University of Amsterdam
System & Network Engineering

Research Project

DNSSEC Troubleshooting

Author:
Niels Monen
nmonen@os3.nl

Coördinator:
Roland van Rijswijk
SURFnet

July 8, 2011
Version 1.0

This page is left blank intentionally

Abstract

SURFnet has seen a problem which involve large DNSSEC packets, and packet filters blocking fragments. Since the root zone has been signed in 2010, there is an increasing amount of validation requests on the SURFnet infrastructure. More and more administrators are enabling DNSSEC, but are not aware that the packet filters are blocking all but the first fragment. DNSSEC is using big keys to validate and authenticate the resource records, which are the cause of the fragmented packets. This problem can be detected by monitoring for ICMP Type 11 Code 1 packets. In five hours time, I monitored the SURFnet infrastructure, and found 3160 unique IP addresses with this problem. However, there are no reasons to block these fragments nowadays.

Contents

1 Introduction

In July 2010 the root zone of the DNS was signed using DNSSEC. Since then, DNSSEC deployment has taken off on a larger scale with many top-level domains signing their zones and making secure delegations available to their registrars and registrants. Many people are signing their zones and measurements on the SURFnet resolver infrastructure shows a steady climb in the validation rate [1]. However, there are some unforeseen problems [2] [3] that have to be researched. These problems are related to firewalls blocking fragmented IP packets. The problem is explained in Figure 1.

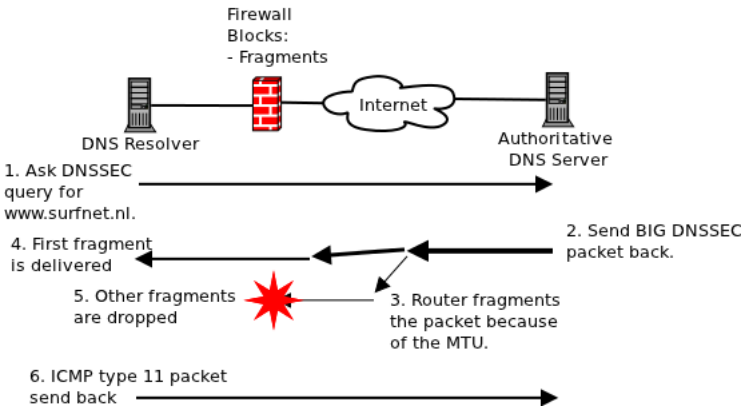


Figure 1: Problem definition

1. The DNS resolver of a client is behind a firewall of the client, and queries the SURFNET authoritative DNS server for a DNSSEC or EDNS0 response.
2. The authoritative DNS server sends a packet which is bigger than the Path MTU back to the resolver.
3. When the packet reaches a router or switch with a small (1500 bytes or less) MTU, the packet gets fragmented.
4. The firewall is configured in such a way that it will allow the first fragment, but drops the next fragments.
5. The IP stack receives the first fragment, and waits a x amount of time.
6. The IP stack drops the fragment, and sends an ICMP type 11 packet back.

1.1 Research

In this project, I will research how the problem shown in Figure 1 is detectable, and can be reproduced in a lab setup. This enables me to detect if the ICMP packet is always send, and by which machine. The results and tests will be in this paper.

By doing this research, I will answer the following question:

> *Is it possible to detect if authoritative DNSSEC responses are blocked at the client side, and in particular when fragmentation occurred?*

The following sub-questions will help to answer the main research question.

- *When and where are the ICMP packets send?*
- *How many of SURFnet clients have this problem?*

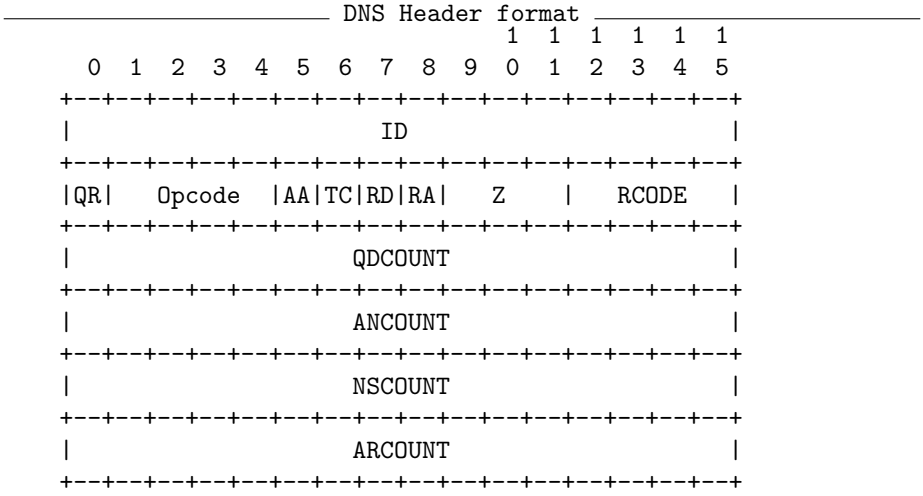
2 Theory research

In this section, the theory, protocols and mechanisms behind the main problem are described.

2.1 DNS

DNS (Domain Name System) [4] is a hierarchical naming system, which allows computers to find the right IP [5] address for a domain name. In RFC 1035 [4], the maximum size of a DNS packet is defined to be 512 bytes. This is the maximum size of a "safe" UDP [6] packet. Such a packet consist of an header, as can be seen below, and several message formats, which I won't specify.

DNS Header format



These days, 512 bytes is not enough for all the information anymore, because of all the "additional" information, or because of DNSSEC. That's why they need to be expanded.

There are two options to enlarge the size of a DNS packet:

Use the TCP protocol DNS can use TCP. However, there is a lot of overhead. The servers first have to establish a connection, and then maintain the state of the connection.

Use the *Extension Mechanism for DNS (EDNS0)* As the name says, it is an extension mechanism which allows more data into the DNS packets. This is done by using reserved bits in the original DNS protocol.

2.1.1 EDNS0

EDNS0 is defined in RFC 2671 [7], which is created in 1999. This extension is developed to extend some fields in the original DNS protocol. The DNS protocol has a limit of 512 bytes, which isn't enough for DNSSEC. DNSSEC needs more space for all the keys involved when the "DO" bit is set. The effected elements of DNS are:

The DNS Message Header's second full 16-bit This is divided into a 4-bit OPCODE, a 4-bit RCODE, and a number of 1-bit flags. All the original reserved bits have now been allocated for various purposes.

The first two bits of a wire format domain label These are used to denote the type of the label. *"0 1"* now indicates an extended label type. *"1 1 1 1 1 1"* is reserved for future expansion.

The limit of 512 octets in size when sent over UDP The maximum reassembly buffer size is still limited to 512 octets of UDP payload, but most of the hosts connected to the Internet are able to reassemble larger datagrams.

There is also an added OPT, namely the pseudo-RR (Resource Record). This record can only be added once, and has a fixed part and a variable set op options. I won't go into detail what the options are how they work. Detailed information about this can be found in RFC 2671 [7].

2.1.2 DNSSEC

DNSSEC is an abbreviations for *"Domain Name System Security Extensions"*, which is first described in RFC 2065 [8]. After this initial RFC, it has been updated several times. The revised specifications are in RFC 2535 [9], and the "Final" specifications are in RFC 4033[10],4034[11],4035[12].

This extension is developed to provides data integrity and authentication. This is done by cryptographic digital signatures, which are included in the zone as resource records. These records are needed to complete the *"Authentication*

Chain" or *"Chain of Trust"* as can be seen in Figure 2. The signatures use the public-key cryptography, which in short means: You create two linked keys with which you can encrypt/decrypt/sign a message. One of the keys is private, and secret to anyone except you. And the other key is public, which can be used by everyone. More information about the public-key cryptography can be found on [13].

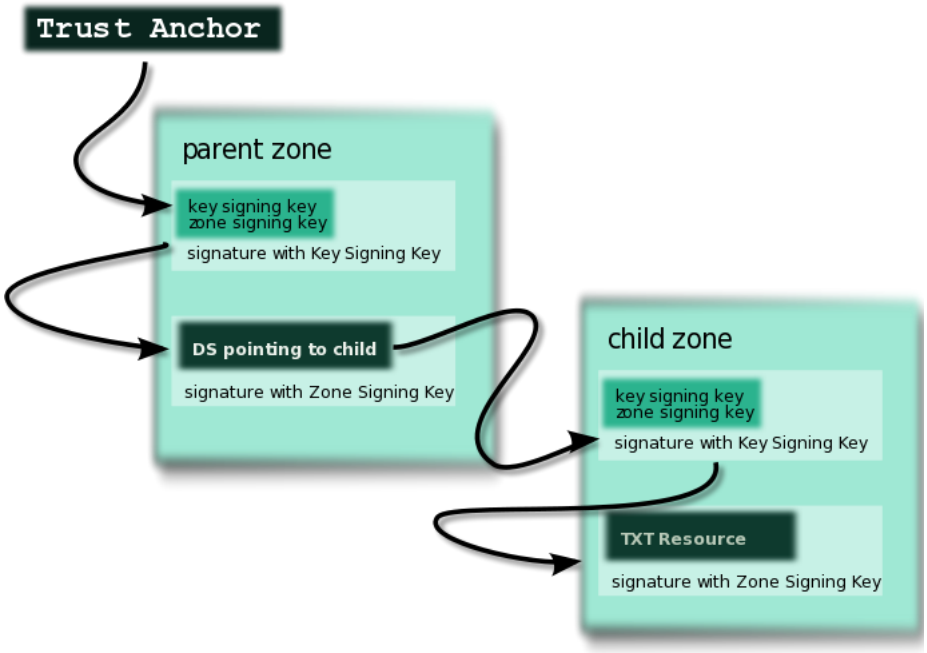


Figure 2: DNSSEC: Chain of Trust. NLnet Labs©.

The records which are needed to setup a chain of trust are:

- DNSKEY
- RRSIG
- NSEC
- DS

These extra resource records create a bigger DNS answer to the client resolver. A normal DNS answer packet from the SURFnet name server can be seen in Appendix D.1. This packet is only 288 bytes big. The DNSSEC answer packet from the same SURFnet name server can be found in Appendix D.2. This packet grew to 1659 bytes, which is bigger than the default MTU(Maximum transmission unit) [14] of Ethernet of 1500 bytes. Because this is the default

size, it has become the de-facto standard on the Internet. Because of the MTU, the packet will be fragmented by the first router which has a link with the MTU set to 1500 bytes.

A client can enable DNSSEC by setting the "DO" bit to 1 in a query. This means the client accepts the DNSSEC resource records, and will check if the chain of trust is right for the requested domain.

2.2 Fragmentation

When a router receives a packet with a bigger PDU (Protocol Data Unit)[15] than the MTU of the next hop, it has two options. The first one is to drop the packet, and send a ICMP type 3 code 4 (Destination Unreachable, fragmentation needed) [16]. The other option is to fragment the packet so it can be send over the link with the smaller MTU. This is done in a standardized way, which is described in RFC 791 [5].

- The Internet Protocol module creates two new Internet datagrams.
- Copy the contents of the Internet header from the original packet to the newly created datagrams.
- Divide the data of the original packet into portion on a 64 bit boundary.
- The *"More-Fragments"* flag is set to one in all but last fragment.
- The *"More-Fragments"* flag is set to zero in the last fragment to indicate it is the last fragment.
- The *"Fragment Offset"* of the fragments is set.

The host will then reassemble the fragments into one normal IP packet using the algorithm described in RFC 815 [17]. While this is absolutely normal behavior on the Internet, some administrators choose to block these fragments.

2.2.1 Why should one block fragments?

Around 1995, there where some fragment attacks which gave attackers the ability to crash or gain access to a host. RFC 1858 [18] is dedicated to these attacks. It describes some attacks and how they work:

Tiny fragment attack This attack is based on the minimum allowed fragment size of a packet, which is defined in RFC 791 [5].

> Every internet module must be able to forward a datagram of 68 octets without further fragmentation. This is because an internet header may be up to 60 octets, and the minimum fragment is 8 octets.

By creating a big TCP header, the TCP flags could be pushed into the next fragment. This gave some problems with packet filters, which just let them through.

Overlapping fragment attack This attack uses fragments which will overwrite a part of a previous fragment. This could be done because the reassembly algorithm didn't check if the fragment offset was right.

Ping of Death This attack is based on the maximum size an IP packet can be. This is defined in RFC 791 [5] to be 65.535 bytes. When such a big packet is send, it will be fragmented. But because of the added *"Fragment Offset"* of 13 bits, the last fragment can only have a maximum offset of 65.528, and data no larger than 7 bytes. A malicious person could send the last fragment with the maximum offset, but with more data. This resulted into an IP packet larger than 65.535 bytes and cause a buffer overflow.

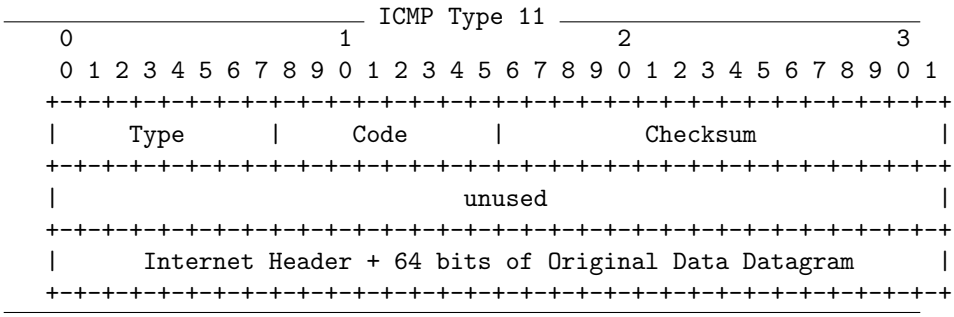
All these attacks are outdated, and won't work nowadays, but administrators are still configuring firewall rules for them. Or the firewall systems are that old and critical, the administrators won't touch those rules.

2.3 ICMP

ICMP is an abbreviations of Internet Control Message Protocol, which are typically used for error reporting in the IP layer. This protocol is defined in RFC 792 [16]. While there are many different types of ICMP packets, I will only focus on Type 11 *"Time Exceeded"* packets.

2.3.1 Type 11: Time Exceeded

This type of ICMP packet can have two meanings. These are differentiated by the *"Code"* field. Code 0 is *"time to live exceeded in transit"* and Code 1 is *"fragment reassembly time exceeded"*. In this project, I will focus on the Code 1 of these ICMP packets. The layout of such a packet can be seen below.



These packets are send when a host cannot complete the reassembly of an IP packet, due to missing fragments within the time limit.

To standardize the behavior on the Internet, RFC 1122 [19] has been written by the IETF [20] in 1989. In this RFC the requirements of communication layers for Internet hosts are defined. While the ICMP RFC says only 64 bits (8 bytes) of the original datagram can be send, RFC 1122 defines that *at least* the first 8 bytes of the original data datagram has to be added, but more MAY be sent. They also define that this timeout MUST be present, and when this timeout expires, the partially-reassembled datagram MUST be discarded, and an ICMP Type 11 Code 1 MUST be sent to the source host. But only when fragment 0 is received. To ensure the ICMP packets are really sent, they say they MUST be passed to the transport layer.

3 Methodology

In this section the methodology that is used for the research is discussed. As stated in Section 1, the theory behind the problem has to be studied, and test if the problem is reproducible in a lab environment. If this problem is reproducible, the amount of clients that have this problem has to be measured.

3.1 Lab setup

To test the ICMP packets in a lab environment, I setup three machines. One DNSSEC enabled resolver, one client, and one machines which acted as packet filter between the client and resolver. The specifications of these machines can be found in Appendix A, and the graphical representation of the setup can be seen in Figure 3.

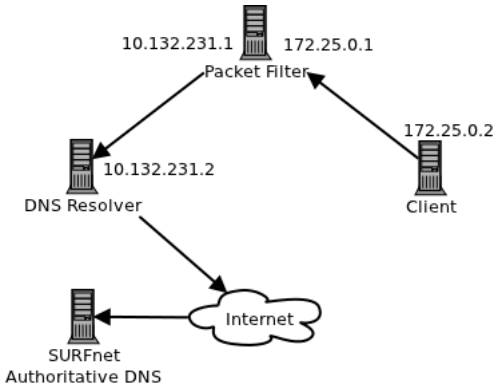


Figure 3: Lab setup

This setup is not exactly the same as the original setup with the problem. If I had to build a setup, exactly like the problem, I had to create a chain of trust for DNSSEC, which would be a lot of extra work. However, the setup created by my can do the same. The DNS resolver acts as a proxy for the client, so I can capture the packets on all interfaces involved.

The client and resolver are configured to use the packet filter as a gateway to the different networks they are connected to. On Linux, this can be done by executing the following commands:

route on client
```
route add -net 10.132.231.0 netmask 255.255.255.0 gw 172.25.0.1 dev eth1
```

route on resolver
```
route add -net 172.25.0.0 netmask 255.255.255.0 gw 10.132.231.1 dev eth1
```

On Windows this can be done in the *"Network Connection properties"*, and setting the right gateway.

To get the whole setup working, IP forwarding has to be enabled in the packet filter machine. This is done by adding or changing the line with gateway_enable:

IP forwarding
```
#gateway_enable="NO"
gateway_enable="YES"
```

3.1.1 Unbound

As DNSSEC enabled resolver, I used *"Unbound 1.4.10"* [21]. This is a validating, recursive, and caching DNS resolver. This package is developed and maintained by NLnet Labs [22]. It has been developed with security in mind. This means that it has DNSSEC options build in, and is easily configurable. The configuration of the machine can be found at Appendix C. In this configuration, NLnet Labs already added an option to set the maximum size of packets, so they won't get fragmented. The default EDNS0 buffer size in *Unbound* is 4096 bytes, but they advise 1480 bytes to solve fragmentation timeouts if they occur. When those fragmentation timeouts are not occuring, the EDNS0 buffer size should not be changed. With this option, *Unbound* will not ask as much additional information as possible from the authoritative server, so this will fill up the packet till the buffer size is reached. An answer with 1480 bytes of buffer size can be found in Appendix D.2.1.

3.1.2 Packet filter

To reproduce the problem in the lab setup, not all the fragments have to be blocked by the packet filter. The first fragment should arrive at the IP protocol stack, and the other fragments should be dropped without notification.
I tested several packet filters which should be able to do this. The packet filters are:

IPtables This is a packet filtering framework inside the Linux 2.4.x and 2.6.x kernel [23]. According to the man pages of this framework, it should be able to block the second and third fragment:

IPtables manpage

```
[!] -f, --fragment
     This means that the rule only refers to second and further frag
     ments of fragmented packets.  Since there is no way to tell  the
     source  or  destination  ports  of such a packet (or ICMP type),
     such a packet will not match any rules which specify them.  When
     the  "!"  argument  precedes  the  "-f" flag, the rule will only
     match head fragments, or unfragmented packets.
```

However, I couldn't get it working as intended. This could be because IPtables had to route the traffic too, and skips some rules. Because of the time limitations for this project, I had to abandon this framework.

PF This is a **P**acket **F**ilter for BSD kernels[24]. I installed Debian with kFreeBSD [25] to test. This packet filter can *"scrub"* fragments. Scrubbing is their name for fragment reassembly. The documentation states that when you do not scrub fragments, they are processed as normal packets. This should mean they can also be blocked because in fragments there is no information about the ports it should go to. However, this also took too much time to get it working as intended.

IPFW This is a *"IPFIREWALL"* [26] sponsored by FreeBSD. Because of this, I installed a clean FreeBSD-8.2 server. This firewall is very easy to setup and configure. The only rules I used on this firewall can be seen in Appendix B. The line:

IPFW deny fragment

```
ipfw -q add deny all from any to any in frag
```

tells the firewall to block every fragment it detects, except the first one.

3.1.3 Extensible Ethernet Monitor

To measure the amount of clients which have this problem, I used a tool SURFnet has developed. This is the *"Extensible Ethernet Monitor"*, or *"eemo"* [27]. This program listens to the live traffic which arrives at a server, and sends it to a back-end server for post-processing. It already has several plug-ins to capture different types of packets, which includes the ICMP packets.

3.1.4 Packet analyzers

To see what is exactly send on the network, I used *"tcpdump"* [28]. This program is a powerful command-line packet analyzer, and uses the *"libpcap"* library to

capture the network traffic. It is lightweight, and can capture data on separate interfaces simultaneously, which makes it perfect for my test setup. Because it uses the *"libpcap"* library, one can open the saved file with any other program which uses this library.

To open and analyze the data, I used *"Wireshark"* [29]. This is a network protocol analyzer, which lets you capture and interactively browse the network traffic. Because it can recognize "conversations" and can "follow streams", I can analyze the captured traffic more easily than with tcpdump.

3.2 Test procedures

For me to study if, when and where the ICMP packets are send, I had to create a small test procedure. This procedure is applied on all the tested packet filters, clients, and buffer sizes.

3.2.1 Test plan: Reproducibility

For the tests, I only had to generate a fragmented DNS packet. This is done from the client in Figure 3.

1. **Tcpdumps** Start tcpdump on each machine for each interface. The commands I executed for this are:

   tcpdump on packet filter
   ```
   $ tcpdump -i le0 host 10.132.231.2 or host 172.25.0.2 -s0 \
    -w ~/le0_dump.pcap
   $ tcpdump -i le1 host 10.132.231.2 or host 172.25.0.2 -s0 \
   -w ~/le1_dump.pcap
   ```

   tcpdump on client and server
   ```
   $ tcpdump -i eth1 src 10.132.231.2 or src 172.25.0.2 -s0 \
   -u -w ~/client_dump.pcap
   $ tcpdump -i eth1 src 10.132.231.2 or src 172.25.0.2 -s0 \
   -u -w ~/server_dump.pcap
   ```

   Because these servers where part of a test lan, there was a lot of other traffic. The "src" or "host" made it possible to only capture the right packets. The "-i" tells tcpdump on which interface it has to listen; "-s0" tells tcpdump to capture every size of package; "-u" tells tcpdump to write unbuffered; and "-w [file]" tells tcpdump to write the data to a file, instead of showing it on the stdout.

2. **Unbound** Start unbound with the right configuration file.

3. **Request** Request a DNSSEC and EDNS0 enabled answer with:

```
dig request
$ dig @10.132.231.2 www.surfnet.nl +dnssec +bufsize=4096 \
+tries=1 +retry=0
```

   Dig is a DNS look up utility. This request asks the unbound server (10.132.231.2) to look up "www.surfnet.nl" with dnssec enabled, and buffer size 4096. This is needed to enable EDNS0. It will only try this 1 times, without any retries. This way I can clearly see the ICMP response.

4. **Stop tcpdumps** Stop the tcpdumps on the machines.

5. **Analyze** Analyze the data with tcpdump, or Wireshark.

3.2.2 Test plan: Live measurement

As said in Section 3.1.3, I used the *"eemo"* package to capture the live traffic. To do this I followed a few steps:

1. **Configure** Configure the *"eemo"* server to send the data to the right back-end server for post-processing. This was a server I controlled.

2. **Start** Start both the *"eemo"* servers.

3. **Capture** The back-end server outputs the received data to "stdout". To capture this, I redirected the output to a file.

4. **Reformat the data** The back-end server outputs the data as seen below:

```
Example output eemo
Received 148 bytes
Sending sensor ID is 0x00000001

Received data is fragmentation monitoring data

host = 62.24.xxx.xxx
host = 85.62.xxx.xxx
host = 62.24.xxx.xxx
host = 211.140.xxx.xxx
host = 211.140.xxx.xxx
host = 216.150.xxx.xxx
...
```

   (The data in the report is anonymized, the real output shows the whole IP address).
   Because of the extra information provided, I had to reformat the data into one big list of IPs. This can be done by executing

Reformatting
```
cat [file] | grep "host = " > [only_ip file]
```

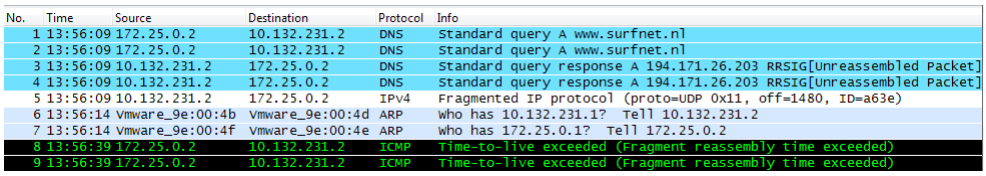
5. **Analyze** Analyze the data. How many requests failed, and from how many unique IPs.

4 Measurements

In this section, I will discuss the results of the tests that are described in the test plans in Section 3.2.

4.1 Reproducibility

After I executed my test plan to reproduce the problem, I found out the problem could exists. The client (172.25.0.2) asks the DNS resolver (10.132.231.2) for "www.surfnet.nl" with the "DO" bit set to 1, and payload size of 4096 bytes. The DNS resolver answers to this query with a fragmented IP packet. This is the query response. Figure 4 is a *merged* network dump of the two interfaces of the packet filter.

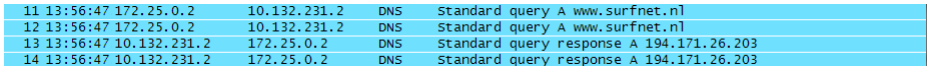


No.	Time	Source	Destination	Protocol	Info
1	13:56:09	172.25.0.2	10.132.231.2	DNS	Standard query A www.surfnet.nl
2	13:56:09	172.25.0.2	10.132.231.2	DNS	Standard query A www.surfnet.nl
3	13:56:09	10.132.231.2	172.25.0.2	DNS	Standard query response A 194.171.26.203 RRSIG[Unreassembled Packet]
4	13:56:09	10.132.231.2	172.25.0.2	DNS	Standard query response A 194.171.26.203 RRSIG[Unreassembled Packet]
5	13:56:09	10.132.231.2	172.25.0.2	IPv4	Fragmented IP protocol (proto=UDP 0x11, off=1480, ID=a63e)
6	13:56:14	Vmware_9e:00:4b	Vmware_9e:00:4d	ARP	Who has 10.132.231.1? Tell 10.132.231.2
7	13:56:14	Vmware_9e:00:4f	Vmware_9e:00:4e	ARP	Who has 172.25.0.1? Tell 172.25.0.2
8	13:56:39	172.25.0.2	10.132.231.2	ICMP	Time-to-live exceeded (Fragment reassembly time exceeded)
9	13:56:39	172.25.0.2	10.132.231.2	ICMP	Time-to-live exceeded (Fragment reassembly time exceeded)

Figure 4: Network dump of the problem

Packets number 2, 3, 5 and 9 are from the interface connected with the DNS resolver. This means it is still unfiltered network traffic. Packet 5 is the second fragment of the DNSSEC packet. Packets number 1, 4 and 8 are from the interface connected to the client. In packet 1 and 2, we see the client querying for "www.surfnet.nl". Packet 3 is a fragment of the response from the resolver, which goes through the packet filter, which can be seen with packet 4. However, the second fragment (packet 5) is only seen on the interface connected with the resolver. This means it is dropped by the packet filter. After 30 seconds, the client sends a ICMP Type 11 Code 1, which can be seen in packet 8. This is going through the packet filter, and delivered to the DNS resolver in packet 9.

To test if not all DNS traffic is being stopped, I also tested if "www.surfnet.nl" would resolve when I was not requesting DNSSEC information. This can be seen in Figure 5.



11	13:56:47	172.25.0.2	10.132.231.2	DNS	Standard query A www.surfnet.nl
12	13:56:47	172.25.0.2	10.132.231.2	DNS	Standard query A www.surfnet.nl
13	13:56:47	10.132.231.2	172.25.0.2	DNS	Standard query response A 194.171.26.203
14	13:56:47	10.132.231.2	172.25.0.2	DNS	Standard query response A 194.171.26.203

Figure 5: Network dump for a normal DNS query

In this figure, packet 12 and 13 are from the interface connected to the DNS resolver, and packet 11 and 14 are from the interface connected to the client. In packet 11, the client asks where "www.sufnet.nl" is located. And in packet 14,

the client get its answer back. This shows the normal DNS traffic is not being blocked.

4.1.1 ICMP

ICMP Type 11 As can be seen in Figure 4, the ICMP Type 11 Code 1 packets are send by the client, to the source (DNS resolver). This is compliant to the RFCs. These ICMP packets are around 1500 bytes. This means it is compliant to RFC 1122, were it specifies at least 8 bytes, but more MAY be added.

Timeout There is no default reassembly timeout defined in the RFCs, but there are suggested values for the timeout in RFC 1122 [19]. In this RFC they suggest a timeout between 60 seconds and 120 seconds. I found out that on Ubuntu 11.04 the default timeout for fragment reassembly is 30 seconds. This is defined in the kernel parameter *"net.ipv4.ipfrag_time"* or *"/proc/sys/net/ipv4/ipfrag_time"*. After this, I looked at several other Linux operating systems, and they all have a 30 seconds timeout. Windows Server 2008 R2 is more compliant with the RFC, and uses a timeout of 60 seconds.

Operating System	Fragment timeout (s)
Ubuntu 11.04	30
Ubuntu 10.04	30
Debian 6.0	30
CentOS 5.6	30
Windows 2008 R2	60

Table 1: Fragment timeout per OS

4.2 SURFnet DNS

I had the opportunity to capture live traffic for five hours (12:00 - 17:00, June 23 2011) and monitor the amount of ICMP Type 11 Code 1 packets on the first name server of SURFnet. In these five hours, a total amount of 15530 ICMP Type 11 packets were captured, which came from 3160 unique IP addresses. From those 15530 packets, 911 came from IPv6 addresses, which originated from 334 unique IPv6 addresses. This is a big part (10.6%) of all the unique IP addresses.

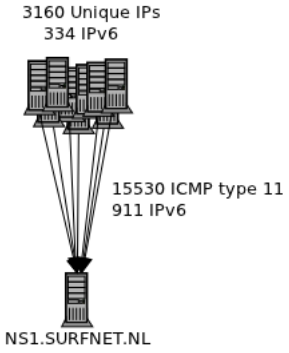


Figure 6: Results of 5 hour scan

To give some perspective to these numbers, the total amount of DNS queries in these five hours is estimated on 600.000. The amount of DNSSEC enabled queries in these five hours is estimated on 300.000.

5 Conclusion and recommendations

In this section I state my conclusions of the results that were discussed in the previous sections.

This research was about studying and reproducing if it is possible to detect if a packet filter between the client DNS resolver and the authoritative DNS server of SURFnet is blocking fragmented DNSSEC traffic. In the lab setup created by me, the client queries the DNS resolver, which will respond to the client with a fragmented DNS answer. The packet filter allows the first fragment, but blocks the seconds one. After 30 seconds, the client sends a ICMP Type 11 Code 1 packet back to the DNS resolver to inform him the fragment reassembly has timed out. This is exactly what is seen on the infrastructure of SURFnet, so the problem is reproducible.

This problem is happening, because of the additional information that DNSSEC is generating. These extra resource records are needed to validate another resource record, or to authenticate the chain of trust. This results into a DNS answer which is 1659 bytes big. This size is bigger than the default MTU on the Internet, which is 1500 bytes, causing it to fragment somewhere between the authoritative DNS server, and the packet filter of the client.

The other side of this research was to see how many of the SURFnet clients have this problem. To study this, I had the opportunity to capture live data on the first name server of SURFnet for five hours. The result of this was that 3160 unique IP addresses had this problem. This means, those 3160 clients could not obtain some of the DNS information in the "surfnet.nl" zone, but they also won't be able to obtain DNS information for other domains which answer with big DNS packets.

I could not find any reason why fragments should be blocked nowadays. I could find attacks from 1995, which were using bugs to get through packet filters, or to crash systems. However, there bugs are patched long ago, so they won't work any more.

6 Future work

This project has some future work which could be researched. They are discussed in this section.

Web application A web page or application can be build, so administrators can test if their DNS resolver has any problem receiving fragmented packets. This can be done by using the probe I build. The only problem I foresee is that the web server is running as a less privileged user. And for creating packets, the root user is needed. The source code of the probe can be found in Appendix E.

Test on bigger scale This problem might be tested on a bigger scale, like the ".nl" domain. This way you can test a bigger "audience". However, I suspect the ".nl" domain does not have this problem, because the DNS(SEC) answers aren't that big. If this is really the case, the problem might be tested on a "bigger" domain.

Test if ICMP packets always arrive The question I had after this research was, "Are ICMP packets always arriving at the source host?". Maybe there are administrators which even block egress ICMP packets. When this is happening, even more DNS resolvers than I have measured in this research, could have this problem. This could be done by checking how many times a DNS resolver tries to get its answers. Most of the times, a DNS resolver will try three times before giving up, or trying DNS over TCP. When this information is known, more and more precise conclusions can be made.

Plug in for DNS packages A small monitor program can be created, which outputs the IPs it finds that have this problem to a list. The DNS package than looks up if an IP is on this list, and lowers the DNS(SEC) answer size to below the fragment size. There could be a time-to-live on this list, so when administrators fix this problem on their side, the server will receive a normal DNS(SEC) answer again.

Research why there are so much IPv6 with this problem In my results, there are relatively a lot of IPv6 addresses. There are some ideas why this could be. The first idea is that IPv6 records (AAAA) are bigger than normal IPv4 records. The other idea is that the Red hat 5 IPv6 implementation is dropping fragments [30].

A Server details

In this section, I will describe the specifications of the used machines.

- DNS Resolver

  Brand VMware

  CPU Intel(R) Xeon(R) CPU L5520 @ 2.27GHz

  Memory 1GB

  OS Ubuntu 11.04 Server edition

  Kernel 2.6.38-8-generic-pae #42-Ubuntu SMP Mon Apr 11 05:17:09 UTC 2011 i686 i686 i386 GNU/Linux

- DNS Client

  Brand VMware

  CPU Intel(R) Xeon(R) CPU L5520 @ 2.27GHz

  Memory 1GB

  OS Ubuntu 11.04 Server edition

  Kernel 2.6.38-8-generic-pae #42-Ubuntu SMP Mon Apr 11 05:17:09 UTC 2011 i686 i686 i386 GNU/Linux

- DNS Client Windows

  Brand VMware

  CPU Intel(R) Xeon(R) CPU L5520 @ 2.27GHz

  Memory 1GB

  OS Microsoft Windows 2008 R2 Server

- Packet Filter

  Brand VMware

  CPU Intel(R) Xeon(R) CPU L5520 @ 2.27GHz

  Memory 1GB

  OS FreeBSD 8.2

  Kernel 8.2-RELEASE FreeBSD 8.2-RELEASE #0: Fri Feb 18 02:24:46 UTC 2011 root@almeida.cse.buffalo.edu:/usr/obj/usr/src/sys/GENERIC i386

B Firewall setup

IPFW ruleset

```
ipfw -q -f flush

cmd="ipfw -q "

$cmd add deny all from any to any in frag
$cmd add allow tcp from any to any
$cmd add allow udp from any to any
$cmd add allow icmp from any to any
```

ipwf list

```
nimo02# ipfw list
00100 deny ip from any to any in frag
00200 allow tcp from any to any
00300 allow udp from any to any
00400 allow icmp from any to any
65535 deny ip from any to any
```

C Unbound configuration

unbound.conf

```
server:
# verbosity number, 0 is least verbose. 1 is default.
verbosity: 1

# specify the interfaces to answer queries from by ip-address.
# The default is to listen to localhost (127.0.0.1 and ::1).
# specify 0.0.0.0 and ::0 to bind to all available interfaces.
# specify every interface[@port] on a new 'interface:' labelled line.
# The listen interfaces are not changed on reload, only on restart.
interface: 0.0.0.0

# port to answer queries from
port: 53

# EDNS reassembly buffer to advertise to UDP peers (the actual buffer
# is set with msg-buffer-size). 1480 can solve fragmentation (timeouts).
# edns-buffer-size: 4096

# buffer size for handling DNS data. No messages larger than this
# size can be sent or received, by UDP or TCP. In bytes.
```

```
# msg-buffer-size: 65552

# control which clients are allowed to make (recursive) queries
# to this server. Specify classless netblocks with /size and action.
# By default everything is refused, except for localhost.
# Choose deny (drop message), refuse (polite error reply),
# allow (recursive ok), allow_snoop (recursive and nonrecursive ok)
access-control: 127.0.0.0/8 allow
access-control: 10.132.231.0/24 allow
access-control: 172.25.0.0/24 allow
```

D SURFnet DNS Answers

D.1 SURFnet normal DNS

DNS answer

```
; <<>> DiG 9.7.3 <<>> www.surfnet.nl
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 33469
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 5, ADDITIONAL: 6

;; QUESTION SECTION:
;www.surfnet.nl.        IN      A

;; ANSWER SECTION:
www.surfnet.nl.     3163    IN      A       194.171.26.203

;; AUTHORITY SECTION:
surfnet.nl.     2337    IN      NS      ns0.ja.net.
surfnet.nl.     2337    IN      NS      ns2.surfnet.nl.
surfnet.nl.     2337    IN      NS      ns1.surfnet.nl.
surfnet.nl.     2337    IN      NS      ns1.zurich.surf.net.
surfnet.nl.     2337    IN      NS      ns3.surfnet.nl.

;; ADDITIONAL SECTION:
ns1.surfnet.nl.     2337    IN      A       192.87.106.101
ns1.surfnet.nl.     2337    IN      AAAA    2001:610:1:800a:192:87:106:101
ns2.surfnet.nl.     2337    IN      A       192.87.36.2
ns2.surfnet.nl.     2337    IN      AAAA    2001:610:3:200a:192:87:36:2
ns3.surfnet.nl.     2337    IN      A       195.169.124.71
ns3.surfnet.nl.     2337    IN      AAAA    2001:610:0:800c:195:169:124:71

;; Query time: 7 msec
```

```
;; SERVER: 192.87.106.106#53(192.87.106.106)
;; WHEN: Thu Jun 16 13:52:50 2011
;; MSG SIZE  rcvd: 288
```

D.2 SURFnet DNSSEC answer

DNSSEC answer

```
; <<>> DiG 9.7.3 <<>> www.surfnet.nl +dnssec
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 39320
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 2, AUTHORITY: 6, ADDITIONAL: 13

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 4096
;; QUESTION SECTION:
;www.surfnet.nl.                        IN        A

;; ANSWER SECTION:
www.surfnet.nl.        3132        IN        A        194.171.26.203
www.surfnet.nl.        3132        IN        RRSIG        A 8 3 3600 20110621225534
20110615070013 30160 surfnet.nl. 9+QfZyefAz1iq6lwrl1+rn1n+47pVPkf
Z8QnhkJjv0PqYEJXAx1vpMDX 3+1nfCijNPac01Q82UJ2Y60ZGhBCufLrM3ezeOWl
9q9CEiWak8ryTFnN pYzIvtkSz6KHlUq16yelNAU+ijtc6qQEhgrQN1jmhSqmmp+I
r8f2T4iN XM4=

;; AUTHORITY SECTION:
surfnet.nl.        2306        IN        NS        ns0.ja.net.
surfnet.nl.        2306        IN        NS        ns2.surfnet.nl.
surfnet.nl.        2306        IN        NS        ns1.surfnet.nl.
surfnet.nl.        2306        IN        NS        ns1.zurich.surf.net.
surfnet.nl.        2306        IN        NS        ns3.surfnet.nl.
surfnet.nl.        2306        IN        RRSIG        NS 8 2 7200 20110622051656
20110615070013 30160 surfnet.nl. Lfo0uyu+meOE7zLYEppSZJ7Prn3itKQLE
Oh6rwA0i+ZjEDYef5/t/Xib DuEmeOqfI8yb/ul5we/UPZweSNfINI9Cs8V4iuYKa/
ydin5Pu6RFHcMY Pd9dwxjhKZAAwoLfTBjfZ5uTJ0ar3Te8rMQRBGrmyQAbz0wxbUj
u9JYp Bq8=

;; ADDITIONAL SECTION:
ns1.surfnet.nl.        2306        IN        A        192.87.106.101
ns1.surfnet.nl.        2306        IN        AAAA        2001:610:1:800a:192:87:106:101
ns2.surfnet.nl.        2306        IN        A        192.87.36.2
ns2.surfnet.nl.        2306        IN        AAAA        2001:610:3:200a:192:87:36:2
ns3.surfnet.nl.        2306        IN        A        195.169.124.71
ns3.surfnet.nl.        2306        IN        AAAA        2001:610:0:800c:195:169:124:71
```

```
ns1.surfnet.nl.         2306    IN      RRSIG   A 8 3 7200 20110622161346
20110615070013 30160 surfnet.nl. Y5DQgVHk6hVv335Mvi6bBSg8aTkQWCcE
AQ+5LPIaMvcQf4tll7kQJnUu k5Zfbxozxd9+JgEh9Q4pMd/UTps0Fho/oaVWLNBA
bXMzVcGf60odEvBU aLKw8AanwauykR1AoUiyb0Rl5G6c1csRT21mZfo1jdrIJgos
QeINMyn2 9Do=
ns1.surfnet.nl.         2306    IN      RRSIG   AAAA 8 3 7200 20110622191527
 20110615070013 30160 surfnet.nl. W9/kRSVhU26eKDwEq9LQ8g830FupycI
X51tWBJqeHohgutHkXqiUvj+q Mrbqz96dOlpXMM/Un/vuqw2LOJzSlRzw7snjOci
DNpp4ILRphA+Yf4Nt aFntN1jgsjmI4FkwMpwbD6p5V/y00o8huzieFNkamPFRKJM
f2XUNAG4n 7jE=
ns2.surfnet.nl.         2306    IN      RRSIG   A 8 3 7200 20110622094157
 20110615070013 30160 surfnet.nl. YLFovBuwyx1N9IU3/4RXwRChMXfIKPG
j7rNVmWeKPpOyLlL7a+YinxPo 16WQdHbGHa0I6ZV2tTXNqTNhk+c3Afj7coUDHL+
fD7IFoICM9J5XJFN5 u71dmc7KXU1krxReWtn8Bly7+yBaCLeugKHm6SJpIkKf32p
w3DWGc6ys qUg=
ns2.surfnet.nl.         2306    IN      RRSIG   AAAA 8 3 7200 20110622060339
 20110615070013 30160 surfnet.nl. Uz2LPenyHscJZP5FB1Rk7gBz8SO2hTT
KpXNy3+tn51Wligg9n4HhnEZ1 gRtjpL3JrpkQSEJyFz5doFMmr20e6zeDLBvRtbN
ob1lAgeEFgB9KzT5V x/Ixitfq1NbhJVI/6EQE0yef3yNLVurPm5pzrWs618Zcr/f
VNV7+wHTQ ipk=
ns3.surfnet.nl.         2306    IN      RRSIG   A 8 3 7200 20110622042940
 20110615070013 30160 surfnet.nl. ewHhKueMzv8yRQn3GaIvX3g5/kpFGTD
8Eo73XRka0DRFZ4CXn40kK8cQ WLHFPumJQN724XqFxG3SjPDUTNsXszNrQam1184
HvCz6ZJfRGDv7tIOe ds3PRz6qsaI89vbeEkhV+DqMymBk/7UN7VGzFPLUz6puHRy
kQHM1GDPV bhE=
ns3.surfnet.nl.         2306    IN      RRSIG   AAAA 8 3 7200 20110622001049
20110615070013 30160 surfnet.nl. kMXR6ftaBMOZOrKCtH8o2F53vrGsPEDv
JT87n3jKNhusdduQ3r69DktA sYaede75quhttyasG7NX+PFe+7VYxhaZUQiZ6V+Y
bZsygA+BuC+snpkn erbRBYy15Fx19zCIWIR/ovCUP6PRN35Epbv7iCxPgFnzwCuv
NY8Itemf 8ow=

;; Query time: 13 msec
;; SERVER: 192.87.106.106#53(192.87.106.106)
;; WHEN: Thu Jun 16 13:53:21 2011
;; MSG SIZE  rcvd: 1659
```

D.2.1 SURFnet DNSSEC answer limited

DNSSEC limited answer

```
; <<>> DiG 9.7.3 <<>> @10.132.231.2 www.surfnet.nl +dnssec
; +bufsize=4096 +tries=1 +retry=0
; (1 server found)
;; global options: +cmd
;; Got answer:
```

```
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 38386
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 6, ADDITIONAL: 11

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 1480
;; QUESTION SECTION:
;www.surfnet.nl.                        IN        A

;; ANSWER SECTION:
www.surfnet.nl. 3600        IN        A        194.171.26.203
www.surfnet.nl. 3600        IN        RRSIG        A 8 3 3600 20110629200223
20110623000007 30160 surfnet.nl. ZWw9z6uSlPpVg2mfqJrjjeDBOGo0H/T7
0NYQeJYr702ioJgNDkMVvo3P k48VqPUCyHshYw7wqjs46Fi8sGACatRa+Ico6k2i
ZhGMbTo7tMOCBnv9V0ioR6DsUXmYISX0sR6Ojt2SJo6DOhY+ext3QQCb1xYVuINlX
ESerL31 yis=

;; AUTHORITY SECTION:
surfnet.nl. 7200        IN        NS        ns1.surfnet.nl.
surfnet.nl. 7200        IN        NS        ns1.zurich.surf.net.
surfnet.nl. 7200        IN        NS        ns0.ja.net.
surfnet.nl. 7200        IN        NS        ns3.surfnet.nl.
surfnet.nl. 7200        IN        NS        ns2.surfnet.nl.
surfnet.nl. 7200        IN        RRSIG        NS 8 2 7200 20110630154809
20110623120007 30160 surfnet.nl. VFlPczRHLx5ClfjQhqpKBkUk1WBrD129
g2Q2T/8h4iXu3kRlWD5MliTq P5+tPFM6LV/uhU9Q4yXHBoEabzVBHbsF3B8J48Jy
QCQVtsTfpafMTeY+ N54AogS6Lb1K8vINAhWmBPmlPJ0mX7uej7BOitHViIpKjQzI
mDl9nUJ/ mV8=

;; ADDITIONAL SECTION:
ns1.surfnet.nl.        7200        IN        A        192.87.106.101
ns1.surfnet.nl.        7200        IN        AAAA        2001:610:1:800a:192:87:106:101
ns2.surfnet.nl.        7200        IN        A        192.87.36.2
ns2.surfnet.nl.        7200        IN        AAAA        2001:610:3:200a:192:87:36:2
ns3.surfnet.nl.        7200        IN        A        195.169.124.71
ns3.surfnet.nl.        7200        IN        AAAA        2001:610:0:800c:195:169:124:71
ns1.surfnet.nl.        7200        IN        RRSIG        A 8 3 7200 20110630133059
20110623200007 30160 surfnet.nl. obariyLkTMUxxePJWzI6I3EmbjS1PYD3
q86q1hTk25BEEgErWtI/jlUe rqxD/6bNRRPn8a2iywdpIS81Mb9pF+9MvAZcx0VB
Y/aQgBmsOEFeoDai g+S8QasjeeY5LGSX7lc+S8dxORZaZpYzzMoL2jvcEVr1QDNH
gC9c/gA8 S18=
ns1.surfnet.nl.        7200        IN        RRSIG        AAAA 8 3 7200 20110701101814
20110624000007 30160 surfnet.nl. s50xJ2O5afxsapjyQuEaCpqefLUt5WhD
yX2siTiXqYWZwdp1ZCfTS8X+ 17MALuIP0/6QqJIMz/mmLZHdX7A2W2Uk78tairej
Mkyw8eCgvA4uTpEd xbLKaCGYs0g2hH8BIRgeFov2roYYYZv1AhtxF+3OuCNFBqGF
vVaCYmyQ Mtc=
ns2.surfnet.nl.        7200        IN        RRSIG        A 8 3 7200 20110630122659
```

```
20110623140007 30160 surfnet.nl. PQbbrQbyvzZe119vDiXyyVX/QMSQgNLy
8cDCEecPXKDt+g9f00xJdGLB uiXyvUQI/1dwaE2fwWs4jXHwTJk69zvWjAV9iInz
3tIdautHbbX9VOWP 0lB1bFxhVtpeTCiwgdxLaLS8nY3JBIeHxG1UFI/2LVDK4EhM
wQ+vPJM0 x7I=
ns2.surfnet.nl.        7200        IN        RRSIG        AAAA 8 3 7200 20110629233326
20110623040007 30160 surfnet.nl. tHQgBc3WPmQ+wVi7ih0s3OpxxWtRWCvP
OnwxzmoS3vYXEU8fgkN1Hzq0 I7OPz19zNIeedFEa048n13MlV9KyPtyy1A5hPoA0
8mz68/txr9OLSCTA +kvx0C8cZfAmkKK9lpbkb29x0QnR/RstjQGDvI8LF/eVMgnr
uJWZJpv/ XuU=

;; Query time: 199 msec
;; SERVER: 10.132.231.2#53(10.132.231.2)
;; WHEN: Fri Jun 24 16:18:08 2011
;; MSG SIZE  rcvd: 1319
```

E Probe

probe.py

```
#! /usr/bin/env python
# Import the needed libraries
from scapy.all import *
import sys

# Variable to check if the machine is probed
probed=0

# Probe the given machine, and check for ICMP packets
def icmp_monitor_callback(pkt):
    global probed
    while probed == 0:
        # dst = argument from commandline
        send(fragment(IP(dst=sys.argv[1])/UDP("X"*1600, dport=[53])))
        probed=probed+1

  add  # If there is a ICMP type 11, code 1 packet, write the IP,MAC to file
    if ICMP in pkt and pkt[ICMP].type == 11 and pkt[ICMP].code == 1:
        print pkt.sprintf("%IP.src% %Ether.src%")
        f = open('/tmp/ips_frag', 'w')
        f.write(pkt.sprintf("%IP.src% %Ether.src%")+'\n')
        f.close()

# Initiate the sniffer, only icmp packets are captured, but not stored
sniff(prn=icmp_monitor_callback, filter="icmp", store=0)
```

References

[1] Dnssec validation at surfnet. https://dnssec.surfnet.nl/?p=665. [Graph of December 20, 2010].

[2] Roland van Rijswijk. Mtu woes again... https://dnssec.surfnet.nl/?p=684, March 2011.

[3] Roland van Rijswijk. Mtu woes. https://dnssec.surfnet.nl/?p=641, November 2010.

[4] P.V. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966.

[5] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[6] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[7] P. Vixie. Extension Mechanisms for DNS (EDNS0). RFC 2671 (Proposed Standard), August 1999.

[8] D. Eastlake 3rd and C. Kaufman. Domain Name System Security Extensions. RFC 2065 (Proposed Standard), January 1997. Obsoleted by RFC 2535.

[9] D. Eastlake 3rd. Domain Name System Security Extensions. RFC 2535 (Proposed Standard), March 1999. Obsoleted by RFCs 4033, 4034, 4035, updated by RFCs 2931, 3007, 3008, 3090, 3226, 3445, 3597, 3655, 3658, 3755, 3757, 3845.

[10] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. RFC 4033 (Proposed Standard), March 2005. Updated by RFC 6014.

[11] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Resource Records for the DNS Security Extensions. RFC 4034 (Proposed Standard), March 2005. Updated by RFCs 4470, 6014.

[12] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Protocol Modifications for the DNS Security Extensions. RFC 4035 (Proposed Standard), March 2005. Updated by RFCs 4470, 6014.

[13] Wikipedia: Public-key cryptography. http://en.wikipedia.org/wiki/Public-key_cryptography. [Wikipedia page about Public-key cryptography, consulted at 06-23-2011].

[14] Wikipedia: Maximum transmission unit. http://en.wikipedia.org/wiki/Maximum_transmission_unit. [Wikipedia page about Maximum transmission unit, consulted at 06-23-2011].

[15] Wikipedia: Protocol data unit. http://en.wikipedia.org/wiki/Protocol_data_unit. [Wikipedia page about Protocol data unit, consulted at 06-23-2011].

[16] J. Postel. Internet Control Message Protocol. RFC 792 (Standard), September 1981. Updated by RFCs 950, 4884.

[17] D.D. Clark. IP datagram reassembly algorithms. RFC 815, July 1982.

[18] G. Ziemba, D. Reed, and P. Traina. Security Considerations for IP Fragment Filtering. RFC 1858 (Informational), October 1995. Updated by RFC 3128.

[19] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Standard), October 1989. Updated by RFCs 1349, 4379, 5884, 6093.

[20] The internet engineering task force. http://www.ietf.org/. [Their mission is to make the Internet work better.].

[21] Unbound. http://unbound.net/. [Unbound is a validating, recursive, and caching DNS resolver].

[22] Nlnet labs. http://www.nlnetlabs.nl/. [NLnet Labs is a research and development group].

[23] Netfilter; firewalling, nat, and packet mangling for linux. http://www.netfilter.org/. [Home to the software of packet filtering framework inside the Linux kernel].

[24] Pf. http://www.openbsd.org/faq/pf/. [OpenBSD FAQ for PF].

[25] Debian gnu/kfreebsd. http://www.debian.org/ports/kfreebsd-gnu/. [consulted at 06-27-2011].

[26] Ipfw. http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/firewalls-ipfw.html. [FreeBSD Handbook].

[27] Svn of eemo. https://svn.surfnet.nl/svn/eemo. [3-clause BSD-style licensed, Copyright SURFnet bv].

[28] tcpdump website. http://www.tcpdump.org/. [Consulted on 07-03-2011].

[29] Wireshark website. https://www.wireshark.org/. [Consulted on 07-03-2011].

[30] Red hatted trouble... ipv6 and bind 9.7 on rhel 5.x. https://dnssec.surfnet.nl/?p=464. [Consulted on 07-06-2011].