eq 192.168.1.121 && ip.dst eq 80.190.148.74" | sort | uniq -c

47689 0x0010

2 0x0011

All packets but 2 have the flag combination 0x0010. This is a hex value equivalent to 16_{10} or 00010000_2. The flag fields in a TCP header are [C E U A P R S F]. The first two bits are for ECN and the last 6 are for Urgent, Acknowledgement, Push, Reset, Synchronize and Fin respectively. The fourth bit (the only one that is set) is used to indicate an acknowledgement, which means that these packets are simply ACKs of a TCP transmission. This result is somewhat anti-climactic but it did showcase how *tshark* can be used to further study specific packets. More analysis can be done:

$ tshark -r tap_capture_Sunday.pcap -n -T fields -e frame.number -c 1 -R "ip.len eq 40 && ip.src eq 192.168.1.121 && ip.dst eq 80.190.148.74"

74426

The *tshark* command above extracts the frame number (*-e frame.number*) of the first frame *(-c 1*) that matches the filter. *Tshark* dutifully responds with 74,426. Since it has already been determined that the destination port of this traffic was TCP/80, perhaps some additional useful data can be found at a higher layer of the OSI model, namely at the application layer:

$ tshark -r tap_capture_Sunday.pcap -n -R "frame.number>74416 && frame.number<74536 && ip.src eq 192.168.1.121 && ip.dst eq 80.190.148.74" -T fields -e http.request.uri -e frame.number | grep ^\/

/package/wks_avira/win32/en/pecl/avira_free_antivirus_en.exe 74427

The filter applied above limits the frames returned by arbitrarily looking at a range of 10 frames before and after the frame of interest (i.e. all frames in the 74,416 -

74,536). A new field is also extracted: the URI of requests made by the client (*-e http.request.uri*). This should contain details of what was requested on TCP/80, provided this was indeed HTTP traffic. As for the *grep* command, it simply filters out some noise such as blanks or otherwise non-printable characters.

The output of the command is definitely more rewarding: file *avira_free_antivirus_en.exe* was requested at frame number 74,427. This also explains all the ACKs between these 2 hosts: a large file was transferred and the ACKs are simply the acknowledgements taking place throughout the file transfer. In Wireshark, such a conversation can be viewed as a stream and luckily, *tshark* also supports streams using *–e tcp.stream*:

```
$ tshark -r tap_capture_Sunday.pcap -n  -R "frame.number>74420 &&
frame.number<74430" -T fields  -e tcp.stream -e frame.number
720     74420
        74421
        74422
744     74423
740     74424
744     74425
744     74426      ← this is where the first ACK was found
744     74427      ← this is where the URI was found
720     74428
744     74429
744     74430
```

In the above *tshark* command, the range of frames of interest is further limited and the stream ID for this file transfer is revealed to be 744. The stream appears to start at frame # 74,423. If that is truly the case, then 74,423 should be a SYN and 74,425 should be a SYN-ACK. This would complete a 3-way handshake with the ACK we found at frame 74,426. By extracting the TCP flags with *tshark*, this can be confirmed:

```
$ tshark -r tap_capture_Sunday.pcap  -n -R "frame.number==74423 ||
frame.number==74425 || frame.number == 74426" -T fields -e tcp.flags
0x0002          Note 0000001₂ => [C E U A P R S F ]
```

0x0012 *Note 00010010$_2$* => [C E U **A** P R **S** F]

0x0010 *Note 00010000$_2$* => [C E U **A** P R S F]

In the *tshark* command above, the logical connector OR (||) is used to look at the 3 packets that are of interest and TCP flags (*-e tcp.flags*) are extracted. This is indeed the 3-way handshake for this particular connection.

At this point, the reader may wonder: "Is there any point to this besides learning a new tool? How is tshark more useful than Wireshark, which provides all this functionality in a nice graphical user interface?" The answer to this lies in the latter question: *Wireshark* is indeed an amazing GUI tool but *tshark*, by its command-line nature, can generate raw data in a form (simple text) that is easily manipulated. This has already been demonstrated by combining *tshark* with *awk*, *sort, uniq*, etc. These command combinations can easily be wrapped in a simple shell script to perform a basic packet capture analysis with no human intervention.

3.2.2. IPv4 – Do-it-yourself IP geolocation

It would be interesting to know where traffic is heading as it exits a network, and this can be achieved using *tshark*. Numerous web sites offer IP geo-location and some even offer a web API. This means that the *for-loop* technique with *curl* of section 3.1 could be used once more. There is a better way though, as *Wireshark* supports IP geo-location (Stewart, 2010) and so does *tshark*. Appendix B shows how to compile and configure *tshark* to make use of Maxmind's IP geo-location database (Maxmind, 2012). Once *tshark* is able to query that database, the following can provide a view of where the traffic is going:

```
$ tshark –o "ip.use_geoip: TRUE" -r tap_capture_Sunday.pcap -n -T fields  -e
ip.dst -e ip.geoip.dst_country -R "ip.dst != 192.168.1.0/24" | sort | uniq -c | sed 's/,//g' |
sort -rn | awk ' $2!="" ' | awk '{for (i=2; i<=NF; i++) printf "%s ",$i;printf ","$1 ;
printf "\n";}' | tee /tmp/externalIPs_With_Location.dat
```

Germany ,49079

Canada ,29513

United States ,21344

United Kingdom ,358

Russian Federation ,41

In the *tshark* command above, IP geolocation is turned on (*-o "ip.use_geoip: TRUE*) and the destination IP (*-e ip.dst*) as well as the destination country (*-e ip.geoip.dst_country*) are extracted. Packets bound for the internal network are excluded (*-R "ip.dst != 192.168.1.0/24"*). The *sort*/*uniq* command that follows creates a tally. The *sed* command is then used to remove extraneous commas. For example, 'Korea, Republic of' becomes 'Korea Republic of'. The following *sort –rn* command organizes the results in reverse numeric order. The first *awk* command checks the second field and filters out lines where it is blank. This filters out destination IP addresses whose countries are unknown to the Maxmind database. Finally, the more complex-looking *awk* simply re-orders the fields to present the country first and the packet count second. The result is in comma-separated format and shows each country with the total number of packets that were sent there. *Gnuplot* can then be run using the commands found in *GEOIPdistrib.gnuplot* (see appendix C) to obtain a distribution graph, as shown in Figure 5.

$ gnuplot < GEOIPdistrib.gnuplot

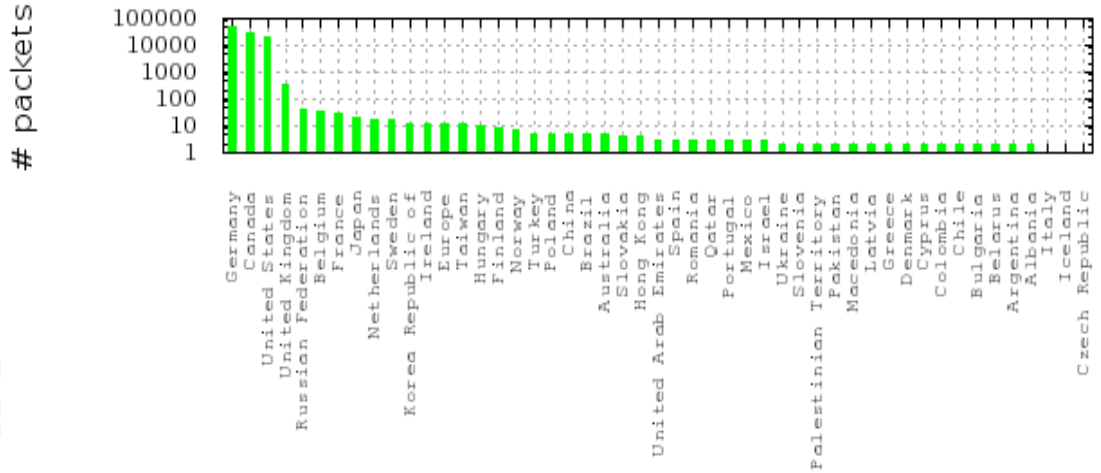

Figure 5 Distribution of destination packets by country

Note that the example given above simply shows the distribution of the number of packets sent to particular countries. It does not illustrate the total number of bytes sent, or the number of separate conversations taking place with hosts located in those countries. Creating these types of reports would require a bit more processing yet the extraction procedure and the tools used would not differ much from what was presented already.

Is this distribution graph really useful? The 2012 Websense Threat report (Websense, 2012) shows the USA, Canada and Germany as the top countries for malware re-directs, hosting malware and phishing attacks. Figure 5 simply appears to confirm these findings. Furthermore a botnet controlled from Russia can use bots located anywhere in the world, as demonstrated by the infamous Blackhole exploit kits (Howard, 2012). On the other hand, if network traffic consistently makes its way to a foreign country that the network owner has no business relationship with, this may warrant further analysis. This means that while the country where network traffic is heading may not always be relevant, it can still prove useful.

3.3. Application layer analysis

3.3.1. DNS – Ad & malware threats correlation

DNS is a key protocol that security analysts rely on when investigating incidents. DNS requests can indicate that a connection was made by a host to a malicious site. Tracking DNS requests is also commonly used when doing dynamic malware analysis (Sikorski & Honig, 2012). It is no wonder then that malicious hackers use DNS-related techniques such as fast-flux to attempt to cover their tracks (Salusky & Danford, 2007). Gathering statistics on DNS queries should therefore be a high priority. Luckily, extracting DNS queries that were successfully made is relatively easy with *tshark*:

$ tshark -r ./tap_capture_Sunday.pcap –nn -T fields -e ip.src -e dns.qry.name -R "dns.flags.response eq 0" | sort | uniq| tee dnsList

192.168.1.110 online-cahoot.com

192.168.1.120 accounts.youtube.com

192.168.1.120 asdsystempro.info

192.168.1.120 best11.co.kr

[...]

The *tshark* command above lists 2 fields: the IP of the host that made the request (*-e ip.src*) and the actual dns query (*-e dns.qry.name*). A filter is applied to ensure that a response was obtained from the queried DNS server (*-R "dns.flags.response eq 0*). The s*ort/uniq* command that follows ensures that a list of unique queries per querying host is produced. Results are printed on the screen as well as saved to a file called *dnsList*.

Such a list can be overwhelming and establishing a baseline of acceptable sites by poring through this list is likely unfeasible. Placing users and hosts behind a proxy server would be a better solution, since a proxy server can easily enforce a strict policy through whitelists and blacklists. With that said, more can be done with the list produced by *tshark*. For instance, that list can be correlated to a database of domains known to be potentially malicious (including sites that distribute malware, engage in phishing, etc.). One such database is hpHosts (hpHosts, 2012). It provides a web API to determine if a domain is listed as well as its classification. Correlating the domains extracted by *tshark* with this API is as simple as a GET request:

$ curl "http://verify.hosts-file.net/?v=pcapAnalyst&s=online-cahoot.com&class=true"

Listed,PSH

In the example above, hpHosts replies with *Listed,PSH* which means that this particular domain is known to hpHosts and is flagged as a **P**hi**SH**ing threat. If the site was not known to hpHosts, the reply would be *Not Listed*. As was done in section 3.1, setting up a *for-loop* with *curl* to iterate through all successful DNS requests is relatively simple:

$ for DOMAIN in `cat dnsList | awk '{print $2}'`; do echo $DOMAIN" :::: `curl \"http://verify.hosts-file.net/?v=pcapAnalyst&s=${DOMAIN}&class=true\" -o hpHosts_reply -s ; cat hpHosts_reply `" ; done | grep -v "Not Listed"

online-cahoot.com :::: Listed,PSH *Note: PSH=Phishing*

asdsystempro.info :::: Listed,EMD *Note: EMD=Malware distribution*

One great advantage of querying hpHosts (or other such web databases that offer an API) is that neither a new DNS query to the domain nor a visit to an actual page is required. The initial analysis is therefore done stealthily. Many other malware/phishing

online databases are available, notably Google's own Safe Browsing API (Google, 2012) which Google has integrated into its Chromium web browser.

3.3.2. IMAP - Extracting email messages

In the Leaking Vault 2011, Suzanne Widup states that: "Many [malware-related security incidents] begin in the form of an email" (Widup, 2012). These emails will typically contain a link to some malicious site, or an attachment that carries a malicious payload. Being able to extract the contents of email messages is therefore something that should matter to security analysts. Since IMAP is one of the main email protocols, this section covers techniques on how to extract email messages from unencrypted IMAP traffic.

The first thing to consider when looking at IMAP traffic is whether or not an email message has been read. Since *tshark* supports the IMAP protocol, it can decode the actual IMAP response, which includes client commands, server responses as well as the content of the emails (header and body). Here are the keywords indicating that a response contains a message:

FETCH (UID 5 BODY[] {17196}

The above response from an IMAP server simply means: *Here is the body of message with UID=5 which you requested me to fetch. It is 17,196 bytes in size*. Knowing this, a *tshark* command can be crafted to seek out all such emails:

```
$ tshark -r mail3.pcap -n -T fields -e tcp.stream -e tcp.ack -e ip.src -e ip.dst -e
imap.response  -R '(imap.response contains "FETCH") && (imap.response contains
"BODY[]")' | sed 's/}.*$/}/g'
1       358     192.168.1.80   192.168.1.10   1 FETCH (UID 5 BODY[] {17196}
1       389     192.168.1.80   192.168.1.10   2 FETCH (UID 6 BODY[] {46120}
```

In the *tshark* command above, the TCP stream number (*-e tcp.stream*), the TCP acknowledgement number (*-e tcp.ack*) as well as the source (*-e ip.src*) and destination (*-e ip.dst*) IP addresses are extracted. The IMAP server response (*-e imap.response*) is also extracted. A filter is applied so that only IMAP responses that contain strings FETCH and BODY[] (*-R '(imap.response contains "FETCH") && (imap.response contains "BODY[]")'*) are considered. Note how single quotes are used to define the filter and

double-quotes are used to define the string to search for. Without double-quotes, *tshark* would fail while searching for BODY[] since the square brackets are considered special characters. The final *sed* command truncates the IMAP server response after the size in bytes. Without this *sed* command, the beginning of the actual email (including headers) would be displayed as shown here:

$ tshark -r mail3.pcap -n -T fields -e tcp.stream -e tcp.ack -e ip.src -e ip.dst -e imap.response -R '(imap.response contains "FETCH") && (imap.response contains "BODY[]")'

*1 358 192.168.1.80 192.168.1.10 1 FETCH (UID 5 BODY[] {17196}**,<fbegin1@franky.tech>,fbegin1@**[…]*

Obviously, an email message will not necessarily fit inside a single packet and the goal is to be able to extract each message individually. In the above example, there appear to be 2 such messages. How can they be told apart?

In section 3.2.1, *tcp.stream* was used to look at a specific communication between two hosts. Unfortunately, the TCP stream number is the same (equal to 1) for both emails. This is easily explained as both were retrieved inside the same IMAP session. Fortunately, the TCP acknowledgement numbers are different (358 and 389). As it turns out, this can be used to extract the messages.

When a client requests to see an email from an IMAP server, it sends the server a special request to look at the body of the email message. Through some simple trial and error, this was found at frame # 133:

$ tshark -r mail3.pcap -n -T fields -e frame.number -e tcp.ack -e tcp.seq -e tcp.len -e ip.src -e ip.dst -e imap.request -R "frame.number==133"

133 1957 327 31 192.168.1.10 192.168.1.80 UID FETCH 5 ***BODY.PEEK[]***

Note that the sequence number used by the client is 327 and that the size of the request is 31 bytes. These add up to 327 + 31 = 358, the acknowledgement value found previously for the first email message. From this point on, as the client keeps acknowledging data received from the server, it will send ACKs that are of size 0 (just as was seen in section 3.2.1). This in turn means that every response packet from the server for the retrieval of that email message can be filtered with *–R "tcp.ack=358"*. This allows for the extraction of the whole message:

$ tshark -r mail.pcap -n -T fields -e imap.response -R "tcp.ack==358"

1 FETCH (UID 5 BODY[]
{17196},<fbegin1@franky.tech>,fbegin1@mail.franky.tech,fbegin1@mail.franky.tech,by mail.franky.tech (Postfix, from userid 1000),8F8E94364A; Wed, 7 Nov 2012 21:27:02 -0700 (MST),The docs you asked
for,<20121108042702.8F8E94364A@mail.franky.tech>,Wed, 7 Nov 2012 21:27:02 -0700 (MST),fbegin1@franky.tech (me@here.com),CHARTER OF RIGHTS AND FREEDOMS,Part I of of the Constitution Act, 1982
[…]
This Part may be cited as the Canadian Charter of Rights and Freedoms.,OK Fetch completed.

The same can be done using filter *-R "tcp.ack=389"* to extract the second message. The astute reader might wonder why the TCP acknowledgement numbers are so small. After all, most modern operating systems should be configured to produce large, random Initial Sequence Numbers (ISN) to prevent a malicious attacker from hijacking a TCP session (Zalewski, 2001). These abnormally small numbers can easily be explained: just like *Wireshark*, *tshark* uses relative sequence numbers by default. This setting can be overridden using *-o tcp.relative_sequence_numbers:FALSE*: to reveal the actual sequence numbers

$ tshark -r mail3.pcap -o tcp.relative_sequence_numbers:FALSE -n -T fields -e frame.number -e tcp.ack -e tcp.seq -e tcp.len -e ip.src -e ip.dst -e imap.request -R "frame.number==133"

*133 1542488278 **752795686** 31 192.168.1.10 192.168.1.80 UID FETCH 5 BODY.PEEK[]*

3.3.3. DHCP - Passive host fingerprinting

Early on, this paper made mention of the *Wireshark Display Filter Reference* as a way to navigate the various protocols. It would be nice, though, if *tshark* could present the analyst with a dissected view of all the protocols it understands for a given set of packets. As it turns out, this view is available by using either the packet details view (*-V*) seen in section 3.1 or by exporting packets using the *Packet Details Markup Language*.

$ tshark -n -r tap_capture_Sunday.pcap -T pdml -R "bootp.option.value[0:1]==08" > dhcp_08

In the above *tshark* command, a filter is applied to extract all packets where the first byte of the BOOTP protocol options is set to 08 (*-R "bootp.option.value[0:1]==08"*). This indicates a client making a DHCP request. Those packets are then saved to a file called *dhcp_08* using the *Packet Details Markup Language* (*-T pdml*).

The resulting file contains all the details of the packets that were extracted. The file is in XML format and is best viewed in a browser or a text editor that supports syntax highlighting. In this particular case, some interesting details are revealed when looking at the various fields, as shown in Figure 6:

```
</field>
<field name="bootp.option.type" showname="Option: (12) Host Name" size="10" pos="294" show="12" valu
  <field name="bootp.option.length" showname="Length: 8" size="1" pos="295" show="8" value="08"/>
  <field name="bootp.option.value" showname="Value: 48616c323530304b" hide="yes" size="8" pos="296"
  <field name="bootp.option.hostname" showname="Host Name: Hal2500K" size="8" pos="296" show="Hal25C
</field>
<field name="bootp.option.type" showname="Option: (60) Vendor class identifier" size="10" pos="304"
  <field name="bootp.option.length" showname="Length: 8" size="1" pos="305" show="8" value="08"/>
  <field name="bootp.option.value" showname="Value: 4d53465420352e30" hide="yes" size="8" pos="306"
  <field name="bootp.option.vendor_class_id" showname="Vendor class identifier: MSFT 5.0" size="8" p
</field>
<field name="bootp.option.type" showname="Option: (55) Parameter Request List" size="15" pos="314" s
  <field name="bootp.option.length" showname="Length: 13" size="1" pos="315" show="13" value="0d"/>
```

Figure 6 Packet Details Markup Language XML export

In DHCP, a well-behaved client will identify itself to a DHCP server by providing information such as a hostname and even its class of system (operating system). This in turn can help craft a *tshark* command to extract this particular information:

$ tshark -n -r tap_capture_Sunday.pcap -T fields -e ip.src -e bootp.option.hostname -e bootp.option.vendor_class_id -R "bootp.option.value[0:1]==08" | sort | uniq

192.168.1.94 Hal2500K MSFT 5.0

192.168.1.64 beginkidspc MSFT 5.0

In the *tshark* command above, the source IP address (*-e ip.src*), the hostname (*bootp.option.hostname*) and operating system (*-e bootp.option.vendor_class_id*) of the hosts making DHCP requests are extracted. Two unique Microsoft Windows hosts are identified. This form of passive fingerprinting can be very useful in determining the hosts that are part of a given network (LaPorte & Kollman, 2007). Specialized OS

fingerprinting tools such as Satori (Kollman, 2012) and p0f (Zalewski, 2012) rely on such techniques as well.

In the case of DHCP fingerprinting, there is even a fingerprint database available (fingerbank.org, 2012). Let us take a closer look at one of the hosts identified as running some version of Windows to determine if a more precise fingerprint can be obtained using this database.

$ tshark -c 1 -n -r tap_capture_Sunday.pcap -T fields -e ip.ttl -e ip.src -e bootp.option.hostname -e bootp.option.value -R 'bootp.option.vendor_class_id =="MSFT 5.0" && bootp.option.hostname=="Hal2500K"'

128 0.0.0.0 Hal2500K 01,01:50:e5:49:c9:6a:50,48:61:6c:32:35:30:30:4b,4d:53:46:54:20:35:2e:30,01:0f:03:06:2c:2e:2f:1f:21:79:f9:2b

In the above *tshark* command, the first (*-c 1*) packet related to DHCP activity from host Hal2500K is analyzed. As well as already familiar fields, the time to live of the IP packet (*-e ip.ttl*) and the raw DHCP options (*-e bootp.option.value*) are extracted. In his paper *Chatter on the Wire* (2007), Eric Kollman demonstrated that a TTL of 128 clearly identifies a DHCP client as a Windows host. In the same paper, he also showed that DHCP options can be used to fingerprint the host more accurately. The value of the options listed above is given in hexadecimal. Since the fingerprint database at fingerbank.org uses base 10 values, a change of base is required. The database can then be searched for a match on a subset of the base 10 values that were extracted (highlighted in yellow below):

$ for HEX in `echo "01:50:e5:49:c9:6a:50,48:61:6c:32:35:30:30:4b,4d:53:46:54:20:35:2e:30,01:0f:03:06:2c:2e:2f:1f:21:79:f9:2b" | sed 's/,/ /g' | sed 's/:/ /g'`; do printf '%d,' "0x${HEX}"; done; echo

1,80,229,73,201,106,80,72,97,108,50,53,48,48,75,77,83,70,84,32,53,46,48,1,15,3,6,44,46,47,31,33,121,249,43

In this particular case, 3 potential matches were found: Windows 7, Windows Server 2008 and Windows Vista. Figure 7 shows the first of these matches:

```
▼<fingerprint name="Windows 7" os_name="Windows 7" os_class="Windows" os_vendor="Microsoft Corp." os_url="http://w
 device_type="" device_vendor="" device_url="" comments="" author="cycocrew@orange.fr" last_updated="2010-11-01">
 ▼<dhcp_tests>
    <test weight="1" matchtype="exact" dhcptype="Request" ipttl="128"/>
    <test weight="1" matchtype="exact" dhcptype="Inform" ipttl="128"/>
    <test weight="1" matchtype="exact" dhcptype="Discover" ipttl="128"/>
    <test weight="5" matchtype="exact" dhcptype="Inform" dhcpoptions="53,61,12,60,55"/>
    <test weight="5" matchtype="exact" dhcptype="Discover" dhcpoptions="53,61,12,60,55"/>
    <test weight="5" matchtype="exact" dhcptype="Request" dhcpoptions="53,61,50,12,81,60,55"/>
    <test weight="5" matchtype="exact" dhcptype="Request" dhcpoptions="53,61,50,54,12,81,60,55"/>
    <test weight="5" matchtype="exact" dhcptype="Request" dhcpoption55="1,15,3,6,44,46,47,31,33,121,249,43"/>
    <test weight="5" matchtype="exact" dhcptype="Discover" dhcpoption55="1,15,3,6,44,46,47,31,33,121,249,43"/>
    <test weight="5" matchtype="exact" dhcptype="Inform" dhcpoption55="1,15,3,6,44,46,47,31,33,121,249,43,252"/>
    <test weight="2" matchtype="exact" dhcptype="Request" dhcpvendorcode="MSFT 5.0"/>
    <test weight="2" matchtype="exact" dhcptype="Inform" dhcpvendorcode="MSFT 5.0"/>
```

Figure 7 DHCP fingerprints from fingerbank.org

Tool *p0f* (v. 3) and *satori* were also used on the same packet capture file. *P0f* identified the host as either Windows 7 or 8 while *satori*'s best guesses were in line with our own findings using *tshark*.

```
.-[ 192.168.1.94/49158 -> 65.54.161.12/80 (syn) ]-
|
| client   = 192.168.1.94/49158
| os       = Windows 7 or 8
| dist     = 0
| params   = none
| raw_sig  = 4:128+0:0:1464:8192,8:mss,nop,ws,nop,nop,sok:df,id+:0
|
`----
```

Figure 8 OS identification by p0f

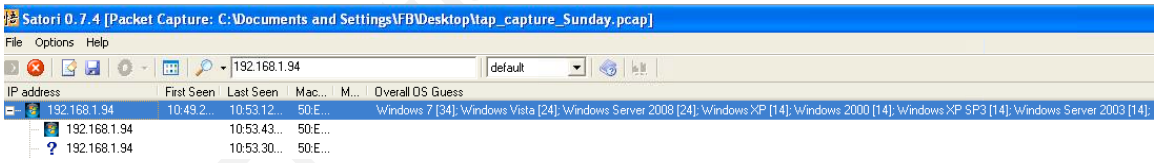

Figure 9 OS identification by Satori

3.3.4. HTTP – Google searches

Research has shown that internet users are influenced by search engines, specifically the order in which results are presented (Hargittai, Fullerton, Menchen-Trevino & Thomas, 2010) and (Pan et al., 2007). This in turn has encouraged malicious attackers to use Search Engine Optimization (SEO) to piggyback onto hot trends to deliver malware, perform phishing attacks, etc. (Flores, 2010). Searches that users perform should therefore be of interest to security analysts. Since Google searches

dominate the US market with more than 65% of explicit core searches (comScore, 2012), this section focuses on that particular search engine.

Extracting Google searches from a packet capture is actually an interesting challenge. In 2008, Google introduced Google Autocomplete (Google, 2012), which provides suggestions as one is typing for keywords, as shown in Figure 10.

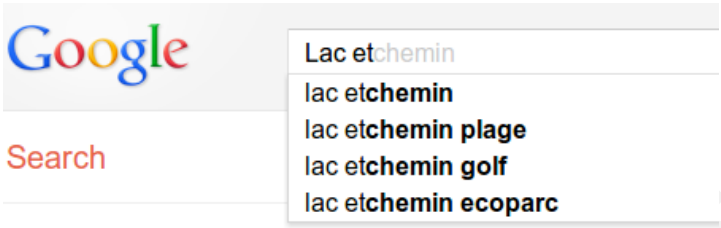

Figure 10 Google Autocomplete

As will be shown shortly, the challenge with Autocomplete turned on is that multiple GET queries take place as letters are typed. A search for "widget" for instance would generate 6 separate GET queries, one for each letter of the word "widget". First though, these queries need to be clearly identified in a packet capture.

$ tshark -r google.pcap –nn -T fields -e ip.src -e http.request.full_uri -R 'http.request.method=="GET" && http.request.full_uri contains "&q=" && http.request.full_uri contains "google.ca" ' | head -1

192.168.1.94
*http://www.google.ca/s?hl=en&gs_nf=3&cp=1&gs_id=1j&xhr=t**&q=**l&pf=p&sclient=psy-ab&oq=&gs_l=&pbx=1&bav=on.2,or.r_gc.r_pw.r_qf.&fp=8963f08acaaa2898&bpcl=35466521&biw=1408&bih=327&bs=1&tch=1&ech=11&psi=8vCNUIWII4jWigLFiYDICA.1351479538297.1*

The above *tshark* command extracts the source IP address (*-e ip.src*) and the full query URI (*-e http.request.full_uri*) of the packet. The filter has 3 conditions:

- The packet is a GET request (*http.request.method=="GET"*)
- The full query URI contains **&q=** (*http.request.full_uri contains "&q="*). This GET parameter defines what is being searched for
- The query is made to Google (*http.request.full_uri contains "google.ca"*).

This particular query to Google shows a GET request for the letter 'l' (*&q=l*). There are numerous other parameters in the GET request that simply introduce noise in the output. A little *sed* magic can be used to provide a more focused output:

```
$ tshark -r google.pcap –nn -T fields -e ip.src -e http.request.full_uri  -R
'http.request.method=="GET" && http.request.full_uri contains "&q=" &&
http.request.full_uri contains "google.ca" ' | sed 's/http.*\&q=/QUERY: /g'  | sed
's/\&.*$//g'
```

```
192.168.1.94   QUERY: l
192.168.1.94   QUERY: la
192.168.1.94   QUERY: lac
192.168.1.94   QUERY: lac%20
192.168.1.94   QUERY: lac%20e
192.168.1.94   QUERY: lac%20et
```

The result is what was expected: the user typed letters **l**, **a**, **c**, a space (%20 in Unicode), then **e** and **t**. This resulted in 6 separate GET queries made to Google.

3.3.5. HTTP - Extracting a file

In section 3.2.1, the analysis of packet sizes led to the discovery that a particular binary (*avira_free_antivirus_en.exe*) had been downloaded, but what if this file contained malware? There are of course challenges with current malware detection techniques, and researchers continue to explore new methods of dealing with this issue (Kolbitsch, et al., 2009).

That being said, retrieving binaries from packet-captures is important to security analysts. Having access to the same binary that may be the source of a security breach would allow the analyst to perform more thorough scans on it. For example, the binary (or its checksum) could be submitted to the Virustotal database (virustotal.com, 2012) to get an 'opinion' on that binary from more than 40 antivirus solutions. Having the binary could also allow a security analyst to deploy it in a virtual sandbox for further analysis.

This section therefore focuses on extracting a binary file from a packet capture. In this particular example, the file was transferred via HTTP. Relying on techniques similar to what was covered in 3.2.1, the stream where the transfer took place can be identified. That stream has a value of 4 and starts at frame 19:

$ tshark -r fileTransfer.pcap -n -T fields -e frame.number -e tcp.stream -e http.request.uri | grep exe$

19 4 /download/keepnote-0.7.8.exe

From frame 20 and onwards, the client should be receiving file *keepnote-0.7.8.exe*. Based on what was seen in section 3.3.2, the server will use the same ACK repeatedly while the client replies with acknowledgements that increase in values as it receives the data:

$ tshark -r fileTransfer.pcap -n -T fields -e frame.number -e ip.src -e ip.dst -e tcp.ack -R "tcp.stream eq 4 && frame.number >=20" | head

20 97.74.144.172 192.168.1.10 682 ← Server's initial ACK

21 97.74.144.172 192.168.1.10 682 ← Server's ACK remains the same

22 192.168.1.10 97.74.144.172 1449 ← Client has received 1,449 bytes

23 97.74.144.172 192.168.1.10 682 ← Server's ACK remains the same

24 192.168.1.10 97.74.144.172 2897 ← Client has received 2,897 bytes

25 97.74.144.172 192.168.1.10 682 ← Server's ACK remains the same

[...]

Assuming that the communication flowed perfectly, the client ACKs should grow without any repeats. This can easily be verified:

$ tshark -r fileTransfer.pcap -n -T fields -e tcp.ack -R "tcp.stream eq 4 && frame.number >=20 && ip.src==192.168.1.10" | sort | uniq -c | sort -rn | head

761 5262033

495 2345761

43 3324609

31 2997361

In the above *tshark* command, a filter is applied to get the right stream as well as source IP address. As was done before, the *sort*/*uniq*/*sort* combination is used to create a tally of unique entries with a count for each. At around the 5,262,033th byte of data transferred, numerous duplicate ACKs (761 to be precise) were sent by the client, as it was requesting a packet that likely got lost in transit. This happened a few more times during this transfer.

This illustrates the difficulty one faces if trying to reconstruct a file from a stream using only *tshark*: these duplicate ACKs would need to be taken into account. Rather

than force the issue, a better tool needs to be introduced: *tcpflow* (Garfinkel, 2012). This tool can capture or replay data and store it in separate flows that are more suitable for analysis.

$ tcpflow -d 0 -r fileTransfer.pcap
$ ls
067.228.181.218.00080-192.168.001.010.40797
192.168.001.010.41823-097.074.144.172.00080
097.074.144.172.00080-192.168.001.010.41823
fileTransfer.pcap

As shown above, *tcpflow* was able to extract 3 flows. The one in bold contains the file that was transferred. Unfortunately, that flow file also contains HTTP headers, as can be seen if that file is read into a hex editor:

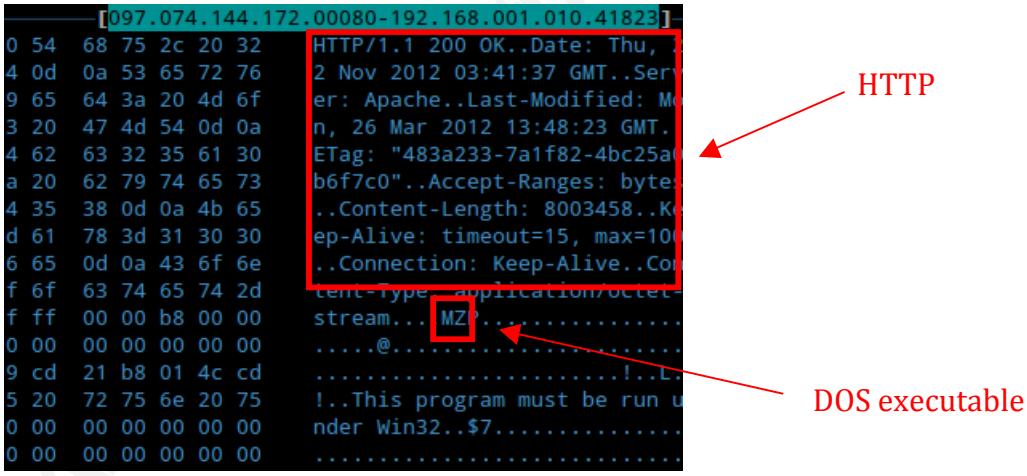

Figure 11 Hex dump of file extracted using tcpflow

To obtain the file as it was sent, all those headers need to be removed. This involves finding the start/end of the file, then extracting between these two boundaries

In this particular case, this can be done using offset 0x12C (decimal 300), where string **MZ** can be seen. This string is an indication we are dealing with an Intel MZ DOS EXE file. From there, *dd* can be used to extract the file, skipping the first 300 bytes:

$ dd if=097.074.144.172.00080-192.168.001.010.41823 bs=1 skip=300 of=fileWithoutHeaders.exe

François Bégin, francois.begin@telus.com

8003458+0 records in
8003458+0 records out
8003458 bytes (8.0 MB) copied, 8.05721 s, 993 kB/s

$ md5sum fileWithoutHeaders.exe keepnote-0.7.8.exe
c7b70ef71eb35aec6aa00fee0168e152 fileWithoutHeaders.exe
c7b70ef71eb35aec6aa00fee0168e152 keepnote-0.7.8.exe

This last exercise proved to be a lot of work. In addition, the end of the file fell cleanly at the end of the flow, which might not always be the case. One might be tempted to consider an alternative to *tshark/tcpflow* for file extraction, but learning how to dissect and analyze packets was one of the goals of this paper. Although the last example was more complicated, it highlights both tools and skills that will serve a security analyst well.

Furthermore, relying on more professional tools may not always yield the expected results. The free edition of *Network Miner* (Netresec.com, 2012) for instance was able to dissect the session but could not extract a file from it, as evidenced by Figure 12. *NetWitness Investigator* (Netwitness.com, 2012) for its part was able to extract the file, but the MD5 checksum of the file it extracted differs from the original file, as shown in Figure 13. Performing dynamic analysis on the retrieved file in a virtual sandbox would therefore prove much more complicated. Since the file extracted is not the same as the one received, submitting it to VirusTotal would likely end up misleading the analyst. Sometimes, a tedious analysis using low-level tools such as *tshark*, *tcpflow* and a hex editor may prove to be the best approach.

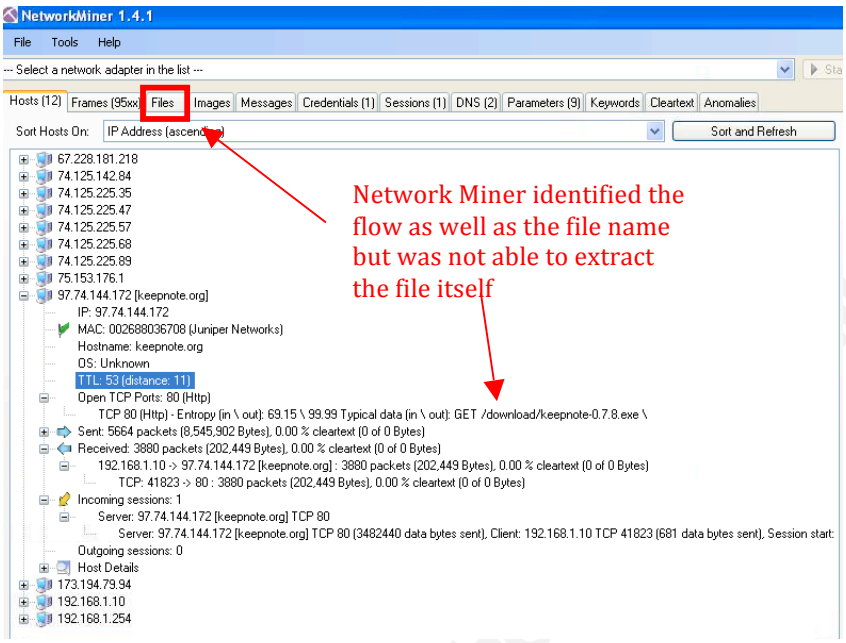

Figure 12 NetworkMiner free edition

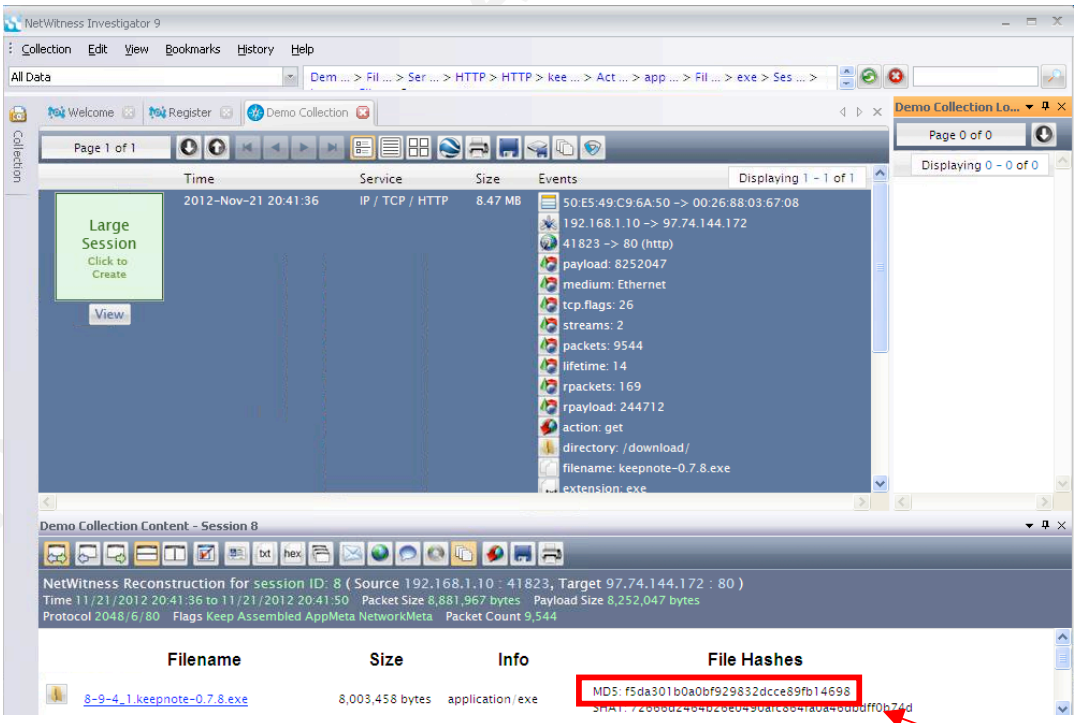

Figure 13 NetworkWitness file extraction

NetworkWitness did not extract the file correctly from the packet capture

François Bégin, francois.begin@telus.com

4. Conclusion

Tshark is a powerful tool for packet capture analysis. Its strength does not derive from an attractive graphical user interface. That is the strength of its big brother *Wireshark*. Where *tshark* does truly shine, though, is in the ability to provide data to an analyst in a simple, text-based output. Such output can then be formatted and fed to external data sources or applications in order to find useful correlations. Moreover, *tshark* commands and their outputs can easily be scripted to obtain these results. This paper showed examples using UNIX shell, allowing an analyst to easily build up a toolkit adapted to their needs. Some examples covered by this paper included plotting geo-ip location, correlating DNS queries to potentially malicious sites, fingerprinting hosts, extracting emails/binaries from a packet capture and even a rudimentary form of keylogging related to Google searches.

As was seen in the latter sections of this paper, more specialized tools that perform the same tasks more efficiently can be found, but using *tshark* has another important advantage: by being such a low-level tool, *tshark* keeps the analyst grounded with key concepts (network streams, ACK numbers) that are sometimes hidden by GUI-driven tools. Staying 'close to the ground' can prove useful. As was seen in the last section, the more specialized tools may not always find the expected results. When these tools fail, it is important to be able to perform the equivalent task with low-level tools.

Hopefully this paper has managed to convey these advantages to using *tshark*, and security analysts will consider adding it to their toolkit.

5. References

Braun, L., Didebulidze, A., Kammenhuber, N., & Carle, G. (2010). *Comparing and Improving Current Packet Capturing Solutions based on Commodity Hardware*. Proceedings of the 10th ACM SIGCOMM conference on Internet measurement. Retrieved from http://dl.acm.org/citation.cfm?id=1879168

Comer, D. (2000). *Internetworking with TCP/IP Volume 1* (4th ed.). Upper Saddle River, NJ. Prentice Hall

comScore (2012). *comScore Releases September 2012 US Search Engine Rankings*. Retrieved from http://www.comscore.com/Insights/Press_Releases/2012/10/comScore_Releases_September_2012_U.S._Search_Engine_Rankings

Curl project (2012). Retrieved from http://curl.haxx.se/

CVE-2007-3798 (2007). *Integer overflow in print-bgp.c.* Common Vulnerabilities Exposure. Retrieved http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-3798

Donahue, G. A. (2011). *Network Warrior*. 2nd edition. Sebastopol. O'Reilly Media Inc.

Evergreensys.com (2007). *Nine Steps to Implement a Successful CMDB Project*. Retrieved from http://www.evergreensys.com/Configuration-Management-Database-How-to-Develop-CMDB/

Fingerbank.org (2012). *DHCP fingerprints*. Retrieved from http://www.fingerbank.org/

Fishburne, D. (2012). *Network Operational Investigating 101: The Network Diagram*. Network World. August 2012. Retrieved from http://www.networkworld.com/community/blog/network-operational-investigating-101-noi-101-network-diagram

Flores, R. (2010). *How Blackhat SEO Became Big*. Trend Micro Research Paper. November 2010. Retrieved from http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp__blackhat-seo-became-big.pdf

Garfinkel, S. (2012). *Tcpflow*. Retrieved from https://github.com/simsong/tcpflow

Gnuplot project (2012). Retrieved from http://www.gnuplot.info

François Bégin, francois.begin@telus.com

Google (2012).

Google Autocomplete.

http://support.google.com/websearch/bin/answer.py?hl=en&answer=106230

Google Safe Browsing API.

https://developers.google.com/safe-browsing/

Hargittai, E, & Fullerton, L, & Menchen-Trevino, E., & Thomas, K. (2010). *Trust Online: Young Adults' Evaluation of Web Content*. International Journal of Communication 4. Retrieved from http://ijoc.org/ojs/index.php/ijoc/article/view/636

Howard, F. (2012). *Exploring the Blackhole Exploit Kit*. SophosLabs, UK. Retrieved from http://sophosnews.files.wordpress.com/2012/03/blackhole_paper_mar2012.pdf

hpHosts (2012). Community managed hosts file providing an additional layer of protection against access to ad, tracking and malicious websites. http://hosts-file.net/

Juels, A. & Yen, T.F. (2012). *Sherlock Holmes and the Case of the Advanced Persistent Threat*. Usenix LEET '12. Retrieved from https://www.usenix.org/conference/leet12/sherlock-holmes-and-case-advanced-persistent-threat

Karpilovsky, E, Gerber, A, Pei, D, Rexford, J, & Shaikh, A. (2009). *Quantifying the Extent of IPv6 Deployment*. PAM '09 Proceedings of the 10th International Conference on Passive and Active Network Measurement. Springer-Verlag. Berlin. Retrieved from http://www2.research.att.com/~ashaikh/papers/ipv6-study-pam09.pdf

Kaufman, C., Perlman, R., & Speciner, M. (2002). *Network security: Private communication in a public world*. (2nd ed.). Upper Saddle River: Prentice Hall.

Kolbitsch, C., Comparetti, P.M., Kruegel, C. Kirda, E., Zhou, X., Wang, X. (2009). *Effective and Efficient Malware Detection at the End Host*. SSYM'09 Proceedings of the 18th conference on USENIX security symposium. Retrieved from http://www.usenix.org/events/sec09/tech/full_papers/kolbitsch.pdf

Kollman, E. (2007). *Chatter on the Wire: A look at DHCP traffic*. Retrieved from http://myweb.cableone.net/xnih/Papers.htm

Kollman, E. (2012). *Satori*. Retrieved from http://myweb.cableone.net/xnih/mortalx.htm

LaPorte, D., & Kollman, E. (2007). *Using DHCP for Passive OS Identification*, BlackHat Japan. Retreived from http://myweb.cableone.net/xnih/download/bh-japan-laporte-kollmann-v8.ppt

MACVendorLookup.com (2012). *MAC Address Lookup API*. http://www.macvendorlookup.com/api

Maxmind (2012). *GeoLite Country Free*. http://dev.maxmind.com/geoip/geolite

Netresec.com (2012). *Network Miner*. Retrieved from http://www.netresec.com/?page=NetworkMiner

Netwitness.com (2012). Netwitness Investigator. Retrieved from http://netwitness.com/products-services/investigator

Norton, Q. (2012). *How Anonymous Picks Targets, Launches Attacks, and Takes Powerful Organizations Down.* Wired magazine, July 2012. Retrieved from http://www.wired.com/threatlevel/2012/07/ff_anonymous

O'Donnell, G, & Casanova, C. (2009). *The CMDB Imperative: How to Realize the Dream and Avoid the Nightmares* (1st ed.). Prentice Hall

Pan, B, & Hembrooke, H, & Joachims, T, & Lorigo, L., & Gay, G, & Granka, L. (2007). *In Google We Trust: Users' Decisions on Rank, Position, and Relevance*. Journal of Computer-Mediated Communication, 12(3). Retrieved from http://jcmc.indiana.edu/vol12/issue3/pan.html

Salusky, W., & Danford, R. (2007). *Know Your Enemy: Fast-Flux Service Networks. An Ever Changing Enemy*. The Honeynet Project. Retrieved from http://www.honeynet.org/papers/ff

Sikorski, M, & Honig, A. (2012). *Practical Malware Analysis*. No Starch Press

Stewart, P. (2010). *Geolocation in Wireshark*. Retrieved from https://learningnetwork.cisco.com/blogs/vip-perspectives/2010/12/11/geolocation-in-wireshark

Turner, A. (2012). *Tcpreplay*. Retrieved from http://tcpreplay.synfin.net/

François Bégin, francois.begin@telus.com

Virustotal.com (2012). http://www.virustotal.com

Websense (2012). *Websense Threat Report 2012*. websense.com. Retrieved from http://www.websense.com/content/websense-2012-threat-report-download.aspx

Widup, S. (2011). *The Leaking Vault 2011. Six Years of Data Breaches.* The Digital Forensics Association. Retrieved from http://www.digitalforensicsassociation.org/storage/The_Leaking_Vault_2011-Six_Years_of_Data_Breaches.pdf

Wireshark Project (2012).

Wireshark. Retrieved from http://wireshark.org

Ethernet Capture Setup: http://wiki.wireshark.org/CaptureSetup/Ethernet.

Wireshark Display Filter Reference. http://www.wireshark.org/docs/dfref/.

Zalewski, M. (2001). *Strange Attractors and TCP/IP Sequence Number Analysis*. Retrieved from http://lcamtuf.coredump.cx/oldtcp/tcpseq.html

Zalewski, M. (2012). *P0f*. Retrieved from http://lcamtuf.coredump.cx/p0f3/

6. Appendix A: Packet capture performance comparison

In the test environment set up for this paper, the Desktop user segment has a bandwidth of 100 mbps. A test was conceived to ensure that the system capturing the data would be able to capture all packets without drops. Prior to introducing the 'sniffing server' into the network, it was attached to an isolated hub along with a high-performance PC. The high performance PC replayed 270 MB of previously captured traffic using *tcpreplay* (Turner, 2012), processing these packets at top-speed (-t) and looping through the capture file 5 times (-l 5)

```
fbegin1@mintBox:capture$ sudo tcpreplay -t -i eth1 -l 5 ./tap_capture_Sunday.pcap
sending out eth1
processing file: ./tap_capture_Sunday.pcap
processing file: ./tap_capture_Sunday.pcap
processing file: ./tap_capture_Sunday.pcap
processing file: ./tap_capture_Sunday.pcap
processing file: ./tap_capture_Sunday.pcap
Actual: 1462645 packets (1399505015 bytes) sent in 115.13 seconds
Rated: 12155867.0 bps, 92.74 Mbps, 12704.29 pps
Statistics for network device: eth1
        Attempted packets:         1462645
        Successful packets:        1462645
        Failed packets:            0
        Retried packets (ENOBUFS): 0
        Retried packets (EAGAIN):  0
```

Figure 14 tcpreplay to test the ability of sniffing server

As shown above, *tcpreplay* was able to generate 92.74 Mbps of traffic on the wire. This is very close to the top theoretical speed of the link to be surveyed (100 Mbps) although this is also something that one is unlikely to witness in real life. *dumpcap* was configured on the sniffing station to capture 15,000 full packets and the test was repeated 3 times.

Surprisingly, *dumpcap* dropped a significant number of packets. The test was repeated with *tcpdump* and that tool only dropped half the number of packets *dumpcap* did. See figure 15 and 16 and Table 1:

```
fbegin1@UbuntuSniffer:/var/tmp/wireshark-1.8.2$ sudo dumpcap -s 0 -q -i eth2 -w /dev/null -c 15000
Capturing on eth2
File: /dev/null
Packets captured: 15000
Packets received/dropped on interface eth2: 15000/1061 (93.4%)
fbegin1@UbuntuSniffer:/var/tmp/wireshark-1.8.2$ sudo dumpcap -s 0 -q -i eth2 -w /dev/null -c 15000
Capturing on eth2
File: /dev/null
Packets captured: 15000
Packets received/dropped on interface eth2: 15000/1085 (93.3%)
fbegin1@UbuntuSniffer:/var/tmp/wireshark-1.8.2$ sudo dumpcap -s 0 -q -i eth2 -w /dev/null -c 15000
Capturing on eth2
File: /dev/null
Packets captured: 15000
Packets received/dropped on interface eth2: 15000/972 (93.9%)
```

Figure 15 dumpcap performance on tcpreplay test

```
fbegin1@UbuntuSniffer:/var/tmp/wireshark-1.8.2$ sudo tcpdump -nn -s 0 -i eth2 -w /dev/null -c 15000
tcpdump: WARNING: eth2: no IPv4 address assigned
tcpdump: listening on eth2, link-type EN10MB (Ethernet), capture size 65535 bytes
15000 packets captured
15538 packets received by filter
516 packets dropped by kernel
fbegin1@UbuntuSniffer:/var/tmp/wireshark-1.8.2$ sudo tcpdump -nn -s 0 -i eth2 -w /dev/null -c 15000
tcpdump: WARNING: eth2: no IPv4 address assigned
tcpdump: listening on eth2, link-type EN10MB (Ethernet), capture size 65535 bytes
15000 packets captured
15536 packets received by filter
529 packets dropped by kernel
fbegin1@UbuntuSniffer:/var/tmp/wireshark-1.8.2$ sudo tcpdump -nn -s 0 -i eth2 -w /dev/null -c 15000
tcpdump: WARNING: eth2: no IPv4 address assigned
tcpdump: listening on eth2, link-type EN10MB (Ethernet), capture size 65535 bytes
15000 packets captured
15496 packets received by filter
495 packets dropped by kernel
```

Figure 16 tcpdump performance on tcpreplay test

Sniffing tool	1st round drops	2nd round drops	3rd round drops	Average drops	Average % dropped
tcpdump	516	529	495	513	3.4%
dumpcap	1061	1085	972	1039	6.9%

Table 1 Packet dropped by tcpdump vs. dumpcap on tcpreplay test

Simple tweaks are available of course. By default, both *dumpcap* and *tcpdump* use a 1 meg capture buffer. Increasing this buffer to 12 Megabytes (-B 12) resulted in the consistent elimination of all the drops in *dumpcap*. *Tcpdump* for its part only required an 8 megs buffer allocation (-B 8192) to achieve a perfect score. Note that the buffer size of *tcpdump* is allocated in Kilobytes rather than Megabytes.

François Bégin, francois.begin@telus.com

7. Appendix B - Compiling *tshark* from source

The following was performed on Ubuntu Server LTS (12.04). First, MaxMind's GeoIP database (for country only) was installed and tested:

$ sudo apt-get install geoip-bin

$ sudo apt-get install libgeoip-dev

$ geoiplookup 66.249.66.204

GeoIP Country Edition: US, United States

In Ubuntu, note that by default, the GeoIP databases ends up in */usr/share/GeoIP*:

$ ls -l /usr/share/GeoIP/

total 6856

-rw-r--r-- 1 root root 1664511 Jan 2 2012 GeoIP.dat

-rw-r--r-- 1 root root 5349447 Jan 2 2012 GeoIPv6.dat

All packages required to build *wireshark* were then installed:

$ sudo apt-get build-dep wireshark

The source code for wireshark 1.8.3 was then retrieved from wireshark.org, uncompressed and untarred. As this was a server and the GUI front-end (*wireshark*) was not going to be used, the *configure* script was run as follows:

$ ./configure --with-geoip --enable-wireshark=no --enable-tshark=yes

$ make

$ sudo make install

To ensure that tshark makes use of the GeoIP databases, a file called *geoip_db_paths* was created in the analyst's home directory under sub-directory *.wireshark*. This file contains the path to the databases

$ ls -l /home/fbegin1/.wireshark/

total 4

-rw-rw-r-- 1 fbegin1 fbegin1 19 Nov 14 22:34 geoip_db_paths

$ cat /home/fbegin1/.wireshark/geoip_db_paths

"/usr/share/GeoIP"

Finally, *tshark* was tested to ensure correct operation with IP geolocation enabled:

$ tshark ***-o "ip.use_geoip: TRUE"*** *-r tap_capture_Sunday.pcap -T fields -e ip.dst* ***-e ip.geoip.country*** *-R "frame.number == 37"*

208.43.253.218 ***United States***

8. Appendix C – gnuplot files

8.1. IPdistrib.gnuplot

gnuplot file to create an IP datagram size distribution graph.

Input: /tmp/IPlen.dat

Output: ./reports/graphs/IPdistrib.png

```
reset
dx=5.
n=2
total_box_width_relative=0.75
gap_width_relative=0.1
d_width=(gap_width_relative+total_box_width_relative)*dx/2.
set term png truecolor
set output "./reports/graphs/IPdistrib.png"
set logscale y 10
set xlabel "Datagram size in bytes"
set ylabel "Number of datagrams"
set grid
set boxwidth total_box_width_relative/n relative
set style fill transparent solid noborder
plot "/tmp/IPlen.dat" u 1:2 w boxes lc rgb"green" notitle
```

8.2. GEOIPdistrib.gnuplot:

gnuplot file to create an IP country location distribution graph.

Input: /tmp/externalIPs_With_Location.dat

Output: ./reports/graphs/ GEOIPdistrib.png

```
reset
dx=5.
n=2
total_box_width_relative=0.75
gap_width_relative=0.1
d_width=(gap_width_relative+total_box_width_relative)*dx/2.
reset
set term png truecolor
set datafile separator ","
set output "./reports/graphs/GEOIPdistrib.png"
set logscale y 10
set size 1, 0.50
set ytics font "courier,10"
set xtics border in scale 1,0.5 mirror rotate by -270 font "courier,8"
set xlabel "Country of destination"
set ylabel "Number of IPs observed"
set grid
set boxwidth total_box_width_relative/n relative
set style fill transparent solid noborder
plot "/tmp/externalIPs_With_Location.dat" u 2:xticlabel(1) w boxes lc
rgb"green" notitle
```