nginx

- HSTS Header "richtig" setzen
- HSTS-Header (HTTP Strict Transport Security) konfigurieren
- Logfiles dynamisch benennen
- nginx page caching
- Permanente Umleitung auf HTTPS
- Query-String an PHP-FPM mit übergeben
- Reverse Proxy
- Reverse Proxy um HTTP Basic Auth zu entfernen
- Rewrite einer Datei auf Datei-$Hostname
- Sichere SSL Konfiguration
- Tuning nginx
- X-Frame-Options setzen
- Zugriff auf .git und .svn Unterverzeichnisse verbieten

HSTS Header "richtig" setzen

Viele nginx-User haben HTTP und HTTPS in einem server-Block zusammengefasst. Trägt man dort nun den add_header Code ein, wird er auch für beide Protokolle ausgeliefert:

```
add_header Strict-Transport-Security "max-age=31536000; includeSubDomains; preload";
```

Das ist eigentlich falsch, da laut RFC 6797 der HSTS Header nicht über unverschlüsseltes HTTP gesendet werden soll.

Auf jeden Fall RFC konform ist diese Lösung und es lassen sich damit auch Port 80 und 443 zusammenfassen:

```
map $scheme $hsts_header {
    https "max-age=31536000; includeSubDomains; preload";
}

server {
    listen  80;
    listen  443 ssl;

    add_header Strict-Transport-Security $hsts_header;
}
```

HSTS-Header (HTTP Strict Transport Security) konfigurieren

HTTP Strict Transport Security (abgekürzt HSTS, definiert in RFC6797) ist ein Sicherheitsfeature einer Webseite, das dem Besucher, bzw. dessen Browser sagt, dass sie nur noch per HTTPS verschlüsselt mit ihm kommunizieren will. Dazu wird ein zusätzlicher HTTP-Header gesetzt, der Angaben zum Zeitraum, Umgang mit Subdomains und der Verwendung der HSTS Preloadliste enthält.

Das Feature funktioniert folgendermaßen: Ein Besucher tippt z.B. www.seite-x.de in seinen Browser ein. Der Webserver leitet ihn auf HTTPS um. In der HTTPS-Verbindung wird ein zusätzlicher HTTP-Header gesendet, der bestimmte Informationen für den Browser enthält. Der Browser merkt sich das für den angegebenen Zeitraum. Bei zukünftigen Besuchen greift der Browser dann sofort auf die HTTPS Seite zu.

Es gibt folgende Keywörter:

- `max-age=63072000;` ? innerhalb dieses Zeitraums wird direkt HTTPS angesteuert (Angabe in Sekunden).
- `includeSubDomains;` ? der Eintrag gilt auch für sämtliche anderen Subdomains (vorsichtig damit, wenn andere Subdomains z.B. nicht per HTTPS funktionieren).
- `preload;` ? Google pflegt eine Liste mit Webseiten, die HSTS aktiviert haben. Diese Liste ist in aktuellen Versionen von Chrome, Firefox, Safari, IE11 und Edge enthalten. Diese Webseiten werden sofort per HTTPS angesurft.

Bei nginx muss dazu folgender Eintrag im HTTPS-Teil der Konfiguration gesetzt werden:

```
add_header Strict-Transport-Security "max-age=63072000; includeSubDomains; preload";
```

Nicht vergessen im Vhost für Port 80 die permanente Umleitung auf HTTPS zu konfigurieren. Wie das für nginx zu machen ist, habe ich hier beschrieben.

Danach muss nginx neu geladen/neu gestartet werden!

Wer die Behandlung von HTTP und HTTPS Traffic bei nginx in einem server-Block zusammenfassen möchte, sollte diesen Eintrag von mir beachten.

Logfiles dynamisch benennen

Diese Anleitung wurde unter Debian Squeeze mit nginx 1.2.1 aus den Squeeze-Backports getestet. Der orginal nginx aus Squeeze ist dafür zu alt.

nginx bietet die Möglichkeit Logfiles dynamisch anzulegen und durch die Verwendung von variablen dynamisch zu benennen.

dazu braucht der nginx-worker, der im Gegensatz zum nginx-master nicht als root sondern als www-data läuft, Schreibrechte im Logverzeichnis:

```
chown -R www-data.root /var/log/nginx
```

Außerdem muss das Verzeichnis /etc/nginx/html angelegt werden, da nginx auf dessen Existenz prüft. Warum ist mir unklar.

```
mkdir /etc/nginx/html
```

Jetzt kann die access_log Direktive (z.B. direkt in der nginx.conf) wie gewünscht angepasst werden. $host wird dabei durch den Namen ersetzt, den der Client aufgerufen hat.

```
access_log /var/log/nginx/$host.access.log;
```

mit „nginx -t“ wird die Konfiguration getestet und mit „nginx -s reload“ geladen.

nginx page caching

Bevor das Caching im Nginx konfiguriert wird, müssen die Seiten, die zwischengespeichert werden sollen oder die Upstream-Webserver, zuerst die passenden Header ausliefern. Das kann z.B. so aussehen:

```
Cache-Control: public, max-age=3600, must-revalidate
```

Was bedeutet das nun?

- public: die Datei ist nicht auf einen bestimmten User zugeschnitten, sondern kann an alle ausgeliefert werden
- max-age: Maximale Lebensdauer in Sekunden
- der Webserver darf den Content nicht mehr ausliefern, bevor er nicht eine auf eine neue Version geprüft hat und ein „304 Not Modified“ erhalten hat

Des weiteren sollte die Applikation nach Ablauf der Lebensdauer des gecachten Objekts korrekt antworten. D.h. es wird der Statuscode „304 Not Modified“ gesendet, wenn sich die Datei nicht geändert hat. Alternativ wird die Seite in der neuen Version geschickt (Code 200 OK) oder „404 File Not Found“ sollte es sie inzwischen nicht mehr geben.

Nginx wird jetzt so konfiguriert, dass er die Dateien in einem Filesystemcache zwischenspeichern kann. Idealerweise legt man den Cache in eine Ramdisk.

```
http {
  # ...
  proxy_cache_path /ramdisk/nginx_cache levels=1:2 keys_zone=cache:5m max_size=100m;

  server {
    # ...

      proxy_pass              http://upstream;
      proxy_cache             cache;
      proxy_cache_key         "$host$request_uri";
      proxy_cache_revalidate  on;
      # Optionally;
      # proxy_cache_use_stale error timeout invalid_header updating http_500 http_502 http_503 http_504;
    }
  }
}
```

Um so z.B. Bilder oder andere statische Dateien durch den Proxy-Nginx cachen zu lassen, kann im Upstream-Nginx diese Einstellung vorgenommen werden:

```
location ~* \.(js|css|png|jpg|jpeg|gif|ico)$ {
  expires max;
  log_not_found off;
}
```

Permanente Umleitung auf HTTPS

Es gibt mehrere Möglichkeiten auf HTTPS umzuleiten. Hier sind die Gängigsten aufgezeigt.

Per return im http-Serverteil, ist auch die schönste Lösung:

```
server {
  listen 80;
  listen [::]:80;
  server_name seite-x.de;
  return 301 https://$server_name$request_uri;
}
```

Eine unschöne Version mit if (ist dann nützlich wenn man den http- und https-Serverteil kombiniert):

```
if ($scheme = http) {
  return 301 https://$server_name$request_uri;
}
```

Oder per rewrite (auch nicht so schön, da die Domain 2x angegeben wird):

```
server {
  listen 80;
  server_name seite-x.de;
  rewrite ^/(.+) https://seite-x.de/$1 permanent;
}
```

Wenn alle Webseiten von HTTP auf HTTPS umgeleitet werden sollen ist dieses Beispiel nützlich. Es funktioniert als Standard-Vhost für alle Seiten, für die es keine explizite Konfiguration gibt:
<code>

```
server {
  listen 80 default_server;
  listen [::]:80 default_server;
  server_name _;
  return 301 https://$host$request_uri;
}
```

Query-String an PHP-FPM mit übergeben

Beim Anflanschen von PHP an nginx habt ihr vermutlich so etwas verwendet:

```
location / {
  try_files $uri $uri/ /index.php;
}
```

Damit auch der Query-String übergeben wird, sollte es folgendermaßen angepasst werden:

```
location / {
  try_files $uri $uri/ /index.php$is_args$query_string;
}
```

Reverse Proxy

Dieser Artikel beschreibt wie man den nginx-Webserver als Proxy vor einen Apache-Server (oder z.B. auch Tomcat) setzt.

Der Apache muss dafür so konfiguriert werden, dass er z.B. nur auf 127.0.0.1:8080 lauscht und nicht den Port 80 blockiert (/etc/apache2/ports.conf).

Im gewünschten nginx-Vhost werden dann die folgenden Einträge gesetzt:

```
server {
  listen       *:80;
  server_name  meineseite.de;

  location / {
    include proxy_params;
    proxy_pass http://127.0.0.1:8080/;
  }
  ...
}
```

Das include-File /etc/nginx/proxy_params sollte so aussehen (kann dann auch für andere Reverse-Proxies wiederverwendet werden):

```
proxy_redirect off;
proxy_set_header Host $http_host;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-Host $host;
#proxy_set_header X-Forwarded-Server $host;
proxy_set_header X-Forwarded-For $remote_addr;
#proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header X-Forwarded-Proto $scheme;

proxy_connect_timeout 300;
proxy_send_timeout    300;
proxy_read_timeout    300;
send_timeout          300;
```

Die nginx-Config kann mit „nginx -t“ getestet und mit „nginx -s reload“ neu geladen werden. Ab dann sollten alle Zugriffe auf „meineseite.de“ per Reverse-Proxy von 127.0.0.1:8080 geladen werden.

Reverse Proxy um HTTP Basic Auth zu entfernen

Manchmal kann es nützlich sein eine mit HTTP Basic Auth geschützte Seite auch ohne Anmeldung (z.B. automatisiert) abzufragen. So blende ich z.B. in meiner Visualisierung der Hausautomation (mit ioBroker) immer ein Livebild einer Webcam ein. Das Bild wird aber durch den Browser geladen (`img src=`). Bis Chrome 59 ging das noch per `http://<user>:<pass>@cam/bild.jpg`. Das wird ab nun, wohl aus Sicherheitsgründen, verhindert. Eine Alternative wäre es die Anmeldedaten für die Cam bei jedem Aufruf wieder einzugeben, ist aber unpraktisch. Daher übernimmt das nun ein kleiner nginx Reverse-Proxy für mich.

Die Lösung ist eigentlich einfach, es muss nur ein zustätzlicher Header an die Cam mitgeschickt werden: `Authorization „Basic dXNlcm5hbWU6cGFzc3dvcnQ=“`

Mein Proxy-Vhost lauscht auf Port 81, die Cam auf Port 80.

Der Base64-codierte Hash wird so erzeugt: `echo -n „username:passwort“ | base64`

Der Hash in diesem Beispiel sieht so aus: `dXNlcm5hbWU6cGFzc3dvcnQ=`

Das -n ist wichtig, sonst wird noch ein Zeilenumbruch angehängt und es funktioniert nicht.

```
#
# reverse proxy with authentication for meine-cam.de
#

server {
  listen 81;
  listen [::]:81;

  root /var/www/html;
  index index.html index.htm index.nginx-debian.html;
  server_name <hostname>;

  location / {
     proxy_pass http://meine-cam.de:80;
     proxy_set_header Host $host;
     proxy_set_header X-Real-IP $remote_addr;
     proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
     proxy_set_header Authorization "Basic <base64 hash einfügen>";
  }

  access_log /var/log/nginx/access.log;
```

```
    error_log /var/log/nginx/error.log;
}
```

Da damit natürlich die Authentifizierung umgangen wird, sollte man seinen Reverse Proxy anderweitig einschränken und zum Beispiel den Zugriff nur von einer bestimmten IP/Netz erlauben.

Rewrite einer Datei auf Datei-$Hostname

Angenommen man will mit einem Vhost mehrere Domains behandeln, da sich z.B. alle einen Documentroot teilen, aber für jeden Auftritt ein eigenes Design per CSS-Datei ausliefern, dann kann man sich mit diesem einfachen Rewrite helfen:

```
rewrite ^/design.css$ /css/$host.css permanent;
```

Jeder Aufruf von z.B. „http://foobar.meine-domain.de/design.css“ wird dann intern umgeleitet auf „http://foobar.meine-domain.de/css/foobar.meine-domain.de.css“.

Sichere SSL Konfiguration

Mit dem „Mozilla SSL Configuration Generator“ läßt sich aktuell wohl am einfachsten und schnellsten eine sichere Konfiguration für den nginx-Webserver erstellen.

Einfach Webserver, Version von Webserver und OpenSSL setzen und zwischen verschiedenen Cipherprofilen wählen, fertig ist eine Beispielkonfiguration.

Der Eintrag zu TLS im Mozilla Wiki ist auch sehr lesenswert.

> Alternativ habe ich hier noch meine (**veraltete**) Anleitung:

Die folgenden Empfehlungen richten sich nach den Empfehlungen von BetterCrypto.org und ihrem Applied Crypto Hardening PDF mit dem Stand vom 21.04.2016.

Folgendes kann man machen, um die Standard-SSL Konfiguration von nginx sicherer zu gestalten:

- Schwache Cipher und Protokolle abschalten
- Webserver soll die Reihenfolge der Ciphers vorgeben
- SSL-Komprimierung ausschalten
- HSTS aktivieren
- Permanente Umleitung von HTTP nach HTTPS aktivieren

Das folgende gilt für nginx unter Debian. Ich trage dabei nur die beiden Direktiven für SSL-Zertifikatsbundle (ssl_certificate) und SSL-Keyfile (ssl_certificate_key) in den jeweiligen server-Block ein. Die restliche SSL-Konfiguration setze ich global in /etc/nginx/nginx.conf (dort gibts bereits einen Teil „SSL Settings“, das Folgende einfach hinzufügen):

```
ssl_prefer_server_ciphers on;
ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
ssl_ciphers
'EDH+CAMELLIA:EDH+aRSA:EECDH+aRSA+AESGCM:EECDH+aRSA+SHA256:EECDH:+CAMELLIA128:+AES128:+
SSLv3:!aNULL:!eNULL:!LOW:!3DES:!MD5:!EXP:!PSK:!DSS:!RC4:!SEED:!IDEA:!ECDSA:kEDH:CAMELLIA128-
SHA:AES128-SHA';
ssl_dhparam dhparams.pem;
```

Auf Diffie-Hellman aufsetzende SSH-Cipher benötigen eine entsprechende Parameterdatei (ssl_dhparam). Die Standardgröße beträgt 1024 bit. Die Empfehlung nach der Logjam-Attacke sind mind. 2048 bit, besser 4096 bit. Die Datei wird folgendermaßen erstellt (bei 4096 bit dauerte es ca. 5-15 Minuten):

```
# 4096 bit
openssl dhparam -out dhparams.pem 4096
# 2048 bit
openssl dhparam -out dhparams.pem 2048
```

Danach muss nginx neu geladen/neu gestartet werden!

Tuning nginx

nginx Tuning

Nach Änderungen nginx neu laden (nginx -t && nginx -s reload):

```
# This number should be, at maximum, the number of CPU cores on your system.
worker_processes 16;

# Number of file descriptors used for Nginx. Should also be set in /etc/security/limits.conf
worker_rlimit_nofile 200000;

# only log critical errors
error_log /var/log/nginx/error.log crit

# Determines how many clients will be served by each worker process.
# (Max clients = worker_connections * worker_processes)
# "Max clients" is also limited by the number of socket connections available on the system (~64
worker_connections 4000;

# essential for linux, optmized to serve many clients with each thread
use epoll;

# Accept as many connections as possible, after nginx gets notification about a new connection.
# May flood worker_connections, if that option is set too low.
multi_accept on;

# Caches information about frequently accessed files. Try to experiment with those values.
open_file_cache max=200000 inactive=20s;
open_file_cache_valid 30s;
open_file_cache_min_uses 2;
open_file_cache_errors on;

# Disable access logs
access_log off;

# Sendfile copies data between one FD and other from within the kernel.
# More efficient than read() + write(), since the requires transferring data to and from the use
sendfile on;

# Tcp_nopush causes nginx to attempt to send its HTTP response head in one packet,
# instead of using partial frames. This is useful for prepending headers before calling sendfile
# or for throughput optimization.
tcp_nopush on;

# don't buffer data-sends (disable Nagle algorithm). Good for sending frequent small bursts of d
tcp_nodelay on;

# Timeout for keep-alive connections. Server will close connections after this time.
keepalive_timeout 30;

# Number of requests a client can make over the keep-alive connection. This is set high for test
keepalive_requests 100000;

# allow the server to close the connection after a client stops responding. Frees up socket-asso
reset_timedout_connection on;

# send the client a "request timed out" if the body is not loaded by this time. Default 60.
client_body_timeout 10;

# If the client stops reading data, free up the stale client connection after this much time. De
send_timeout 2;

# Compression. Reduces the amount of data that needs to be transferred over the network
gzip on;
gzip_min_length 10240;
gzip_proxied expired no-cache no-store private auth;
gzip_types text/plain text/css text/xml text/javascript application/x-javascript application/xml
gzip_disable "MSIE [1-6]\.";
```

TCP-Stack Tuning

Nach Änderungen „sysctl –system“ ausführen:

```
# Increase system IP port limits to allow for more connections
net.ipv4.ip_local_port_range = 2000 65000

net.ipv4.tcp_window_scaling = 1

# number of packets to keep in backlog before the kernel starts dropping them
net.ipv4.tcp_max_syn_backlog = 3240000

# increase socket listen backlog
net.core.somaxconn = 3240000
net.ipv4.tcp_max_tw_buckets = 1440000

# Increase TCP buffer sizes
net.core.rmem_default = 8388608
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216
net.ipv4.tcp_rmem = 4096 87380 16777216
net.ipv4.tcp_wmem = 4096 65536 16777216

net.core.default_qdisc=fq
net.ipv4.tcp_congestion_control=bbr
```

X-Frame-Options setzen

Über den X-Frame-Options Header (RFC 7034) kann eine Webseite den Browser anweisen, dass sie nicht (ggf. entfernt) über <frame> oder <iframe> geladen werden darf. Das soll sog. Clickjacking verhindern.

In nginx läßt sich das so setzen (innerhalb eines server-Blocks):

```
add_header X-Frame-Options "DENY";
```

Danach nginx reloaden/neu starten.

Folgende Optionen sind möglich:

- DENY ? Es ist keinerlei Einbettung per Frame erlaubt, egal welche Seite es versucht.
- SAMEORIGIN ? Es ist nur die Einbettung in Seiten vom gleichen Ursprung erlaubt.
- ALLOW-FROM ? Es ist nur die Einbettung in Seiten vom angebenen Ursprung erlaubt.

> aktuell wird ALLOW-FROM nur von Firefox und Internet Explorer / Edge unterstützt.

Zugriff auf .git und .svn Unterverzeichnisse verbieten

Mit diesem Einzeiler (in jeden server{}-Block einfügen) läßt sich der Zugriff auf eventuell vorhandene .git und .svn Verzeichnisse unterbinden:

```
location ~ /\.(svn|git) { deny all; }
```