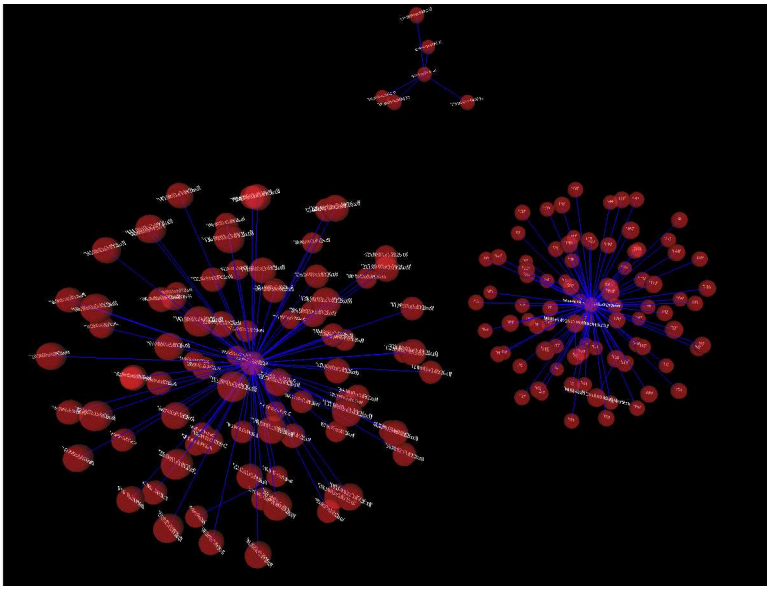
GCIA Practical Assignment
Version 4.0

The Big Barnyard

Raffael Marty, CISSP
<ram@[cryptojail.net|arcsight.com]>
Mountain View, CA 94041

Submission Date: 19th December 2004



Contents

Chapter 1

Executive Summary

This report shows an in-depth analysis of intrusion detection logs gathered between the months of August and November 2002.

The data analyzed shows signs of intrusions and machines compromised by backdoors and worms. The following findings need immediate attention; the analysis in this report gives the necessary details and support for calling the attention to these incidents:

- Multiple machines in the internal network are compromised by a worm. The compromised machines are actively trying to infect other vulnerable machines (see Section 3.1).
- At least one machine is compromised by an FTP attack (see Section 3.1 under "`FTP CWD overflow attempt`). We recommend taking these machines offline immediately.
- External sources are utilizing reconnaissance mechanisms to gather information about the internal networks (see Section 3.2).
- It seems that a DoS attack is targeting a number of internal machines (see Section 3.5[1]). These findings have to be confirmed with some additional source of information. In case of a positive confirmation, ACLs on the border devices should be configured to block all the aggressors.
- Peer to peer traffic was detected by the IDS (see Section 3.5). We recommend to possibly block peer to peer traffic at the border. If this is not a possibility, an awareness session with all the internal users should help reduce this type of activity.

Whereas the above finding should be immediately addressed, there are some more recommendations that should be addressed:

[1]Sections 3.5 and 3.6 are published at `http://raffy.ch/projects/Raffael_Marty_GCIA_Additional_Chapters.pdf`. The end of this Section explains the reason for this.

- The Snort signatures should be tuned. We will see all throughout the report, that the number of false positives is immense.
- We were only given log files from one Snort instance. The amount and type of attacks this sensor can capture is very limited. We recommend deploying more sensors, especially in the internal network. We will see later how another sensor could help us verifying some of the attacks.
- Given the high number of false positives, it would make sense to utilize some kind of a correlation software or even a security information management (SIM) solution. The additional context that a SIM could add to an event would greatly improve their accuracy.
- As we will see later, there do not seem to be any access control devices deployed at all. Even if this is an academic environment, where networks have to be as open as possible, we recommend to filter certain traffic, such as peer to peer that can be abused for illegal purposes (see Section 3.5).
- To limit the amount of malicious traffic (e.g., peer to peer, irc servers, etc.), we recommend to first issue a security policy which disallows these services and second to enforce the policy.

We found that the data sets were already analyzed by many other GCIA students. Instead of repeating their analysis, we decided to approach the problem slightly different, by emphasizing the ways of analyzing such a data set. The paper is therefore a little weaker on the analysis of specific incidents and focuses more on the approach of getting a handle of a big data set. We will first give a very generic overview of the data found in the log files. After determining the topology, we will outline some anomalies (Chapter 2) and then establish some hypotheses on how to find suspicious behavior in a generic way (see Chapter 3).

We had to take two Sections out of this paper as it grew too big. We did not want to loose our main Sections where we came up with interesting ways of analyzing the data but put some of the "ordinary" analysis Sections on a Web page: `http://raffy.ch/projects/Raffael_Marty_GCIA_Additional_Chapters.pdf`.

Before I forget: Thanks to Christian for helping me with AfterGlow[14] and Colby for having a quick glance at the paper before submission.

Chapter 2

Analysis

The analysis for this paper was done on a Linux system, we used tcpdump[23] to read the logs provided. Mergecap(1)[3] was used to combine multiple log files into one, in order to facilitate the analysis of more than one file. To process output from tcpdump, standard UNIX utilities like sed(1) and awk(1) were utilized. For all these tools, their respective man pages will give more information on how exactly they can be used. In addition to all of these tools, We decided to utilize graphical libraries and utilities to generate event graphs for visual analysis.

This chapter will first set the stage for what exactly has been done and what files have been analyzed. Then some first analysis steps are executed and event graphs introduced. This first Chapter will give us a very good understanding of what type of traffic we are dealing with in all the log files. Chapter 3 will then go into some in-depth analysis of a few findings. The exact tools used during the analysis process are explained in Appendix B. There we will walk through an elaborate example of how to graph events. Space limitations for this paper did not allow us to show all the steps necessary to generate each graph in this paper[1].

2.1 Log Data

For the purpose of this analysis, we decided to use 52 log files out of the collection that is made available by SANS[9]. All the files analyzed were recorded by a Snort[16] instance running in binary logging mode. This means that only packets triggering a signature appear in the logs. This fact is going to play an important role later when we analyze the data and try to draw some conclusions.

To work with all the 52 log files, we merged them into one big tcpdump file using mergecap(1).[2] The merging of the log files yielded 324.000 recorded packets as the following command shows:

[1] It would probably also be boring to read through all of them.

[2] The command to generate the merged capture file is: `mergecap -w /tmp/sans 2002.*`

```
tcpdump -nnelr /tmp/sans | wc -l
324461
```

2.2 MAC Addresses

To start the analysis, we want to understand the environment in which the log files were collected. A few simple queries should reveal some of the topology. Firstly it is interesting to figure out what the network looks like that the snort instance was running on. To get an idea of the topology, let us have a look at the MAC addresses[3]:

```
tcpdump -nnelr /tmp/sans | awk '{print $2}' | sort | uniq -c
        141216    00:00:0c:04:b2:33
        183245    00:03:e3:d9:26:c0
```

The output shows the source MAC addresses and the number of times they showed up. It is important to understand that the number of packets from each of the two devices does not have to be the same. It seems interesting that the number of packets that triggered an alert from both devices are about the same. One would think that the number of packets triggering a snort rule would be higher for connections coming into a network and would therefore show a clear asymmetry in these numbers. This might be a first clue about the network topology that we are dealing with. It might not be that we have a clear internal vs. external situation, but something more complex. Let us continue with some statistics and then see what these numbers can tell us.

The next step in our analysis is to find the destination MAC addresses and their counts:

```
tcpdump -nnelr /tmp/sans | awk '{print $4}' | sort | uniq -c
        183239    00:00:0c:04:b2:33
        6         00:00:c0:6b:e9:c6
        141216    00:03:e3:d9:26:c0
```

We have a new device showing up. We will see later what this device is. The other two counts are directly linked to the counts we got before and we actually see that the new device only received traffic and only from device `00:03:e3:d9:26:c0`. Figure 2.1 summarizes our findings in a visual representation.

We can see that the snort sensor is surrounded by three other devices. To understand the role of these three devices, we can try to look up what vendor produces them. The IEEE OUI[8] assignments reveal the following:

```
00:00:0c:04:b2:33       Cisco Systems, Inc.
00:00:c0:6b:e9:c6       WESTERN DIGITAL CORPORATION
00:03:e3:d9:26:c0       Cisco Systems, Inc.
```

[3]The second field in the tcpdump output represents the source MAC address, see Appendix A.

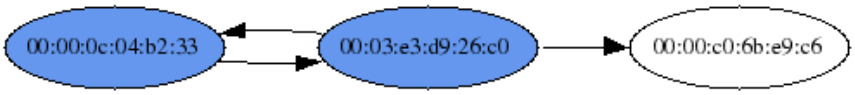


Figure 2.1: Source and destination MAC addresses showing up in the log files.

We cannot further break these assignments down to figure out what type of device these MAC addresses represent. However, we learned that we are surrounded by two network devices and most likely an end-systems. The western digital device is one that is a little strange. The company is known to manufacture harddrives[25]. However, a search on their Web page indicated that they also build firewire network interfaces. What we could be dealing with here is a storage area network (SAN). We did not find any further information about this device. None of the other GCIA practicals had a sound explanation on what this device was. We have to leave it at the speculation of it being a SAN.

2.3 Subnets

After spending a few hours issuing queries against tcpdump and trying to look at different statistics of the data, we decided to develop a parser that would take tcpdump output and put it into a MySQL[17] database. The parser can be found on the AfterGlow[14] Web page along with all the scripts to generate the graphs in this paper.

To further understand the environment we are dealing with, it would be helpful to know what subnets are behind all the three devices. Figure 2.2 shows a communication graph. All IP addresses are aggregated into A classes. This gives us a first and rough understanding of the address spaces and the topology[4].

Looking at Figure 2.2[5], we see a few interesting things:

- 00:00:c0:6b:e9:c6 only shows packets that enter its network (arrows only point away from the device). This supports our finding that this machine is an end system. The mystery is that it has three different IP addresses.

- Some address spaces only show up as sources (red nodes), some as destinations (white nodes) and most of them as both (blue nodes). This seems interesting in the sense that we are looking at snort logs and would not expect to see rules triggering for incoming and outgoing traffic to and from the same subnets. Normally we would expect to get mainly incoming attack events from aggressors in the Internet.

[4]The choice to summarize the IP addresses into A classes was made to keep the graph legible and is sufficient for the propose of determining the topology.

[5]As mentioned in the very beginning, Appendix B explains in detail how this graph was generated.

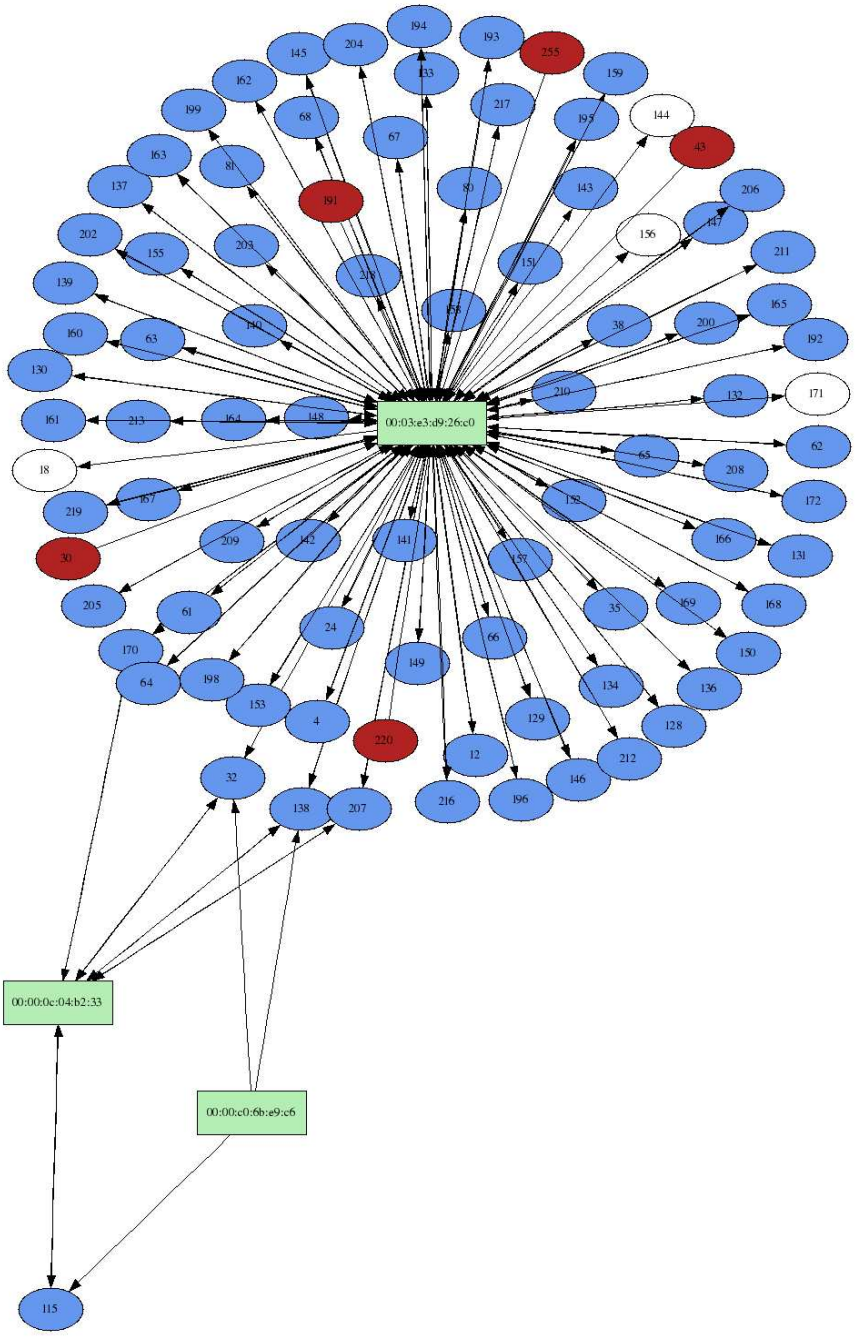


Figure 2.2: Topology showing IP subnets (circles, summarized in A classes) and their border devices (boxes). The arrows indicate the direction of traffic from the device. An arrow leaving a device (box) indicates that traffic targeted this subnet. An arrow entering a device (box) indicates traffic originating from this subnet (circle).

- Some subnets on the external network do only show up as targets (white nodes). That means snort either generated false alarms or internal machines are attacking the outside. In Section 3, we will further analyze this.

2.4 Topology

Putting all prior analysis together, we end up with a topology as it is shown in Figure 2.3.

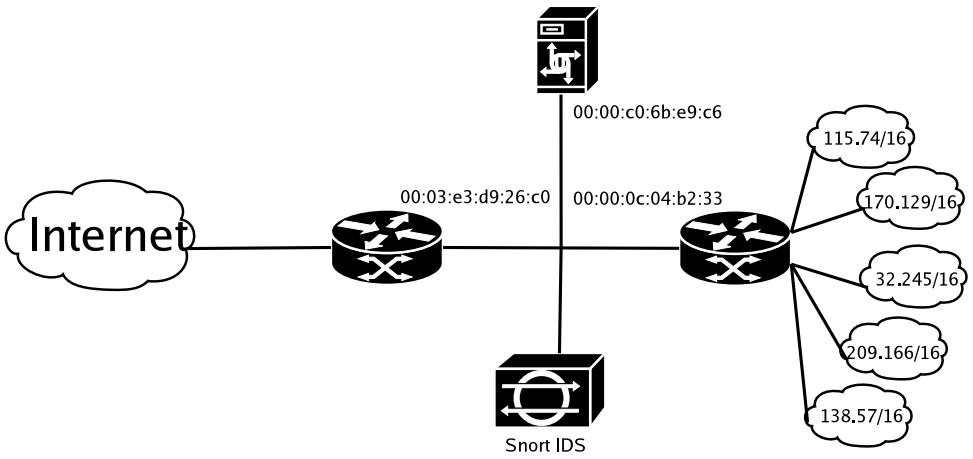


Figure 2.3: Topology showing network devices, the snort IDS and the internal subnets.

2.5 Communications

Now that we have a rough idea of how the topology looks, we can try to further break down the communications. Figure 2.4 shows the outbound connections. We see that only ten machines are showing up. These are the number of destination that these machines are connecting to[6]:

IP	Count	IP	Count
207.166.87.157	6760	207.166.87.40	31
32.245.166.236	2972	32.245.166.119	31
138.97.18.88	2539	138.97.18.225	12
115.74.249.65	358	115.74.249.202	6
170.129.50.120	332	170.129.50.3	2

Figure 2.5 shows the inbound connections. After generating the first version of the graph, it became apparent that there are too many IP addresses to be displayed:

[6] `select sourceip, count(distinct(destip)) from sans where sourcemac="00:00:0c:04:b2:33" group by sourceip`

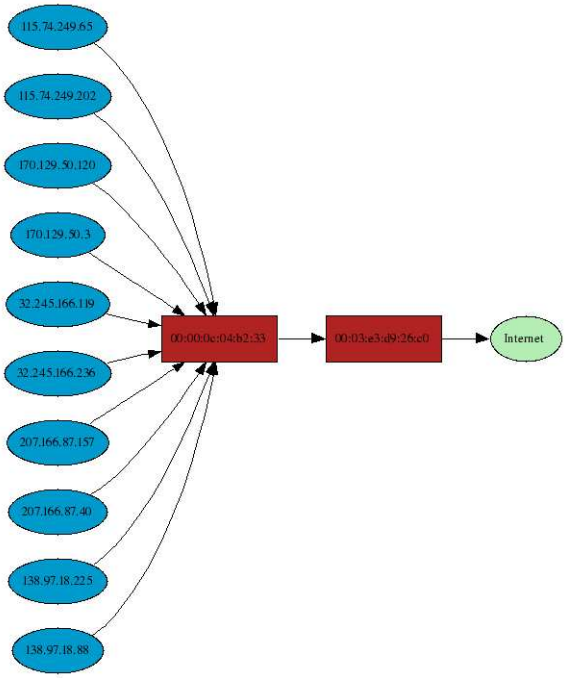


Figure 2.4: Outbound connections.

- Number of destination IPs:[7] 83634
- Number of destination IPs aggregated in C classes:[8] 1208
- Number of destination IPs aggregated in B classes:[9] 5

These numbers also suggest that there are five class B networks protected by this device.

2.6 Top Talkers

Figure 2.6 shows the top talkers on each of the networks. To see whether there are machines on the network that need special attention, either because they show up as targets of many attack attempts or they seem to be unusual aggressors.

The ten machines in Figure 2.6 are machines on the internal network. There are potentially compromised systems, as already mentioned in Section 2.3. We will have a closer look at this in Section 3.3.

[7] `select count(distinct(destip)) from sans where sourcemac="00:03:e3:d9:26:c0"`

[8] `select count(distinct(substring_index(destip,".",3))) from sans where sourcemac="00:03:e3:d9:26:c0"`

[9] `select count(distinct(substring_index(destip,".",2))) from sans where sourcemac="00:03:e3:d9:26:c0"`

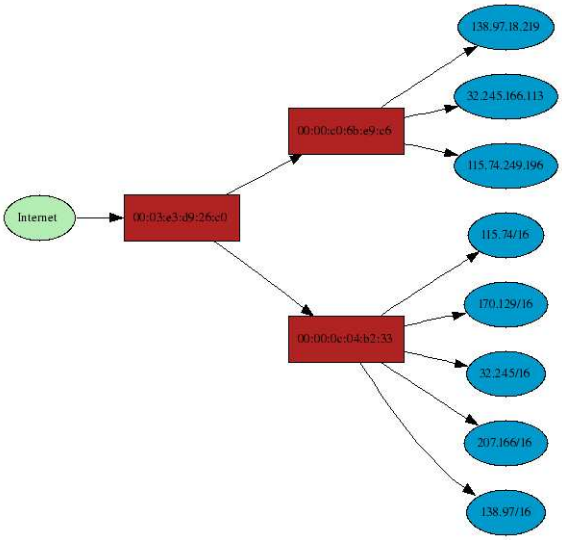


Figure 2.5: Inbound connections.

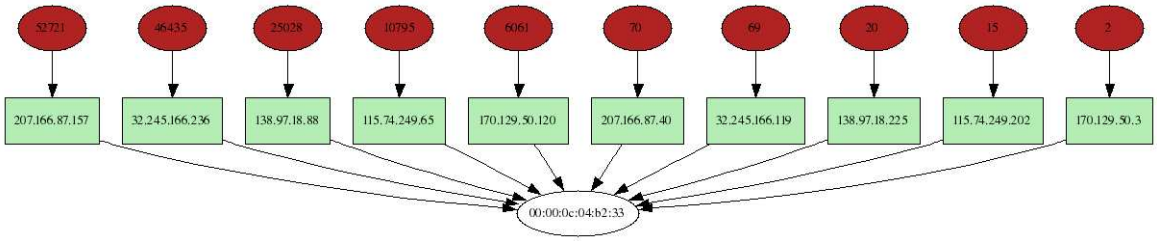


Figure 2.6: Top 10 *sources* (green nodes) originating behind source MAC address 00:00:0c:04:b2:33, i.e., events triggered by internal machines. The red nodes show the number of times the source shows up.

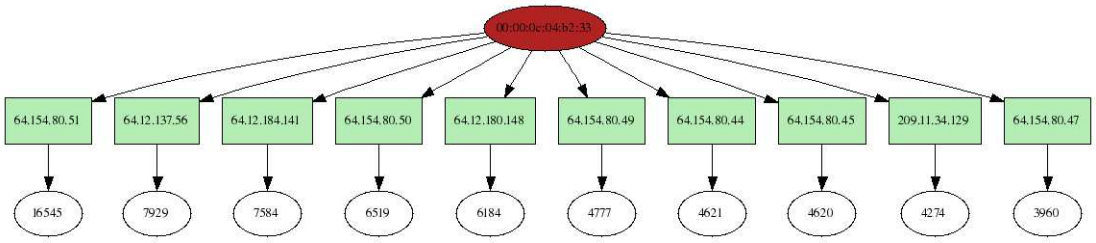


Figure 2.7: Top 10 *targets* (green nodes) in the external network targeted by machines situated behind MAC address 00:00:0c:04:b2:33. These are all the internally originating events targeting systems in the Internet. The red nodes show the number of times the target occurs.

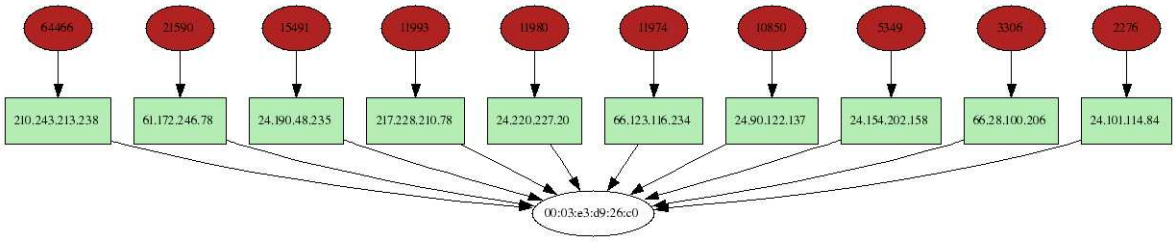


Figure 2.8: Top 10 *sources* originating behind source MAC address 00:03:e3:d9:26:c0. These are the top 10 systems on the external network attacking internal machines. The red nodes show the number of times the source shows up.

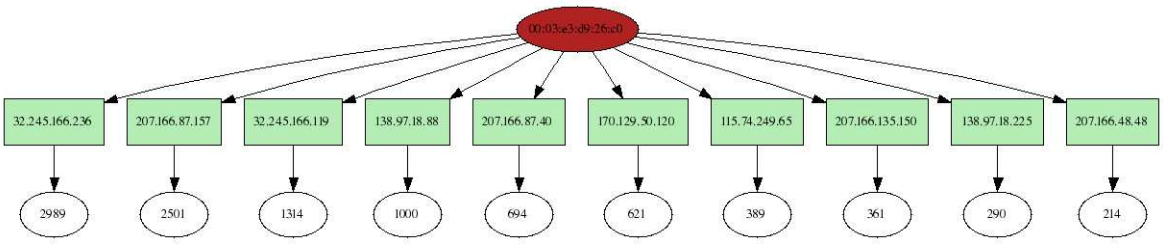


Figure 2.9: Top 10 *targets* (green nodes) originating from source MAC address 00:03:e3:d9:26:c0. This shows the top 10 machines in the internal network, targeted by external machine. The red nodes show the number of times the source shows up.

2.7 Gateway's Role

We still do not know much about the topology we are dealing with. The fact that only ten machines show up as sources (Figure 2.4) lets us speculate that the device with the MAC address `00:00:0c:04:b2:33` is a network address translation (NAT) gateway. To support this claim, we ran a query that shows the IP time to live (TTL) values per source IP addresses. We would expect exactly one TTL per IP address, except for a network address translation (NAT) gateway which guards machines with different operating systems. Why different operating systems? Assuming that the NAT gateway protects a non-routed network, the resulting TTLs for packets leaving the network would be the same for all the packets, unless machines used different initial TTL values. As [24] shows, different operating systems utilize different initial TTL numbers. Therefore our claim that a NAT gateway shows a variety of TTLs if it protects different operating systems. Let us verify whether this is the case in our data set. Figure 2.10 shows the TTLs per source IP.

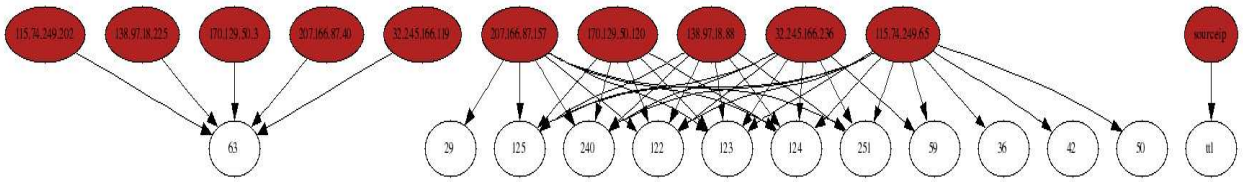


Figure 2.10: TTLs per source IP for traffic originating behind 00:00:0c:04:b2:33.

The result supports our speculation that we are dealing with a NAT device. As a next step we tried to determine the operating systems associated with the IP addresses. Using [24], we mapped these addresses to operating systems, which turned out to be impossible. The initial TTL vales for operating systems are commonly 30, 32, 60, 64, 128 or 255. Assuming the end systems are not too far away from the gateway, most of the values showing up should be in close proximity to these numbers. What we see in the logs is very different though. There are values like 36, which has to be associated with either 60 or 64. This would result in a hop count of (60-36=24). This does not seems right. Someone must be playing with these values.

The fact that we have TTLs that are in a contiguous range, suggests that there is a routed network behind the gateway. However, the chain of 122, 123, 124 and 125 would indicate that there is a machine in each of the subnets. Something seems to be wrong. We will later investigate this along with the strange TTLs discussed in the last paragraph (see Section 3.6 in [15]).

Another fact that remains unclear is why there are ten different IP addresses showing up as sources from device 00:00:0c:04:b2:33. That would indicate that we have a firewall with ten translated IP addresses. However, the IP addresses are in very different networks. This leads to the speculation that the firewall has ten interfaces. Another explanation

would be that the log files were tempered with to present this very picture.

2.8 Services

The next step in our analysis is to figure out what services the machines behind the NAT gateway are offering. We are first interested in how many snort signatures require the destination machine to accept connections in order to fire (i.e., meaning that the service exists on the target system). The `flow` keyword in snort signatures makes sure that the signature only triggers after a connection to the destination service has been established. In addition, snort signatures have a `content` keyword, which looks at packet contents. This can, to a certain extent, make sure that snort does not trigger on bogus traffic where no destination service is present. Assuming that TCP SYN packets do not carry any content (this is not always true), we can argue that only after a successful handshake between the attacker and the target service, content can be transmitted over the wire. This would require the target service to be present. Here are some numbers we collected:

- Total number of rules: 2510[10].
- Number of rules checking the flow: 2045[11].
- Number of rules not checking the flow: 425[12].
- Number of rules not checking the flow and not checking for content: 132 [13].

The fact that 425 signatures are triggering without making sure that a flow is established, is somewhat disappointing. The 132 signatures which do not check for a flow nor do they check the content, are even worse. Firstly these signatures are very prone to false positives. Even if the destination machine does not offer the service these signatures are still going to trigger. Secondly, when we are going determine what services the destination machines offer, we cannot take these signatures into account. We will continue on the idea of mapping out destination services for machines a little later.

To quickly see whether it makes sense to map the destination services to machines, despite the findings from above, we ran a couple of queries to see what target ports are accessed. The results in Table 2.1 are again very interesting and nicely support our findings from before. The top signatures triggering are those with very simple definitions and only look at the destination port along with some other properties[14]. In some cases source and destination are inverted. Normally a TCP connection is initiated from a source

[10] `grep alert *.rules | wc -l`
[11] `grep ‘‘flow:’’ *.rules | grep -v ‘‘stateless’’ | wc -l`
[12] `grep -v ‘‘flow:’’ *.rules | grep alert | wc -l`
[13] `grep -v ‘‘flow’’ *.rules | grep alert | grep -v content | wc -l`
[14] See the following signatures: `SCAN Squid Proxy attempt`, `SCAN Proxy Port 8080 attempt`, `SCAN SOCKS Proxy attempt`, ...

port that is bigger than 1024. In the case of Linux, it's even above 32768[15] and services offered by a machine are typically below 1024. The ports in Table 2.1, however, show some quite anomalous ports. Destination ports above 60000 are very uncommon.

Query	Count	Port
Number of distinct destination ports on the internal network	2556	
Top destination ports	64493	21
	45931	3128
	45001	8080
	11901	1080
	4402	80
	1917	515
	840	0
	424	9511
	352	53
	282	62513
	213	2673
	203	139
	131	137
	84	6346
	55	61310
	52	63414
	52	62830
	39	61939

Table 2.1: Destination port statistics.

Figure 2.11 shows a few source, destination port combinations after filtering out the most common destination ports. Again, very uncommon port pairs show up. For example, there are many source ports of 80. This probably indicates that there are snort signatures which trigger on Web responses as opposed to Web requests.

Taking the data from Figure 2.11 and adding the count per connection yields Figure 2.12. Some interesting port pairs suddenly show up. After filtering out the most frequent connections, connections from port 6666 to port 62513 are surfacing. There are also connections from port 0 to port 0, as well as connections from port 53 to port 53. Some of the connections from port 0 to port 0 are there because we used a database and the ports show up as zero, if they are not set. These are most likely ICMP packets or IP fragments. We also have 53 packets that are from an FTP data connection (source port of 20). A lot of the connections in Figure 2.12 expose very strange port pairs, which do not make much sense.

[15] `cat /proc/sys/net/ipv4/ip_local_port_range`

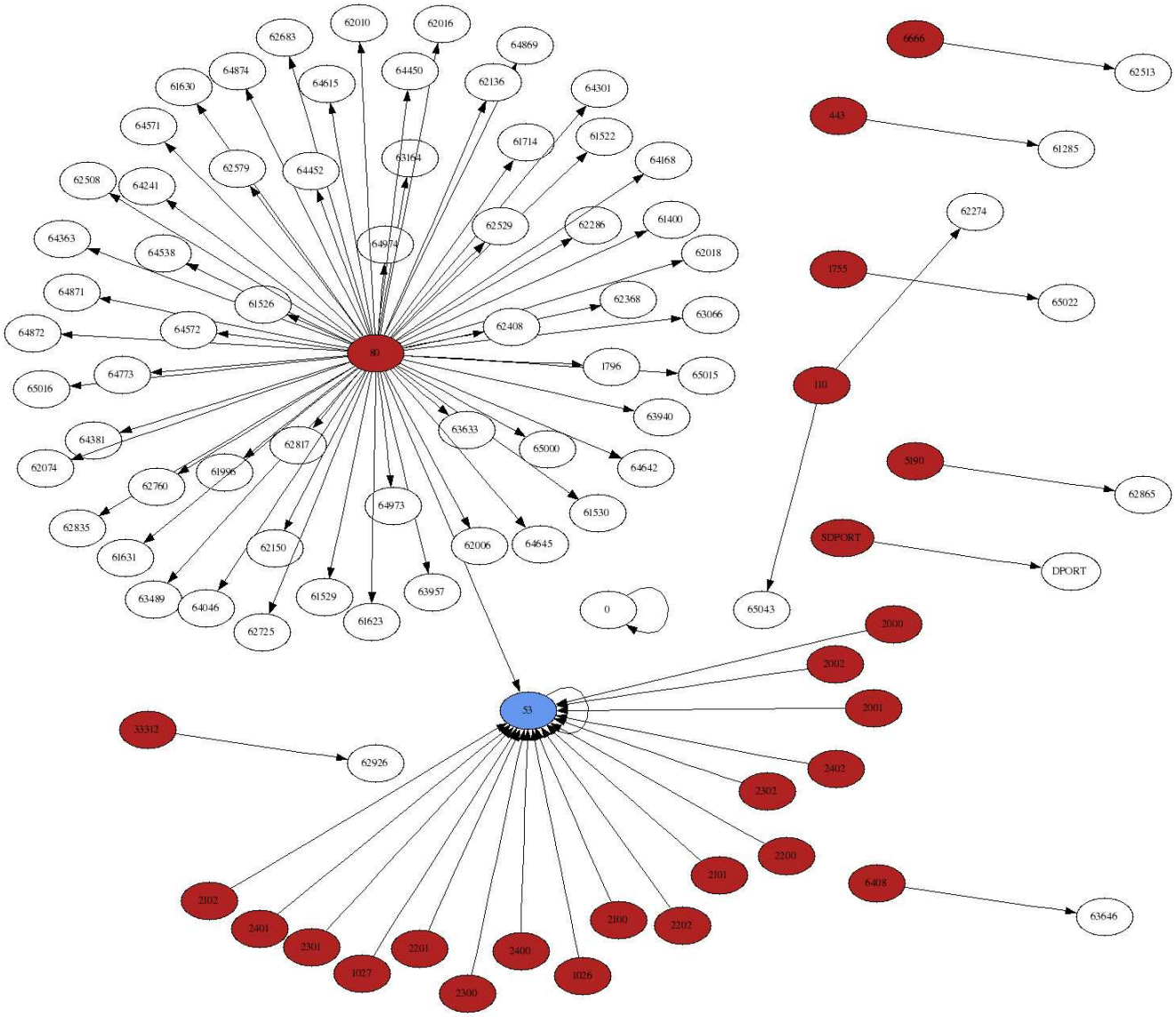


Figure 2.11: Source and destination port pairs after filtering the most common destination ports.

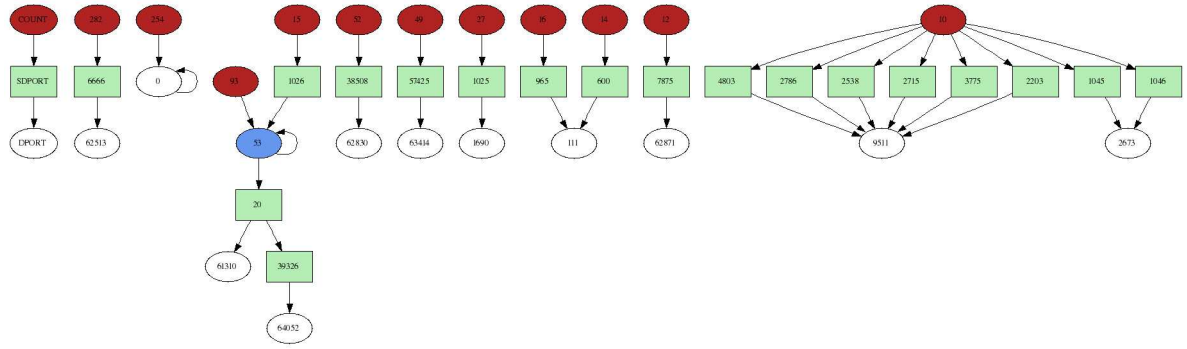


Figure 2.12: Connections and their number of occurrence where the most frequent target ports and source ports of 80 are filtered out.

More interesting packets resulted when we generated a graph of all the communications from a port below 1024 to a port below 1024 (see Figure 2.13. This is not something that normally happens.

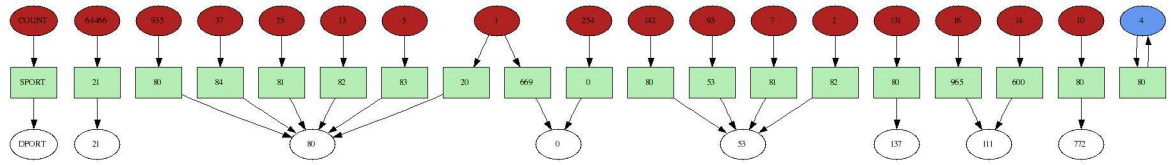


Figure 2.13: Connections and their number of occurrence where source and destionation ports are below 1024.

We now have some understanding of what is going on, but still have no idea what services are running on the machines. To address this problem, we first generated a snort alert file:

```
snort -c /etc/snort/snort.conf.complete -UDek none -r /tmp/sans -l snortlog
```

Note that the configuration file we passed to Snort is one that `includes` all the `.rules` files. By default this is not the case. We removed all the `flow:` parameters from the rules. Otherwise only a fraction of all the log entries will generate a snort alert, because snort is expecting a flow to be established. This will never be the case, because the logs only contain the packets immediately triggering the rules. The `-U` flag in the snort command changes the time format to use UTC. This is used to match the timestamps with the alerts in the database. The database with the packet information was then updated with the additional snort alerts.[16]. In a next step, we flagged all the entries in the database

[16]Check snortalerts.pl at [14] for the exact script to annotate the snort log entries with their corresponding alerts.

that were generated by rules requiring an established flow[17]. All this helpts to make a statement about what services the machines really offer. Figure 2.14 is a first pass on showing the services available on target machines[18]. As you can see, there are some strange services showing up:

- All events associated with `X11 outbound client connection detected` have the ports inverted because the snort alert triggers on the response packets.
- For the following alerts, the source and targets are inverted:

  `WEB-CLIENT readme.eml autoload attempt`

  `P2P Napster Client data`

  `Virus - Possible MyRomeo Worm`

  `CHAT AIM receive message`

  `Virus - SnowWhite Trojan Incoming`

  `CHAT IRC message`

- `P2P GNUTella client request` packets are associated with varying high ports. These ports cannot be associated with services.

All the information combined results in Figure 2.15, which shows a corrected version of the graph, with the real services exposed by the machines.

[17] Check snortservice.pl at [14] to see how exactly the database was updated.

[18] The query for the graph was: `select destip,destport,replace(snort_alert," ","_") from sans where service=1 and destmac="00:00:0c:04:b2:33` combined with `select destip,sourceport, replace(snort_alert," ","_") from sans where service=2 and destmac="00:00:0c:04:b2:33`.

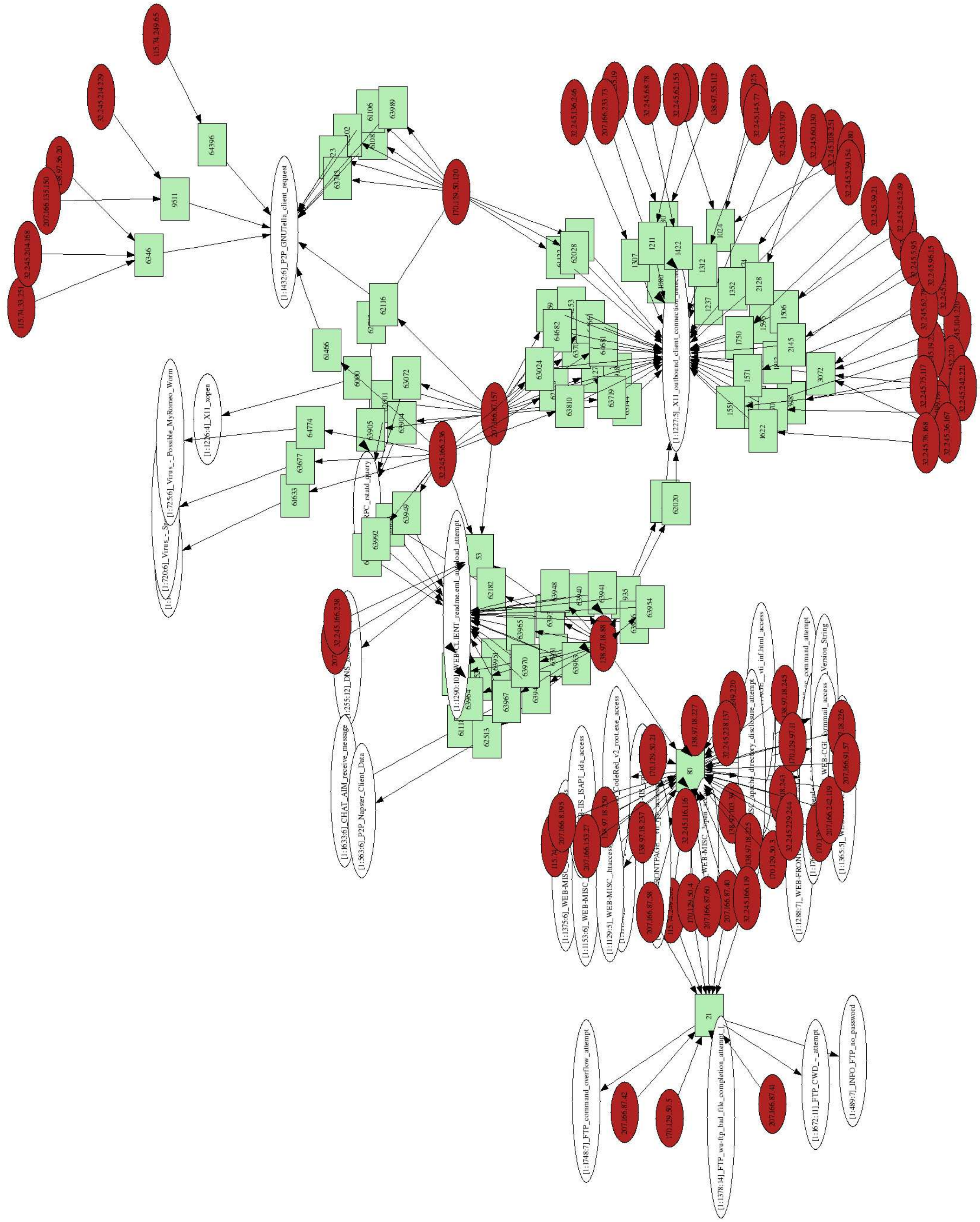


Figure 2.14: A first attempt to figure out what services are running on the target machines.

During the process of identifying the role of machines, we found two of them which need some further investigation. Both machines are running some kind of a server and X11 outbound connections were identified to the machines. It seems that the security policy should prohibit such behavior. We will investigate those two machines (`32.245.166.236` and `138.97.18.88`) in Section 3.4.

To determine the operating systems running on the machines, we ran the dataset through p0f[26]. The results were both interesting as well as disappointing. p0f heavily relies on analyzing TCP SYN packets. In our logs, however, there are not very many of those and the analysis fails for all the hosts of interest. We already tried to identify the operating systems via the TTLs in the network traffic, which also failed. This pretty much leaves us in the dark with regards to the OSes of the machines.

All this information taken together, we were able to build Table 2.2 that shows the machines in the protected network and their probable roles.

2.9 Missing Snort Alerts

There is still one issue we have not resolved. Why are there about 324461 log entries, but for about 70,000 of them we were not able to generate a snort alert. Out of these 70,000 events, more than 69.000 are related to Web traffic (targeting or originating from port 80).

There are multiple possible reasons for this:

- Our snort rules were different, either because the original snort was tuned, custom signatures were used or the signatures got updated over time.
- We did not have the same settings for the preprocessors or originally, other preprocessors were used.
- We set the `HOME_NET` or other variables differently.

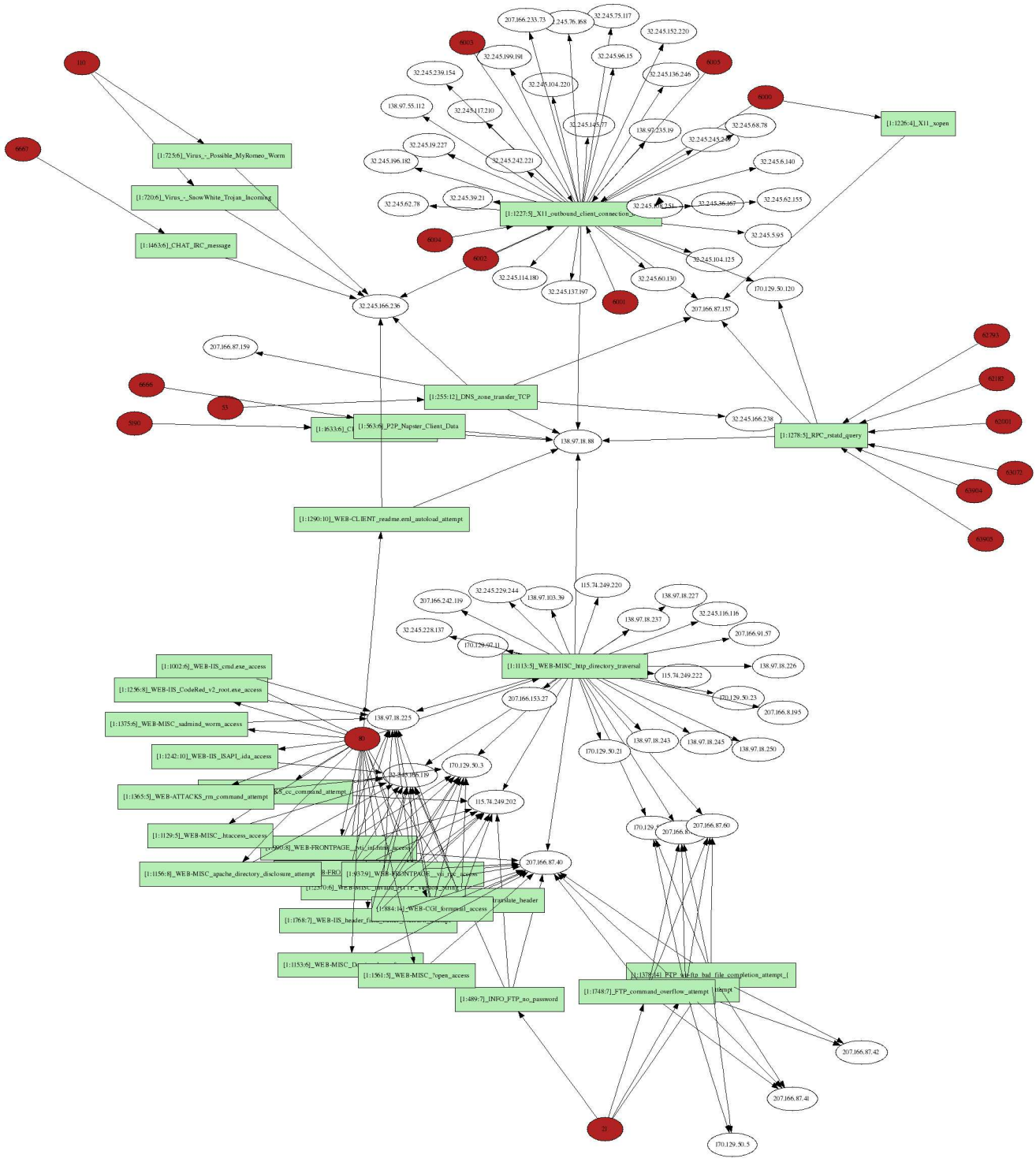


Figure 2.15: The definite list of services running on the target machines showing also the snort rules that lead to the detection.

Role	**Machine**
Web, DNS, X Server and more	138.97.18.88
Web, DNS, POP and more	32.245.166.236
DNS Server	207.166.87.157
DNS Server	207.166.87.159
DNS Server	32.245.166.238
FTP Server	170.129.50.5
FTP Server	207.166.87.40
FTP Server	207.166.87.41
FTP Server	207.166.87.42
Web and FTP Server	115.74.249.202
Web and FTP Server	170.129.50.4
Web and FTP Server	207.166.87.58
Web and FTP Server	207.166.87.60
Web and FTP Server	32.245.166.119
Web Server	115.74.249.220
Web Server	115.74.249.222
Web Server	138.97.103.39
Web Server	138.97.18.225
Web Server	138.97.18.226
Web Server	138.97.18.227
Web Server	138.97.18.237
Web Server	138.97.18.243
Web Server	138.97.18.245
Web Server	138.97.18.250
Web Server	170.129.50.21
Web Server	170.129.50.23
Web Server	170.129.50.3
Web Server	170.129.97.11
Web Server	207.166.153.27
Web Server	207.166.242.119
Web Server	207.166.8.195
Web Server	207.166.87.40
Web Server	207.166.91.57
Web Server	32.245.116.116
Web Server	32.245.228.137
Web Server	32.245.229.244
RPC Server	207.166.87.157
X Server	a few

Table 2.2: Machines and their roles.

Chapter 3

Investigations

In this Chapter we will first have a glance at all snort alerts we were able to generate with the log files analyzed. This will help us understand the trends in the data set. In the following sections we will then have a closer look at some of the issues we identified in the first Chapter. In addition to the discussion of those findings we will come up with different methods to analyze datasets. We will see that there are a number of machines utilizing automated ways of attacking machines. We also found a lot of false positives in the alerts triggered by snort. A rough guess is probably around 50%. This is very disappointing and shows that something fundamental is wrong in the world of network-based intrusion detection systems. We will suggest a possible fix at the end of the next Section.

3.1 Snort Alert Investigations

Table 3.1 shows a complete list of the snort alerts triggered by the data analyzed. These alerts were generated using the method discussed in Section 2.8. The alerts in bold are the ones we will have a closer look at. The selection of events was done based on three criteria:

1. The severity of the snort alert triggered (e.g., a buffer overflow event is more severe than the detection of IRC traffic).

2. The level of precision of the snort rule (e.g., some snort rules only look for certain port numbers in the traffic, which is very prone to false positives).

3. Some analysis of the packets triggering the snort rules (e.g., most of the rules looking for Web traffic related attacks, are false positives; they are very loosely written and merely check for the presence of a certain string in the HTTP protocol).

A lot of GIAC students have already analyzed these datasets and have published their findings[10]. We do not want to repeat their findings here. We will however provide a short list of the most important activities found in the log data. We will quickly describe

Count	Alert	Count	Alert
64466	SCAN synscan portscan	64466	SCAN SYN FIN
45820	SCAN Squid Proxy attempt	44872	SCAN Proxy Port 8080 attempt
29008	WEB-IIS header field buffer overflow attempt	21102	P2P GNUTella client request
21082	P2P Outbound GNUTella client request	20714	P2P Inbound GNUTella client request
11790	SCAN SOCKS Proxy attempt	9188	(http_inspect) BARE BYTE UNICODE ENCODING
3690	(http_inspect) OVERSIZE REQUEST-URI DIRECTORY	1813	(http_inspect) IIS UNICODE CODEPOINT ENCODING
1801	**BACKDOOR Q access**	1485	SCAN nmap TCP
1470	P2P Napster Client Data	1137	CHAT IRC nick change
835	CHAT MSN message	775	BAD-TRAFFIC tcp port 0 traffic
690	WEB-IIS view source via translate header	528	(http_inspect) APACHE WHITESPACE (TAB)
446	WEB-FRONTPAGE /_vti_bin/ access	402	WEB-MISC http directory traversal
385	SHELLCODE x86 NOOP	379	WEB-CGI formmail access
366	(http_inspect) NON-RFC HTTP DELIMITER	351	(http_inspect) DOUBLE DECODING ATTACK
330	WEB-FRONTPAGE _vti_rpc access	325	WEB-FRONTPAGE _vti_inf.html access
324	WEB-FRONTPAGE shtml.exe access	257	WEB-IIS %2E-asp access
246	WEB-IIS cmd.exe access	237	WEB-CGI redirect access
180	(http_inspect) WEBROOT DIRECTORY TRAVERSAL	169	ATTACK-RESPONSES 403 Forbidden
111	BAD-TRAFFIC same SRC/DST	93	X11 outbound client connection detected
93	WEB-MISC Invalid HTTP Version String	86	WEB-CGI formmail arbitrary command execution attempt
77	SHELLCODE x86 inc ebx NOOP	75	DNS zone transfer TCP
65	WEB-MISC weblogic/tomcat .jsp view source attempt	52	(snort_decoder) WARNING: TCP Data Offset is less than 5!
43	BAD-TRAFFIC ip reserved bit set	41	WEB-IIS ISAPI .ida access
40	WEB-IIS ISAPI .ida attempt	33	BAD-TRAFFIC data in TCP SYN packet
32	MISC Tiny Fragments	24	BAD TRAFFIC Non-Standard IP protocol
22	**SHELLCODE x86 setuid 0**	21	WEB-CLIENT readme.eml autoload attempt
14	WEB-MISC search.dll access	13	WEB-CGI calendar access
12	**SHELLCODE x86 setgid 0**	12	SCAN FIN
10	WEB-ATTACKS id command attempt	10	INFO Outbound GNUTella client request
10	FTP CWD attempt	8	WEB-CGI search.cgi access
8	WEB-ATTACKS cc command attempt	8	(snort_decoder): Short UDP packet, length field > payload length
8	RPC portmap mountd request UDP	7	WEB-CGI campus access
6	WEB-MISC /doc/ access	6	WEB-FRONTPAGE shtml.dll access
6	RPC portmap pcnfsd request UDP	6	INFO FTP no password
5	X11 xopen	5	**SHELLCODE x86 0xEB0C NOOP**
5	RPC rstatd query	5	FTP wu-ftp bad file completion attempt
5	**FTP CWD overflow attempt**	5	FTP command overflow attempt
5	**BACKDOOR NetMetro File List**	4	WEB-MISC ?open access
4	MISC source port 53 to ¡1024	3	WEB-MISC Domino log.nsf access
3	WEB-CGI eXtropia webstore access	2	WEB-MISC .htaccess access
2	WEB-MISC /home/www access	2	WEB-IIS encoding access
2	WEB-CGI phf access	2	WEB-ATTACKS rm command attempt
2	**Virus - Possible MyRomeo Worm**	2	MISC xdmcp query
2	(http_inspect) OVERSIZE CHUNK ENCODING	2	CHAT IRC message
1	WEB-MISC sadmind worm access	1	WEB-MISC intranet access
1	WEB-MISC ICQ Webfront HTTP DOS	1	WEB-MISC handler access
1	WEB-MISC cross site scripting attempt	1	WEB-MISC apache directory disclosure attempt
1	WEB-IIS CodeRed v2 root.exe access	1	WEB-IIS asp-dot attempt
1	WEB-CGI zsh access	1	Virus - SnowWhite Trojan Incoming
1	SHELLCODE x86 stealth NOOP	1	SHELLCODE x86 0x90 unicode NOOP
1	MISC Source Port 20 to ¡1024	1	(http_inspect) U ENCODING
1	CHAT AIM receive message	1	ATTACK-RESPONSES id check returned userid
1	**ATTACK-RESPONSES id check returned root**		

Table 3.1: All the snort alerts triggered by the data analyzed. The alerts in bold are the ones that will be further investigated.

the findings, do a severity analysis and give a reference to more information (potentially to other GCIA practicals that have analyzed this type of traffic). In some cases we will do a quick analysis and show that snort generated false positives.

There is worm traffic (e.g., Nimda and Code Red) showing up in the log files. We do not discuss this here, but direct the interested reader to [13]. For a short description of how the severities can be computed see Appendix C.

- **BACKDOOR Q access**

  Description: Pete Storm[21] did a great job of analyzing this backdoor traffic. However, there is one point where we disagree with him. On page 15 of his paper, he talks about the snort signature for the backdoor, which checks for packets coming from the 255.255.255.0/24 subnet. He tries to explain why packets *coming* from this subnet show up in the log files. We believe that these addresses are not sources, but destinations! The snort rule looking at the backdoor traffic does not look for incoming connections, but for the responses:

  ```
  alert tcp 255.255.255.0/24 any -> $HOME_NET any (msg:"BACKDOOR Q access";
  dsize:>1; flags:A+; flow:stateless; reference:arachnids,203;
  classtype:misc-activity; sid:184; rev:6;)
  ```

  Our hypothesis is that the backdoor sends traffic to one of the broadcast addresses to see whether any machine answers. If an answer comes back from one of the machines, the packet will be a response coming from the 255.255.255.0/24 subnet.

  Severity: 3

  Criticality: 2, we do not have any further information about this asset. This machine did not seem to be involved in any other important traffic either.

  Lethality: 5, if the "Q" trojan is on a system, anything could have happened to it and potentially other systems.

  System Countermeasures: 2, we do not have information about the protection mechanisms on the host and therefore assume a two.

  Network Countermeasures: 2, there seem to be no firewalls or ACLs on the routers; however, an IDS is deployed.

  Reference: Pete Storm, GCIA Practical[21]

- **SHELLCODE x86**

  Description: Possible shellcode discovered on the wire. These snort rules are very prone to false positives. They only check for sequences of certain bytes which could show up in benign traffic such as emails, ftp transfers, ... Here are the conditions the rules trigger on:

  ```
  SHELLCODE x86 setgid 0                content:"|B0 B5 CD 80|"
  SHELLCODE x86 setuid 0                content:"|B0 17 CD 80|"
  SHELLCODE x86 NOOP                    content:"|90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90|"; depth:128
  SHELLCODE x86 stealth NOOP            content:"|EB 02 EB 02 EB 02|"
  SHELLCODE x86 0x90 unicode NOOP       content:"|90 00 90 00 90 00 90 00 90 00|"
  SHELLCODE x86 inc ebx NOOP"           content:"CCCCCCCCCCCCCCCCCCCCCCCC"
  SHELLCODE x86 NOOP"                   content:"aaaaaaaaaaaaaaaaaaaa"
  SHELLCODE x86 0xEB0C NOOP             content:"|EB 0C EB 0C EB 0C EB 0C EB 0C EB 0C EB 0C EB 0C|"
  SHELLCODE x86 0x71FB7BAB NOOP         content:"q|FB|{|AB|q|FB|{|AB|q|FB|{|AB|q|FB|{|AB|"
  SHELLCODE x86 0x71FB7BAB NOOP unicode  content:"q|00 FB 00|{|00 AB 00|q|00 FB 00|{|00 AB 00|q|00 FB 00|{|00 AB 00|q|00 FB 00|{|00 AB 00|"
  ```

Some of the traffic triggering these rules is HTTPS (which is a false positive because the traffic is encrypted and the signatures are written to match the unencrypted traffic. Analyzing the traffic triggering these alerts, we found that almost all of the connections were from very high port numbers to other very high port numbers. If these were real shellcodes, we would expect the traffic to target well-known port numbers in order to exploit a certain service running on a target system. It does not make sense that this traffic would target high port numbers. We therefore conclude that the traffic represents false positives except for the ones targeting port 20, which are 72 of them. Even among this traffic, we suspect to find a high number of false positives. This is all binary traffic and is very likely to contain some of the sequences in the signatures. If the raw tcpdump logs would be available, we could make sure that prior to the FTP data connections, there was a command connection that requested a certain file or started an upload of a file. That would show that this traffic did not contain shell code. It is also possible that somebody is transferring shellcode via FTP and is not actively trying to exploit a system. This could be verified by looking at the offset of the traffic in the data connection. If the offset is a high one, we are dealing with someone potentially transferring shellcode in an FTP connection. If the offset is very small, it is a potential buffer overflow attack.

It would be worthwile figuring out what the rest of this traffic represents (the one from high port numbers to other high port numbers). However, with the data given, this was impossible. A hypothesis is that the traffic is RPC traffic and the shellcode tries to exploit an RCP service. Again, this would have to be verified with other sources of information.

Severity: 3

Criticality: 3, we do not have any further information about these assets. We do not know whether the attacks were successful.

Lethality: 4, if this is indeed shellcode an no false positives, the attack could potentially execute commands on the target systems. Although, it is not clear what the capabilities of executing code would be.

System Countermeasures: 2, we do not have information about the protection mechanisms on the host and therefore assume a two.

Network Countermeasures: 2, there seem to be no firewalls or ACLs on the routers. Firewalls do not seem to be present either. Otherwise the communications underlying this attack might have been blocked. However, an IDS is deployed.)

- **FTP CWD overflow attempt**

Description: This is an attempt to use the FTP `CWD` command to overflow a buffer. Analyzing some packets, we found that there are some real buffer overflow attempts and the rule triggered correctly. This is the snort rule:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"FTP CWD overflow attempt";
flow:to_server,established; content:"CWD"; nocase; isdataat:100,relative;
pcre:"/^CWD\s[^\n]{100}/smi"; sid:1919; rev:19;)
```

And this is the offending packet:

```
03:09:06.526507 00:03:e3:d9:26:c0 > 00:00:0c:04:b2:33, ethertype IPv4 (0x0800),
length 574: IP (tos 0x0, ttl  45, id 55450, offset 0, flags [DF], length: 560,
bad cksum 9bb8 (->516e)!) 195.232.55.6.1701 > 207.166.87.42.21: P
[bad tcp cksum 7135 (->25e2)!] 2184450005:2184450513(508) ack 1127458918 win
5840 <nop,nop,timestamp 1040178 3948516>
        0x0000:  0000 0c04 b233 0003 e3d9 26c0 0800 4500  .....3....&...E.
        0x0010:  0230 d89a 4000 2d06 9bb8 c3e8 3706 cfa6  .0..@.-.....7...
        0x0020:  572a 06a5 0015 8234 0fd5 4333 a866 8018  W*.....4..C3.f..
        0x0030:  16d0 7135 0000 0101 080a 000f df32 003c  ..q5.........2.<
        0x0040:  3fe4 4357 4420 3030 3030 3030 3030 3030  ?.CWD.0000000000
        0x0050:  3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
        0x0060:  3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
        0x0070:  3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
        0x0080:  3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
        0x0090:  3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
        0x00a0:  3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
        0x00b0:  3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
        0x00c0:  3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
        0x00d0:  3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
        0x00e0:  3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
        0x00f0:  3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
        0x0100:  3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
        0x0110:  3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
        0x0120:  3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
        0x0130:  3030 3030 3030 3030 3030 3030 3030 3030  0000000000000000
        0x0140:  3030 3030 3030 f0fc 4031 0708 985f 0808  000000..@1..._..
        0x0150:  eb0c eb0c eb0c eb0c eb0c eb0c eb0c eb0c  ................
        0x0160:  eb0c eb0c eb0c eb0c eb0c eb0c eb0c eb0c  ................
        0x0170:  eb0c eb0c eb0c eb0c eb0c eb0c eb0c eb0c  ................
        0x0180:  eb0c eb0c eb0c eb0c eb0c eb0c eb0c eb0c  ................
        0x0190:  eb0c eb0c eb0c eb0c eb0c eb0c eb0c eb0c  ................
        0x01a0:  eb0c eb0c eb0c eb0c eb0c eb0c eb0c eb0c  ................
        0x01b0:  eb0c eb0c eb0c eb0c eb0c eb0c eb0c eb0c  ................
        0x01c0:  eb0c eb0c eb0c eb0c eb0c eb0c eb0c eb0c  ................
        0x01d0:  eb0c eb0c eb0c eb0c eb0c eb0c eb0c eb0c  ................
        0x01e0:  eb0c eb0c eb0c eb0c eb0c eb0c eb0c eb0c  ................
        0x01f0:  eb0c eb0c eb0c 9090 9090 9090 9090 9090  ................
        0x0200:  9090 31db 43b8 0b74 510b 2d01 0101 0150  ..1.C..tQ.-....P
        0x0210:  89e1 6a04 5889 c2cd 80eb 0e31 dbf7 e3fe  ..j.X......1....
```

```
0x0220:  ca59 6a03 58cd 80eb 05e8 ed0a ca59 6a03  .Yj.X........Yj.
0x0230:  58cd 80eb 05e8 edff ffff ffff ff0a       uX............
```

The packet contains shell code at address 0x0202. Disassembling the hex code yields the following assembly code:

```
00000000: 31DB      xor         bx,bx
00000002: 43        inc         bx
00000003: B80B74    mov         ax,740B
00000006: 51        push        cx
00000007: 0B2D      or          bp,[di]
00000009: 0101      add         [bx+di],ax
0000000B: 0101      add         [bx+di],ax
0000000D: 50        push        ax
0000000E: 89E1      mov         cx,sp
00000010: 6A04      push (w)    +04
00000012: 58        pop         ax
00000013: 89C2      mov         dx,ax
00000015: CD80      int         80
00000017: EB0E      jmps        file:00000027
00000019: 31DB      xor         bx,bx
0000001B: F7E3      mul (w)     bx
0000001D: FECA      dec         dl
0000001F: 59        pop         cx
00000020: 6A03      push (w)    +03
00000022: 58        pop         ax
00000023: CD80      int         80
00000025: EB05      jmps        file:0000002C
00000027: E8ED0A    calln       +0AED
0000002A: CA596A    retf        6A59
0000002D: 0358CD    add         bx,[bx+si-33]
00000030: 80EB05    sub         bl,+05
00000033: E8EDFF    calln       file:00000023
00000036: FFFF      ??? (w)     di
00000038: FFFF      ??? (w)     di
0000003A: FF0A      dec         [bp+si]
```

Obviously, this shellcode is from the TESO 7350wurm.c[22] exploit. We found an in-depth analysis of this very attack on the dshield mailinglist. Stephen Hall[12] has done a very thorough analysis of this exploit for his GCIH practical. We do not repeat his elaborate findings here, but want to also mention Way Farers's post on dshield.org[7]. He mentions that the FTP exploit has two possible outcomes. Firstly a DoS attack can be launched by sending three packets of 504 bytes each. Secondly a buffer overflow can be caused, given that a prior login to the ftp service was possible; for example with an anonymous account. The packet we found was not trying to

crash the service as no three packets with the size of 504 bytes were detected. Due to the fact that we already found the shellcode embedded in one of the FTP packets, this was an attack of the latter kind, trying to gain root-access on the remote system. The source was probably not spoofed as otherwise the FTP connection would not have been established[1]. This system should definitely be taken offline and analyzed for a successful intrusion. To protect other machines, they should all be patched and updated to the latest FTP daemons. We also recommend installing a local firewall and only allow people to access the FTP server who should have access to it. This might not be possible for a public FTP server. In such a case, the FTP server should only serves the content it needs to and t should not serve any other purposes. This ensures that a compromised FTP server does not leak any other information.

Severity: 7

Criticality: 4, this is an FTP server. It is possible that the access to this system is business critical.

Lethality: 5, this is a root access attempt, therefore possibly compromising the system.

System Countermeasures: 2, we do not have information about the protection mechanisms on the host and therefore assume a two.

Network Countermeasures: 2, there seem to be no firewalls or ACLs on the routers, according to the traffic in the logs. However, an IDS is deployed.)

References: Bugtraq ID 1227[1], TESO 7350wurm[22], dshield.org posting[7].

- **BACKDOOR NetMetro File List**

  Description: This is most likely a false positive triggered by FTP traffic. The snort signature is very loosely defined:

  ```
  alert tcp $EXTERNAL_NET any -> $HOME_NET 5032 (msg:"BACKDOOR NetMetro
  File List"; flow:to_server,established; content:"--"; ...
  ```

  The traffic triggering this alert has source ports of 20 and 80, and a destination port of 5032. These are therefore valid FTP data and HTTP connections[2]. Matching the *contents*-part of the signature ("`--`") easily happens in this type of data.

  Severity: 0 (false positive)

- **Virus - Possible MyRomeo Worm**

  Description: This alert is a false positive again. The rule triggers on an email which contains the string specified in the snort signature:

[1] It is very hard to actually spoof sources and simulate a successful communication.
[2] Replies, to be correct.

```
alert tcp any 110 -> any any (msg:"Virus - Possible MyRomeo Worm";
flow:established; content:"ble bla"; nocase; classtype:misc-activity;
sid:725; rev:6;)
```

The signature looks for POP connections containing the characters "`ble bla`". The email triggering the rule contained the following sentence:

```
with the purchase of an eligible Blast motorcycle.
                              +-----+
```

Severity: 0 (false positive)

- **ATTACK-RESPONSES id check returned root**

  Description: Another false positive triggered by a Web page containing information about a SITE EXEC vulnerability. The snort rule for this:

```
alert ip any any -> any any (msg:"ATTACK-RESPONSES id check returned root";
content:"uid=0|28|root|29|"; classtype:bad-unknown; sid:498; rev:6;)
```

And the packet triggering the alert (note the `uid=0(root)` part):

```
23:59:56.156507 00:03:e3:d9:26:c0 > 00:00:0c:04:b2:33, ethertype IPv4 (0x0800),
length 1514: IP (tos 0x0, ttl  46, id 25786, offset 0, flags [DF], length: 1500,
bad cksum 898a (->9ea2)!) 65.118.58.104.80 > 32.245.166.236.64857: P
[bad tcp cksum a34e (->b866)!] 382268946:382270406(1460) ack 2266701906 win 6432

[snip]

To test for this hole, type (when logged in as a real user, not anonymous) :
<BR>
<tt>ftp&gt; SITE EXEC bash -c id</tt>
<P>
If you get a return with '200-uid=0(root) gid=0(root)' in it, you have the problem.
```

Severity: 0 (false positive)

After all this analysis, it is quite disappointing how many false positives snort generated. And this was after we already sorted out some of the events because we either knew or suspected that they are false positives. Our initial recommendation for having some kind of context along with the intrusion detection system starts to make more and more sense. It would help a great deal to reduce the number of false positives. Another suggestion would be to improve snort such that it is aware of the protocol the traffic represents. In the case of the `BACKDOOR NetMetro File List`, the rule could be rewritten to not just look for a target port of 5032, but also make sure that the traffic is not FTP, nor HTTP, nor any other known protocol, but some kind of proprietary backdoor traffic.

3.2 Scripted and Automated Activity

To identify which attacks (i.e., the corresponding snort alerts) are generated by someone executing a script or some sort of automated program, we wrote a tool[14] to calculate the time differences between alerts of the same connection. In most cases, automated tasks are highly predictable. Authors of attack scripts hardly ever build randomness into their tools. This normally reflects in the network traffic from those attack tools. Especially scanners often leave packet traces that are very monotonous. They can often be identified inside traffic trace by looking at timestamps of packets. Packets of the same connection arriving at constant intervals are probably related to some kind of automated behavior.

An attempt to detect automated activity was made by looking at the target machines, target ports, source machines and their packet inter arrival times. The source ports are neglected as they change during a connection. The time resolution we chose was one second. Anything below one second would have included too much noise and not resulted in clean deltas. For the following analysis we decided to drop the target ports as well and just looked at the connections between machines and their packet inter arrival times.

3.2.1 The Automated Behavior

It is very interesting to see that our method uncovered most of the SCAN alerts. Here is a list of snort alerts found[3]:

```
WEB-IIS view source via translate header
WEB-MISC http directory traversal
P2P GNUTella client request
WEB-IIS header field buffer overflow attempt
(http_inspect) OVERSIZE REQUEST-URI DIRECTORY
(http_inspect) DOUBLE DECODING ATTACK
(http_inspect) BARE BYTE UNICODE ENCODING
(http_inspect) IIS UNICODE CODEPOINT ENCODING
BAD-TRAFFIC tcp port 0 traffic
CHAT MSN message
P2P Napster Client Data
SCAN SOCKS Proxy attempt
SCAN Squid Proxy attempt
SCAN Proxy Port 8080 attempt
SCAN nmap TCP
SCAN synscan portscan
```

For some of these events, we expected monotony. We therefore discarded all the SCAN, CHAT and P2P events. The remaining list looks very impressive. There are a few sources (most of them even on the internal network) which are showing automated behavior.

[3] `select snort_alert, count(*) c from sans where delta2>0 group by sourceip,delta2 having c>20`

A quick look at the coverage we achieved with our method: There are a total of 168558 `SCAN` alerts in the logs. Out of those, we uncovered 50466, which seems pretty good. One might object that this does not really help much. This is right, but we only want to show that our method has a certain amount of success. It would be very interesting to test our method with raw tcpdump logs to detect automated behavior!

Four of the top sources utilizing automated techniques are on the internal network.

```
Count IP                      Count IP
   67 32.245.166.236             23 148.64.16.128
   40 207.166.87.157             13 170.129.50.120
   25 138.97.18.88               11 115.74.249.65
   ...
```

Graphs summarizing the activity are shown in Figure 3.1, which shows per target port what snort alerts were triggered, and Figure 3.2 showing the deltas and corresponding snort alerts. The packets for which we could not associate a snort alert (NULL-nodes) were eliminated in this graph. It is interesting to see that the delta times for the `SCAN Proxy Ports` are very high. These deltas do not just show up once or twice, but several hundred times. This very nicely shows that our method of finding automated behavior works!

3.2.2 First Event

After removing some of the obvious alerts, one of the remaining alerts is the `(http_inspect) BARE BYTE UNICODE ENCODING`. The snort documentation[19] describes this as follows:

> "bare byte encoding is an iis trick that uses non-ascii chars as valid values in decoding utf-8 values. this is not in the http standard, as all non-ascii values have to be encoded with a %. bare byte encoding allows the user to emulate an iis server and interpret non-standard encodings correctly."

Looking at this traffic in ASCII[4] yields only gibberish for the entire payload. Possibly, this traffic is not Web traffic (although targeting port 80), but some kind of tunneled traffic. We do not think the traffic is encrypted because encrypted HTTP traffic would use port 443, not port 80. To find the reason for the scrambled payloads, we tried to correlate this traffic with the other traffic these machines generate. This was unfortunately not very successful. Looking at the events, it turned out that all the machines were incredibly active and there was no obvious correlation with other types of traffic. We were not able to uncover what this traffic really represents. It could be tunneled traffic, a false positive of the snort preprocessor, be non-HTTP traffic. In Section 3.5[5] we will revisit this traffic and see that there are some very interesting anomalies in this traffic (changing TTLs for the same connection, etc).

[4] `tcpdump -s 0 -A -nnevr /tmp/sans src host 115.74.249.65 and dst port 80 and src port 62785`

[5] To be found at `http://raffy.ch/projects/Raffael_Marty_GCIA_Additional_Chapters.pdf`

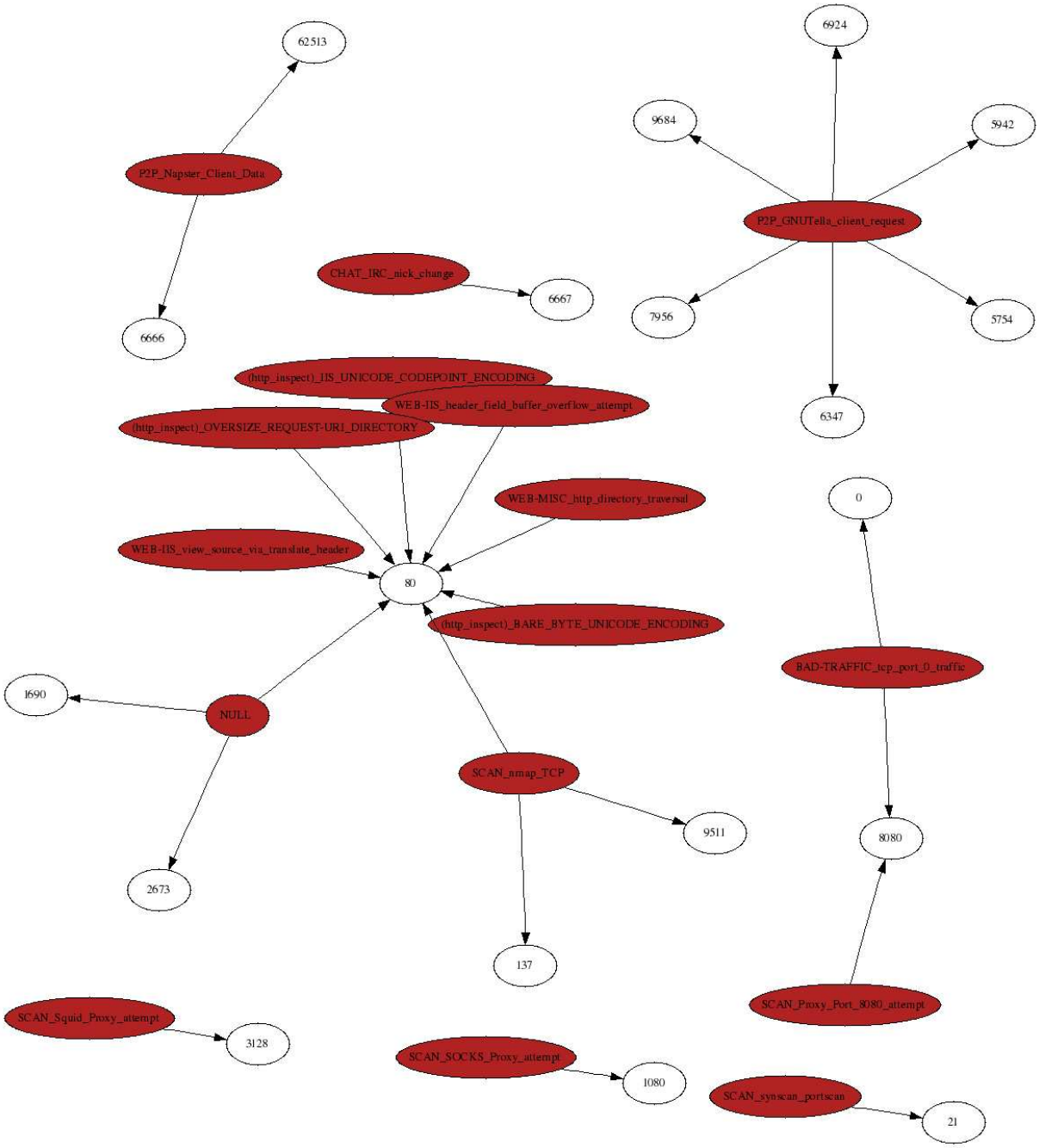


Figure 3.1: Snort alerts identified to be generated by automated behavior. Per target port the snort alerts are drawn. Only detects with an occurrence of 20 or more are shown.

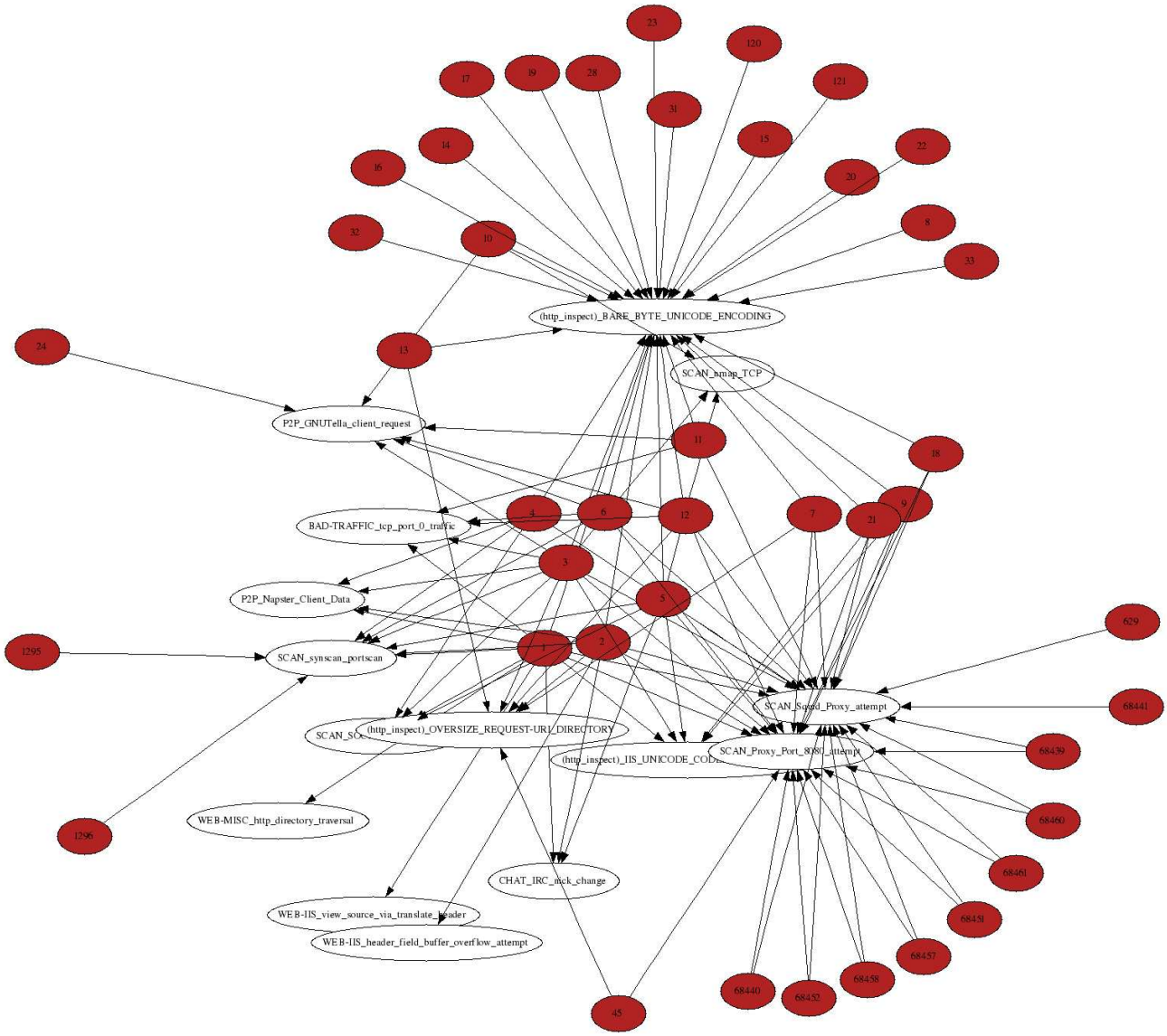


Figure 3.2: Snort alerts identified to be generated by automated behavior. Per snort alert the delta time between packets that triggered this traffic, are shown.

3.2.3 Second and Third Event

Moving on in the list of automated activity, there are two http_inspect messages: `IIS UNICODE CODEPOINT ENCODING` and `OVERSIZE REQUEST-URI DIRECTORY`. The former alert triggers on unicode encoded characters embedded in HTTP requests. The latter indicates an overly long directory name in a URL. The machines triggering these events are a subset of the ones triggering the `(http_inspect) BARE BYTE UNICODE ENCODING` event. This traffic might be related to it. It turned out that some of the offending payload was from the `Cookie:` entries in the HTTP headers. These entries seem to be huge and unicode encoded. Hence, probably a false positive! What is a little strange however, is the regularity in which these packets show up.

3.2.4 Fourth Event

The next three events we found to be automated behavior are from external machines trying to access internal IP addresses:

```
WEB-IIS view source via translate header
WEB-IIS header field buffer overflow attempt
WEB-MISC http directory traversal
```

Investigating the first event we found that it represents real HTTP traffic. The packet capture shows most of the HTTP headers (unlike in the last case where only part of the header was captured). The user-agents used in the HTTP requests are the following[6]:

```
User-Agent: Microsoft Data Access Internet Publishing Provider Protocol Discovery
User-Agent: Mozilla/2.0 (compatible; MS FrontPage 5.0)
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR 1.0.3705)
User-Agent: MSFrontPage/5.0
```

The `Host:`-field of the HTTP traffic has one interesting value: `Host: www.XXXXXXXX` We assume this is part of the obfuscation that was done by SANS and in reality this would show a valid host name. The requests sent in these packets are as follows[7]:

```
GET /../images/bullet.jpg HTTP/1.1
GET /_vti_inf.html HTTP/1.1
OPTIONS /emc/eval.html HTTP/1.1
OPTIONS / HTTP/1.1
OPTIONS /main/catalog/usb97c210.html HTTP/1.1
OPTIONS /main/catalog/usbprods.html HTTP/1.1
OPTIONS /usb/eval210.html HTTP/1.1
```

[6] `tcpdump -Annr /tmp/sans src host 68.36.170.9 and dst port 80 | grep User-Agent | sort | uniq`

[7] `tcpdump -Alnnevr /tmp/sans src host 68.36.170.9 and dst port 80 | grep -E "(OPTIONS|GET|POST)" | sed -e 's/.*\(OPTIONS\|GET\|POST\)/\1/g' | sort | uniq`

```
OPTIONS /usb/eval3000.html HTTP/1.1
OPTIONS /usb/evs3000.html HTTP/1.1
OPTIONS /usb HTTP/1.1
OPTIONS /usb/ HTTP/1.1
OPTIONS /usb/usbprods.html HTTP/1.1
POST /_vti_bin/shtml.exe/_vti_rpc HTTP/1.1
```

The targets of these events are only two: `32.245.166.119` and `207.166.87.40`

It is hard to tell whether this traffic is normal behavior or worm traffic. The regular intervals of the alerts lets us believe that we are dealing with a worm. We found a posting on the ARIS list[18] inquiring about very similar log entries. The sender also suggests that the traffic is automated. Unfortunately there are no responses available. This traffic could also stem from people using either WebDAV or Frontpage to upload Web content to a server.

3.2.5 Fifth Event

The `WEB-IIS header field buffer overflow attempt` alerts are all associated with traffic that was generated by Internet Explorer (see User-Agent field in HTTP request). However, a lot of this traffic is gibberish again. We think that this might be images or zipped content embedded in the HTTP connections! For the traffic which is legible, we found that firstly, the snort the triggering on a false positive. The traffic revealed some absolutely benign HTTP requests. The snort rule is very weak and only checks whether three specific bytes are in the HTTP traffic. Secondly, we concluded that the traffic does not indicate automated or scripted behavior. The detection as automated behavior is merely a statistical failure due to the vast amount of HTTP traffic in the logs.

3.2.6 Sixth Event

The last snort alert we identified to be automated behavior is `BAD-TRAFFIC tcp port 0 traffic`. The sources triggering the alerts are only five:

```
211.47.255.20          211.47.255.24
211.47.255.21          66.250.114.252
211.47.255.22
```

All these addresses did not trigger any other alerts. Something seems to be wrong with the network stacks of these machines; or a firewall/gateway in the network-path garbled the port numbers in some strange way. Another possibility for this type of traffic is fingerprinting[20] activity using port 0. This type of fingerprinting requires sending different packets to a machine from and to port zero. The responses from the targeted machines are then analyzed for specific behavior. Although port 0 is a valid TCP / UDP port number, it is highly recommend that one should block any traffic using this port. No program should be listening on port 0 and no program should connect from port 0. A

tool called gobbler-2.0.1-alpha[11] can be used to perform port 0 fingerprinting and might be the source for these detects.

It would be interesting to do a similar analysis we did but instead of using packet inter arrival times, calculating the difference between IP IDs of packets from the same connection. One problem with only having the packets triggered by snort alerts is that this type of analysis would not be very successful as the gaps in the IDs would not be monotonous enough. This analysis would make more sense on raw tcpdump logs.

3.3 Attack Chains

Finding attacks in a large amount of snort alerts turned out to be a real challenge. Almost all the events investigated seemed to be a side-effect of some sort of benign (or at least quite harmless) traffic. However, we have one more idea which seems to be interesting to pursue: Assume we are not interested in users in the internal network attacking other users on the inside. This assumption seems legitimate because the deployment of the snort sensor already excludes the monitoring of this type of activity. The chosen deployment of snort only reveals attacks either entering or leaving the internal networks (see also Section 2.4). To identify an attack, we use the following hypothesis: An attacker triggers a snort alert while he tries to break into a machine on the internal network. This alert does not necessarily have to be an attack. It might simply be reconnaissance activity. Recording the time of this inbound attack, we can search the rest of the snort alerts for alerts showing up at a later time *originating* from the machine that was targeted in the first alert. This method reveals internal machines that were either compromised or an attack response triggered an alert (e.g., `id check returned root`). We hope to find activity where an external machine attacks an internal one, compromises it and launches some follow-up activity from that machine. A disadvantage of this method is that using the data captures given, this method will not return a great amount of activity. The reasons for that are twofold:

1. An attacker might compromise an internal machine and not use that machine to issue any further attacks; therefore leaving no traces after the initial attack.

2. If an external attacker compromises an internal system and never launches an attack targeting an outside machine the snort logs will not reveal anything.

Despite the limitations of this approach, it seemed worthwhile generating some graphs. Figure 3.3 shows the output of a script we wrote to detect attack-chains[14]. It turns out to be quite difficult to analyze the findings:

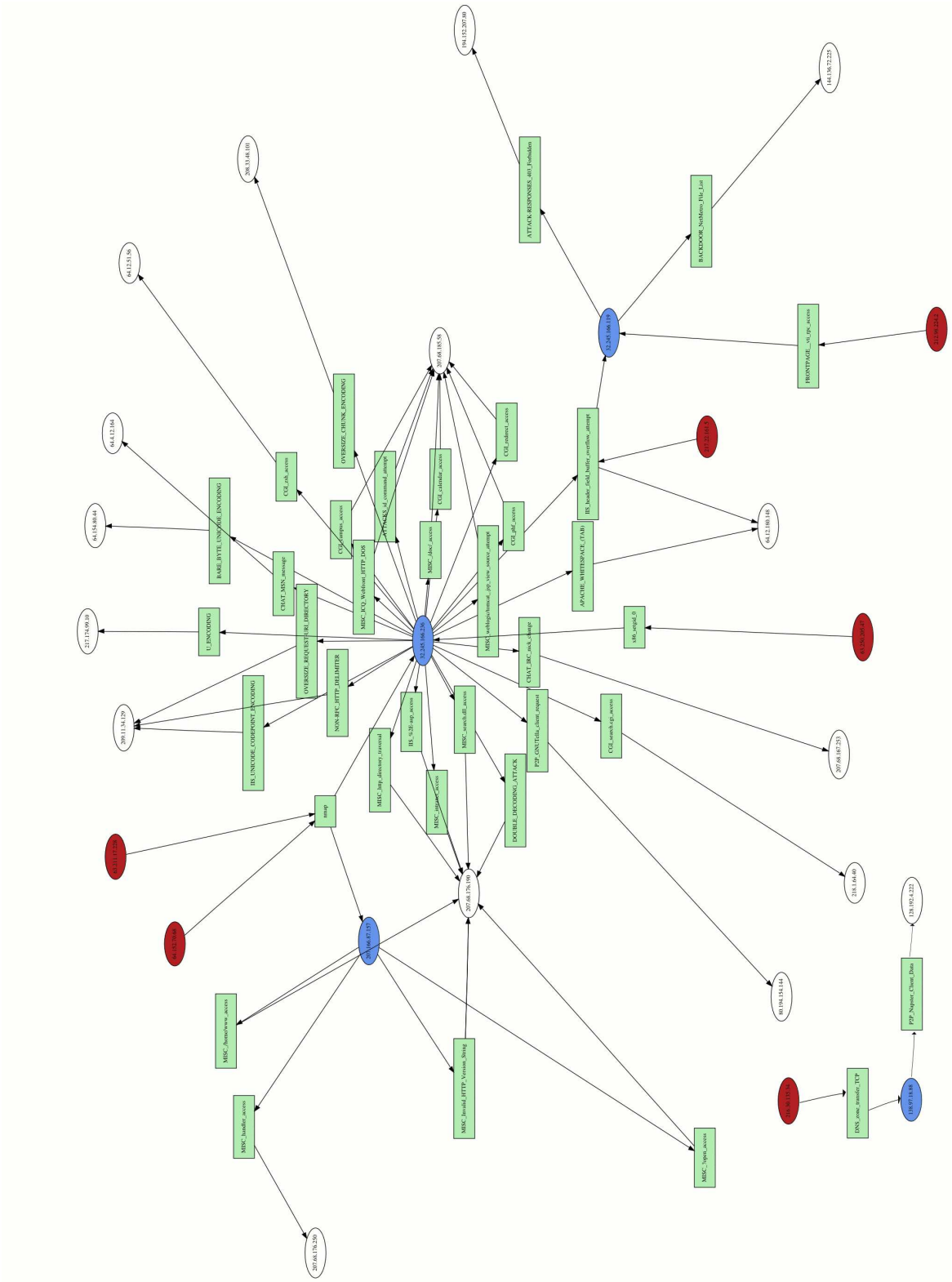


Figure 3.3: Output from attack-chain generation script showing six attackers (red) attacking four targets (blue). After the initial attacks, the targets launched attacks against other machines (white).

- A total of six machines were identified launching attacks against internal machines and then showing up as sources of other snort alerts (red nodes).
- Four machines were targets of the attacks (blue nodes).
- Only a small number of machines were involved in traffic originating from targeted machines (white nodes). We expected to see machines that were first compromised and then started some kind of scanning activity, resulting in more end-points contacted. The expected scanning activity could be anything from nessus or nmap scan to scans for specific cgi scripts on Web servers.
- One chain can be eliminated from the graph in Figure 3.3. It shows a Web request (`FRONTPAGE _vti rpc access`) followed by a `ATTACK-RESPONSES 403 Forbidden`. This seems to be normal traffic showing someone uploading a Web page via frontpage and then getting a response back indicating that his privileges did not allow for the action. (This traffic is shown on the lower right of Figure 3.3.)
- The chain of events on the lower left of the graph, which is completely disconnected from the rest, can be removed as well. It shows a `DNS zone transfer TCP` followed by a `P2P Napster Client Data` event. The zone transfers are real zone transfer attempts. However, the subsequent P2P message is not related to the zone transfer at all. Therefore we decided to remove these nodes from the investigation as well.
- Some snort alerts triggered by one "compromised" machine (32.245.166.236) are quite strange. For example the `CGI phf access`. The PHF attack is a very old one which exploited a vulnerability in the phf cgi script [2]. This cgi has been removed from Web server distributions for years already. Nessus[6] has a built-in check which checks for the presence of the phf cgi script. This could be an indicator that the machine launched some kind of vulnerability scan against other machines. However, it seems strange that only one machine was targeted with this event. Investigating this event showed that the URL triggering the alert was: `www.health.state.ny.us/nysdoh/phforum/jobs/hriload.htm`. This turns out to be a false positive. The snort rule is too loose and triggers on all the `uricontent:"/phf";`.
- The machines 207.68.185.58, 207.68.176.190, 207.68.176.250 resolve to "msnsearch.com". This explains all the other events targeting this machine. They are all false positives, triggering on either the HTTP requests or the responses from the search engine.

Figure 3.4 shows a sanitized version of the attack-chain graph, where all the nodes identified to be unimportant or even false positives, were removed.

There are three different initial attack events: `SCAN nmap TCP`, `ISS header field buffer overflow attempt` and `SHELLCODE x86 setgid 0`. All of these events represent quite aggressive attacks and seem to be good indicators for a successful compromise of an internal system. To finish an analysis of all events in Figure 3.4, we need some more information about the topology and the role of the single machines. As an immediate

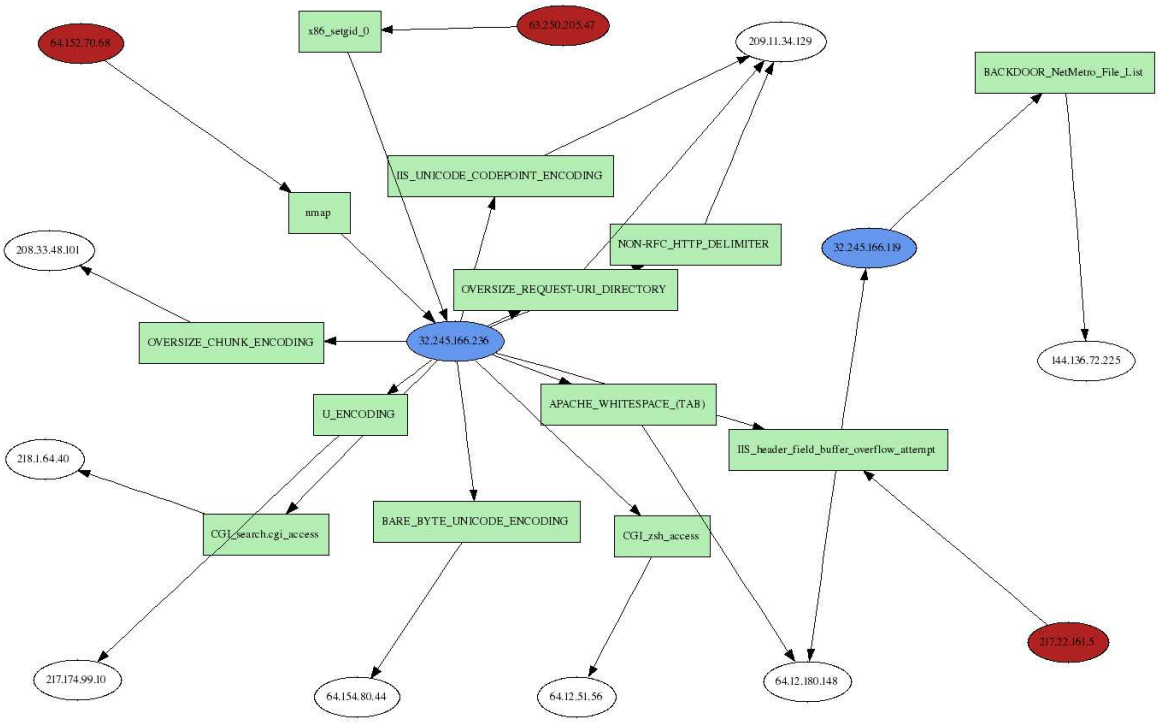


Figure 3.4: Attack-chain graph with false positives and other unimportant nodes removed.

remedial step, we recommend having a look at the two remaining stepping stone machines: "32.245.166.236" and "32.245.166.119". Especially because it is not the first time that we tripped over them during our analysis.

3.4 Proxy Servers

During all the analysis we realized that only a small number (10) of machines from the internal network triggered rules[8]:

```
115.74.249.202      170.129.50.120      32.245.166.119
115.74.249.65       170.129.50.3        32.245.166.236
138.97.18.225       207.166.87.157
138.97.18.88        207.166.87.40
```

A very interesting fact is that two machines are located in each of the internal subnets. Furthermore, the snort alerts for all the machines look very much the same. Having analyzed some of the traffic already earlier, we issue a new hypothesis about these machines. They are *proxy servers*. A few reasons to support this claim:

- All of the 10 machines trigger about the same variety of events.

[8] `select distinct(sourceip) from sans where sourcemac="00:00:0c:04:b2:33"'`

- Only 10 machines, out of a total of 83634 machines were contacted, show up as sources of events. The others are only targets.
- It seems very unlikely that no other machine in the internal network would trigger a single event.
- The internal machines (i.e., the proxies) seem to have many roles. One of them triggers AOL instant messenger events, participates in DNS zone transfers and offers Web pages. This seems very unlikely.

We can unfortunately not further support this claim without knowing more about the topology.

Appendix A

TCPDump Output

To understand the tcpdump(1) output throughout the paper, we provide a simple key here. tcpdump(1) generates the following kind of output if invoked via `tcpdump -nne`. The first `-n` tells tcpdump not to resolve hostnames and the second does prevent it from resolving the ports to service names. The parameter `-e` is used to get the MAC addresses of the traffic in the output.

```
19:27:01.454488 00:00:0c:04:b2:33 > 00:03:e3:d9:26:c0, ethertype IPv4 (0x0800),
[1]             [2]               [3]                      [4]
length 1687: IP 138.97.18.88.63259 > 64.154.80.51.80: P    0:   1633(1633) ack  1634 win 33580
       [5]      [6]          [7]     [8]          [9] [10] [11] [12] [13]  [14]

[1] TimeStamp           [8]  Destination IP
[2] SourceMac           [9]  Destination Port
[3] DestinationMac      [10] TCP Flags
[4] Network Protocol    [11] TCP Sequence Number
[5] IP Packet Length    [12] TCP Last Sequence Number
[6] Source IP           [13] TCP Length
[7] Source Port         [14] ACK flag
```

A parser for tcpdump(1) output can be found in `afterglow-database.tar.gz` at `http://sourceforge.net/project/showfiles.php?group_id=125211`. The script is called tcpdump2sql.pl. It is jsed to populate our MySQL database with the events in order to easier access them (see also Appendix B).

Appendix B

Graphing Event Data

Event graphs like the ones we showed throughout the paper are generated by following these steps:

1. Creating MySQL database for all the details of the snort alerts.
2. Populating database with snort (tcpdump) output.
3. Extracting fields from database to generate comma separated lists.
4. Convert the lists into a format that is readable by the graphic library.
5. Run a graphical interpreter on the data to produce a graph.

Step one requires a database schema that can be populated with the information from snort. Our schema is the following:

```
# MySQL Server version: 4.1.6
# The Schema contains some extra entries utilized for certain
# steps of our analysis
CREATE TABLE sans (
  id int(11) NOT NULL auto_increment,
  `timestamp` datetime NOT NULL default '0000-00-00 00:00:00',
  sourcemac varchar(17) NOT NULL default '',
  destmac varchar(17) NOT NULL default '',
  sourceip varchar(15) NOT NULL default '',
  destip varchar(15) NOT NULL default '',
  sourceport int(5) NOT NULL default '0',
  destport int(5) NOT NULL default '0',
  proto varchar(10) default NULL,
  tcpflags varchar(10) default NULL,
  length int(11) NOT NULL default '0',
  ttl int(11) default NULL,
  ipid int(11) default NULL,
```

```
  iptos varchar(10) default NULL,
  ipflags varchar(5) default NULL,
  `offset` int(11) default NULL,
  snort_alert varchar(100) default NULL,
  service int(11) default NULL,
  delta int(11) default NULL,
  delta2 int(7) default NULL,
  PRIMARY KEY  (id),
  KEY `timestamp` (`timestamp`),
  KEY id (id),
  KEY sourceip (sourceip),
  KEY destip (destip),
  KEY snort_alert (snort_alert)
)
```

The database keys greatly help to improve the speed when issuing queries to the database.

Step two, the population of the database can be done with a script that is available in `afterglow-database.tar.gz` at `http://sourceforge.net/project/showfiles.php?group_id=125211`. To start it, point tcpdump to your snort binary log and pipe the output to the script: `tcpdump -vttttnnelr /tmp/sans | tcpdump2sql.pl`.

Step three requires an example. Let us assume we want to graph the source MAC addresses and the IP addresses that are located behind them. The SQL query for this would be: `select sourcemac, sourceip from sans`. This output should now be converted into a comma separated form in order to feed it to the graphical library. To do so, we used Linux and some command-line tools. The final command we used:

```
echo 'select sourcemac, sourceip' | mysql -s -u root -ppass tcpdump
| awk '{printf "%s,%s\n",$1,$2}' > list.csv
```

The file list.csv now contains the following lines:

```
00:03:e3:d9:26:c0,255.255.255.
00:00:0c:04:b2:33,138.97.144.
00:03:e3:d9:26:c0,255.255.255.
00:00:0c:04:b2:33,138.97.82.
00:03:e3:d9:26:c0,24.84.106.
00:00:0c:04:b2:33,138.97.18.
00:03:e3:d9:26:c0,24.84.106.
00:00:0c:04:b2:33,138.97.18.
00:03:e3:d9:26:c0,24.84.106.
00:00:0c:04:b2:33,138.97.18.
00:03:e3:d9:26:c0,24.84.106.
00:00:0c:04:b2:33,138.97.18.
00:03:e3:d9:26:c0,24.84.106.
```

For step five, we need to explain some more things: We decided to use a package called GraphViz[4] from AT&T Research to generate all the graphs in this paper. GraphViz requires the input to be in a specific language that expresses a graph. A very simple example of a graph definition is the following:

```
digraph G {
    a -> b -> c
}
```

Passing this to the graphviz libraries[1], will generate the graph shown in Figure B.1. For a complete description of the language, have a look at the GraphViz documentation[5].

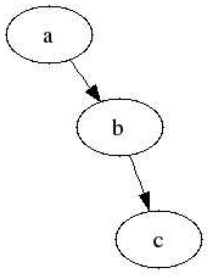


Figure B.1: Simple GraphViz example graph: digraph G $\{a- > b- > c\}$.

Now that we know how the input to the graphical library looks and we know how to generate comma separated output from entries in our database, we need a module that translates the CSV output into GraphViz's language. In order to facilitate this process, we utilized a tool called AfterGlow[14][2]. AfterGlow expects two values on each line. Each line then represents two nodes and a connection between the nodes[3] Using the input, AfterGlow will produce output that can be passed on to one of the utilities from GraphViz.

Continuing on our example, we would now pass the information from `list.csv` into AfterGlow: `cat list.csv | ./afterglow.pl -t > list.dot`. The output is a file that can then be passed into neato[4]: `cat list.dot | neato -Tgif -o list.gif`.

This is the full process of generating graphs. All the steps can be taken together and executed as follows:

```
echo 'select sourcemac, sourceip from sans' | mysql -s -u root -ppass tcpdump |
awk '{printf "%s,%s\n",$1,$2}' | ./afterglow.pl -t | neato -Tgif -o list.gif
```

[1]The command to generate the graph: `echo ``digraph G{a->b->c}'' | neato -Tgif -o list.gif`

[2]The tool was written by Christian Beedgen and myself for the purpose of graphing security events. More information about the project can be found on the Web page: http://afterglow.sourceforge.net/

[3]This is for the two-node mode of AfterGlow.

[4]Neato is one of the tools provided by GraphViz, which takes a GraphViz description of a graph and generates an image as output.

Appendix C

Severity Analysis

The severity of an attack is a measurement for how severe an attack is. Not all attacks have the same impact on an organization. There are multiple factors which have an impact on how an attack potentially impacts the targeted machine. The formula defining the severity is the following:

$$severity = (criticality + lethality) - (systemcountermeasures + networkcountermeasures)$$

The single elements making up the severity are:

Criticality How critical is the target system to the organization. It is important to note that this factor has to be viewed from a business perspective. The more important the target is for the business, the higher this value should be. 5 indicates a very critical system.

Lethality How likely is it that the attack will do harm to the target machine? This factor can potentially be used to reduce the impact of false positives generated by an IDS. Assume for example that an attack targets port 80. Assume further that the port is not open on the target machine. If the IDS does still generate an alarm for this attack, the lethality helps decrease the importance for this event. 5 indicates that an attacker could gain root access to the entire network and is therefore very lethal.

System Countermeasures What countermeasures are in place on the target system? A patched and up to date system with extra hardening tasks performed will get a high number (5). A system which is missing some patches or runs an older operating system will get a lower number.

Network Countermeasures What network countermeasures are deployed? Is there a firewall on the attack path? Is an IPS system in place? Are there multiple access paths to the target system? Do they all employ the same security standards? Again, a 5 illustrates good countermeasures.

Bibliography

[1] *ArGoSoft FTP Server 1.0 Multiple Buffer Overflow Vulnerabilities* http://www.securityfocus.com/bid/1227/info/. 31

[2] *CERT Advisory CA 1996-06*, Cert Coordination Center, http://www.networkpenetration.com/Gobbler-2.0.1-Alpha1.tar.gz. 41

[3] *Ethereal - Network Protocol Analyzer* http://www.ethereal.com. 7

[4] *AT&T GraphViz* http://www.research.att.com/~erg/graphviz. 49

[5] AT&T GraphViz - *Documentation* http://www.research.att.com/~erg/graphviz/info/lang.html. 49

[6] Renaud Deraison, *Nessus*, http://www.nessu.org. 41

[7] *Post on dshield.org about FTP EXPLOIT CWD overflow* http://www.dshield.org/pipermail/intrusions/2002-October/005497.php. 30, 31

[8] *IEEE OUI assignments* http://standards.ieee.org/regauth/oui. 8

[9] *GIAC Certification Practical Logs*, http://isc.sans.org/logs/Raw. 7

[10] *GCIA Practicals* http://www.giac.org/GCIA.php. 25

[11] *Gobbler 2.0.1alpha* http://www.networkpenetration.com/Gobbler-2.0.1-Alpha1.tar.gz. 39

[12] Stephen Hall, *GCIH Practical* http://www.giac.org/practical/GCIH/Stephen_Hall_GCIH.pdf. 30

[13] Ian Martin - *GCIA practical* http://www.giac.org/practical/GCIA/Ian_Martin_GCIA.pdf. 27

[14] Raffael Marty and Christian Beedgen, *AfterGlow* http://afterglow.sourceforge.net. 6, 9, 19, 20, 33, 39, 49

[15] Raffael Marty, *GCIA Practical Additional Chapters* http://raffy.ch/projects/Raffael_Marty_GCIA_Additional_Chapters.pdf 15

[16] Martin Roesch, *Snort IDS*, http://www.snort.org. 7

[17] *MySQL Database* http://www.mysql.org. 9

[18] *New Virus/Worm - Frontpage?* http://lists.jammed.com/incidents/2002/01/0203.html. 38

[19] *Snort User Documentation* http://www.snort.org/docs/snort_manual/node10.html. 34

[20] *Port 0 OS Fingerprinting by Ste Jones NetworkPenetration.com* http://www.networkpenetration.com/port0.html. 38

[21] Pete Storm - *GCIA practical* http://www.giac.org/practical/GCIA/Pete_Storm_GCIA.pdf. 27

[22] *TESO 7350wurm* http://www.packetstormsecurity.org/removed/7350wurm.c. 30, 31

[23] *tcpdump/libpcap*, http://www.tcpdump.org. 7

[24] *Default TTL Values in TCP/IP*, http://secfr.nerim.net/docs/fingerprint/en/ttl_default.html, 15

[25] *Western Digital* http://www.wdc.com. 9

[26] Michal Zalewski *P0f v2*, http://lcamtuf.coredump.cx/p0f. 22