University of Amsterdam
System & Network Engineering

Naxsi performance measurement

April 1, 2013

Authors:
Dennis Pellikaan dennis.pellikaan@os3.nl
Lutz Engels lutz.engels@os3.nl

Abstract

The goal of the research described in this paper is to find out what the performance impact is when Naxsi is used as a web application firewall for the Nginx web server. The performance is measured for two different scenarios. First, by making the Nginx web server returning only `HTTP 200 OK` responses, the Naxsi performance impact becomes isolated. Secondly, a more realistic scenario is looked at by measuring the performance impact when Naxsi is protecting a Wordpress website. The overall performance impact of Naxsi is minimal. However, when analysing the results, it becomes clear that the performance impact of Naxsi becomes greater when the number of URL parameters increase. Also, when thousands of requests per second need to be handled by the Nginx web server, Naxsi shows a more noticeable impact on the web server.

Contents

1 Introduction

More and more people will have access to the Internet [1] and they will have access to many web applications. Commercial businesses have access to an ever growing market, but also criminals become more intelligent on how to reach end-users and abuse the web services users are accessing on a daily basis. According to the web survey held by Netcraft in February 2013 [2], the majority of web servers is running Apache and is followed by Microsoft. However, Nginx is gaining more popularity and it is not only used as a standard web server, but also as a reverse proxy for load balancing purposes. Naxsi (*Nginx Anti Xss & Sql Injection*) is developed as a response to common attacks that often occur on the Internet. Naxsi is a firewall designed to work with Nginx. It works as a DROP-by-default firewall, and rules should be added to ACCEPT certain traffic. This concept is also known as whitelisting. This paper focusses on the performance impact when using Naxsi to protect a web application, which leads to the following research question:

How does Naxsi influence the performance of the Nginx web server?

To further answer this question, the performance measurement is looked at from two perspectives, which lead to the following two sub-questions.

- What is the performance of the Nginx firewall when Naxsi is isolated?
- What is the performance of the Nginx firewall in a real-life scenario?

2 Naxsi

Naxsi is written to be a fast, light and scalable web application firewall (WAF) for the Nginx web server. Naxsi stands for *Nginx Anti Xss & Sql Injection* and it has a positive approach for web traffic inspection by using a whitelisting method. This means that traffic is blocked by default, and "good" traffic must explicitly be allowed. Naxsi uses two different files, which contain the rules. First, at the server level configuration. Second, at the HTTP location level configuration. The first one is called the core rules, and it contains regular expressions for most of the characters that are usually involved in an attack. Upon match, it will increase the score of the request. The location level configuration has site specific rules and, thus allows for multiple virtual hosts for having different whitelists and thresholds for the score of a request. Naxsi will deny the request, when the threshold is exceeded. The core level configuration can be referred to as blacklist. Furthermore, it is more or less a fixed list and, according to the Naxsi website [8], it is not expected to evolve rapidly. On the other hand, the location level configuration is a site specific configuration, and thus needs to be created. Creating the rules is done by putting Naxsi in learning mode. When

[1] `http://data.worldbank.org/indicator/IT.NET.USER.P2/countries?display=graph`

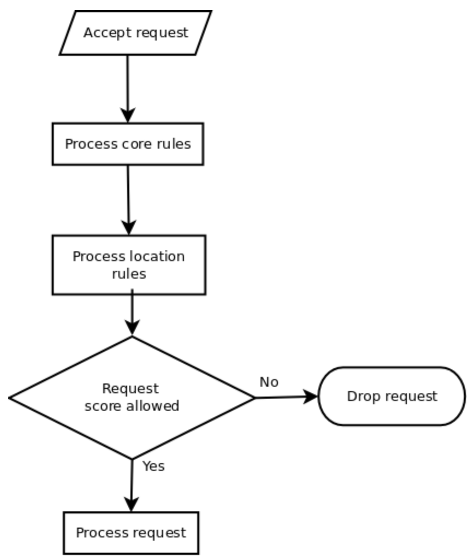
[2] `http://news.netcraft.com/archives/category/web-server-survey/`, February 2013

Naxsi is in learning mode, requests are not blocked, but are rather seen as valid traffic and used for creating the whitelist rule set.

2.1 Request flow

When Naxsi is in production mode, it actively gives each request a score. Depending on the location level configuration rule set, the request may be allowed or dropped. Figure 1 shows how logically each request is processed by Naxsi. First, the request is checked for "dangerous" symbols and SQL keywords. Second, the request is checked by the location level rules. Location level rules may overrule the core rules. Lastly, the request score is checked against the rule set. Depending on the score, the request is either blocked, which means that the request gets forwarded to the *DeniedURL*, or the request is further processed. The *DeniedUrl* is set in the location level configuration whitelist file. This allows the administrator to specify what should be returned to the client when a bad request is sent.

Figure 1: Naxsi request flow



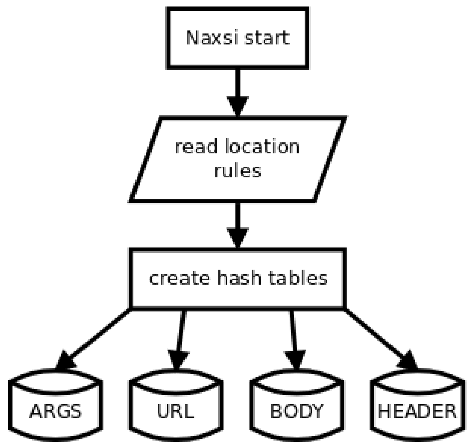
2.2 Whitelist processing

As already mentioned, Naxsi does have a short blacklist. However, the usage of whitelists is more extensive, as each location level configuration contains a separate set per virtual host. At startup Naxsi generates up to four zone hash tables from the location level whitelist rules in memory, as can be seen in figure 2. The respective zones are:

- ARGS (GET arguments)

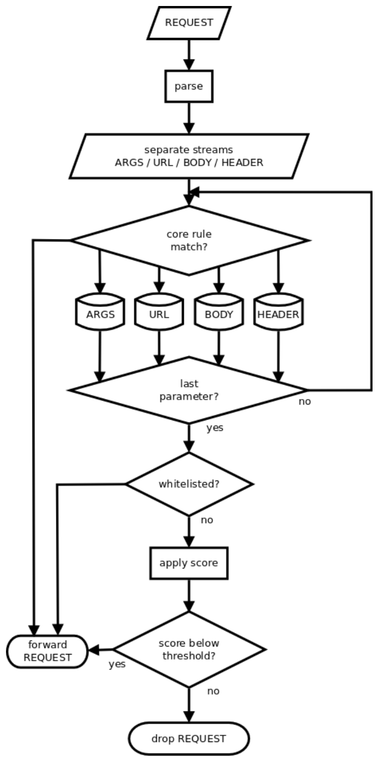
- URL (the full URI)
- BODY (POST arguments)
- HEADER (HTTP headers)

Figure 2: Creating hash tables from location level rules



As visualized in figure 3, once an HTTP request is accepted, it is parsed and split up into four streams corresponding to the zones: ARGS, URL, BODY, HEADERS. Each stream, that may consist of more parameters, is iterated independently, but in the same manner. Each parameter is checked for suspicious patterns, by consulting the core rules. Furthermore, a lookup in the respective zone hash table is done, as to determine whether this specific match is whitelisted. If it is not whitelisted, the score that is attached to the matching core rule is applied. When all streams are processed, then eventually it is determined if the total score for SQL, RFI, TRAVERSAL and XSS is below the threshold. If it is, the request is forwarded. If not, it is denied.

Figure 3: Naxsi whitelist processing



This approach shows that matching is done very efficient. Creating hash tables from the whitelists in memory, accelerates the process of lookups. Furthermore, through splitting up the REQUEST data in the same manner as the hash tables, the processing of parameters is handled separately. This allows for lookups to be only done, when there is a need for it. If, for example, a request contains a few URL parameters, but only one of them is 'suspicious', each parameter is matched against the (short list of) core rules, but only the suspicious one is further analysed.

3 Methods

This section discusses the methods and tools that are used to measure the performance of Naxsi. Before both the approach and the methods are discussed in more depth, it is important to understand how the basic configuration looks like, which is discussed in the next section. Next, the tools that are used for measuring the performance and measuring the system resources, are briefly described. Lastly, the methods of the performance measurements are discussed.

3.1 Experimental setup

A configuration for standard web hosting that is seen quite often, is to separate the web hosting services on a functional basis [5]. First, there is the web server, which is the front-end from a client's perspective. Second, there is the application layer. The application layer takes care of most of the processing power that is needed to process all the application logic. Third, there is the data layer, which is often a database server. It is not necessary to allocate these layers over different servers. Depending on the requirements, it is possible to host all services on one server. However, in order to do a performance measurement on one of these services, it is important that not all services run on the same server. Therefore, in this setup, every layer is taken care of by a dedicated server.

Figure 4 shows the setup that is used for the experiments. Table 1 gives a short description of each server and the service(s) that run(s) on it. Server01 is the front-end server, but it will also act as a software router for basic communication with the servers behind it. Server02 processes all the application data, which for a large part consists of the processing of PHP code. Server04 is used to execute the performance measurements. The hardware specifications and the software that is used, can be found in appendix B.

Figure 4: Experimental setup

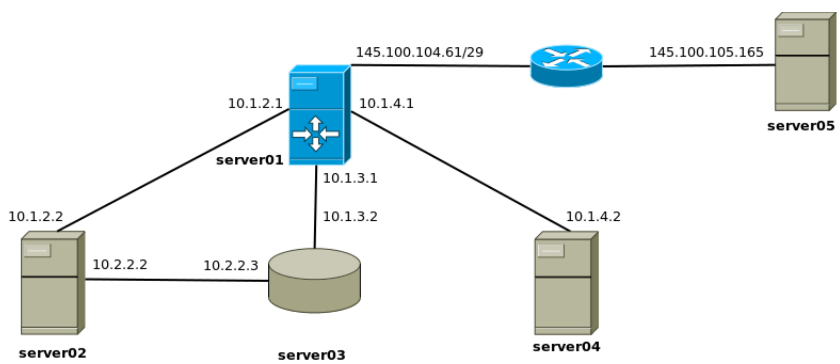


Table 1: Experimental infrstructure

Hostname	Service	Short description
server01	Nginx + Naxsi	Front-end server and router
server02	Nginx + Fastcgi	Application layer
server03	MySQL	Data layer
server04	Benchmark tools	Performance measurements
server05	Collectd	Resource usage collector

3.2 Performance measurement tools

A variety of performance measurement tools (also called benchmarking tools) exist. However, not all of them are suited to perform tests in this specific experimental setup. This section discusses a selection of the available tools.

3.2.1 Apache benchmark

Apache benchmark is an open source web server benchmarking tool developed by Apache initially to test Apache web server installations. [1] However, it is not limited to Apache web servers, as it simulates regular HTTP/1.1 requests. During the execution of the performance tests, *Apache benchmark* is used when the number requests per second exceed the number of concurrent connections that are realistically possible. To be able to stepwise increase the number of URL parameters, as well as to calculate the average of repeated tests, a perl script is written to automate these operations [3]. These experiments are explained later in this section.

3.2.2 Httperf/Autobench

Httperf is a web server performance measurement tool developed by David Mosberger [7] at Hewlet-Packard Research Labs [6]. It supports both HTTP/1.1 and SSL and aims to be robust enough to generate server overload. *Autobench* [2] is a Perl script that wraps around *Httperf*. It aims at automating the benchmarking process and offers extensive options, amongst which the ability to stepwise increase concurrent connections or the number of requests per second.

In the course of this project, *Autobench* is used to confirm and visualize the threshold of the number of concurrent connections that Naxsi can handle (as described in section 4.1.1). The automation of incrementing the URL parameters is done by wrapping a script [4] around Autobench.

3.2.3 Collectd

Collectd [3] is a daemon for unix-based operating systems that gathers performance statistics. It has a modular design, meaning that the collection of different kind of statistics (e.g. cpu-load or network-usage) is enabled or disabled by toggling the respective plugins and thereby minimizing resource usage. Furthermore, it only handles the data collection, leaving out the logic to create graphs, but storing the data in round-robin Database (RRD) files. Programs like RRDtool in turn can easily create graphs from the RRD files. It is written in the fast and low impact programming language C [9] and it is designed to be run on e.g. embedded devices.

Collectd is configured on all servers of the experimental setup. *Server01* to *server04* are configured as clients, collecting their own local performance

[3] https://github.com/lutzengels/naxsiperftest/blob/master/tools/bench.pl

[4] https://github.com/lutzengels/naxsiperftest/blob/master/tools/measure_naxsi_with_param_increments.pl

data. *server05* also collects local performance data, but moreover receives the performance data from the clients. Eventually the *Collection 3* [4] web-based front-end uses RRDtool to graph the collected data. Details can be found in appendix B.2.

3.3 Performance measurements

The performance measurements are done in two main phases. The first phase is to measure the baseline performance of the Nginx web server without Naxsi compiled into the Nginx server. These measurements are needed in the second phase when Naxsi is compiled into the Nginx server and when it is enabled. Details of how the Nginx server is compiled with and without Naxsi can be found in appendix B.2.2. In both phases, the performance is once measured with a Wordpress website on a back-end server, and a second time with the Nginx server returning an `HTTP 200 OK` reponse for every request. A Wordpress [10] website is introduced in the measurement to see the effects of a more realistic scenario. Based on the content management system (CMS) popularity measurements of March 1st by w3techs [5], Wordpress is the most populair CMS on the Internet. According to their survey, 17.4 % of all web servers run Wordpress and it holds 54.6 % of the CMS market share. Wordpress has, by far, the greatest market share and is therefore a popular target for cyber criminals. Naxsi's goal of course, is to block these cyber attacks.

Naxsi's design aims to process whitelist rules in an efficient manner. As explained in section 2.2 the performance impact of the number of rules should therefore be minimal. However, by looking at the number of uniform resource locator (URL) parameters that need to be parsed by Naxsi, a performance decrease is expected. Naxsi inspects each individual URL parameter that is concatenated to a URL. Based on the content of each parameter, Naxsi decides what to do next. The performance impact of the number of URL parameters is measured from zero to twenty parameters. This measurement is done in two steps. First, the number of URL parameters is incremented with valid content that is allowed by Naxsi. Second, the number of parameters is incremented with valid content, except for the last one, which should result in a *RequestDenied*. This way, it is also possible to measure the performance when Naxsi has to handle bad requests.

3.3.1 Wordpress

For the performance measurements that integrate a Wordpress website on the back-end server, httperf has proven to be a reliable tool. By measuring the response time of each request and by monitoring the resources on the front-end server, it shows the impact of a real life scenario (namely hosting of a Wordpress website). This is repeated for both the baseline measurement, as well as for when Naxsi is actively protecting the website with a reasonable set of whitelist rule [6].

[5] `http://w3techs.com/technologies/overview/content_management/all`
[6] `http://imil.net/wp/2012/12/30/wordpress-3-5-and-naxsi/`

3.3.2 HTTP 200 OK

In performance measurements where Nginx replies with `HTTP 200 OK` messages, httperf is not suitable anymore. Nginx handles request at such a fast speed, that the number of requests per second exceed the number of concurrent connections that are realistically possible. Therefore, the Apache benchmark tools are used. These tools, however, also come with a drawback. The number of requests per second are far greater than in the case of the hosted Wordpress website. Because of the high number of concurrent connections, Apache benchmark shows some inconsistencies that need to be taken into consideration. Each individual Apache benchmark command is repeated 5 times and for a maximum of 60 seconds for each command. A set of 5 commands is called a step. Of each step, the lowest, highest and average values are graphed. Each connection is used for only one request, thus the number of connections is equal to the number of possible requests. The number of concurrent connections is incremented with 10 after each step, starting with 1 and ending with 1,000 concurrent connections.

3.3.3 URL parameters

As explained in the beginning of this section, Naxsi processes each URL parameter individually. By incrementing the number of URL parameters that Naxsi has to parse, the performance is expected to decrease linearly. First, the URL parameters are only incremented with valid parameters, as shown in figure 5. This means, that Naxsi allows this traffic to be passed through.

Figure 5: Valid URL parameters

```
http://www.example.com/
http://www.example.com/?foo1=bar1
http://www.example.com/?foo1=bar1&foo2=bar2
...
```

Next, the URL parameters are incremented with valid content, but only the last one has invalid content, as shown in figure 6. Naxsi processes all the requests until it reaches the last one where it bails out. The last parameter tries to use path traversal, which is not allowed by Naxsi. When Naxsi does not allow a request, it returns an `HTTP 403` error code. By only returning an HTTP error code, there is no overhead of processing an error page. The configuration details can be found in appendix B.2.3.

Figure 6: Invalid URL parameters

```
http://www.example.com/
http://www.example.com/?../
http://www.example.com/?foo1=bar1&../
...
```

Also this time, for Wordpress the measurement are performed with *httperf*, which gives consistent response values. Because of the higher number of request per seconds, Apache benchmark is used for the `HTTP 200 OK` performance measurements.

4 Experiments

Basically, there are two main different scenarios for measuring the performance. The first one is by measuring the performance of Nginx when Naxsi is disabled. The second one is by measuring the performance when Naxsi is enabled. First, a baseline measurement is performed with a Wordpress website on the back-end server. Second, the performance baseline is measured where Nginx only returns a `HTTP 200 OK` response. The latter baseline measurement gives the lowest overhead and maximum performance of the Nginx webserver.

4.1 Baseline performance measurements

4.1.1 Wordpress

In order to measure the performance of Naxsi, it is important to know the bottleneck of the back-end server processing all the application logic. Table 2 shows that ≈ 21 % of the 10,000 requests produce a `HTTP 5xx` response when using 250 concurrent connections. Details can be found in appendix A.1.1.

Table 2: httperf measurement 1

	1xx	**2xx**	**3xx**	**4xx**	**5xx**
Connections	0	0	8542	0	1458

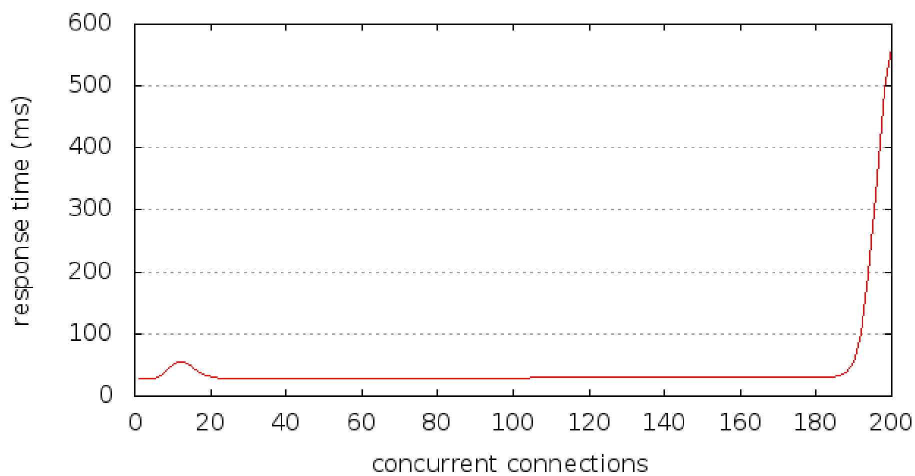
By using a half-interval search method, the optimal number of concurrent connections is found, which gives the highest rate of 190 request per second. As can be seen in figure 3, the total number of `HTTP 3xx` responses is 10,000. Details can be found in appendix A.1.2.

Table 3: httperf measurement 2

	1xx	**2xx**	**3xx**	**4xx**	**5xx**
Connections	0	0	10000	0	0

Based on the values derived from the measurement above, the performance is measured by stepping through the concurrent connections from 1 to 200. The response time is calculated by taking the average of each measurement of each step. Each step has a measurement duration of 60 seconds. When staying under 190 concurrent connections, the response time is $\approx 30ms$. The resource usage on server01 is very minimal, of which the details can be found in appendix A.1.3.

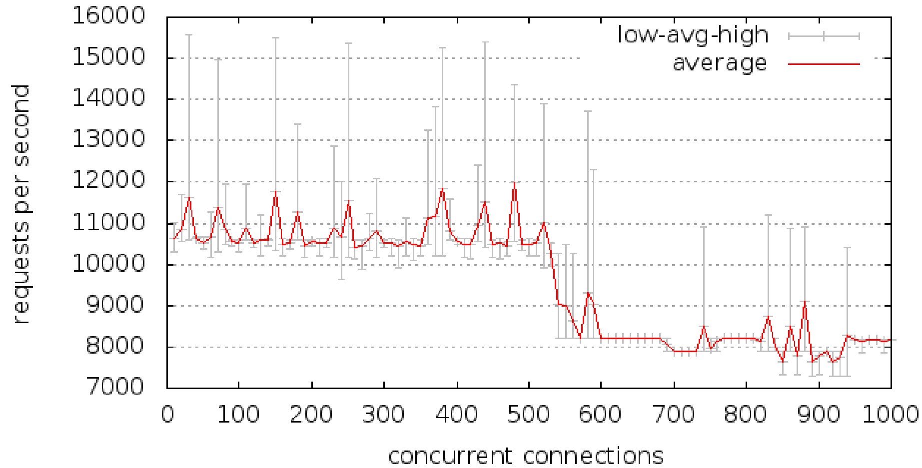
Figure 7: Wordpress: baseline measurement



4.1.2 HTTP 200 OK

To measure the performance of the Nginx web server, the number of concurrent connections is incremented with steps of 10. Figure 8 shows some erratic behaviour when looking at the number of request per second that is measured. This is because the number of requests per second is relatively high and Apache benchmark is not capable of measuring consistent results. The gray line shows the lowest measured value and the highest measured value. The red line shows the average value for each step. The maximum number of requests per second is $11,973$, which is reached when 380 concurrent connections are used. Therefore, 380 concurrent connections is considered as the optimal number of concurrent connections for this configuration. The resource usage of server01 can be found in appendix A.1.4.

Figure 8: HTTP 200 OK: baseline measurement



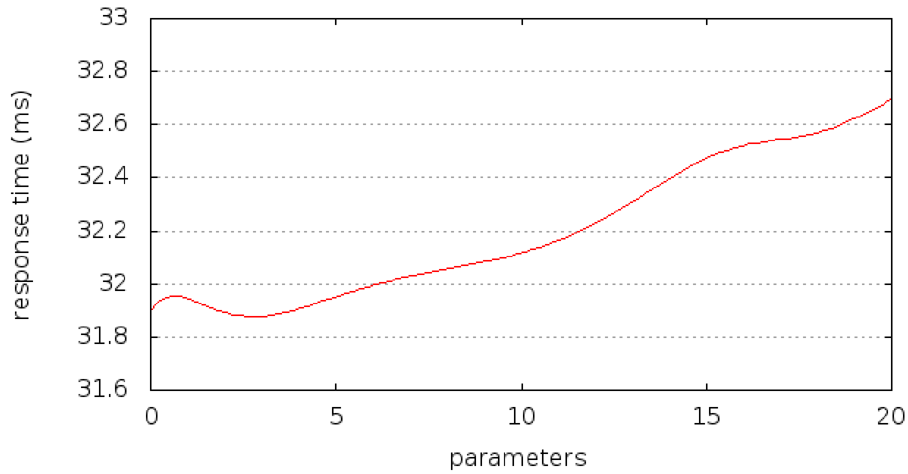
4.2 Performance measurements with Naxsi

4.2.1 Wordpress

Allowed parameters

Figure 9 shows how the response time evolves when the number of valid URL parameters increases. As determined during the baseline measurements, the number of optimal concurrent connections is 190. Each measurement is performed with the same number of concurrent connections, but the number of URL parameters are increased. As can been seen in figure 9, when 20 parameters are used, the response time is 0.8 milliseconds longer than when no parameters are used.

Figure 9: Wordpress: valid URL parameters



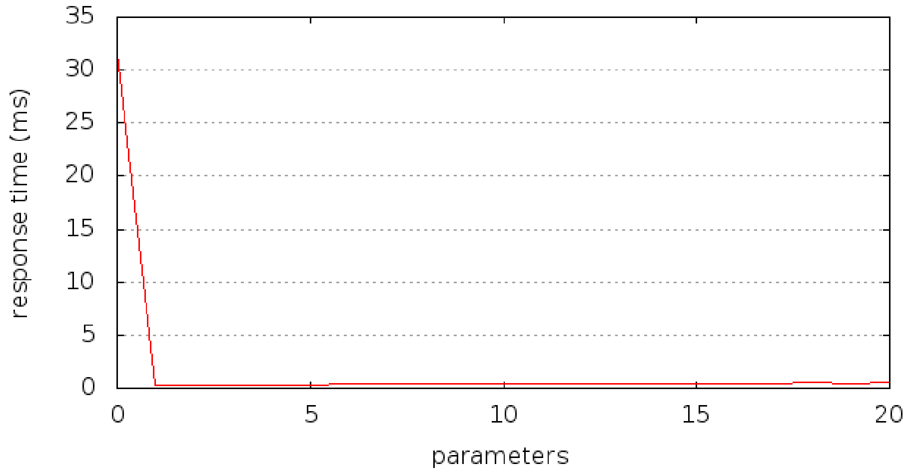
Disallowed parameters

At first sight, the graph in figure 10 shows strange behaviour. This is because as soon as Naxsi sees an invalid URL parameter, it returns an `HTTP 403` error code to the requesting client. The first measurement, where no URL parameter is used, the requests gets forwarded to the back-end server that processes the request, and the results get back to the client. This takes about the same amount of time as is measured during the baseline measurements. As soon as an `HTTP 403` error code is returned, it takes only very little time, as can be seen in table 4.

Table 4: Wordpress: response time with invalid URL parameters

Parameters	**Response time (ms)**
0	31.7
1-5	0.3
6-17	0.4
18-20	0.5

Figure 10: Wordpress: invalid URL parameters

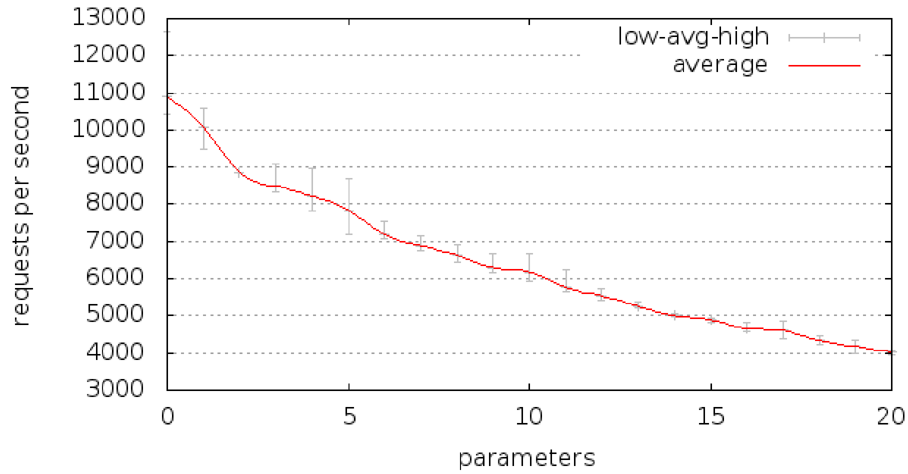


4.2.2 HTTP 200 OK

Allowed parameters

Although, the influence on the response time caused by the number of URL parameters is already visible during the Wordpress measurements, it is much more visible when the Nginx server only returns `HTTP 200 OK` responses, as can be seen in figure 11. The number of requests per seconds decrease somewhat linearly when the number of URL parameters increase. When 20 URL parameters are used, the number of requests per second drop with $\approx 35\%$ compared to when no parameters are used.

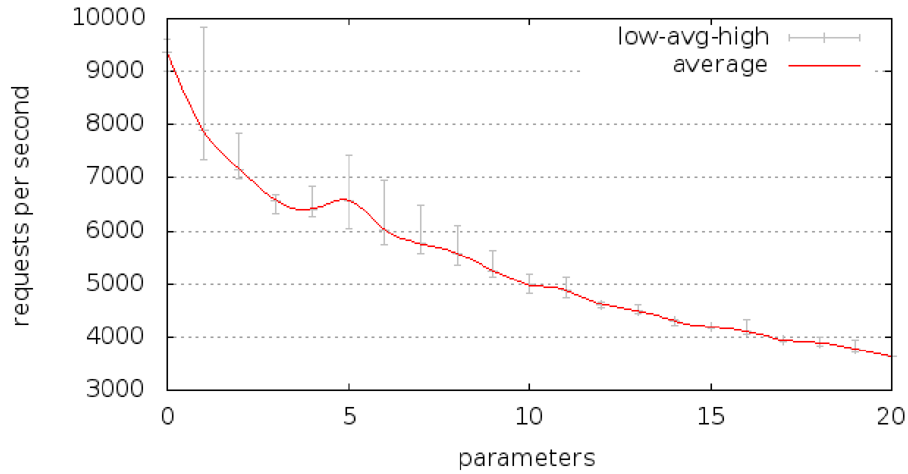
Figure 11: HTTP 200 OK: valid URL parameters



Disallowed parameters

When Naxsi allows the request to be further processed, then Nginx returns an `HTTP 200 OK` response the client. When Naxsi denies a request, then Nginx returns an `HTTP 403` error code to the client. In both cases, there is no processing overhead when returning an HTTP code. Although the line in figure 12 proceeds similar to the one in figure 11, it shows, that denying a request comes with some extra performance loss. The number of requests per second do not decrease linearly, however, but they appear to converge to the same number of requests per second as would be the case when only valid URL parameters are used. When comparing the measurements of both valid URL parameters and invalid URL parameters, the number of concurrent requests per second is approximately 1,000 requests less when 10 parameters are used of which the last URL parameter has bad content.

Figure 12: HTTP 200 OK: invalid URL parameters



5 Conclusion

This paper is about how Naxsi influences the performance of the Nginx web server. The performance of Naxsi is looked at from two perspectives. First, Naxsi is isolated by making Nginx return only `HTTP 200 OK` responses, and thus allowing Nginx to handle a high amount of requests per second. Second, the performance is measured with a more realistic scenario. This is done by hosting a Wordpress website on a back-end server. In both cases, a baseline performance measurement is conducted to find the maximum performance when Naxsi is not compiled into the Nginx web server. When Nginx returns only `HTTP 200 OK` responses, the Nginx server is capable of handling up to $11,973$ requests per second when 380 concurrent connections are used. The back-end server that is hosting the Wordpress website is capable of handling 190 concurrent connections, with a response time of $\approx 30ms$ per request. For measuring the impact of the performance of Naxsi, the number of URL parameters is incremented from 1 to 20. First, all parameters have valid content and Naxsi allows the request to be further processed. With the `HTTP 200 OK` responses, the number of requests per second drops with $\approx 35\%$ when 20 URL parameters are used, compared to when no URL parameters are used. The response time for serving a Wordpress website increases with $0.8ms$ when 20 URL parameters are used, compared to when no parameters are used. When Naxsi encounters an invalid URL parameter, there is a small performance loss for denying the request. In our setup, Naxsi is configured to return an `HTTP 403` error code when it encounters an invalid request. Compared to when 10 valid URL parameters are used, Naxsi is capable of handling $\approx 6,000$ requests per second. But, when the 10th URL parameters has invalid content, then Naxsi drops to $\approx 5,000$ requests per second. Invalid URL parameters have a different result when hosting the Wordpress website, because processing a Wordpress web page takes considerable longer than simply returning an `HTTP 403` error code.

The usage of Naxsi greatly depends on the application that needs to be protected by Naxsi. When a Wordpress website is hosted, then there is a very little performance loss compared to hosting the website without the protection of Naxsi. In this case, Naxsi shows very little impact on the system resources and the response time only slightly increases. When several thousands of requests per second need to be processed by Naxsi, then the performance loss becomes more noticeable.

6 Further research

This paper focusses only on the impact of Naxsi when URL parameters are parsed. However, Naxsi not only inspects the URL parameters, but it also inspects POST arguments, HTTP headers and the full URL, as is explained in section 2.2. The performance impact of these checks may be different compared to those that are measured during this research. Also, a combination of these checks may decrease the overall performance of the web server even further.

References

[1] ab - apache http server benchmarking tool. `http://httpd.apache.org/docs/2.2/programs/ab.html`.

[2] Autobench. `http://www.xenoclast.org/autobench/`.

[3] Collectd – the system statistics collection daemon. `http://collectd.org/`.

[4] Collection 3. `https://collectd.org/wiki/index.php/Collection_3`.

[5] Wayne W. Eckerson. Three tier client/server architecture: Achieving scalability, performance, and efficiency in client server applications. *Open Information Systems*, 10(1), 1995.

[6] Httperf - a tool for measuring web server performance. `http://www.hpl.hp.com/research/linux/httperf/`.

[7] David Mosberger and Tai Jin. httperf—a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[8] Naxsi - Nginx Anti Xss & Sql Injection. `https://code.google.com/p/naxsi/`.

[9] Lutz Prechelt. An empirical comparison of c, c++, java, perl, python, rexx and tcl. *IEEE Computer*, 33(10):23–29, 2000.

[10] Wordpress. `http://wordpress.org/`.

A Experiment results

A.1 Baseline performance measurements

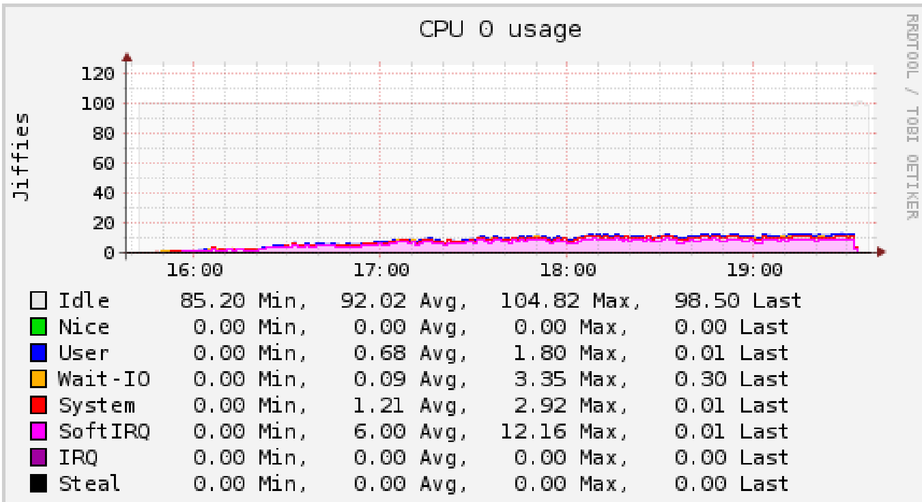
A.1.1 Wordpress httperf measurement 1

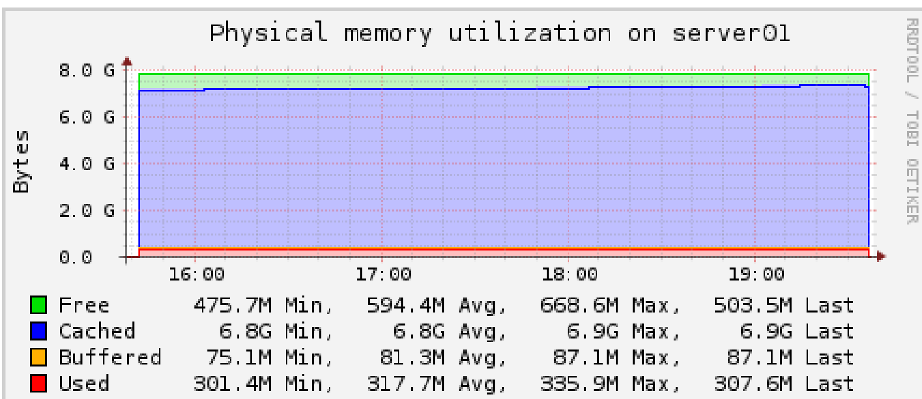
```
~# httperf --server wp_without_naxsi.test.nl --uri /index.php \
--num-call 1 --rate 230 --num-conn 10000
httperf --client=0/1 --server=wp_without_naxsi.test.nl --port=80 \
--uri=/index.php --rate=230 --send-buffer=4096 --recv-buffer=16384 \
--num-conns=10000 --num-calls=1
Maximum connect burst length: 1

Total: connections 10000 requests 10000 replies 10000 test-duration 44.164 s

Connection rate: 226.4 conn/s (4.4 ms/conn, <=136 concurrent connections)
Connection time [ms]: min 0.7 avg 562.7 max 723.3 median 687.5 stddev 255.3
Connection time [ms]: connect 0.2
Connection length [replies/conn]: 1.000

Request rate: 226.4 req/s (4.4 ms/req)
Request size [B]: 86.0

Reply rate [replies/s]: min 203.2 avg 226.7 max 230.2 stddev 9.5 (8 samples)
Reply time [ms]: response 562.5 transfer 0.0
Reply size [B]: header 296.0 content 25.0 footer 1.0 (total 322.0)
Reply status: 1xx=0 2xx=0 3xx=8542 4xx=0 5xx=1458

CPU time [s]: user 1.30 system 42.87 (user 2.9% system 97.1% total 100.0%)
Net I/O: 90.3 KB/s (0.7*10^6 bps)

Errors: total 0 client-timo 0 socket-timo 0 connrefused 0 connreset 0
Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0
```

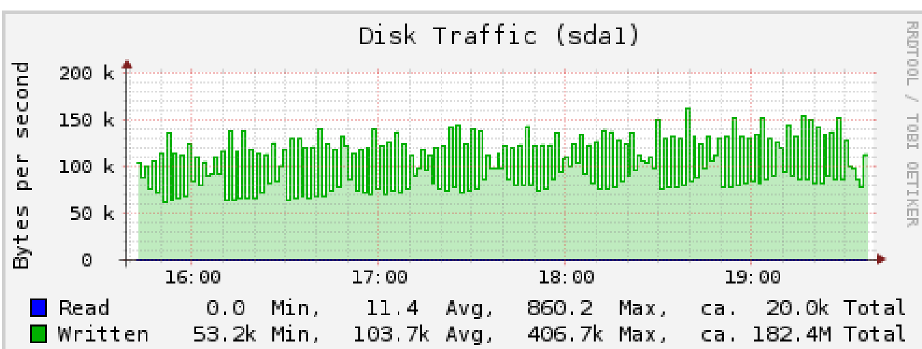
A.1.2 Wordpress httperf measurement 2

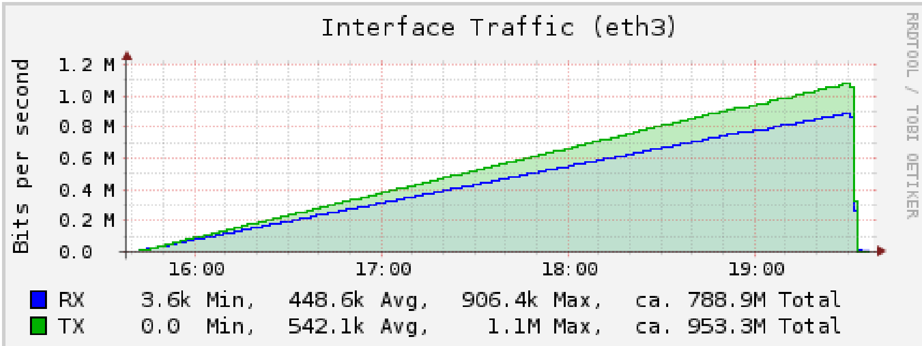
```
# httperf --server wp_without_naxsi.test.nl --uri /index.php --num-call 1 \
--rate 190 --num-conn 10000
httperf --client=0/1 --server=wp_without_naxsi.test.nl --port=80 \
--uri=/index.php --rate=190 --send-buffer=4096 --recv-buffer=16384 \
--num-conns=10000 --num-calls=1
Maximum connect burst length: 1

Total: connections 10000 requests 10000 replies 10000 test-duration 52.656 s

Connection rate: 189.9 conn/s (5.3 ms/conn, <=10 concurrent connections)
Connection time [ms]: min 28.0 avg 31.8 max 51.9 median 31.5 stddev 2.0
Connection time [ms]: connect 0.2
Connection length [replies/conn]: 1.000

Request rate: 189.9 req/s (5.3 ms/req)
Request size [B]: 86.0

Reply rate [replies/s]: min 188.8 avg 189.9 max 190.2 stddev 0.4 (10 samples)
Reply time [ms]: response 31.6 transfer 0.0
Reply size [B]: header 321.0 content 0.0 footer 2.0 (total 323.0)
Reply status: 1xx=0 2xx=0 3xx=10000 4xx=0 5xx=0

CPU time [s]: user 8.93 system 43.73 (user 17.0% system 83.0% total 100.0%)
Net I/O: 75.5 KB/s (0.6*10^6 bps)

Errors: total 0 client-timo 0 socket-timo 0 connrefused 0 connreset 0
Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0
```

A.1.3 Wordpress: resource usage

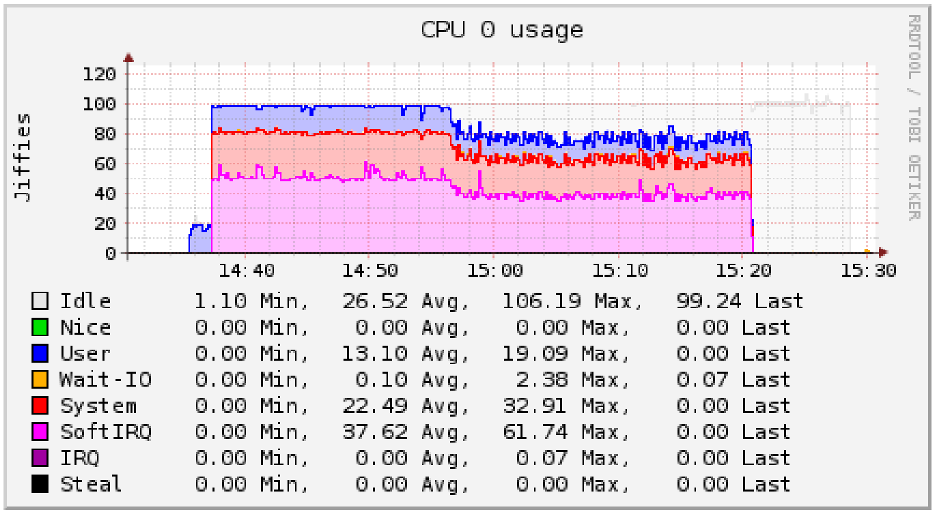


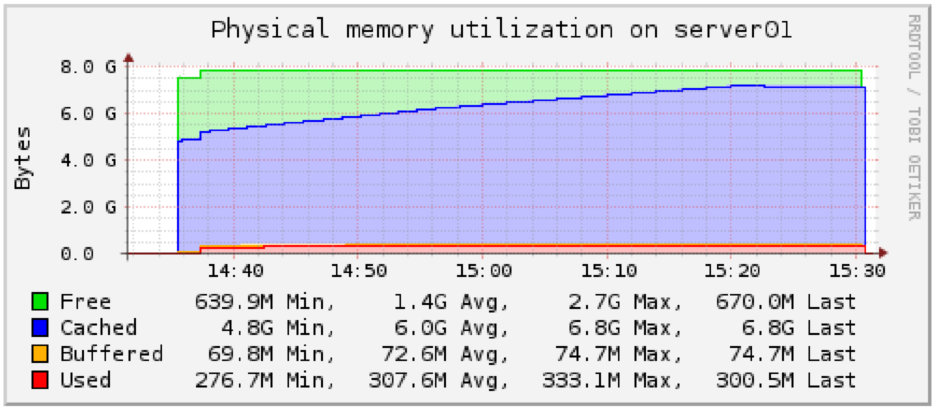


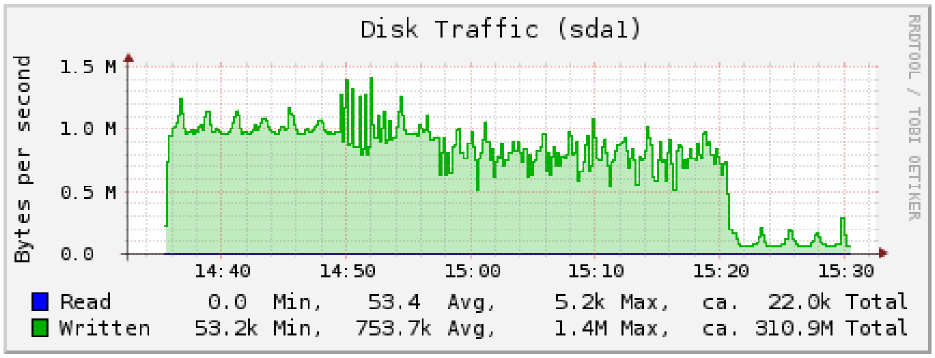


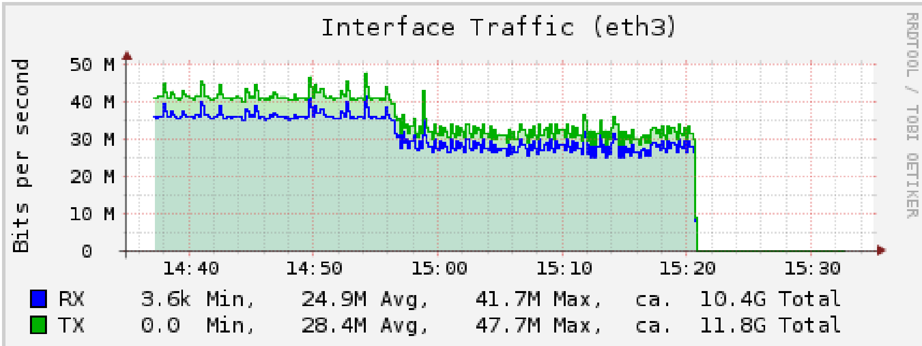

Interface Traffic (eth3)
Bits per second
1.2 M
1.0 M
0.8 M
0.6 M
0.4 M
0.2 M
0.0
16:00
17:00
18:00
19:00
RX 3.6k Min, 448.6k Avg, 906.4k Max, ca. 788.9M Total
TX 0.0 Min, 542.1k Avg, 1.1M Max, ca. 953.3M Total
RRDTOOL / TOBI OETIKER

A.1.4 HTTP 200 OK: resource usage



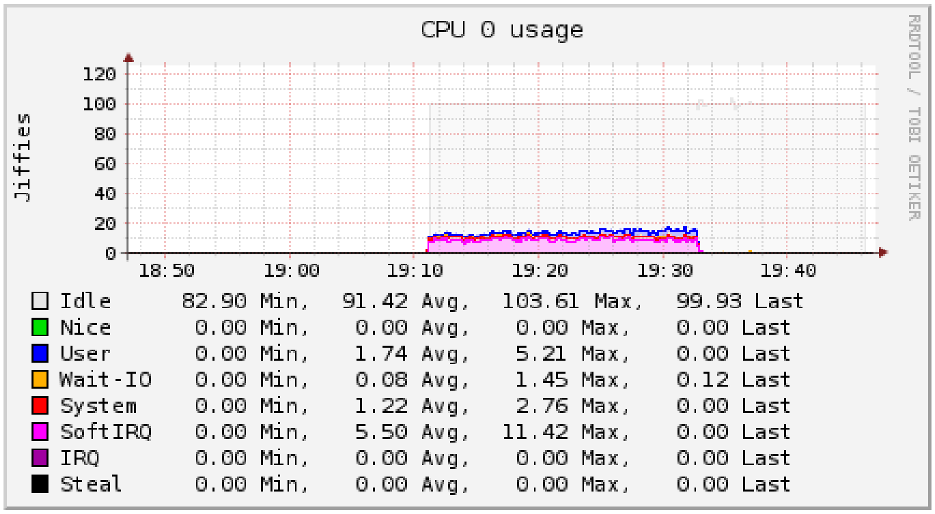


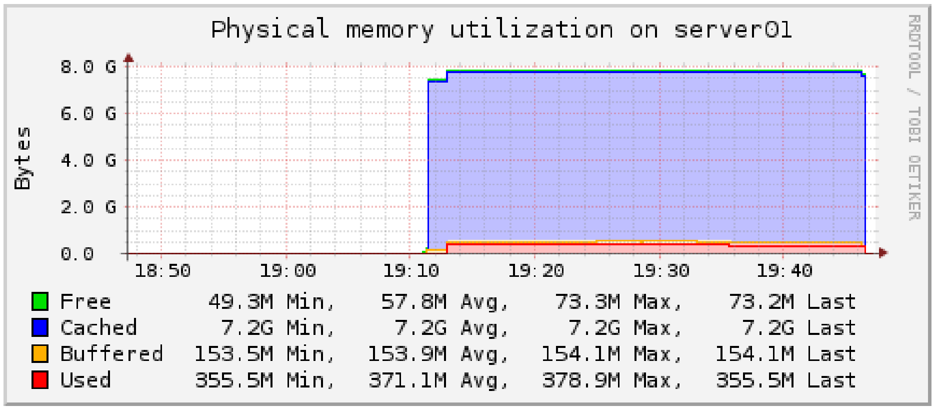


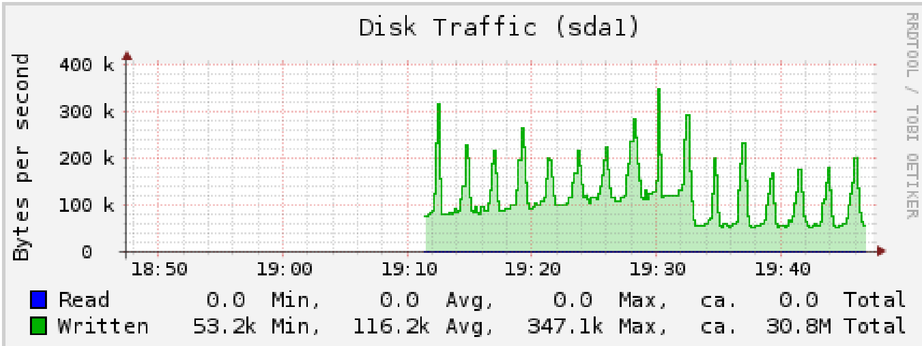

Interface Traffic (eth3)
Bits per second
50 M
40 M
30 M
20 M
10 M
0
14:40
14:50
15:00
15:10
15:20
15:30
RRDTOOL / TOBI OETIKER
RX 3.6k Min, 24.9M Avg, 41.7M Max, ca. 10.4G Total
TX 0.0 Min, 28.4M Avg, 47.7M Max, ca. 11.8G Total

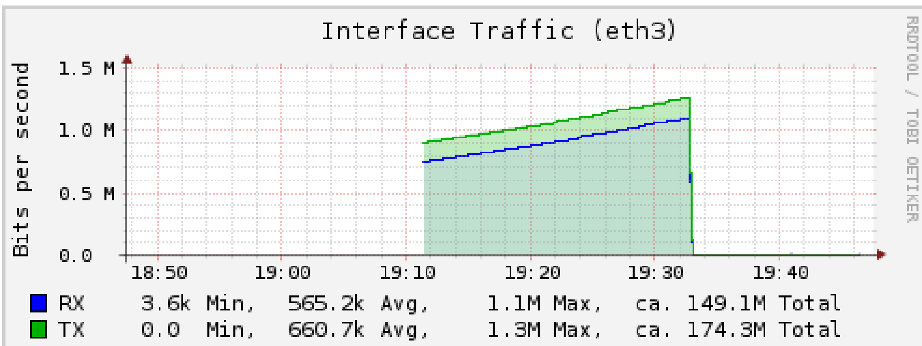
A.2 Naxsi performance measurements

A.2.1 Wordpress: resource usage with valid URL parameters

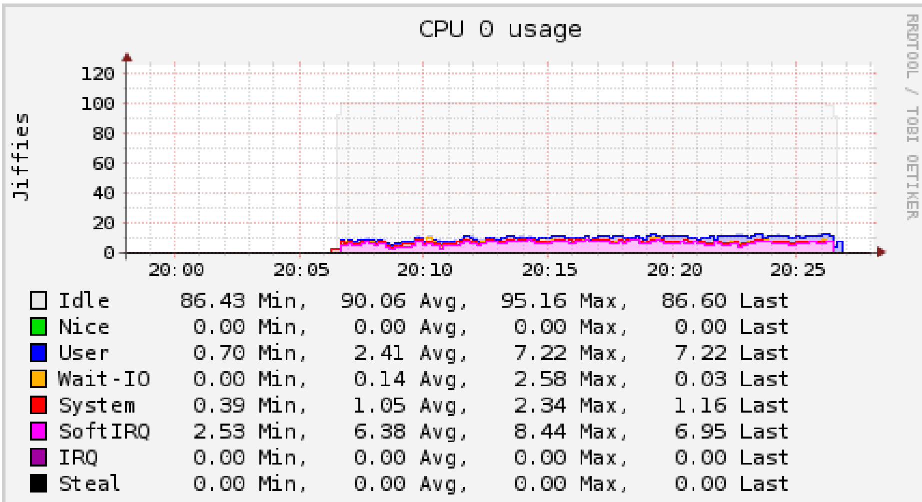


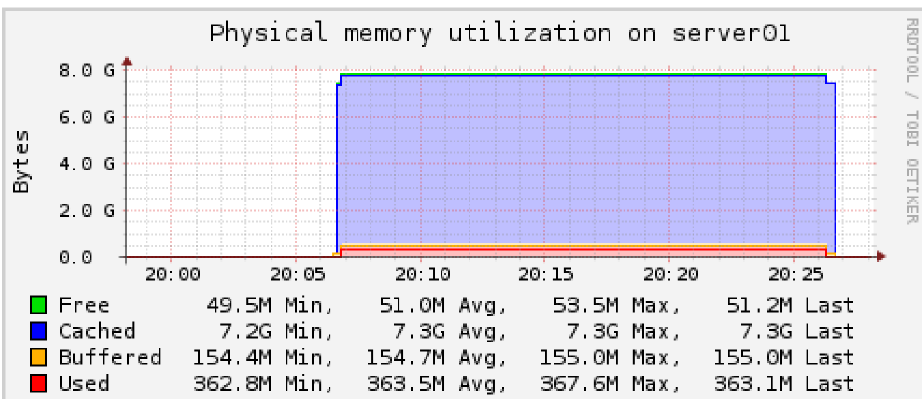


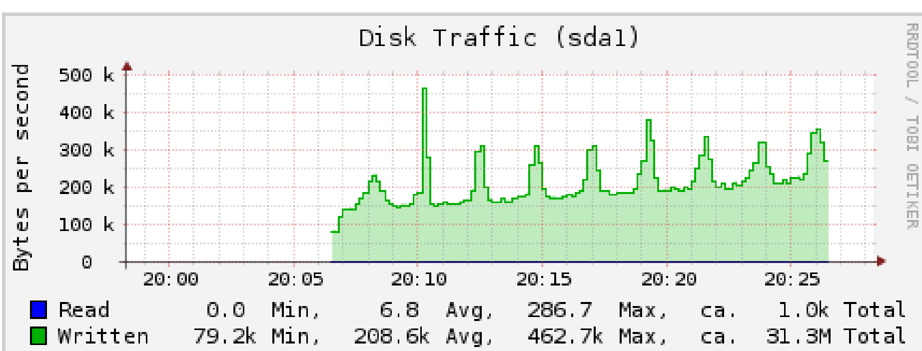

Disk Traffic (sda1)
Bytes per second
400 k
300 k
200 k
100 k
0
18:50
19:00
19:10
19:20
19:30
19:40
Read 0.0 Min, 0.0 Avg, 0.0 Max, ca. 0.0 Total
Written 53.2k Min, 116.2k Avg, 347.1k Max, ca. 30.8M Total
RRDTOOL / TOBI OETIKER

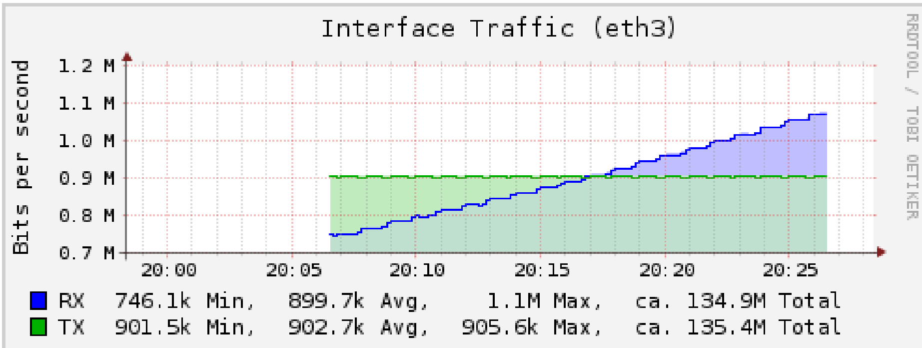

Interface Traffic (eth3)
Bits per second
1.5 M
1.0 M
0.5 M
0.0
18:50
19:00
19:10
19:20
19:30
19:40
RX 3.6k Min, 565.2k Avg, 1.1M Max, ca. 149.1M Total
TX 0.0 Min, 660.7k Avg, 1.3M Max, ca. 174.3M Total
RRDTOOL / TOBI OETIKER

A.2.2 Wordpress: resource usage with valid URL parameters

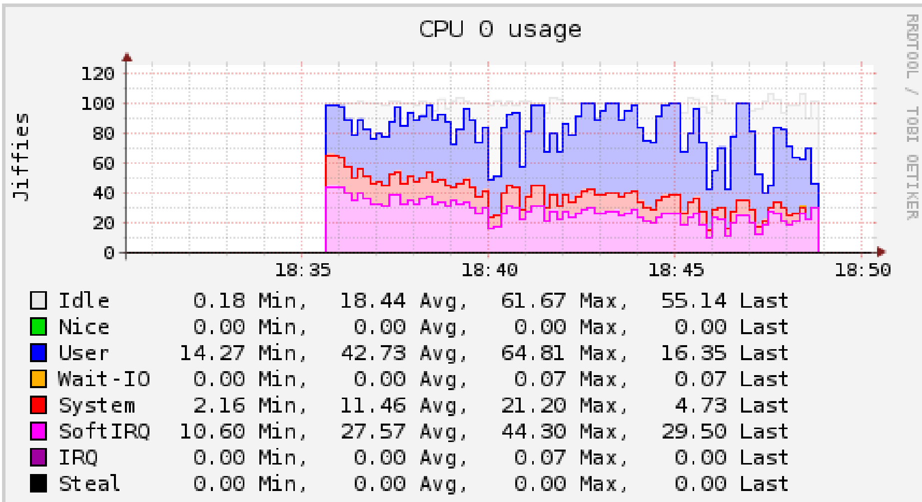


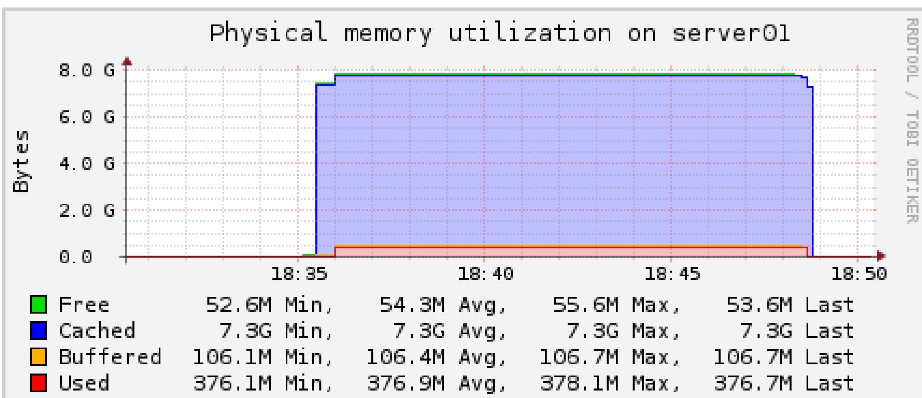


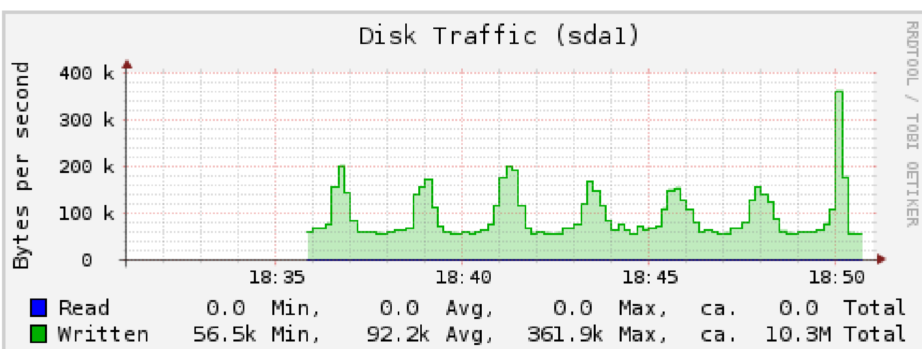


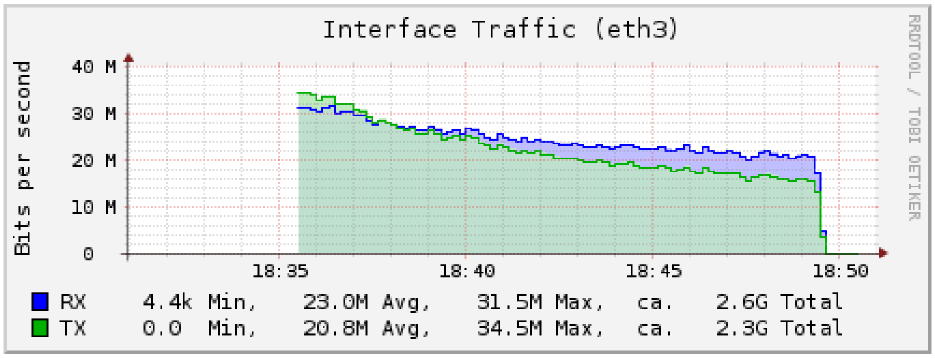

Interface Traffic (eth3)
Bits per second
1.2 M
1.1 M
1.0 M
0.9 M
0.8 M
0.7 M
20:00
20:05
20:10
20:15
20:20
20:25
RRDTOOL / TOBI OETIKER
RX 746.1k Min, 899.7k Avg, 1.1M Max, ca. 134.9M Total
TX 901.5k Min, 902.7k Avg, 905.6k Max, ca. 135.4M Total

A.2.3 HTTP 200 OK: resource usage with valid URL parameters

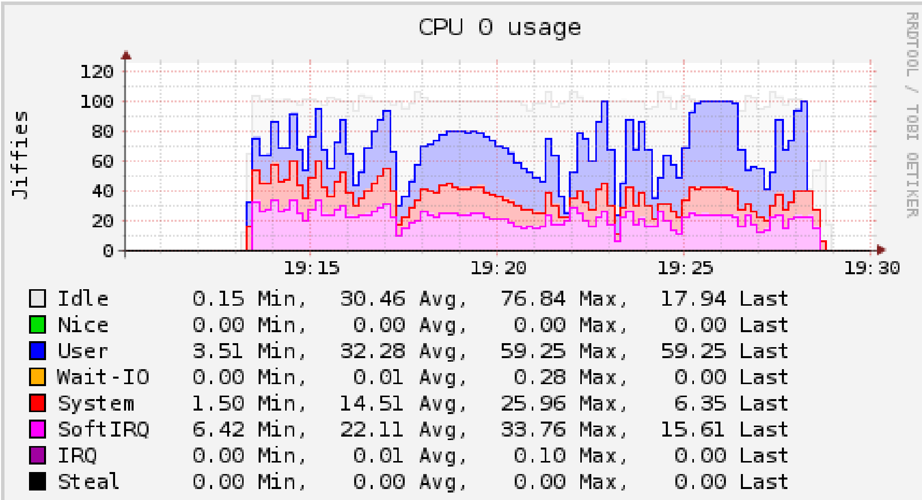


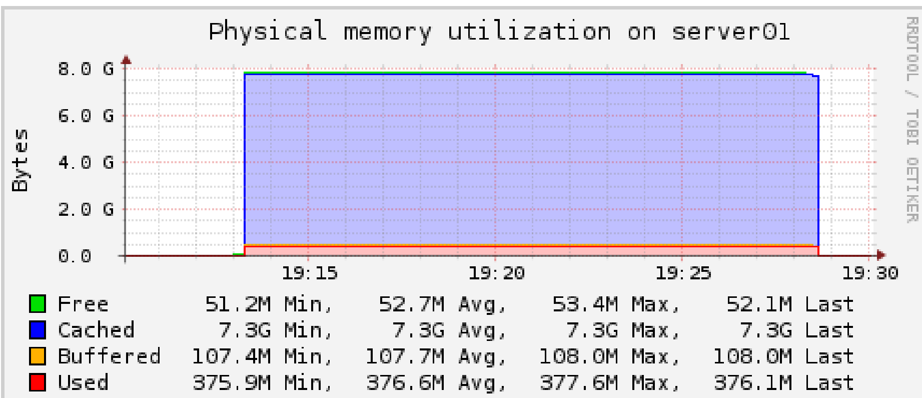


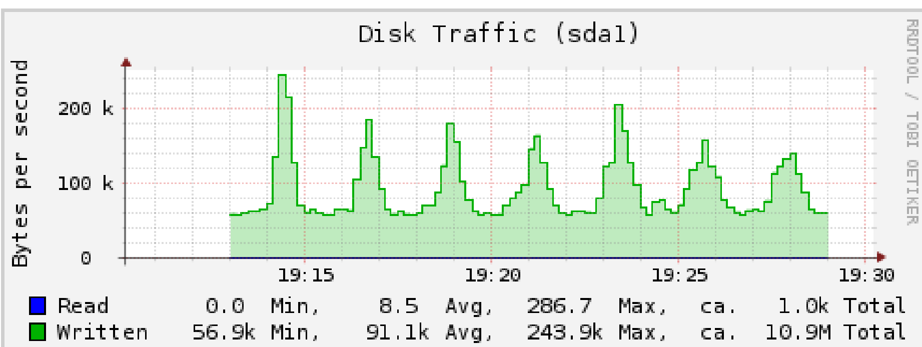


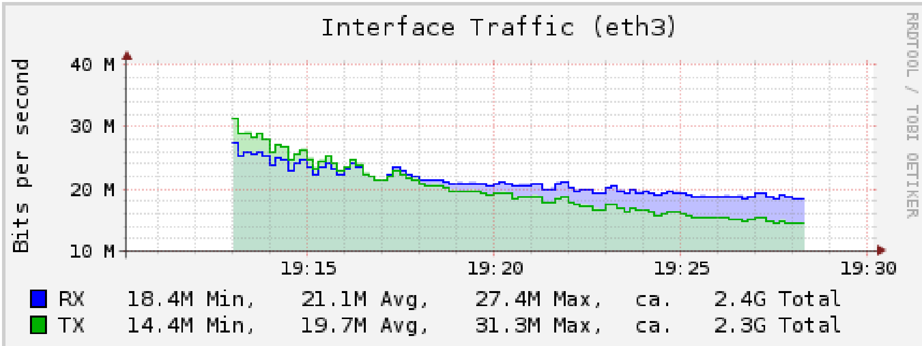

Interface Traffic (eth3)
Bits per second
40 M
30 M
20 M
10 M
0
18:35
18:40
18:45
18:50
RRDTOOL / TOBI OETIKER
RX 4.4k Min, 23.0M Avg, 31.5M Max, ca. 2.6G Total
TX 0.0 Min, 20.8M Avg, 34.5M Max, ca. 2.3G Total

A.2.4 HTTP 200 OK: resource usage with invalid URL parameters








Interface Traffic (eth3)
Bits per second
40 M
30 M
20 M
10 M
19:15
19:20
19:25
19:30
RRDTOOL / TOBI OETIKER
RX 18.4M Min, 21.1M Avg, 27.4M Max, ca. 2.4G Total
TX 14.4M Min, 19.7M Avg, 31.3M Max, ca. 2.3G Total

B Experimental setup

B.1 Hardware

Table 5: server01

Brand	Dell
Model	PowerEdge R210
CPU	Intel(R) Xeon(R) CPU L3426 @ 1.87GHz (2x)
Memory	8 GB RAM, 1066 MHz (4x 2GB)
Disk	500 GB, SATA 3.0 Gbps (2x)
Interfaces	Integrated 10/100/1000 Mbps NIC (2x)
	Broadcom NetXtreme II BCM5716 1000Base-T (C0) PCI Express Dual-port
OS	Debian Squeeze 6.0.7

Table 6: server02

Brand	SunMicro
Model	X7DBT/X7DGT
CPU	Intel(R) Xeon(R) CPU E5420 @ 2.50GHz (2x)
Memory	16 GB RAM, 1333 MHz (8x 2GB)
Disk	160 GB, SATA 3.0 Gbps (2x)
Interfaces	Intel(R) PRO/1000 Network (2x)
OS	Debian Squeeze 6.0.7

Table 7: server03

Brand	SunMicro
Model	X7DBT/X7DGT
CPU	Intel(R) Xeon(R) CPU E5420 @ 2.50GHz (2x)
Memory	16 GB RAM, 1333 MHz (8x 2GB)
Disk	160 GB, SATA 3.0 Gbps (2x)
Interfaces	Intel(R) PRO/1000 Network (2x)
OS	Debian Squeeze 6.0.7

Table 8: server04

Brand	SunMicro
Model	X7DBT/X7DGT
CPU	Intel(R) Xeon(R) CPU E5420 @ 2.50GHz (2x)
Memmory	16 GB RAM, 1333 MHz (8x 2GB)
Disk	160 GB, SATA 3.0 Gbps (2x)
Interfaces	Intel(R) PRO/1000 Network (2x)
OS	Debian Squeeze 6.0.7

Table 9: server05

Brand	Dell
Model	PowerEdge R210
CPU	Intel(R) Xeon(R) CPU L3426 @ 1.87GHz (2x)
Memory	8 GB RAM, 1066 MHz (4x 2GB)
Disk	500 GB, SATA 3.0 Gbps (2x)
Interfaces	Integrated 10/100/1000 Mbps NIC (2x)
	Broadcom NetXtreme II BCM5716 1000Base-T (C0) PCI Express Dual-port
OS	Debian Squeeze 6.0.7

B.2 Configuration

B.2.1 server01, server02, server03 and server04

Listing 1: cronjob -l

```
@hourly /usr/sbin/ntpdate 0.nl.pool.ntp.org 1.nl.pool.ntp.org 2.nl.pool.ntp.org 3.nl.pool.ntp.org
```

Listing 2: Collectd

```
# apt-get install -y vim build-essential librrd-dev
# wget http://collectd.org/files/collectd-5.2.1.tar.bz2
# tar jxf collectd-5.2.1.tar.bz2
# cd collectd-5.2.1
# ./configure --enable-rrdtool --enable-rrdcache
# make
# make install

# cat /opt/collectd/etc/collectd.conf | grep '^[^#]'
LoadPlugin syslog
<Plugin syslog>
    LogLevel info
</Plugin>
LoadPlugin aggregation
LoadPlugin cpu
LoadPlugin csv
LoadPlugin df
LoadPlugin disk
LoadPlugin interface
LoadPlugin load
LoadPlugin memory
LoadPlugin network
LoadPlugin rrdtool
<Plugin "aggregation">
  <Aggregation>
    #Host "unspecified"
    Plugin "cpu"
    Type "cpu"
    GroupBy "Host"
    GroupBy "TypeInstance"
    CalculateNum false
    CalculateSum false
    CalculateAverage true
    CalculateMinimum false
    CalculateMaximum false
    CalculateStddev false
```

```
  </Aggregation>
</Plugin>
<Plugin network>
     Server "145.100.105.165" "1000"
</Plugin>
<Plugin rrdtool>
     DataDir "/opt/collectd/var/lib/collectd/rrd"
     CacheTimeout 120
     CacheFlush   900
</Plugin>

# cat /etc/rc.local | grep '^[^#]'
/opt/collectd/sbin/collectd
exit 0
```

Listing 3: Lines add to /etc/sysctl.conf

```
fs.file-max = 5000000
net.core.netdev_max_backlog = 400000
net.core.optmem_max = 10000000
net.core.rmem_default = 10000000
net.core.rmem_max = 10000000
net.core.somaxconn = 100000
net.core.wmem_default = 10000000
net.core.wmem_max = 10000000
net.ipv4.conf.all.rp_filter = 1
net.ipv4.conf.default.rp_filter = 1
net.ipv4.ip_local_port_range = 1024 65535
net.ipv4.tcp_congestion_control = bic
net.ipv4.tcp_ecn = 0
net.ipv4.tcp_max_syn_backlog = 12000
net.ipv4.tcp_max_tw_buckets = 2000000
net.ipv4.tcp_mem = 30000000 30000000 30000000
net.ipv4.tcp_rmem = 30000000 30000000 30000000
net.ipv4.tcp_sack = 1
net.ipv4.tcp_syncookies = 0
net.ipv4.tcp_timestamps = 1
net.ipv4.tcp_wmem = 30000000 30000000 30000000

# optionally, avoid TIME_WAIT states on localhost no-HTTP Keep-Alive tests:
#    "error: connect() failed: Cannot assign requested address (99)"
# On Linux, the 2MSL time is hardcoded to 60 seconds in /include/net/tcp.h:
# #define TCP_TIMEWAIT_LEN (60*HZ)
```

```
# The option below is safe to use:
net.ipv4.tcp_tw_reuse = 1

# The option below lets you reduce TIME_WAITs further
# but this option is for benchmarks, NOT for production (
    NAT issues)
net.ipv4.tcp_tw_recycle = 1
```

Listing 4: Lines added to /etc/security/limits.conf

```
* soft nofile 200000
* hard nofile 200000
```

B.2.2 server01

Listing 5: /etc/udev/rules.d/70-persistent-net.rules

```
KERNEL=="eth*", ATTR{address}=="b8:ac:6f:8b:81:bd", NAME
    ="eth0"
KERNEL=="eth*", ATTR{address}=="b8:ac:6f:8b:81:be", NAME
    ="eth1"
KERNEL=="eth*", ATTR{address}=="00:15:17:74:7b:3c", NAME
    ="eth2"
KERNEL=="eth*", ATTR{address}=="00:15:17:74:7b:3d", NAME
    ="eth3"
```

Listing 6: /etc/network/interfaces

```
auto lo
iface lo inet loopback

auto lo
iface lo inet loopback

# The primary network interface
allow-hotplug eth0
iface eth0 inet dhcp

auto eth1
iface eth1 inet static
        address 10.1.2.1
        netmask 255.255.255.0
        network 10.1.2.0
        broadcast 10.1.2.255

auto eth2
```

```
iface eth2 inet static
        address 10.1.3.1
        netmask 255.255.255.0
        network 10.1.3.0
        broadcast 10.1.3.255

auto eth3
iface eth3 inet static
        address 10.1.4.1
        netmask 255.255.255.0
        network 10.1.4.0
        broadcast 10.1.4.255
```

Listing 7: /etc/resolv.conf

```
# sysctl -p
net.ipv4.ip_forward = 1
```

Listing 8: /etc/resolv.conf

```
nameserver 145.100.96.11
nameserver 145.100.96.22
```

Listing 9: /etc/network/interfaces

```
#!/bin/bash

iptables -F INPUT
iptables -t nat -F

# Allow internet for internal network
iptables -t nat -A POSTROUTING -o eth0 -s 10.1.0.0/16 ! -d 10.1.0.0/8 -j SNAT --to-source 145.100.104.61

# Allow ssh to reach internal network
iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 1002 -j DNAT --to-destination 10.1.2.2:22
iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 1003 -j DNAT --to-destination 10.1.3.2:22
iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 1004 -j DNAT --to-destination 10.1.4.2:22

# Block http from the internet
iptables -A INPUT -i eth0 -p tcp --dport 80 -j DROP
```

Listing 10: Compiling Nginx with Naxsi

```
wget http://nginx.org/download/nginx-1.2.7.tar.gz
wget http://naxsi.googlecode.com/files/naxsi-core-0.48.tgz
tar xvzf naxsi-core-0.48.tgz
tar xvzf nginx-1.2.7.tar.gz
cd nginx-1.2.7/
./configure --conf-path=/etc/nginx/nginx.conf \
--add-module=../naxsi-core-0.48/naxsi_src/ \
--error-log-path=/var/log/nginx/error.log \
--http-client-body-temp-path=/var/lib/nginx/body \
--http-fastcgi-temp-path=/var/lib/nginx/fastcgi \
--http-log-path=/var/log/nginx/access.log \
--http-proxy-temp-path=/var/lib/nginx/proxy \
--lock-path=/var/lock/nginx.lock \
--pid-path=/var/run/nginx.pid \
--with-http_ssl_module \
--without-mail_pop3_module \
--without-mail_smtp_module  \
--without-mail_imap_module  \
--without-http_uwsgi_module \
--without-http_scgi_module \
--with-ipv6 \
--prefix=/usr
make
make install
mkdir /etc/nginx/sites-enabled
mkdir -p /var/lib/nginx/body
```

Listing 11: Compiling Nginx without Naxsi

```
wget http://nginx.org/download/nginx-1.2.7.tar.gz
tar xvzf nginx-1.2.7.tar.gz
cd nginx-1.2.7/
./configure --conf-path=/etc/nginx/nginx.conf \
--error-log-path=/var/log/nginx/error.log \
--http-client-body-temp-path=/var/lib/nginx/body \
--http-fastcgi-temp-path=/var/lib/nginx/fastcgi \
--http-log-path=/var/log/nginx/access.log \
--http-proxy-temp-path=/var/lib/nginx/proxy \
--lock-path=/var/lock/nginx.lock \
--pid-path=/var/run/nginx.pid \
--with-http_ssl_module \
--without-mail_pop3_module \
--without-mail_smtp_module  \
--without-mail_imap_module  \
```

```
--without-http_uwsgi_module \
--without-http_scgi_module \
--with-ipv6 \
--prefix=/usr
make
make install
mkdir /etc/nginx/sites-enabled
mkdir -p /var/lib/nginx/body
```

Listing 12: /etc/nginx/nginx.conf

```
worker_processes  1;

events {
    worker_connections  1024;
}

http {
    # Uncomment to enable Naxsi core rules
    #include           /etc/nginx/naxsi_core.rules;

    include           mime.types;
    default_type      application/octet-stream;

    sendfile          on;

    keepalive_timeout  65;

    include           /etc/nginx/sites-enabled/*;
}
```

Listing 13: /etc/nginx/sites-enabled/wordpress

```
upstream backend {
  server 10.1.2.2;
}

server {
  listen                80;

  server_name           _;

  location / {
    proxy_pass          http://backend;
    proxy_redirect      off;
    proxy_set_header    Host              $host;
```

```
    proxy_set_header     X-Real-IP          $remote_addr;
    proxy_set_header     X-Forwarded-For    $proxy_add_x_forwarded_for;
  }
}
```

Listing 14: Minimal whitelist rules

```
# cat /etc/nginx/nbs.rules

SecRulesEnabled;
DeniedUrl "/RequestDenied";

## check rules
CheckRule "$SQL >= 8" BLOCK;
CheckRule "$RFI >= 8" BLOCK;
CheckRule "$TRAVERSAL >= 4" BLOCK;
CheckRule "$EVADE >= 4" BLOCK;
CheckRule "$XSS >= 8" BLOCK;
```

Listing 15: Wordpress whitelist rules

```
# cat /etc/nginx/nbs.rules

## check rules
CheckRule "$SQL >= 8" BLOCK;
CheckRule "$RFI >= 8" BLOCK;
CheckRule "$TRAVERSAL >= 4" BLOCK;
CheckRule "$EVADE >= 4" BLOCK;
CheckRule "$XSS >= 8" BLOCK;

# WordPress naxsi rules

### HEADERS
BasicRule wl:1000,1001,1005,1007,1010,1011,1013,1100,1200,
        1308,1309,1315 "mz:$HEADERS_VAR:cookie";
# xmlrpc
BasicRule wl:1402 "mz:$HEADERS_VAR:content-type";

### simple BODY (POST)
# comments
BasicRule wl:1000,1010,1011,1013,1015,1200 "mz:$BODY_VAR:post_title";
BasicRule wl:1000 "mz:$BODY_VAR:original_publish";
BasicRule wl:1000 "mz:$BODY_VAR:save";
```

```
BasicRule wl:1008,1010,1011,1013,1015 "mz:$BODY_VAR:sk2_my_js_payload";
BasicRule wl:1001,1009,1005,1016,1100,1310 "mz:$BODY_VAR:url";
BasicRule wl:1009,1100 "mz:$BODY_VAR:referredby";
BasicRule wl:1009,1100 "mz:$BODY_VAR:_wp_original_http_referer";
BasicRule wl:1000,1001,1005,1008,1007,1009,1010,1011,1013,1015,
    1016,1100,1200,1302,1303,1310,1311,1315,1400
    "mz:$BODY_VAR:comment";
BasicRule wl:1100 "mz:$BODY_VAR:redirect_to";
BasicRule wl:1000,1009,1315 "mz:$BODY_VAR:_wp_http_referer";
BasicRule wl:1000 "mz:$BODY_VAR:action";
BasicRule wl:1001,1013 "mz:$BODY_VAR:blogname";
BasicRule wl:1015,1013 "mz:$BODY_VAR:blogdescription";
BasicRule wl:1015 "mz:$BODY_VAR:date_format_custom";
BasicRule wl:1015 "mz:$BODY_VAR:date_format";
BasicRule wl:1015 "mz:$BODY_VAR:tax_input%5bpost_tag%5d";
BasicRule wl:1100 "mz:$BODY_VAR:siteurl";
BasicRule wl:1100 "mz:$BODY_VAR:home";
BasicRule wl:1000,1015 "mz:$BODY_VAR:submit";
# news content matches pretty much everything
BasicRule wl:0 "mz:$BODY_VAR:content";
BasicRule wl:1000 "mz:$BODY_VAR:delete_option";
BasicRule wl:1000 "mz:$BODY_VAR:prowl-msg-message";
BasicRule wl:1100 "mz:$BODY_VAR:_url";
BasicRule wl:1001,1009 "mz:$BODY_VAR:c2c_text_replace%5btext_to_replace%5d";
BasicRule wl:1200 "mz:$BODY_VAR:ppn_post_note";
BasicRule wl:1100 "mz:$BODY_VAR:author";
BasicRule wl:1001,1015 "mz:$BODY_VAR:excerpt";
BasicRule wl:1015 "mz:$BODY_VAR:catslist";
BasicRule wl:1005,1008,1009,1010,1011,1015,1315 "mz:$BODY_VAR:cookie";
BasicRule wl:1101 "mz:$BODY_VAR:googleplus";
BasicRule wl:1007 "mz:$BODY_VAR:name";
BasicRule wl:1007 "mz:$BODY_VAR:action";
BasicRule wl:1100 "mz:$BODY_VAR:attachment%5burl%5d";
BasicRule wl:1100 "mz:$BODY_VAR:attachment_url";
BasicRule wl:1001,1009,1100,1302,1303,1310,1311 "mz:$BODY_VAR:html";
BasicRule wl:1015 "mz:$BODY_VAR:title";
BasicRule wl:1001,1009,1015 "mz:$BODY_VAR:recaptcha_challenge_field";
```

```

### BODY|NAME
BasicRule wl:1000 "mz:$BODY_VAR:delete_option|NAME";
BasicRule wl:1000 "mz:$BODY_VAR:from|NAME";

### Simple ARGS (GET)
# WP login screen
BasicRule wl:1100 "mz:$ARGS_VAR:redirect_to";
BasicRule wl:1000,1009 "mz:$ARGS_VAR:_wp_http_referer";
BasicRule wl:1000 "mz:$ARGS_VAR:wp_http_referer";
BasicRule wl:1000 "mz:$ARGS_VAR:action";
BasicRule wl:1000 "mz:$ARGS_VAR:action2";
# load and load[] GET variable
BasicRule wl:1000,1015 "mz:$ARGS_VAR:load";
BasicRule wl:1000,1015 "mz:$ARGS_VAR:load[]";
BasicRule wl:1015 "mz:$ARGS_VAR:q";
BasicRule wl:1000,1015 "mz:$ARGS_VAR:load%5b%5d";

### URL
BasicRule wl:1000 "mz:URL|$URL:/wp-admin/update-core.php";
BasicRule wl:1000 "mz:URL|$URL:/wp-admin/update.php";
# URL|BODY
BasicRule wl:1009,1100 "mz:$URL:/wp-admin/post.php|$BODY_VAR:_wp_http_referer";
BasicRule wl:1016 "mz:$URL:/wp-admin/post.php|$BODY_VAR:metakeyselect";
BasicRule wl:11 "mz:$URL:/xmlrpc.php|BODY";
BasicRule wl:11 "mz:$URL:/wp-cron.php|BODY";
BasicRule wl:2 "mz:$URL:/wp-admin/async-upload.php|BODY";
# URL|BODY|NAME
BasicRule wl:1100 "mz:$URL:/wp-admin/post.php|$BODY_VAR:_wp_original_http_referer|NAME";
BasicRule wl:1000 "mz:$URL:/wp-admin/post.php|$BODY_VAR:metakeyselect|NAME";
BasicRule wl:1000 "mz:$URL:/wp-admin/user-edit.php|$BODY_VAR:from|NAME";
BasicRule wl:1100 "mz:$URL:/wp-admin/admin-ajax.php|$BODY_VAR:attachment%5burl%5d|NAME";
BasicRule wl:1100 "mz:$URL:/wp-admin/post.php|$BODY_VAR:attachment_url|NAME";
BasicRule wl:1000 "mz:$URL:/wp-admin/plugins.php|$BODY_VAR:verify-delete|NAME";
BasicRule wl:1310,1311 "mz:$URL:/wp-admin/post.php|$BODY_VAR:post_category[]|NAME";
BasicRule wl:1311 "mz:$URL:/wp-admin/post.php|$BODY_VAR:
```

```
    post_category|NAME";
BasicRule wl:1310,1311 "mz:$URL:/wp-admin/post.php|
    $BODY_VAR:tax_input[post_tag]|NAME";
BasicRule wl:1310,1311 "mz:$URL:/wp-admin/post.php|
    $BODY_VAR:newtag[post_tag]|NAME";
# URL|ARGS|NAME
BasicRule wl:1310,1311 "mz:$URL:/wp-admin/load-scripts.
    php|$ARGS_VAR:load[]|NAME";
BasicRule wl:1000 "mz:$URL:/wp-admin/users.php|$ARGS_VAR:
    delete_count|NAME";
BasicRule wl:1000 "mz:$URL:/wp-admin/users.php|$ARGS_VAR:
    update|NAME";

# plain WP site
BasicRule wl:1000 "mz:URL|$URL:/wp-admin/update-core.php
    ";
BasicRule wl:1000 "mz:URL|$URL:/wp-admin/update.php";
# URL|BODY
BasicRule wl:1009,1100 "mz:$URL:/wp-admin/post.php|
    $BODY_VAR:_wp_http_referer";
BasicRule wl:1016 "mz:$URL:/wp-admin/post.php|$BODY_VAR:
    metakeyselect";
BasicRule wl:11 "mz:$URL:/xmlrpc.php|BODY";
BasicRule wl:11 "mz:$URL:/wp-cron.php|BODY";
# URL|BODY|NAME
BasicRule wl:1100 "mz:$URL:/wp-admin/post.php|$BODY_VAR:
    _wp_original_http_referer|NAME";
BasicRule wl:1000 "mz:$URL:/wp-admin/post.php|$BODY_VAR:
    metakeyselect|NAME";
BasicRule wl:1000 "mz:$URL:/wp-admin/user-edit.php|
    $BODY_VAR:from|NAME";
BasicRule wl:1100 "mz:$URL:/wp-admin/admin-ajax.php|
    $BODY_VAR:attachment%5burl%5d|NAME";
# URL|ARGS|NAME
BasicRule wl:1310,1311 "mz:$URL:/wp-admin/load-scripts.
    php|$ARGS_VAR:load[]|NAME";
BasicRule wl:1000 "mz:$URL:/wp-admin/users.php|$ARGS_VAR:
    delete_count|NAME";
BasicRule wl:1000 "mz:$URL:/wp-admin/users.php|$ARGS_VAR:
    update|NAME";
```

[7]

[7] http://imil.net/wp/2012/12/30/wordpress-3-5-and-naxsi/

B.2.3 server02

Listing 16: /etc/network/interfaces

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
        address 10.1.2.2
        netmask 255.255.255.0
        network 10.1.2.0
        broadcast 10.1.2.255
        gateway 10.1.2.1

auto eth1
iface eth1 inet static
        address 10.2.2.2
        netmask 255.255.255.0
        network 10.2.2.0
        broadcast 10.2.2.255
```

Listing 17: /etc/resolv.conf

```
nameserver 145.100.96.11
nameserver 145.100.96.22
```

Listing 18: necessary packages

```
# apt-get install nginx spawn-fcgi php5-common php5-mysql php5-xmlrpc php5-cgi php5-curl php5-gd php5-cli php-apc php-pear php5-dev php5-imap php5-mcrypt
```

Listing 19: /etc/php5/cgi/php.ini

```
# echo "extension=mysql.so" >> /etc/php5/cgi/php.ini
# echo "extension=mysqli.so" >> /etc/php5/cgi/php.ini
```

Listing 20: Wordpress 3.5.1 installation

```
# mkdir -p /srv/www/with_naxsi.test.nl/logs
# cd /srv/wwww/with_naxsi.test.nl/
# wget http://wordpress.org/latest.tar.gz
# tar zxvf latest.tar.gz
# chown -R www-data:www-data /srv/www/with_naxsi.test.nl
```

Listing 21: /usr/bin/php-fastcgi (needs chmod +x /usr/bin/php-fastcgi)

```
#!/bin/bash

FASTCGI_USER=www-data
FASTCGI_GROUP=www-data
SOCKET=/var/run/php-fastcgi/php-fastcgi.socket
PIDFILE=/var/run/php-fastcgi/php-fastcgi.pid
CHILDREN=6
PHP5=/usr/bin/php5-cgi

/usr/bin/spawn-fcgi -s $SOCKET -P $PIDFILE -C $CHILDREN -
    u $FASTCGI_USER -g $FASTCGI_GROUP -f $PHP5
```

Listing 22: /etc/nginx/sites-enabled/wordpress

```
server {
    server_name wordpress;
    root /srv/www/wordpress;

    location / {
        index  index.php;
    }

    location ~ \.php$ {
        include /etc/nginx/fastcgi_params;
        fastcgi_pass unix:/var/run/php-fastcgi/php-
            fastcgi.socket;
        fastcgi_index index.php;
        fastcgi_param SCRIPT_FILENAME /srv/www/
            wordpress$fastcgi_script_name;
    }
}
```

Listing 23: /etc/nginx/nginx.conf

```
user www-data;
worker_processes  1;

events {
    worker_connections  1024;
}

http {
    include       mime.types;
    default_type  application/octet-stream;

```

```
    sendfile        on;

    keepalive_timeout  65;

    access_log  off;

    include /etc/nginx/sites-enabled/*;
}
```

Listing 24: /etc/init.d/php-fastcgi

```
#!/bin/bash

PHP_SCRIPT=/usr/bin/php-fastcgi
FASTCGI_USER=www-data
FASTCGI_GROUP=www-data
PID_DIR=/var/run/php-fastcgi
PID_FILE=/var/run/php-fastcgi/php-fastcgi.pid
RET_VAL=0

case "$1" in
    start)
      if [[ ! -d $PID_DIR ]]
      then
        mkdir $PID_DIR
        chown $FASTCGI_USER:$FASTCGI_GROUP $PID_DIR
        chmod 0770 $PID_DIR
      fi
      if [[ -r $PID_FILE ]]
      then
        echo "php-fastcgi already running with PID `cat
            $PID_FILE`"
        RET_VAL=1
      else
        $PHP_SCRIPT
        RET_VAL=$?
      fi
  ;;
    stop)
      if [[ -r $PID_FILE ]]
      then
        kill `cat $PID_FILE`
        rm $PID_FILE
        RET_VAL=$?
      else
        echo "Could not find PID file $PID_FILE"
```

```
      RET_VAL=1
    fi
  ;;
   restart)
    if [[ -r $PID_FILE ]]
    then
      kill `cat $PID_FILE`
      rm $PID_FILE
      RET_VAL=$?
    else
      echo "Could not find PID file $PID_FILE"
    fi
    $PHP_SCRIPT
    RET_VAL=$?
  ;;
   status)
    if [[ -r $PID_FILE ]]
    then
      echo "php-fastcgi running with PID `cat $PID_FILE`"
      RET_VAL=$?
    else
      echo "Could not find PID file $PID_FILE, php-fastcgi does not appear to be running"
    fi
  ;;
   *)
    echo "Usage: php-fastcgi {start|stop|restart|status}"
    RET_VAL=1
  ;;
esac
exit $RET_VAL
```

Listing 25: starting services

```
chmod +x /etc/init.d/php-fastcgi
update-rc.d php-fastcgi defaults
/etc/init.d/php-fastcgi start
/etc/init.d/nginx start
```

B.2.4 server03

Listing 26: /etc/network/interfaces

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
        address 10.1.3.2
        netmask 255.255.255.0
        network 10.1.3.0
        broadcast 10.1.3.255
        gateway 10.1.3.1

auto eth1
iface eth1 inet static
        address 10.2.2.3
        netmask 255.255.255.0
        network 10.2.2.0
        broadcast 10.2.2.255
```

Listing 27: /etc/resolv.conf

```
nameserver 145.100.96.11
nameserver 145.100.96.22
```

Listing 28: MySQL Server 5.1

```
# apt-get install mysql-server
```

Listing 29: MySQL configuration

```
# sed -i 's/127.0.0.1/10.2.2.3/' /etc/mysql/my.cnf
# /etc/init.d/mysql restart

# mysql -u root -p
Enter password:
mysql> CREATE DATABASE wordpress;
Query OK, 1 row affected (0.01 sec)

mysql> GRANT ALL PRIVILEGES ON wordpress.* TO 'naxsi'@
    '10.2.2.2' IDENTIFIED BY 'naxsi';
Query OK, 0 rows affected (0.00 sec)
```

B.2.5 server04

Listing 30: /etc/network/interfaces

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
        address 10.1.4.2
        netmask 255.255.255.0
        network 10.1.4.0
        broadcast 10.1.4.255
        gateway 10.1.4.1
```

Listing 31: /etc/resolv.conf

```
nameserver 145.100.96.11
nameserver 145.100.96.22
```

Listing 32: httperf and autobench

```
# apt-get install build-essentials gawk httperf libgd2-xpm-dev
# wget http://www.xenoclast.org/autobench/downloads/autobench-2.1.2.tar.gz
# tar zxvf autobench
# cd autobench-2.1.2
# make
# make install
# cd ..
# sed -i 's/echo set data style linespoints >> gnuplot.cmd/echo set style data linespoints >> gnuplot.cmd/' /usr/local/bin/bench2graph

# wget http://sourceforge.net/projects/gnuplot/files/gnuplot/4.6.1/gnuplot-4.6.1.tar.gz/download
# tar zxvf download
# cd gnuplot-4.6.1
# ./configure
# make
# make install
```

B.2.6 server05

Listing 33: Debian packages

```
# apt-get install apache2 librrds-perl libconfig-general-perl libhtml-parser-perl libregexp-common-perl librrd2-dev rrdtool
```

Listing 34: Collectd

```
# wget http://collectd.org/files/collectd-5.2.1.tar.gz
# tar zxvf collectd-5.2.1.tar.gz
# cd collectd-5.2.1/
# ./configure --enable-rrdtool --enable-rrdcache
# make
# make install
# cp -r contrib/collection3/* /var/www
# chown -R www-data:www-data /var/www

# ln -s /opt/collectd/var/lib/collectd/ /var/lib/collectd
# head -1 /var/www/etc/collection.conf
DataDir "/opt/collectd/var/lib/collectd/rrd"

# cat /opt/collectd/etc/collectd.conf | grep '^[^#]'
LoadPlugin syslog
<Plugin syslog>
    LogLevel info
</Plugin>
LoadPlugin aggregation
LoadPlugin cpu
LoadPlugin csv
LoadPlugin df
LoadPlugin disk
LoadPlugin interface
LoadPlugin load
LoadPlugin memory
LoadPlugin network
LoadPlugin rrdtool
<Plugin "aggregation">
  <Aggregation>
    #Host "unspecified"
    Plugin "cpu"
    Type "cpu"
    GroupBy "Host"
    GroupBy "TypeInstance"
    CalculateNum false
    CalculateSum false
```

```
    CalculateAverage true
    CalculateMinimum false
    CalculateMaximum false
    CalculateStddev false
  </Aggregation>
</Plugin>
<Plugin network>
    Listen "145.100.105.165" "1000"
</Plugin>
<Plugin rrdtool>
    DataDir "/opt/collectd/var/lib/collectd/rrd"
    CacheTimeout 120
    CacheFlush   900
</Plugin>
```

Listing 35: Apache

```
# cat /etc/apache2/sites-available/collectd
<VirtualHost *:80>
    ServerAdmin webmaster@localhost

    DocumentRoot /var/www/

    <Directory /var/www/>
        AddHandler cgi-script .cgi
        DirectoryIndex bin/index.cgi
        Options +ExecCGI
        Order Allow,Deny
        Allow from all
    </Directory>
</VirtualHost>

# a2ensite collectd
# a2dissite default
# /etc/init.d/apache2 restart
```