University of Amsterdam

System & Network Engineering

Preventing DNS Amplification Attacks using white- and greylisting

July 23, 2013

Author:
Ralph Dolmans ralph.dolmans@os3.nl

Abstract

The amplification factor caused by the Domain Name System (DNS) can be used to perform massive Distributed Denial-of-Service (DDoS) attacks. In an attempt to mitigate this problem, RFC5358 describes that a Recursive Resolving Name Server (RRNS) should provide its service solely to its intended users. These intended users can be listed on a local whitelist.

For an Authoritative Name Server (ANS) there are no standards to restrict access to exclusively its intended users, which would defy attacks from unintended users. The intended users of ANSs are RRNSs. However, DNS Amplification Attacks are typically targeted at worldwide services, such as webservers and ANSs. By creating a global whitelist that contains all RRNSs these attacks can be mitigated.

A global whitelist can be created by logging source addresses from requests that are received at an ANS. Custom ANS software is built that uses the behavior of CNAME handling to emulate handshakes and prove that requests are not spoofed. Software firewalls can be deployed to effectively limit the number of responses sent to non intended users.

This research shows that the proposed solution covers all the attack vectors. It also offers a practical implementation that seems feasible to execute, assuming enough RRNS servers can be collected.

Contents

Chapter 1

Introduction

Many actions on the Internet, such as visiting websites and sending emails, involve the translation from domain names to IP addresses. The Domain Name System (DNS)[10][11] is therefore a critical component of the Internet. Another use that DNS provides, is the ability to perform distributed denial-of-service (DDoS) attacks. The DDoS attack on Spamhaus (March 2013) is currently known as the biggest DDoS attack and was executed using DNS[14].

The DNS infrastructure consists of two elements. One element is the Recursive Resolving Name Server (RRNS). The RRNS is provided by Internet Service Providers (ISPs) to its customers and used as central point to query and cache Resource Records. The other element is the Authoritative Name Server (ANS). An ANS contains the original Resource Records.

The reason why DNS can be used to perform DDoS attacks is the combination of the used transport-layer protocol and the simplicity of the DNS protocol. The User Datagram Protocol (UDP)[12] is used for the transport of DNS requests and responses. Unlike the Transmission Control Protocol (TCP)[13], UDP does not use handshaking dialogues. The lack of handshaking dialogues creates the possibility to sent a UDP packet that contains a variable sized payload and a spoofed source address. When a DNS request is received by a DNS server it will reply with one response. Meaning, there is no detection of spoofed source addresses in the application layer either. This allows a DNS server to transfer data to a node that did not request for it.

The danger of sending unsolicited data using DNS depends on the effect of the amplification factor. A DNS response is typically bigger than a DNS request, hence an attacker can send significantly more data to the victim compared to original request. When all the downstream bandwidth of the victim's network is used to receive the DNS responses, the victim will not be capable to handle legitimate requests anymore.

According to the original DNS specification, a DNS response will fit in one UDP package and thereby is at most 512 bytes. Hence, when a 64 bytes request is sent the maximum amplification factor is 8:1. In 1999 the Extension mechanisms for DNS (EDNS0)[17] were introduced to exceed the maximum size of the response packet, hereby exceeding the maximum amplification factor as well. Nowadays DNS is also used to store cryptographic keys, for example for DNSSEC[3] and DKIM[2]. The relatively big size of these keys can be abused to create a large amplification factor. During the DNS Amplification Attack on

Spamhaus, requests were made to retrieve DNS Resource Records for ripe.net. The amplification factor of these requests was 100:1[15].

Chapter 2

Previous work

Previous research is done on methods to prevent DNS Amplification Attacks from occurring on RRNSs and ANSs.

2.1 Resolvers

Due to RFC5358[4], DNS Amplification Attacks that use a RRNS can be prevented by providing the DNS service to intended clients only. One of the suggested ways to do this, is by using IP address based authorization. The RRNS should only reply when the request is coming from a local (and authorized) IP. The network containing the RRNS should be protected against external address spoofing.

Restricting the access of a RRNS to local users is the advise in a report from the National Cyber Security Division, Department of Homeland security[6] too.

2.2 Authoritative

Preventing a DNS Amplification Attack on an ANS by confining access to local users is not a solution, since the service should be accessible to any user. Solutions to mitigate DNS Amplification Attacks on an ANS recommended in previous research can be divided into three categories: creation of stateful connections, BCP38 related solutions and ANS request thresholds.

2.2.1 Creation of stateful connections

When the state of the connection between client and server can be stored, the possibility to spoof the source address can be avoided. The state is generated during the handshaking dialogue. DNS servers should only handle requests that contain a state agreed upon by both parties, thereby introducing a simple challenge-response mechanism.

Stateful connections can be created in the transport-layer. One way to achieve this is by handling the DNS traffic over TCP instead of UDP. Another way to create stateful connections is by saving states in the application-layer. Guo et al. proposed a method to implement this idea by using cookies between the requester and a firewall in front of the ANS[8]. Kambourakis et al. proposed

a method requiring a validation on a RRNS whether a request was made after receiving a response[9]. TCPCT has been suggested to replace UDP for DNS, amongst others by Allen and William[1].

The disadvantages of storing states at a server is the establishment of new attack vectors. For example, it might be possible to create an unusual high amount of states which would result in an overload of the server. Hence, making it unavailable.

2.2.2 BCP38 related solutions

Another way to prevent source address spoofing is by excluding the possibility of an UDP packet containing a source address of a node that is not in the sending network segment. A method to accomplish this is by implementing BCP38[7].

The disadvantage of this method is that the responsibility of preventing source address spoofing is situated at edge networks. When a malicious ISP refuses to implement source address inspection, the possibility to perform DNS Amplification Attacks from within that network remains unaltered.

2.2.3 ANS request thresholds

The third category contains solutions in which the ANS decides whether the request should be answered or not.

Vixie proposed to do this, by using DNS Response Rate Limiting (DNS RRL)[16]. In DNS RRL the server keeps track of the amount of transmitted responses per subnet. When the amount of responses exceed a threshold, the server will stop sending replies to a specific IP or subnet. Donnerhacke introduced DNS Dampening[5]. DNS Dampening works with penalty points. Every DNS request will give the source address one point. When the amount of points exceed a threshold the server will drop all request containing that particular source address.

The disadvantage of using thresholds is the occurrence of false-positives. When many legitimate requests comes from one subnet, these requests might be dropped. On the other hand, a problem is the occurrence of false-negatives. When the attack is distributed over a number of different ANSs the number of requests of each ANS can remain under the specified threshold, meaning that the possibility to amplify requests can not be mitigated.

Chapter 3

Proposed solution

A solution without the mentioned disadvantages is needed. For a RRNS this solution lies in solely handling requests coming from intended users. The IPs of these intended users are specified on a local whitelist. This model is displayed in figure 3.1. RFC5358[4] withholds the practical use of this solution at an ANS, for the obvious reason that is will limit the access to a domain.

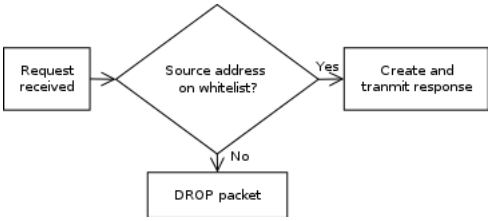

Figure 3.1: RRNS whitelisting model

When looking at the distinction of the ANS architecture an the RRNS architecture, we can deduct that the intended users for ANSs are RRNSs. When looking at victims of DNS Amplification Attacks the opposite is shown. Victims are typically servers whose services are intended to be used by anybody, for example a webserver or an ANS, and not a RRNS.

By using a global whitelist that contains all RRNSs, the impact of DNS Amplification Attacks can be mitigated. ANSs should respond to requests regularly, when the source address in the requests is on the global whitelist. The same model as with RRNSs can be used. This solution excludes the possibility to perform a DNS Amplification Attack on an ANS or a non-DNS server.

At times it might be useful to have direct access to an ANS without the intervention of a RRNS, for example when debugging. Therefore, requests coming from an address that is not on the whitelist should not instantly get dropped. Instead, the IPs that are not on the whitelist should be handled as greylisted. When an IP is considered to be on the greylist, the ANS would limit the number of responses. This model is displayed in figure 3.2. The limit would prevent the possibility to transfer data with such a big amount that it could overload a server.

The server side source address verification rules for the proposed solution

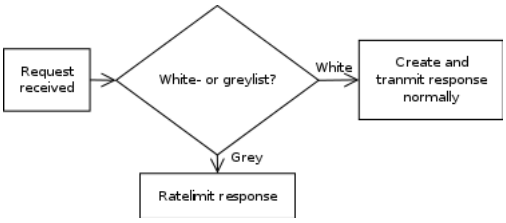

Figure 3.2: ANS white- and greylisting model

are:

RRNS-1 source address of DNS request packet on the whitelist → handle request normally;

RRNS-2 source address of DNS request packet not on the whitelist → drop request;

ANS-1 source address of DNS request packet on the whitelist → handle request normally;

ANS-2 source address of DNS request packet not on the whitelist → ratelimit response.

Chapter 4

Proposed solution effectiveness

To prove the effectiveness of the proposed solution it is necessary to create an overview of all the variations on a DNS Amplification Attack. The different attack scenarios can be divided into four categories, which contain two or four scenarios:

4.1 Amplification using a remote RRNS

Scenario 1 Attacker not located in network of victim; amplifier not located in network of victim and attacker; RRNS is used for amplification.

Scenario 2 Attacker not located in network of victim; amplifier located in network of victim; RRNS is used for amplification.

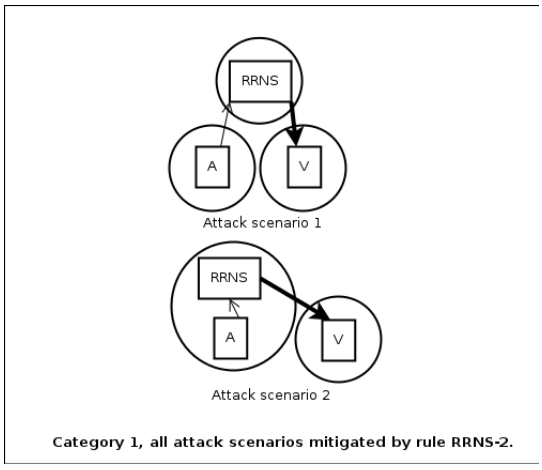

Figure 4.1: Amplification using a RRNS. RRNS is not located in victims network, victim can be any type of node.

4.2 Amplification using a local RRNS

Scenario 3 Attacker not located in network of victim; amplifier is located in network of attacker; RRNS is used for amplification.

Scenario 4 Amplifier, attacker and victim are located in the same network; RRNS is used for amplification.

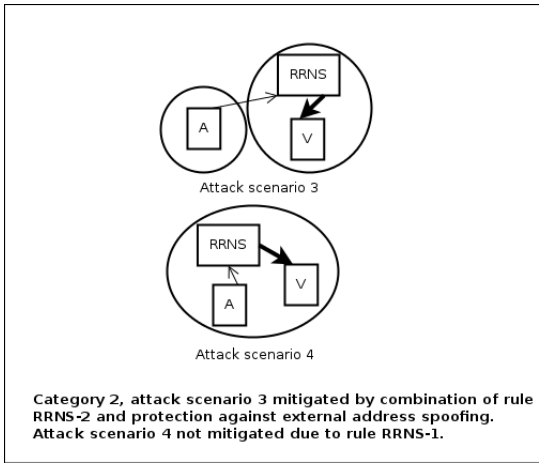

Figure 4.2: Amplification using a RRNS. RRNS is located in victims network, victim can be any type of node.

4.3 Amplification using an ANS to an ANS or non-DNS node

Scenario 5 Attacker not located in network of victim; amplifier not located in network of victim and attacker; ANS is used for amplification; victim is not a RRNS.

Scenario 6 Attacker not located in network of victim; amplifier located in network of victim; ANS is used for amplification; victim is not a RRNS.

Scenario 7 Attacker not located in network of victim; amplifier is located in network of attacker; ANS is used for amplification; victim is not a RRNS.

Scenario 8 Amplifier, attacker and victim are in the same network; ANS is used for amplification; victim is not a RRNS.

4.4 Amplification using an ANS to a RRNS

Scenario 9 Attacker not located in network of victim; amplifier not located in network of victim and attacker; ANS is used for amplification; victim is a RRNS.

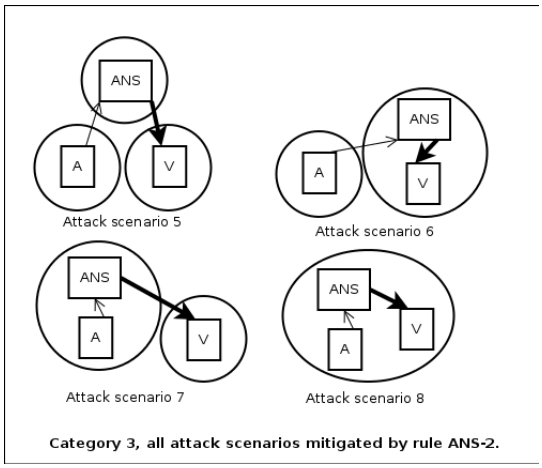

Figure 4.3: Amplification using a ANS. Victim can be an ANS or non-DNS node.

Scenario 10 Attacker not located in network of victim; amplifier located in network of victim; ANS is used for amplification; victim is a RRNS.

Scenario 11 Attacker not located in network of victim; amplifier is located in network of attacker; ANS is used for amplification; victim is a RRNS.

Scenario 12 Amplifier, attacker and victim are in the same network; ANS is used for amplification; victim is a RRNS.

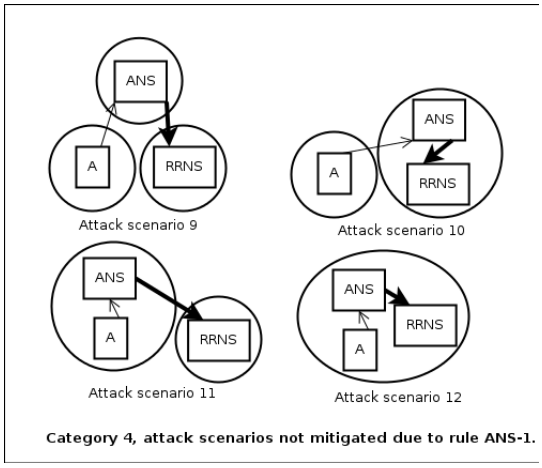

Figure 4.4: Amplification using a ANS. Victim is a RRNS.

4.5 Effectiveness per scenario

When a RRNS is configured to reply to requests coming from intended users only (rule RRNS-2), the attacks scenarios listed in the first category are prevented.

When the network containing the RRNS is protected against external address spoofing, attack scenario 3 from the second category is prevented as well.

The attack to a local server using a RRNS, as shown in attack scenario 4, will remain possible due to rule RRNS-1. This possibility is not a big issue, since the victim can solve this problem by contacting his ISP. When BCP38 is implemented at the ISP, this type of attack will only work when the attacker and victim are located in the same network segment.

By implementing the global whitelist validation, all attack scenarios from the third category are prevented (rule ANS-2). The victims in these scenarios are not on a global whitelist, therefore the responses sent to those victims are limited.

Attacking a RRNS by amplifying data using an ANS will still be possible in the suggested solution. This attack scenario is, however, not common. Hence, all relevant attack scenarios are mitigated by implementing the proposed solution.

Chapter 5

Practical approach

5.1 Local whitelist

RRNSs form the client-side part of the DNS and are therefore located at the network of an ISP or at a local network. At these locations it is known which IPs are used inside the network. Therefore, creating a local whitelist is an easy task.

5.2 Global whitelist

The need of a complete and up-to-date global whitelist demands an automated method to collect as many RRNSs as possible. Scanning the complete IP space as done by the Open Resolver Project[1] is not possible, because RRNSs using a local whitelist should not reply to request coming from the scanning node. Therefore, the action to whitelist a RRNS should be triggered from within the RRNS's network.

To make the whitelist as complete as possible, it should be possible for all the users of the network to participate in the collection process. Users should preferably be able to participate without the need to install software or without executing difficult tasks.

One way to use crowd sourcing to collect RRNS addresses, is by logging source addresses for DNS requests that are received at an ANS. Participants only have to resolve the IP of a domain in that case. The ANS for that domain should be configured to log all requests. There is, however, by default no way to verify that the source addresses in the requests are not spoofed.

Source address spoofing can be prevented by using handshaking dialogues. A handshaking dialogue can be implemented by generating an unique validation token for every received request at the ANS. The generated validation token will be included in the response that is sent to the RRNS. The RRNS should now send the validation token back to the ANS, thereby verifying that the token is received at a RRNS. It is clear that the original request does not contain a spoofed source address when the validation token sent by the RRNS equals the original generated token at the ANS.

[1] `http://openresolverproject.org/`

It is possible to have the RRNS automatically re-send the validation token to the ANS. Section 3.6.2 (Aliases and canonical names) in RFC1034[10] states:

> CNAME RRs cause special action in DNS software. When a name server fails to find a desired RR in the resource set associated with the domain name, it checks to see if the resource set consists of a CNAME record with a matching class. If so, the name server includes the CNAME record in the response and restarts the query at the domain name specified in the data field of the CNAME record. The one exception to this rule is that queries which match the CNAME type are not restarted.

This behavior can be used to create the handshaking dialogue. The ANS used for the whitelist generation should always respond with a CNAME containing the validation token. The RRNS will now resent the validation token in the attempt to resolve the CNAME. The steps for executing a handshaking dialogue over the DNS protocol are:

1. Client sent request for A record to RRNS
2. RRNS sent request for A record to ANS
3. ANS generates validation token and stores this token, together with the source address of the request, in a database
4. ANS replies to RRNS with a CNAME resource record. The validation token will be included in the data field of the CNAME record. The validation token will be succeeded with the domain name of the ANS.
5. The RRNS received a CNAME while requesting an A record, therefore a new request will be executed. A request to resolve the CNAME containing the validation token is sent to the ANS.
6. The ANS can mark the source address corresponding with the validation token as validated. Finally the requested A record can be returned.

These steps are displayed in the sequence diagram in figure 5.1.

By exporting all source addresses from the database that are marked as validated, a global whitelist is generated.

An attacker can add a server to the whitelist if he is able to retrieve the validation token transmitted to that server. This can be done by having access to the system or by successfully performing a Man-in-the-Middle attack. Another way of retrieving the right validation token is by using brute forcing techniques, i.e. trying all possibilities. The risk of brute force attacks can be mitigated by limiting the time window in which a validation token is valid.

For this research, the custom ANS software is developed using python and the Twisted[2] library. The validation tokens and whitelist entries are stored in a MySQL database.

[2] `http://twistedmatrix.com/`

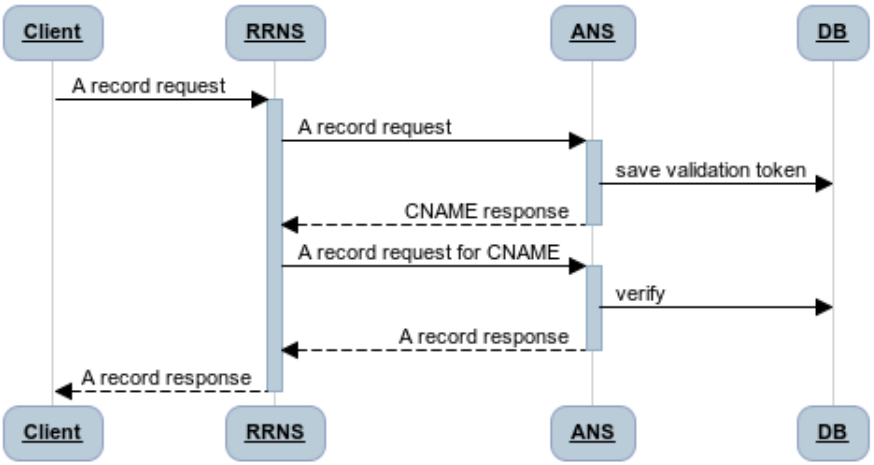

Figure 5.1: Handshaking dialogue using CNAME resource records

Chapter 6

Practical approach feasibility

6.1 Local whitelist

All modern RRNS software offers the possibility to limit requests using a local whitelist. Enabling the use of local whitelists therefore only requires a change in the RRNS configuration.

6.2 Global whitelist

ANS software does not provide the possibility to rate limit greylisted IPs. The access control solutions that are available are designed for the use of a local whitelist. It is not suitable for the high amount of entries that are available in the global whitelist.

One location to include this additional software is in the code of the ANS software. Unfortunately this will introduce some drawbacks. This solution requires that the whitelist check has to be included in all available ANS software. It will take a long time to finis, while there is a demand for a solution as soon as possible.

Another method to perform global whitelist validation is by using firewall software. Firewalls are specialized in dropping packets that meet a specified rule. When using a firewall, a portable solution for multiple platforms is available.

To ensure a high adoption of global whitelist validation, the impact on the performance of the DNS service caused by the validation should be limited. Two benchmarks were executed to measure the impact.

Two identical servers (Dell PowerEdge R210, Intel Xeon L3426, 8GB RAM) were used for the benchmarks. The two servers were connected with each other using an Ethernet crossover cable. On the first server the DNS software and firewall were installed. BIND[1] was used as DNS software, iptables[2] as firewall. Ipset[3], an iptables extension, was used to store the whitelist entries in a way

[1] `http://www.isc.org/downloads/bind/`
[2] `http://www.netfilter.org/projects/iptables/index.html`
[3] `http://ipset.netfilter.org/`

that can be used in an iptables rule.

The second server was deployed as monitor for the measurements. Dnsperf[4] was used to send DNS requests and to measure the latency between sending the request and receiving a response. The DNS requests were coming from the test file[5] offered by Nominum.

6.2.1 Latency benchmark

The goal of the first benchmark was to see the difference in latency when using a large amount of entries in ipset. The benchmark was executed with 21 different ipset lists. The number of entries in these lists range from 0 to 1 million IPs, with intervals of 50 thousand.

The script used to create pseudo-random IPs from the whitelists is listed in Appendix A. The script used to create the ipset lists is listed in Appendix B. The iptable rules used during the benchmark on the DNS server are:

```
iptables -A INPUT -m set --match-set <ipsetname> src -j DROP
iptables -A INPUT -j ACCEPT
```

When the source IP matches the whitelist the lookup will stop, resulting in a lower latency. To prevents this it was made sure that the IP of the monitor system was not whitelisted. The command used to execute the benchmarks is:

```
dnsperf -d queryfile-10million -s 192.168.1.50 -Q 200000
```

The Q parameter limits the maximum number of DNS requests send per second. By limiting at 200 thousand requests we can make sure that the CPU of the monitor server is not completely used, which could alter the results.

For all performed benchmarks, using the 21 different ipsets, the average latency over one million requests is around 0.13 milliseconds. From these results, the conclusion can be drawn that the latency for DNS lookups does not increase when a whitelist containing one million IPs is used. The benchmark results are displayed in the chart in figure 6.1. Appendix C provides a complete overview of the results.

6.2.2 CPU load benchmark

The goal of the second benchmark is to measure difference in CPU load on the DNS server when it is configured to limit access to global whitelisted IPs. The benchmark consists of three test-scenarios:

No iptables Measurements of the DNS server's CPU load when no global whitelist validation is performed.

Whitelisted Measurements of the DNS server's CPU load which is configured to perform global whitelist validation. In this scenario, the IP of the monitor server is on the whitelist.

[4] http://www.nominum.com/support/measurement-tools/

[5] ftp://ftp.nominum.com/pub/nominum/dnsperf/data/queryfile-example-10million-201202.gz

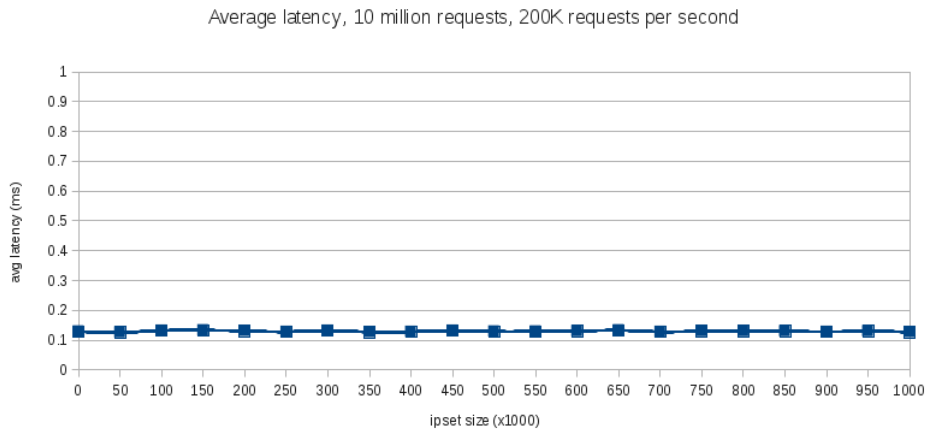

Figure 6.1: Latency when using ipset

Greylisted Measurements of the DNS server's CPU load which is configured to perform global whitelist validation. In this scenario, the IP of the monitor server is not on the whitelist. The firewall should therefore rate limit request coming from the monitor server.

For the last two scenarios a whitelist containing one million IPs was used.

The CPU utilization on the DNS server is the average of four measurements, each running for 15 seconds. The tool used to measure the CPU load is *sar*. The executed command is:

```
sar -u 15 4
```

The iptables rules used in this benchmark that accept packets coming from whitelisted IPs and ratelimit packets coming from greylisted IPs are:

```
iptables -A INPUT -p udp --dport 53 -m set --match-set whitelist
  src -j ACCEPT
iptables -A INPUT -p udp --dport 53 -m recent --rcheck --seconds
  10 --hitcount 2 --name GREYLIST -j DROP
iptables -A INPUT -p udp --dport 53 -m recent --set --name
  GREYLIST -j ACCEPT
```

The dnsperf command was executed nine times for each scenario. During the nine measurements, the amount of requests send per second was increased. The number of requests send per second ranged from 0 to 200 thousand and increased with an interval of 25 thousand at each measurement. The executed dnsperf command is:

```
dnsperf -d queryfile-10million -s 192.168.1.50 -Q <requests_per_second>
```

This benchmark showed that the CPU utilization of the DNS server does not increase when iptables is configured to perform global whitelist validation. Rate limiting requests did not consume enough CPU load to be visible in the results. Because BIND does not have to handle limited requests, the CPU load is a lot less when using rate limiting. These results are displayed in the chart in figure 6.2. See Appendix D for the complete benchmark results.

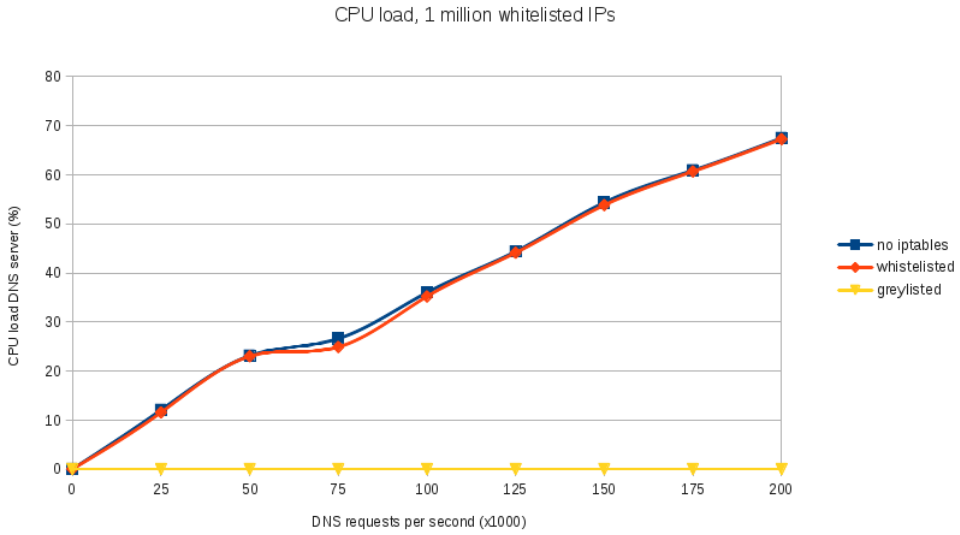

Figure 6.2: CPU load when using ipset

6.2.3 Whitelist manager

To simplify the implementation of global whitelist validation, the global whitelist management tool is developed. This tool downloads the most recent list of RRNS IPs and validates its integrity using md5 hashes. After downloading the list an ipset is created and all downloaded IPs are added to the list. Finally the tools add the right rules to iptables. The tool is provided under the MIT license and is listed in Appendix E.

Chapter 7

Considerations and future work

The feasibility of the practical approach depends on whether or not enough RRNS servers can be collected by our custom ANS software. After all RRNSs are collected by means of crowd sourcing, the global whitelist validation can be applied without negative impact.

For this research benchmarks were performed to measure the impact of validating global whitelists using iptables. Iptables is included in the Linux kernel and therefore not portable to different operation systems. Future research can be done for performance benchmarks on firewall software that is used by different operating systems. Widely used firewall software for BSD based operating systems is *PF*[1]. We do not see a reason to suspect worse performance when using different firewall software.

[1] `http://www.openbsd.org/faq/pf/`

Chapter 8

Conclusion

This research proposes to mitigate DNS Amplification Attacks by limiting DNS services to intended users. Intended users for RRNSs are local users, intended users for ANSs are RRNSs.

Whitelists are needed to specify the intended users. Creating a whitelist for a RRNS is easy, since the local users are known by the network administrator. This paper proposes a method to generate a whitelist containing all global RRNS servers to be used by an ANS.

Applying a global whitelist can be done using standard firewall software. This research shows that there is no noticeable performance impact when 'iptables' is used for global whitelist validation. It is suggests that greylisting is applied to ANS servers to allow for debugging.

To prove that the proposed solution mitigates DNS Amplification Attacks, all relevant attack scenarios are evaluated. This research also proves that the proposed solution is practically feasible, assuming enough RRNS servers can be collected.

Bibliography

[1] William Allen. improving tcp security with robust cookies.

[2] E Allman, J Callas, M Delany, M Libbey, J Fenton, and M Thomas. Domainkeys identified mail (dkim) signatures. Technical report, RFC 4871, May, 2007.

[3] Roy Arends, Rob Austein, Matt Larson, Dan Massey, and Scott Rose. Dns security introduction and requirements. Technical report, RFC 4033, March, 2005.

[4] J Damas and F Neves. Preventing use of recursive nameservers in reflector attacks. Technical report, BCP 140, RFC 5358, October, 2008.

[5] L Donnerhacke. Dns dampening, september 2012.

[6] Homeland Security Federal Network Security. Domain name system (dns) security reference architecture. Technical report, 2011.

[7] Paul Ferguson. Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing. 2000.

[8] Fanglu Guo, Jiawu Chen, and Tzi-cker Chiueh. Spoof detection for preventing dos attacks against dns servers. In *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*, pages 37–37. IEEE, 2006.

[9] Georgios Kambourakis, Tassos Moschos, Dimitris Geneiatakis, and Stefanos Gritzalis. A fair solution to dns amplification attacks. In *Digital Forensics and Incident Analysis, 2007. WDFIA 2007. Second International Workshop on*, pages 38–47. IEEE, 2007.

[10] Paul Mockapetris. Rfc 1034: Domain names-concepts and facilities. 1987.

[11] Paul Mockapetris. Rfc 1035: Domain names: implementation and specification. 1987.

[12] Jon Postel. Rfc 768: User datagram protocol. 1980.

[13] Jon Postel. Rfc 793: Transmission control protocol. 1981.

[14] Matthew Prince. The ddos that almost broke the internet. `http://blog.cloudflare.com/the-ddos-that-almost-broke-the-internet`, March 2013.

[15] Matthew Prince. The ddos that knocked spamhaus offline (and how we mitigated it). http://blog.cloudflare.com/the-ddos-that-knocked-spamhaus-offline-and-ho, March 2013.

[16] P Vixie. Dns response rate limiting (dns rrl), april 2012.

[17] Paul Vixie. Extension mechanisms for dns (edns0). 1999.

Appendix A

Pseudo-random IP list generation script

```
1  import random
2
3  ips= dict()
4  for i in range(13000000):
5    ips['.'.join(str(random.randint(1,255)) for x in range
         (4))] = 1
6
7  print "IPs generated, time to write"
8
9  ipfile = open('ipfile_10million', 'w')
10 for ip in ips.keys():
11   ipfile.write("%s\n" % ip)
12
13 print "%d unique IPs added to file" % len(ips.keys())
```

Appendix B

Ipset whitelist generation script

```
1  #!/bin/bash
2  for i in {0..1000000..50000}; do
3    echo `date`
4    echo "ipset create whitelist$i hash:ip maxelem $i"
5    `ipset destroy whitelist$i`
6    `ipset create whitelist$i hash:ip maxelem $i`
7    ips=`head -n $i ipfile_10million`;
8    for ip in $ips; do
9      `ipset add whitelist$i $ip`
10   done
11 done
```

Appendix C

Latency benchmark results

# of entries in ipset	Average latency over one million requests (in ms)
0	0.129
50,000	0.126
100,000	0.132
150,000	0.133
200,000	0.131
250,000	0.128
300,000	0.132
350,000	0.127
400,000	0.128
450,000	0.132
500,000	0.129
550,000	0.129
600,000	0.13
650,000	0.133
700,000	0.128
750,000	0.13
800,000	0.13
850,000	0.13
900,000	0.128
950,000	0.13
1,000,000	0.127

Appendix D

CPU utilization benchmark results

Requests per second	CPU load no iptables	CPU load requester whitelisted	CPU load requester greylisted
0	0.06%	0.06%	0.06%
25,000	12.11%	11.59%	0.06%
50,000	23.14%	23.02%	0.06%
75,000	26.63%	24.92%	0.06%
100,000	36.03%	35.22%	0.06%
125,000	44.37%	44.12%	0.06%
150,000	54.36%	53.83%	0.06%
175,000	60.9%	60.68%	0.06%
200,000	67.49%	67.28%	0.06%

Appendix E

Global whitelist management tool

```
1  #!/usr/bin/python
2  import commands
3  import subprocess
4  import urllib
5  import tarfile
6  import time
7  import os
8  import hashlib
9  import shutil
10
11 def check_requirements():
12   ipset_status = commands.getstatusoutput("hash ipset")
13   iptables_status = commands.getstatusoutput("hash
         iptables")
14   if ipset_status[0] != 0 or iptables_status[0] != 0:
15     raise Exception("Iptables and/or ipset not found,
           please intstall these dependencies first")
16
17 def md5_for_file(f):
18   md5 = hashlib.md5()
19   for ip in f:
20     md5.update(ip)
21   return md5.hexdigest()
22
23 def retrieve_whitelist():
24   whitelist = urllib.urlopen("http://reliablenameservers.
         org/whitelists/latest_whitelist")
25   temp_path = "/tmp/whitelist/%d" % int(time.time())
26   temp_name = os.path.join(temp_path, "latest_whitelist")
27   os.makedirs(temp_path)
28   temp = open(temp_name, "wr")
29   temp.write(whitelist.read())
```

```
30    temp.flush()
31    temp.close()
32
33    temp = open(temp_name)
34    tar = tarfile.open(mode="r:gz", fileobj=temp)
35    tar.extractall(temp_path)
36
37    latest_whitelist = open(os.path.join(temp_path, "
          whitelist"), "r")
38
39    if md5_for_file(latest_whitelist) != open(os.path.join(
          temp_path, "whitelist.md5")).read():
40      raise Exception("Download seems to be corrupt, please
            run again.")
41
42    return latest_whitelist
43
44  def fill_ipset(latest_whitelist):
45    subprocess.call(["ipset", "create", "whitelist", "hash:
          ip", "maxelem", "1000000"])
46    latest_whitelist.seek(0)
47    for ip in latest_whitelist:
48      subprocess.call(["ipset", "add", "whitelist", ip])
49
50  def add_iptables_rules():
51    subprocess.call("iptables -A INPUT -p udp --dport 53 -m
          set --match-set whitelist src -j ACCEPT", shell=
          True)
52    subprocess.call("iptables -A INPUT -p udp --dport 53 -m
          recent --rcheck --seconds 10 --hitcount 2 --name
          GREYLIST -j DROP", shell=True)
53    subprocess.call("iptables -A INPUT -p udp --dport 53 -m
          recent --set --name GREYLIST -j ACCEPT", shell=True
          )
54
55  def cleanup():
56    shutil.rmtree(temp_path)
57
58  if __name__ == '__main__':
59    print "Reliable Nameserver installer"
60    check_requirements()
61    print "Download latest whitelist",
62    latest_whitelist = retrieve_whitelist()
63    print " - Done"
64    print "Add whitelisted IPs to ipset",
65    fill_ipset(latest_whitelist)
66    print " - Done"
67    iptables = ""
68    while iptables.lower() not in ["y", "n"]:
```

```
69      iptables = raw_input("Let installer add iptable rules
            ? (y/n): ")
70    if iptables.lower() == "y":
71      print "Add iptables rules",
72      add_iptables_rules()
73      print " - Done"
```