

Nginx Secure Web Server

with HTTP, HTTPS SSL and Reverse Proxy Examples

Nginx is a secure, fast and efficient web server. It can be configured to serve out files or be a reverse proxy depending on your application. What makes this web server different from [Apache](#), [Lighttpd](#) or [thttpd](#) is the overall efficiency of the daemon, the number of configuration options and how easy it is to setup.

Nginx ("engine x") is a high-performance HTTP server and reverse proxy server. Nginx was written by Igor Sysoev for rambler.ru, Russia's second-most visited website, where it has been running in production for over two and a half years. Igor has released the source code under a BSD-like license. Although still in beta, Nginx is known for its stability, rich feature set, simple configuration, and low resource consumption. [Nginx](#)

Security methodology behind our configuration

In the following example we are going to setup some web servers to serve out web pages to explain the basics. The daemon will load a few mime include files, compress outgoing data in real time and set the expires header to reduce bandwidth of client cached traffic. Full logging is on, in the default Apache format with the addition of compressed file size and the amount of time the server took to fulfill the request. Finally, we are going to set up restriction filters by ip to limit access to the "/secure" directory structure where you might put more sensitive non-public data.

The security mindset of the configuration is paranoid. There are a significant amount of bots, scanners, broken clients and just bad people which will abuse your site if given the opportunity. These clients will waste your bandwidth and system resources for the sole purpose of personal gain. In response, we will not trust any client to access our server without first making sure all of the request parameters are met. This means the remote client must be asking for our site by the proper host name and must request any support files, like pictures and css, with the referrer headers properly set. Any deviation from these rules will lead to Nginx dropping the client's connection with a return code 404. Even though Nginx does have support for the mod_security module we prefer to make our own access rules. Note that even though these rules are strict, normal web traffic and bots like Google can access the site without issue.

Our goal is to setup a fast serving and CPU/disk efficient web server, but most importantly a

secure web server. This configuration will work for the latest version of Nginx as well as the development versions. For the purpose of this example we built the latest development version of Nginx from source.

Below you will find a few different example nginx.conf configuration files in scrollable windows. The formats are available to make it easier for you to review the code. They are all fully working configuration files with the exception of setting up a few variables for your environment like listen port or ip.

You are welcome to copy and paste the following working examples. Before using the configuration file take a look it and then scroll down this page to the section titled, "Explaining the directives in nginx.conf".

Option 1: Nginx http web server for static files

This is a basic webserver running on port 80 (http) serving out web pages. Though we have added quite a few security checks, this is as basic a server as you can get. On an AMD64 3GHz machine this config will easily serve out thousands of pages a minute.

```
#####
### Calomel.org /etc/nginx.conf BEGIN
#####
#
worker_processes      4;      # one(1) worker or equal the number of _real_
worker_priority       15;     # renice workers to reduce priority compared t
                                # machine health. worst case nginx will get ~2
#worker_rlimit_nofile 1024; # maximum number of open files

events {
#worker_connections 512; # number of parallel or concurrent connections p
#accept_mutex         on;   # serially accept() connections and pass to work
#accept_mutex_delay 500ms; # worker process will accept mutex after this d
}

http {

    ## Size Limits
    #client_body_buffer_size      8k;
    #client_header_buffer_size   1k;
    #client_max_body_size        1m;
    #large_client_header_buffers 4 4k/8k;

    # Timeouts, do not keep connections open longer then necessary to reduce
    # resource usage and deny Slowloris type attacks.
    client_body_timeout          5s; # maximum time between packets the client c
    client_header_timeout        5s; # maximum time the client has to send the e
    keepalive_timeout            75s; # timeout which a single keep-alive client
    send_timeout                 15s; # maximum time between packets nginx is all
```

If you dislike malicious ip addresses scanning your web server causing excessive errors then take a look at our [Web Server Abuse Detection perl script](#) to keep track of the abusers and block them in real time.

Option 2: Nginx serving only SSL and redirecting http to https

This example configuration is for a webserver that serves out SSL (https) traffic only. We will redirect all clients that try to go to port 80 (http) to port 443 (https) with a permanent 301 redirect. This type of redirect works for Google bot and other search bots too. We will also stop illegal linking and document hijacking. On a AMD64 3GHz machine this config can serve out thousands of fully encrypted https sessions per minute without a problem.

Notice we have set worker_priority at 15 to reduce the priority of the web server compared to system processes. We renice nginx so if the webserver starts using too many resources the system can still use CPU, memory and I/O to keep itself healthy. You never want to allow a public, externally accessible daemon to deny the system of resources. Case in point: Imagine your server is under a web based attack. The nginx daemon is using all of the system resources and other daemons are suffering. Your monitoring system and even users are reporting the system is sluggish and unresponsive. To find out what is going wrong you try to ssh into the box, but ssh times out because sshd does not have enough CPU time. We have been in this situation before and do not ever wish to be there again. But, does renice'ing nginx to a lower priority make the daemon slower? No. Nginx will work perfectly fine, including low latency sends, if the machine is not under a high load from some other process which has a higher priority. Worst case scenario is nginx at nice level 15 will be allowed to use no more then 25% system resources under high system load ($(20-15)/(20-0) = 0.25$ or 25%).

If you need help with setting up a SSL certificate with a certificate authority like Comodo check out the section below titled, "How to setup a SSL cert from Comodo through NameCheap for Nginx". If you want to learn more about SSL in general then check out our [Guide to Webserver SSL Certificates](#).

```
#####
### Calomel.org /etc/nginx.conf BEGIN
#####

#
worker_processes      4;      # one(1) worker or equal the number of _real_
worker_priority       15;     # renice workers to reduce priority compared t
                                # machine health. worst case nginx will get ~2
#worker_rlimit_nofile 1024; # maximum number of open files

events {
#worker_connections 512; # number of parallel or concurrent connections p
#accept_mutex         on;   # serially accept() connections and pass to work
#accept_mutex_delay 500ms; # worker process will accept mutex after this d
}

http {

## Size Limits
#client_body_buffer_size      8k;
#client_header_buffer_size    1k;
#client_max_body_size         1m;
#large_client_header_buffers  4 4k/8k;

# Timeouts, do not keep connections open longer then necessary to reduce
# resource usage and deny Slowloris type attacks.
    client_body_timeout        4s; # maximum time between packets the client c
    client_header_timeout      4s; # maximum time the client has to send the e
    keepalive_timeout          75s; # timeout which a single keep-alive client
```

Option 3: Nginx optimized for FreeBSD 11 with ZFS and OpenSSL 1.1.1

The following is an example of nginx, built from source, running on a FreeBSD 11 box with the ZFS file system. Nginx is built against OpenSSL v1.1.1 in order to take advantage of TLS v1.3 ciphers and because OpenSSL is between 2.3x to 6.7x times faster than LibreSSL using ChaCha20, AES-128-GCM and AES-256-GCM ciphers. Check out our [AES-NI SSL Performance](#) page for the results of a study of AES-NI acceleration using LibreSSL and OpenSSL.

The source build will install Nginx on FreeBSD 11 with the same paths as the pkg version of Nginx. Also, if you wanted to install the package version of nginx first (pkg install nginx), you can save a copy of the /usr/local/etc/rc.d/nginx startup script and then uninstall the pkg version of nginx before the following our build process. This way you can use the FreeBSD rc.d method to start, stop and reload our source built nginx daemon as well as put "nginx_enable="YES" in /etc/rc.conf for nginx to start on boot.

```
# install dependencies
pkg install git-lite pcre py36-brotli

# Download Google's Nginx brotli module for (.br) file support
cd /tmp && git clone https://github.com/google/nginx_brotli.git && cd /tmp/nginx_brotli

# Download OpenSSL v1.1.1-DEV with TLS v1.3 support
cd /tmp && git clone https://github.com/openssl/openssl.git openssl

# Download the latest Nginx
export VER=1.15.0; cd /tmp && curl -O https://nginx.org/download/nginx-$VER.tar.gz

# Build Nginx against OpenSSL and minimal module support

./configure --with-openssl=/tmp/openssl --with-openssl-opt=enable-tls1_3 --p
```

If you are doing a new install we recommend installing FreeBSD root filesystem to ZFS; [FreeBSD ZFS Root Install Script](#). ZFS's Adaptive Replacement Cache (ARC) will cache most recently used (MRU) and most frequently used (MFU) files in RAM which reduces read and write latency to microseconds without real time disk I/O.

The sendfile conundrum: Nginx with encryption and HTTP/2 can not make use of FreeBSD 11's newest iteration of asynchronous sendfile. The reason sendfile is so efficient is sendfile takes a file on the disk and sends the data directly to the network stack using a kernel socket without first making a copy in RAM. This direct path means sendfile can only be used for HTTP/1 (plaintext) transfers from a non-ZFS filesystem because the data is not modified in any way before being sent to the FreeBSD network stack. Our Nginx server is using https (TLS) to encrypt the files, HTTP/2 to transfer files in binary frames and ZFS ARC to store the files. Both TLS and HTTP/2 require a copy of the data from disk in system RAM to do these extra operations _before_ sending the data to the network stack. Nginx with HTTP/2 and TLS can not use sendfile because Nginx needs to manipulate the data and sendfile does not allow data manipulation. Finally, ZFS does not support sendfile calls. But, I read Netflix is using Nginx on FreeBSD and they are using sendfile?! Netflix and has built a custom, micro TLS stack into the Netflix version of sendfile on FreeBSD and Netflix is using the UFS2 filesystem. This allows Netflix to transfer data directly from the disk using sendfile, encrypt the data using TLS in Netflix's sendfile and pass that encrypted data directly to the network stack. FreeBSD 11 does not have Netflix's version of sendfile with TLS so FreeBSD's sendfile is of no use to our server. And, even if we had a copy of sendfile from Netflix, Nginx would still need a copy of data in RAM for HTTP/2 binary frame processing.

worker_cpu_affinity: In order to decrease latency and increase cache hits you may want to research pinning the Nginx process to a CPU core if, and only if, the server is lightly loaded. A highly loaded system may incur higher latency as the CPU core Nginx is bound to might already be used by another process thus Nginx has to wait until the bound CPU is free. Modern processors have a hierarchy of caches; blocks of on-chip memory which are faster to work with than system memory through the CPU's external, slow buses. Intel processors typically have three levels of cache: the first, quickest and smallest is the Level 1 or L1 cache. When a processor reads or writes memory data, the CPU checks whether the data is already in the L1 cache (this is a cache hit), in which case the CPU uses that copy in L1 cache. If the data is not in L1 (this is a cache miss), the CPU checks the next level, the larger but slower L2 cache; if the data is in L2, the cache controller delivers the data to the processor but also moves the chunk of L2 data containing the requested information into L1.

If the data was not in L2, then the process repeats with the yet larger and slower L3 cache (L3 is the only cache shared between Intel cores), again cascading the data through L2 and L1 cache hierarchy. All cache checks and copies take differing amounts of time and can leave the cache in different states, regardless of whether code was speculative or successful.

Some general notes on the FreeBSD nginx.conf: The ciphers our server will accept support both TLS v1.3 and TLS v1.2, but we will not accept connections from clients with lower TLS versions or any SSLv2/3 connections. One large output_buffer is most efficient for static data read from ZFS ARC. The output_buffers have been increased to one(1) megabyte which will hold up to eight(8) ZFS records; 128 kilobytes is the default recordsize of the ZFS filesystem. The location blocks are setup just like a firewall with a deny all rule first and specific location match blocks afterwards. If the client request does not match one of the location strings then nginx sends back an error 410, Gone. Error 410 tells the client the file is gone, has never existed and will never come back which is more absolute compared to an error 404. Location matching in this format is CPU efficient with the lowest latency to find a URI match. Nginx should use asynchronous I/O so we turn "aio" on. AIO is supported by default in the FreeBSD 11 kernel. Do not use "aio threads" on FreeBSD with ZFS as "aio on" through the kernel is faster and more efficient. The system level pcre on FreeBSD is compiled with JIT support so enable just-in-time compilation (PCRE JIT) which offers a performance boost to regular expression processing. Ideally you should have enough RAM to hold all of your static web files. Disable ssl_session_tickets and ssl_session_cache because enabling either option [will break PFS](#), Perfect Forward Secrecy.

ACCF Data Filter: The string "accept_filter=dataready" added to the https listen directive enables FreeBSD to buffer incoming connections until the client sends a full request. Make sure to enable accf_data_load="YES" in the /boot/loader.conf to load the ACCF_DATA(9) kernel module.

Socket Sharding: To lower latency and increase requests per second by 2 to 3 times, make sure to enable "listen: reuseport". For FreeBSD 11.2 and earlier, enable "reuseport" only when using a single nginx worker due to limits in FreeBSD 11's SO_REUSEPORT. For DragonFly BSD, FreeBSD 12 and later, as well as all versions of Linux, enable at least one worker per cpu core and enable "reuseport" to use SO_REUSEPORT_LB; the load balanced version of SO_REUSEPORT. Also, take a look at the [Socket Sharding in NGINX](#) performance study.

Please take a look at our [FreeBSD Network Tuning](#) guide. With the following Nginx configuration and some FreeBSD tuning we are able to serve out static files read from ZFS ARC with full HTTP/2 TLSv1.3 negotiation in less than ten(10) milliseconds on a local one(1) gigabit LAN with 0.2 ms latency.

```
#####
### Calomel.org /etc/nginx.conf BEGIN
#####
#
## Calomel.org  nginx.conf optimized for FreeBSD on ZFS
#

user www www;
worker_processes 1;
#worker_cpu_affinity auto;

pcre_jit on;

events { }

http {

    # ECDSA certificate
    ssl_certificate /certificates/calomel.org_ecc.crt;
    ssl_certificate_key /certificates/calomel.org_ecc.key;

    # RSA certificate
    #ssl_certificate /certificates/calomel.org_rsa.crt;
    #ssl_certificate_key /certificates/calomel.org_rsa.key;

    # ECDSA ciphers, TLSv1.3 and TLSv1.2
    ssl_ciphers TLS_CHACHA20_POLY1305_SHA256:TLS_AES_256_GCM_SHA384:TLS_AES_

    # ECDSA and RSA ciphers, TLSv1.3 and TLSv1.2

```

Option 4: Nginx reverse proxy to a few back end web servers

A reverse proxy is a type of proxy server that retrieves resources on behalf of a client from one or more servers. These resources are then returned to the client as though it originated from the reverse proxy itself. Reverse proxies are a great way to reduce the load on any one machine as well as help secure the web cluster from external influences. Some advantages for using a reverse proxy (source wikipedia) are:

- Reverse proxies can hide the existence and characteristics of the origin server(s).
- Application firewall features can protect against common web-based attacks. Without a reverse proxy, removing malware or initiating take downs, for example, can become difficult.
- A reverse proxy can distribute the load from incoming requests to several servers, with each server serving its own application area. In the case of reverse proxying in the neighborhood of web servers, the reverse proxy may have to rewrite the URL in each incoming request in order to match the relevant internal location of the requested resource.

- A reverse proxy can reduce load on its origin servers by caching static content, as well as dynamic content. Proxy caches of this sort can often satisfy a considerable amount of website requests, greatly reducing the load on the origin server(s). Another term for this is web accelerator. This technique is also used for the Wikipedia servers. A reverse proxy can optimize content by compressing it in order to speed up loading times.
- In a technique known as "spoon feeding",[2] a dynamically generated page can be produced all at once and served to the reverse-proxy, which can then return it to the client a little bit at a time. The program that generates the page is not forced to remain open and tying up server resources during the possibly extended time the client requires to complete the transfer.
- Reverse proxies can be used whenever multiple web servers must be accessible via a single public IP address. The web servers listen on different ports in the same machine, with the same local IP address or, possibly, on different machines and different local IP addresses altogether. The reverse proxy analysis each incoming call and delivers it to the right server within the local area network.

This config is for a reverse proxy server in front of a few back end web servers. The client will connect to the proxy and depending on the hostname and path the request the proxy will forward the request to the proper back end server.

You may also want to think about switching from Apache's reverse proxy configuration (if you currently use it) due to the amount of serious vulnerabilities that have come up. By using different operating systems and software throughout your organization you can better protect yourself from a single vulnerability taking out the entire company.

In the example we have also added the ability to cache data on the proxy so requests do not have to go all the way to the back end servers. Commonly ask for data is quickly served from the proxy reducing the load of your infrastructure. We have also added client request rate limiting. This is a good idea to limit bad clients from abusing your web servers. You can also configure Nginx to compress http calls back to the client in real time, thus saving bandwidth.


```
#####
### Calomel.org /etc/nginx.conf BEGIN
#####

worker_processes      4;      # one(1) worker or equal the number of _real_
worker_priority       15;      # renice workers to reduce priority compared t
                                # machine health. worst case nginx will get ~2
#worker_rlimit_nofile 1024; # maximum number of open files

events {
worker_connections 512; # number of parallel or concurrent connections p
accept_mutex        on;  # serially accept() connections and pass to work
#accept_mutex_delay 50ms; # worker process will accept mutex after this de
}

http {

    ## Size Limits
    #client_body_buffer_size      8k;
    #client_header_buffer_size    1k;
    #client_max_body_size         1m;
    #large_client_header_buffers  4 4k/8k;

    # Timeouts, do not keep connections open longer then necessary to reduce
    # resource usage and deny Slowloris type attacks.
    client_body_timeout          3s; # maximum time between packets the client c
    client_header_timeout        3s; # maximum time the client has to send the e
    keepalive_timeout            75s; # timeout which a single keep-alive client
    send_timeout                 9s;  # maximum time between packets nginx is all

```

Building the Nginx Reverse Proxy example

To make it easy we have included the source build line we used to make the reverse proxy above. It just removes all of the unneeded nginx modules and leaves only those which are needed for the proxy, caching and client rate limit restrictions.

```
make clean; ./configure --with-file-aio --without-http_autoindex_module
--without-http_browser_module --without-http_geo_module
--without-http_empty_gif_module --without-http_map_module
--without-http_memcached_module --without-http_userid_module
--without-mail_pop3_module --without-mail_imap_module
--without-mail_smtp_module --without-http_split_clients_module
--without-http_uwsgi_module --without-http_scgi_module
--without-http_referer_module --without-http_upstream_ip_hash_module && make
make install

```

For more information about OpenBSD's Pf firewall, CARP and HFSC quality of service options check out our [PF Config \(pf.conf\)](#), [PF CARP](#) and [PF quality of service HFSC](#) "how to's".

Building nginx from source

To get started, you need to first install nginx on your machine. The source code is available from the [nginx home page](#) and practically every distribution has pre-made packages if you prefer those. The install is easy and it will not take you much time.

We highly recommend you build Nginx from source. This way you can modify the code if you need to and make sure you apply the latest patches when they come out.

You need to make sure that the package for **PCRE** is installed when using OpenBSD. Use the command "pkg_add -i pcre" to install from your chosen PKG_PATH repository. BTW, you may want to also look at the Perl script [pkg_find](#) for OpenBSD package management.

OPTIONAL: Remove the Server: string of your host

The Server: string is the header which is sent back to the client to tell them what type of http server you are running and possibly what version. This string is used by places like Alexa and Netcraft to collect statistics about how many and of what type of web server are live on the Internet.

There is no practical reason to send the server string back to a client. The following window shows the two files you need to edit to completely remove the server: string from a Nginx header reply. The first file edit is takes care of standard http and https keepalive replies. The second file edit is to remove the server header from any https SPDY replies.

```

## EDIT FILE #1
## vi src/http/nginx_http_header_filter_module.c

# around line 48
# remove the nginx name and server version strings
static char ngx_http_server_string[] = "";
static char ngx_http_server_full_string[] = "";

# around line 280
# change "- 1" to "- 0:" on both lines 281 and 282 like this
if (r->headers_out.server == NULL) {
    len += clcf->server_tokens ? sizeof(ngx_http_server_full_string) - 0:
                                sizeof(ngx_http_server_string) - 0;
}

## EDIT FILE #2
## vi src/http/nginx_http_spdy_filter_module.c

# around line 174
# comment out the following function with /* and */ as this method is not
/*  if (r->headers_out.server == NULL) {
    len += ngx_http_spdy_nv_nsize("server");
    len += clcf->server_tokens ? ngx_http_spdy_nv_vsize(NGINX_VER)
                                : ngx_http_spdy_nv_vsize("nginx");
}
*/

```

OPTIONAL: anonymize you server string in the auto generated error pages

When nginx sends an error back to the client it can auto generate the error page. This error page has the error code at the top, a single horizontal line and then the string "nginx" and possibly the version number. If you want to you can take out the server string in the error page by editing the source code in the file **src/http/nginx_http_special_response.c** on lines 21 and 28. The following line would make the nginx generated error pages show your domain name for example.

This same file contains all of the default HTML error pages Nginx will send to the client if there is an error. Look for the functions that start with the line **static char ngx_http_error_** and make any changes you find necessary. Note that the HTML text is only shown to the user and that all errors sent by Nginx will have the proper error code in the HTML headers. This means you can put anything you want into the HTML code.

```

## vi src/http/nginx_http_special_response.c

# You can also change all of the built in error
# messages with just a carriage return.

static u_char ngx_http_error_full_tail[] = ""CRLF;
static u_char ngx_http_error_tail[] = ""CRLF;
static u_char ngx_http_msie_padding[] = "";
static u_char ngx_http_msie_refresh_head[] = "";
static u_char ngx_http_msie_refresh_tail[] = "";

static char ngx_http_error_301_page[] = "";
static char ngx_http_error_302_page[] = "";
static char ngx_http_error_303_page[] = "";
static char ngx_http_error_307_page[] = "";
static char ngx_http_error_400_page[] = "";
static char ngx_http_error_401_page[] = "";
static char ngx_http_error_402_page[] = "";
static char ngx_http_error_403_page[] = "";
static char ngx_http_error_404_page[] = "";
static char ngx_http_error_405_page[] = "";
static char ngx_http_error_406_page[] = "";
static char ngx_http_error_408_page[] = "";
static char ngx_http_error_409_page[] = "";
static char ngx_http_error_410_page[] = "";
static char ngx_http_error_411_page[] = "";
static char ngx_http_error_412_page[] = "";
static char ngx_http_error_413_page[] = "";
static char ngx_http_error_414_page[] = "";
static char ngx_http_error_415_page[] = "";

```

OPTIONAL: modify the tls record size

TLS record size can have significant impact on the page load time performance of your application. The the server is transmitting static data then a small buffer will incur extra processing. If the buffer is too large then a dynamic site with small updates will delay the client from Time To First Byte (TTFB).

We suggest setting a large TLS record size if you are serving large static files. The larger buffer will MAC sign larger chunks of data and get them sent out to the wire faster. The speed is gained by not having to split a large file into many 16KB chunks and also sign them. Instead, we split the file into larger 256KB chunks leading to less server and client load when verifying each chunks MAC.

On the other hand, if the server sends out small real time updates then you might want to reduce the buffer size so the entire payload and MAC can fit into one packet. A Google engineer wrote a piece title, [Optimizing NGINX TLS Time To First Byte \(TTTfB\)](#), and he mentioned Google sets their buffer at 1400 bytes.

```
# Option 1: increase the tls buffer for static sites to reduce server
           and client processing

## vi src/event/nginx_event_openssl.h (around line 109)

- #define NGX_SSL_BUFSIZE  16384
+ #define NGX_SSL_BUFSIZE  262144

# Option 2: decrease the tls buffer for dynamic sites decreasing the
           Time To First Byte (TTFB). A smaller buffer will increase
           processing time on both the client and server due to small
           signed MAC "packages".

## vi src/event/nginx_event_openssl.h (around line 109)

- #define NGX_SSL_BUFSIZE  16384
+ #define NGX_SSL_BUFSIZE  1400

## vi src/event/nginx_event_openssl.c (around line 572)

- (void) BIO_set_write_buffer_size(wbio, NGX_SSL_BUFSIZE);
+ (void) BIO_set_write_buffer_size(wbio, 16384);
```

OPTIONAL: change any of the default error codes

Normally you DO NOT want to change any of the standard error codes specified by RFC. But, in case you really need to you can edit the file **src/http/nginx_http_request.h** and look for the variables starting with NGX_HTTP_REQUEST. For example, if we wanted to change the default error code for REQUEST_URI_TOO_LARGE from 414 to 999 we could:

```
vi src/http/nginx_http_request.h (line 83)
- #define NGX_HTTP_REQUEST_URI_TOO_LARGE 414
+ #define NGX_HTTP_REQUEST_URI_TOO_LARGE 999
```

Compiling the code for the static server with SSL support

Building nginx for the AMD64 architecture

As a general example, Nginx can be built with the following arguments. Make sure to check if you need to use one of the modules that we omit during the build. We use the methodology, "if you do not need it then do not build it in." Our example nginx.conf (option 1 and 2) works fine with the following:

```
make clean; ./configure --with-http_ssl_module --with-http_spdy_module
--with-http_gzip_static_module --with-file-aio --without-http_autoindex_mod
--without-http_browser_module --without-http_fastcgi_module --without-http_
--without-http_empty_gif_module --without-http_map_module --without-http_pr
--without-http_memcached_module --without-http_ssi_module --without-http_us
--without-mail_pop3_module --without-mail_imap_module --without-mail_smtp_m
--without-http_split_clients_module --without-http_uwsgi_module
--without-http_scgi_module --without-http_limit_conn_module
--without-http_referer_module --without-http_cache
--without-http_upstream_ip_hash_module && make && make install
```

Once Nginx is built and installed in place it is time to take a look at the config file.

Explaining the directives in nginx.conf

Now we need to edit the config file for your environment. Lets take a look at each of the directives that need attention.

pid /var/run/nginx.pid : This is the location of the process id file that holds the pid number of the master Nginx process. If you wanted to re-read the nginx.conf file without restarting the daemon you could cat this file and send a HUP like so, "kill -HUP `cat /var/run/nginx.pid`".

user nginx nginx : Is the user and group the child processes will run as. You may need to make this user and group if you install Nginx from source. Make sure this user is completely unprivileged or at least runs with the least privileges necessary to make the server work. You can also run nginx as the generic "user daemon daemon".

worker_processes : Is the number of workers to spawn. A worker is similar, but not exactly like, a child process in Apache. Apache uses multiple threads to provide each client request with its own thread of execution to avoid blocking when using synchronous I/O. Nginx takes advantage of asynchronous I/O, which makes blocking a non-issue. The only reason nginx uses multiple worker processes, is to make full use of multi-core systems. Even with SMP support, the kernel cannot schedule a single thread of execution over multiple CPUs. Nginx requires at least one worker process per logical CPU core. So, the difference between Apache and the Nginx model is, nginx requires only enough worker processes to get the full benefit of SMP, whereas Apache's architecture necessitates creating a new thread (each with it's own stack of around ~8MB) per request. At high concurrency, Apache will use much more memory and suffer greater overhead from maintaining large numbers of threads when compared to Nginx.

How many worker_processes should use? A single worker processes can handle thousands of requests per second. Once a cpu core reaches 100% then nginx will slowdown. A simple rule is to set the worker_processes equal to the amount of REAL cpu cores in the machine. If you have a low volume web server, set the worker_processes to one(1) and turn accept_mutex off for efficiency.

When running Nginx as an SSL server or SSL terminator, you must use more than one(1) worker_processes in order to avoid CPU core blocking. Nginx workers block on the disk I/O and the SSL handshake. When a worker is accessing disk I/O the CPU can do other tasks as well, interleaving jobs. In contrast, the SSL handshake is a CPU core locking event meaning the CPU core can not do anything else until the CPU time share slice ends for the current process/thread. For example, a typical time share slice is 1ms and if we have a 2048 bit key SSL handshake negotiation time of 0.5ms, then the chances another process/thread could be blocked on the same CPU core is 50%. This means that 50% of the time you could see contention locking and delays if a worker process lands on the same physical CPU core. The "fix" is to make sure you set the worker_processes to the same value as the amount of real CPU cores spreading the load.

For speed testing, we suggest using the httpperf or Apache benchmark binary (ab) to stress your server and see how many connections your machine can handle. "ab" can be found in any apache_utils install and httpperf is a stand alone package. To calculate how many total concurrent connections nginx can support, multiply "worker_processes" times "worker_connections". Our example is setup to handle $4 \times 1024 = 4096$ total concurrent connections. Clients who attempt to connect after 4096 concurrent clients are already connected will be denied access. It is better to deny clients than overload the machine possibly causing a DOS. Make sure not to set these values to high as your machine might just crumble under the strain of too many clients. Take a look for our httpperf testing procedure lower down on this page.

worker_rlimit_nofile : is the maximum number file descriptors (ulimit -n) that can be opened by EACH worker_processes. In the case of Nginx this translates to the amount of open network connections to remote clients in addition to proxied backend connections. Understand that the open file limit is really regulated by the operating system. This directive simply allows Nginx to try to set "ulimit -n value" when nginx starts. When you set this directive you will not need to set a separate "ulimit -n 'value' " in the OS. Default value (the ulimit -n value) will be overridden. If worker_rlimit_nofile is not specified, your default ulimit -n number for the user who is running nginx will take effect. It is important to note the worker_rlimit_nofile value should be greater or equal to worker_connections. If your open file limit is too low for the amount of connections nginx is making you will see the error, "Too many open files" in the error log. For example, OpenBSD has a really low open file limit of 128 files for a normal user; i.e. not root. You will want to set this directive to at least 1024 to avoid errors. You will also want to make sure that if you are keeping connections open for a long time with keepalive statements to watch this value. Each open connection is a used open file descriptor and thus counts against your "ulimit -n" value.

worker_connections : This is the amount of client connections a single child process will handle by themselves at any one time. (default: 1024) Note: Multiply worker_processes times worker_connections for the total amount of connections Nginx will handle. Our example is setup to handle $4 \times 1024 = 4096$ concurrent connections in total. Clients who connect after the max has been reached will be denied access.

Why not set the worker_processes and worker_connections really high ?

If your server sits idle most of the day it is better to increase worker_processes and reduce worker_connections. The reason is worker_connections will recycle and clean up better and you will not run into possible serial issues. However, if the server is consistently busy you

should increase `worker_connections` and reduce `worker_processes` so the server is not trying to do garbage collection too fast or too soon.

In truth, we found during testing that a 4 core server works quite well with only one(1) to three(3) `worker_processes`. We leave at least one core for the system and its interrupts and the other cores for the web server. As for `worker_connections` we found that a server responding to two thousand (2000) requests per second worked perfectly fine with only 64 `worker_connections` if you have a short timeout period. If you expect clients to connect and keep the connection active for long periods of time you will want to increase the `worker_connections` value.

The best idea is to test your server with real data in its final configuration with a tool like `httpperf`. We have an example of `httpperf` uses lower down on this page.

accept_mutex : `accept_mutex` is used to serialize `accept()` syscalls thus making the division of labor to the `worker_processes` more efficient. If you have more than two(2) `worker_processes` then keep `accept_mutex` on. The other solution is to make "`accept_mutex off`;" and allow the sockets to be non-blocking. In this case the `accept` call won't block the children, and they will be allowed to fight for the `accept()` request immediately. But this fighting wastes CPU time by about 1% on average and makes the system a little less efficient. For example, if `accept_mutex` is off then suppose you have ten idle `worker_processes` in `select`, and one connection arrives. Then ten of those children will wake up, try to accept the connection, nine(9) will fail, and loop back into `select`, accomplishing nothing. In the mean time, none of those nine(9) children are servicing requests that occurred on other sockets until they get back up to the `select` again. The only time you may want to turn `accept_mutex` off is if you only have one(1) child process and are expecting moderate to low traffic to the site. By moderate traffic we mean less than around 30 requests per second. Since you are only using a single worker processes there is no need to use a locking mutex and thus server accepts will be faster.

multi_accept : `multi_accept` tries to `accept()` as many connections as possible after `nginx` gets notification about a new connection. By default this is turned off to make sure a single remote client can not take up more than their fair share of the server's connections. The purpose of `multi_accept` is to accept all connections from listen queue all at once. Normally, only one connection will be accepted on each return from event function. In the worst case scenario, with `multi_accept "on"`, if you have constant stream of incoming connections at a high rate the connections may overflow your `worker_connections` and they will not have a chance to process the previously accepted connections. Keep in mind that "`multi_accept on`;" is ignored for `kqueue` which is used on `OpenBSD` and `FreeBSD` as there is a limit of accepted connections set by the kernel, and `nginx` accepts all connections up to this limit. It is highly recommended to limit clients on the server using the `limit_req_zone` and `limit_req` directives like in the examples above. This will help keep malicious clients at bay.

MIME types : This section allows `nginx` to identify files by extension. For example, if we serve out a `.txt` file then the mime type would be defined as `text/plain`.

include mime.types is the definition file `nginx` loads to identify all of the mime types. These directive simply allow our server to send the the proper file type and application type to the clients. Alternatively you can take out this line and instead define your own Mime types by using the following "`type`" directive".

types {...} Instead of using the "`include mime.types`" directive you can define your own

mime types. This is especially useful option if you want to use the same mime types on many different systems or do not want to rely on a secondary definition file. You also have the option of defining a mime type for a non-standard extension. In our example we define the extension "bob" as a text/plain.

default_type application/octet-stream is the default type if a file extension has not already be defined in the mime.types file. This is useful if you serve out files with no extension or of a non standard extension. Either way, clients will be able to retrieve the file un-obstructed.

Size Limits : These directive specify the buffer size limitations on the amount of data we will consider to be valid for a request. If the client sends to much data in one request, for example in a buffer overflow attack, then the request will be denied.

client_body_buffer_size If the request body is more than the buffer, then the entire request body or some part is written in a temporary file.

client_header_buffer_size is the limit on the size of all of the http headers the client can send to the server. For the overwhelming majority of requests a buffer size of 1K is sufficient. The only time you would need to increase this is if you have a custom header or a large cookie sent from the client.

client_max_body_size is the maximum accepted body size of client request, indicated by the line "Content-Length" in the header of request. If size exceeds this value the client gets sent the error "Request Entity Too Large" (413). If you expect to receive files uploaded to your server through the POST request method you should increase this value.

large_client_header_buffers is the limit of the URI request line which can not be larger than the buffer size multiplied by the amount of buffers. In our example we accept a buffer size of 1 kilobyte and there is only one(1) buffer. So, will not accept a URI which is larger than (1x1K=1K) 1 kilobyte of data. If the client sends a bigger request then Nginx will return an error "Request URI too large" (414). The longest header line of the request must also be less than the size of (1x1K=1K) 1 kilobyte, otherwise the client get the error "Bad request" (400). Limiting the client URI is important to keep a scanner or broken client from sending large requests and possibly cause a denial of service (DOS) or buffer overflow.

Timeouts : These values specify the amount of time in seconds that Nginx will wait for the client to complete the specified action. A keepalive of 300 seconds is a good value as browsers will wait from 120 to 300 seconds to drop the keepalive connection. If you close the connection and browser thinks it is still open the result to the client will look as if the web site is responding slowly; in truth the browse sends request over a closed keepalive connection, has to timeout and then resend the request. Also, 300 seconds is the timeout of most SSL keys. For the other timeout 60 seconds (default of 60 seconds) is normally fine.

client_body_timeout is the read timeout for the request body from client. If after this time the client sends nothing, nginx returns error "Request time out" (408). You may want to lower this value to around 5 seconds protect yourself from attacks like Slowloris DoS attack explained lower on this page.

client_header_timeout is the timeout reading the title of the request of the client. If after this time the client send nothing, nginx returns error "Request time out" (408). Just like stated before, this value can be lowered to as little as 5 seconds to help mitigate attacks like the Slowloris DoS attack explained lower on this page.

keepalive_timeout the first value is for keep-alive connections with the client. The second parameter assigns the value "Keep-Alive: timeout=time" in the header of answer. With the complexity today's websites keep-alive are critical to keeping clients load times to a minimum. Establishing a TCP connection is expensive in terms of time. It is efficient for a client to establish its first set of connections (Firefox makes 4 for example) and request all of the objects on the page through those connections. A problem with keepalives to be aware of is every browser, and every version of each browser, has a different timeout the use for keep alives. Firewalls also have their own connection timeouts which may be shorter then the keep alives set on either the client or server. This means browsers, servers and firewalls all have to be in alignment so that keeps alives work. If not, the browser will try to request something over a connection which will never work which results in pausing and slowness for the user. Google Chrome got around this timeout issue by sending a keepalive every 45 seconds until the browser's default 300 second timeout limit. You can do your part by setting firewall timeouts no less then 600 seconds and allowing clients to keep a connection open to your server for at least 300 seconds.

send_timeout is response timeout to the client. Timeout is established not on the entire transfer of answer, but only between two operations of reading, if after this time client will accepts nothing, then nginx is shutting down the connection. You may want to look at lowering this value (5 seconds is ok) if you have malicious clients opening connection and not closing them like in the Slowloris DoS attack explained lower on this page.

Want more speed? Make sure to also check out the [Network Speed and Performance Guide](#). With a little time and understanding you could easily double your firewall's throughput.

General Options :

ignore_invalid_headers on throws away non-standard headers in the client request. If you do not expect to receive any custom made headers then make sure to enable this option.

keepalive_requests are the number of requests which can be made over a keep-alive connection. If a client makes 4 connections to your server (Firefox's default is 4 initial connections for example) they have 4 keepalive channels. If you set keepalive_requests to 20 like in our example above that would give a client 80 objects to request through those 4 connections. Keep alive requests are a lot faster and more efficient then having the client make 80 individual TCP connections. Try to set your keepalive_requests high enough so that a browser can request at least double what an average user requests in 300 seconds (keepalive_timeout 300 300;). So, lets say I have a web site and the average amount of objects (pictures, css, html, js, ect.) is 20 per page. Lets also say that the average browser looks at 2 pages in 300 seconds (5 minutes). You may want to keep keepalive_requests set at 20 as this would serve 2 pages times 20 objects which equals 40 requests in 300 seconds easily. Remember keepalive_requests at 20 times 4 connections was 80 requests total. Why not just set keepalive_requests really high ? You always want to put a limit on clients in case a malicious user attacks your site.

keepalive_disable by default Nginx disables keepalive requests from both Safari and MSIE6. There is a known bug in the Mac OSX TCP stack which has problems uploading (the POST command) to web servers which have keepalives enabled. The result are pauses and stalled connections seen by the Safari client. This bug with OSX has been known about since 2005 with no fixes proposed. This means that Apple products like the iPod, iPad and iPhone as well as Mac OSX machines will have to make new TCP connections for every object on your site. Creating a separate TCP connection for every item is quite slow. The good news is if your server does not allow the client to upload (POST) data you can enable keepalives with Safari like we did in the examples above. You may want to test your server with Safari to test if you see any issues. It is up to you, but there are enough Safari browsers on the market to argue that we are willing to risk some Safari users experiencing pauses compared to increasing the load speeds of all other Safari users. Make an informed decision if you really want to enable Safari to do keepalives for your site.

max_ranges clients use the range header to download parts of a file. This method is useful for transferring large files to clients who get disconnected during a transfer. Lets say a client is downloading a 1 gigabyte file from you and they get 80% and then get disconnected. If max_ranges is set to at least one(1) the client can send a request for the same file, but starting at 80% so they do not have to start all over again. We recommend setting max_ranges to no more then 1 for this situation. The times you do not want to use max_ranges is if you only serve out small files. For a small static server which sends out small files, sizes less then 1 meg for example, set max_ranges to 0. Another hazard to be aware of is not to set the max_ranges high. If you allow max_ranges to 10 then a client could request a single file split into 10 parts. Request 1 asks for 0%-10%, request two asks for 11%-20%, ect. This behavior is what "download accelerators" do to attempt to speed up their downloads and this can put quite a heavy burden on your server. On a side note, Apache had a vulnerability which included abuse of the range header. You can search on Google for, "Apache Range Header DoS Attack" for more information.

open_file_cache is nginx's ability to cache file inode locations in ram for quick access to the files and distribution to the clients. Understand that the file itself is not cached, just the pointer to the inode.

How does open file caching work? When nginx needs a file it asks the file system, which then

looks up the location for the inode and then the file is retrieved. By using `open_file_cache` you bypass the slowest step which is the file system look up for the inode. A quick analogy is when you are looking for a web page. If you do not know what the URL is then you search through Google for page and then click on the link for the URL; this is slow just like when you are asking the file system to go find the inode pointing to a file. A fast way is to already have the web page bookmarked and this is just like having the inode location of the file in cache.

This directive uses two arguments; "max" is the maximum amount of objects in the cache and inactive is the amount of time an object can stay in cache. Understand that an object can be removed from cache if the inode has changed and the timeout set on "open_file_cache_valid" was reached. If "max" is hit the oldest, least used object is removed to make room for the new object. Cached files can significantly increase the speed at which nginx can find the object the client requested and begin the transfer to them.

We highly suggest using caching if it works in your environment. But, you have to be aware of some problems caching could cause when you make changes to files. The `open_file_cache` directive looks at the inode of a file to see if it has changed or not. If you go in and edit a file and save it as the same name the inode is the same. So nginx may not catch the change and continue serving out the old version of the file till the "invalid" timeout is reached. The other item to remember is that every worker process has its own cached items. If you change a file and one worker has a cached copy it will serve that out. If another worker did not have a copy then it will pick up the new version of the file. So, it is possible that you could have multiple versions of the same file being served at the same time. Truthfully, this is not a problem for the speed you gain by using caching and there are some was around the delay. When you edit a file you could just wait till the "open_file_cache_valid" timeout for nginx to check if the file is still the same. You could also copy the original html file you want to edit, edit the file and then mv (move) the new copy on top of the old file. This would change the inode, update the cache and keep the same file name.

open_file_cache_min_uses is the amount of times a file needs to be requested before it is added to the `open_file_cache`. The files also need to be requested this many times within the "inactive" time frame in the `open_file_cache` directive. So if "inactive=1h;" like in the example above and `open_file_cache_min_uses` is 3 then any client needs to request the same file at least three(3) times in 1 hour for the open file pointer to be cached. The default is 1 for this directive which is fine for most uses. If you also enable `open_file_cache_errors` then set this directive a little higher. No need to have our cache filled with errors which were only called once.

open_file_cache_errors is simply allowing nginx to cache errors as well. Errors could be any request for a file that does not exist for example. This may be a good idea to keep confused clients or scanners from forcing your server to do I/O calls to the disk for the same invalid URLs.

open_file_cache_valid is the amount of time before nginx is going to go check if a file, which is identified by its inode and is currently in cache, has changed or not. If you are using caching then you are trying to reduce the amount of disk I/O. Set this value high enough so that nginx does not have to check the disk over and over for changed files. This is especially true for static sites. You may want to set this value to tens of seconds to make sure changes are seen or many hours for a highly static site. BTW, if an object is in a worker's cache and has not changed in this directive's time frame then it will continue to stay in the cache until "open_file_cache invalid=(time)" timeout.

output_buffers 1 512k; tells nginx to use one(1) 512KB chunk output buffer if, and only if, sendfile is off. The default size is "1 32k" and you may want to disable sendfile if you are sending large files to clients and you are seeing disk lock contention and slowdown due to insufficient disk IO speed. The output_buffers will instead be used to buffer the data to be sent back to the client. Search on this page for "sendfile" for more information.

postpone_output is the amount of data Nginx will get ready before sending out a packet to the client. By default postpone_output is set to 1460 bytes which should be fine in most cases, but you might want 1440. The reason is if you are using [TCP segmentation offload \(TSO\)](#) on your network card then your nic is going to segment that stream of packets in 1448 byte chunks to put on the network. If you make your packet stream slightly smaller you will notice the nic does not have to re-segment the stream and latencies will be lower. You will want to set this value to your Maximum Segment Size (MSS) of your external network interface. For example, if your MTU is 1500 then your MSS is 1460 or 1440 to be safe. If you have a MTU of 9000 then set this value to 8192. Test to make sure you send out full packets, but do not fragment. Understand that if a remote client has a smaller MTU your server will send the packets at a small MSS. Not much you can do about those cases. You can use "tcpdump" to look at MSS sizes.

limit_req_zone \$binary_remote_addr zone=gulag:1m rate=60r/m; sets up a table we will call "gulag" which uses no more than 1 megabyte of ram to store session information keyed by remote ip address. This directive is used in conjunction with **limit_req zone=gulag burst=200 nodelay;**. The ngx_http_limit_zone_module restricts the amount of requests an ip address can make. An error 503 will be returned to the client if request processing is being blocked at the socket level and new requests from the same ip continue. Limiting requests is a good way of keeping a single client ip address from hammering your server. This directive will limit requests no matter how the client connects; for example they could connect through many individual TCP connections or use only a few TCP connections and pass many keepalive requests. This directive currently does not take effect on SPDY connections, but should in the future when the SPDY patch is joined into the nginx source.

This is not a directive to limit the total number of open, "established" connections to the server per ip address!! You could use your iptables or PF firewall to limit the total amount of connections. The [OpenBSD and FreeBSD PF firewall \(pf.conf\)](#) uses max-src-conn or max-src-states to limit the amount of established connections to your server.

You can increase the size of the "gulag" table from 1 megabyte if you need to. A zone size of 1M can handle 32000 sessions at a default size of 32 bytes/session. You can also change the name of the table we called "gulag" to any string you want. We thought this was a good name due to Nginx's country of origin combined with the purpose of this directive.

The HTTP 1.1 specification, circa 1999, recommends that browsers and servers limit parallel requests to the same hostname to two. Most browsers comply with the multi-threading recommendation of the specification, although downgrading to HTTP 1.0 boosts parallel downloads to four. So most web browsers are effectively throttled by this limit on parallel downloads if the objects in the web page they download are hosted on one hostname. We set this limit to 5 so browsers can open 4 connections with one slot left over as a buffer. Download accelerators can open many hundreds of connections to download a file so this directive will help to alleviate abuses.

recursive_error_pages allows the use of the error_pages directive specified later in the config.

reset_timeout_connection will make sure nginx closes stale client connections. If our server is done talking to the client or if the client is unresponsive, we should make sure the connection is closed and resources are released which frees up socket-associated memory. Bad, lazy or abusive clients may try to open up many connections and keep them open to use up available sockets and thus deny any future clients from connecting.

sendfile enables the use of sendfile() function. For FreeBSD reading from ZFS make sure sendfile is OFF to avoid redundant data caching. Sendfile relies on the OS to do IO and this may be a good idea if you are serving small files from a fast disk. With the sendfile() zero-copy approach, the data is read immediately from the disk into the OS cache memory using Direct Memory Access (DMA) hardware, if possible. The TLB cache is left intact. The performance of applications utilizing sendfile() primitive is great because the system call does not directly point to memory and, therefore, minimizes performance overhead. Data to be transferred is usually taken directly from system buffers, without context switching, and without trashing the cache. Thus, the usage of sendfile() in server applications can significantly reduce CPU load. Now, sendfile is not without problems. Nginx uses the sendfile asynchronous model concept without regard to disk I/O. The disk might be busy with a lot of simultaneous reading or writing requests (keep in mind limited random seek speed at ~130 random operations per second for spinning platters) and when Nginx process tries to read data from that disk, it freezes and nginx comes to a halt till the disk responds. Not good. You also need to be careful about using sendfile if the file being sent has any possibility of being modified (especially truncated) while the operation is in progress since some odd behavior (like the process crashing) can happen on some platforms. You may just want to test your server with sendfile on and off. We like to keep sendmail on as our site is serving small static files from a combination of a SSD drive and RAM disk.

sendfile_max_chunk 512K set the maximum amount of data the system will read in a single sendfile() operation. 512 kilobytes is an example if every object, like jpg picture or html page are smaller then this value. On a fast local connection sendfile() may send tens of megabytes per one syscall blocking other connections. sendfile_max_chunk allows to limit the maximum size per one sendfile() operation. This problem is especially evident in Linux, but not really a problem on FreeBSD. You may want to set this directive at least as large as your largest object on the page, like your biggest jpg picture. If you are serving large uploads to your clients you will want to test out some values depending on how fast your disk I/O is and how many concurrent users you get. Start low and keep increasing until you start to see nginx slowdown due to wait states on the disk I/O or you see nginx increasing its response times. On linux "dstat" is a nice tool. You are looking for the limits on your hard drive which will be thrashing due to over saturation of the bus or random read limitations. Remember nginx is a non-blocking asynchronous program and the _only_ time nginx will start to be blocked is when the hard disk I/O is saturated. Nginx will wait for the disks to respond and in that time the hardware is blocking nginx and your internet clients are waiting on you.

server_name_in_redirect if off disables the server's ability to substitute the client supplied "Host" header with the virtual server variable "server_name" when a client is redirected.

server_tokens if off disables nginx's version numbers in the auto generated error pages. We do not want to display this information for security purposes.

TCP options : These options say how we should use the TCP stack.

tcp_nodelay TCP_NODELAY is for a specific purpose; to disable the Nagle buffering algorithm. It should only be set for applications that send frequent small bursts of information without getting an immediate response, where timely delivery of data is required (the canonical example is mouse movements). By default tcp_nodelay is on and this is a good setting as most web page objects are small and sending them out quickly reduces load times.

tcp_nopush If set to on, do not send out partial frames. This option is useful for pre-pending headers before calling sendfile(2) and for throughput optimization. As currently implemented, there is a 200 millisecond ceiling on the time for which output is corked by TCP_CORK. If this ceiling is reached, queued data is automatically transmitted. tcp_nopush is off by default. We highly recommend testing your server with this option enabled. We noticed a decrease in latency of at least 100 milliseconds with most real world clients when we enabled tcp_nopush.

Compression : These values tell nginx how to compress outgoing data. Remember that all files of the specified mime.type (gzip_types) are compressed in real time. On a P3 500MHz a 100KB HTML file takes 0.05 seconds (5 hundredths of a second) to gzip at level 9 compression (highest).

gzip on turn real time compression on. You may want to disable this option if you decide to use the more efficient gzip_static instead. You always want to gzip compress your HTML pages as most clients have plenty of extra CPU cycles, but limited internet bandwidth. Also, using this option to compress data in real time adds a response delay back to the client. If you are compressing the same HTML code to every client then this is just wasting time. Disable this real time gzip and use gzip_static instead. Then you can pre-compress your html files and increase your response times.

gzip_static on; allows one to have pre-compressed .gz files served instead of compressing files on the fly. This is the most efficient method of serving compressed data. To use this option simply have a compressed copy of the same .html file in document root. For example, if we have the index.html file in place we will also have a pre-compressed index.html.gz file. You will have a non-compressed copy for older clients which do not accept compression and a pre-compressed copy for all other clients. gzip_static does not depend on the gzip filter nginx module so you can use gzip_static without compiling the gzip filter. gzip_static can also be set to always to send out the compressed file no matter if the client specifies compression or not.

The following script will publish a compressed gzip file from a given html file. When you are done editing the html file execute this script to make a compressed copy ready for distribution. As soon as it is in place Nginx will serve it out. Also, make sure the date on the compressed .gz is always newer or equal to the original as Nginx will always serve out the most recent copy:

```
#!/bin/sh
#
## Calomel.org  publish_html2gz.sh
## usage: ./publish_html2gz.sh index.html
#
## Make a tmp copy of the original HTML file
cp $1 $1.tmp

## Remove the old gz if there is one
rm -rf $1.gz

## Brotli compression defaults to level 11.
#brotli-3.6 -f -i $1.tmp -o $1.br

## Compress the tmp HTML copy. Use level 9
## compression and do not store dates or file names
## in the gzip header. BTW, if the compressed gz is
## larger then the original file a gzip will NOT be made.
gzip -9 -n $1.tmp -o $1.gz

## Clean up any tmp files
rm -rf $1.tmp

echo ""
echo "Verify files"
ls -al $1*

echo ""
echo "Compression statistics"
gzip -vl $1.gz
```

When Nginx sees the .gz file it will send this out to clients who accept compression instead of compressing the file in real time. Make sure you have built your nginx binary with the argument "--with-http_gzip_static_module". Execute "nginx -V" to see the compiled options.

gzip_buffers allows 16 slots of 8k buffers used to respond to clients with a gzip'd response. This means the max size of our compressed responses can be no larger than $16 \times 8 = 128$ kilobytes. By default Nginx limits compressed responses to $4 \times 8k = 32$ kilobytes. If you expect to return responses which compressed size is more than 32KB in size then increase the number of buffers (e.g. 16). The single buffer size of 8K can not be increased.

gzip_comp_level set at 1 compresses files to the lowest compression level. Level 1 is the fastest/lowest compression and level 9 is the slowest/best compression. During testing the time difference between level 1 and 9 was around 2 hundredths of a second per file on a P3 500MHz. Two(2) hundredths of a second is about the same amount of time the entire page should be rendered.

Which compression ratio is right for your server ? To test we took a large HTML file and compressed it on a AMD64 2.4GHz machine using gzip levels 1 though 9. Level 1 compressed the file 61.3% and gzip level 9 took twice as long to compress the file to 67.6%. Level 1 has the best compression to time ratio.

When should each level be used? If you are doing real time gzip compression we suggest level 1 due to its low cpu useage, low latency and good compression. The problem with gzip level 1 is the file size is slightly larger then level 9 and the larger size will add a delay to the transmission of the file to the client. The problem with level 9 compression is the extra client side cpu time need to decompress the file which leads to slower page draw times. If you are using the gzip_static to pre-compress html then use level 1 compression. Pre-compressing to gzip level 1 will give you a small file saving bandwidth and transfer time, but also lower client side decompression time. Since you are precompressing, server side CPU time and compression cpu latency does not exist.

On the client side, meaning the browser, today's computers are fast enough that a user is unlikely to notice slightly more CPU usage compared to longer download times. In fact, decompression of the test file using gzip 1 through 9 only took around 0.004 seconds on our test box. The client will be more thankful to save the bandwidth compared to the slight increase in CPU time to decompress the file. This is especially true on mobile devices.

Compression of an HTML test file on the server side					
gzip level	ratio	cpu time	size	cpu_time/ratio	
original	0.0%	0m0.000s	206694	-	
level 1	61.3%	0m0.007s	80063	1.141E-4	<- most efficient for
level 2	62.9%	0m0.007s	76731	1.112E-4	
level 3	64.1%	0m0.008s	74257	1.248E-4	
level 4	66.1%	0m0.008s	70126	1.248E-4	
level 5	67.1%	0m0.010s	68002	1.490E-4	
level 6	67.5%	0m0.012s	67180	1.777E-4	
level 7	67.6%	0m0.013s	67036	1.923E-4	
level 8	67.6%	0m0.014s	66992	2.071E-4	
level 9	67.6%	0m0.014s	66989	2.071E-4	
Decompression of the same HTML test file on the client side					
gzip level	ratio	cpu time	size		
original	0.0%	0m0.000s	206694		
level 1		0m0.004s			<- notice decompression times are about t
level 2		0m0.004s			
level 3		0m0.004s			
level 4		0m0.004s			
level 5		0m0.004s			
level 6		0m0.003s			
level 7		0m0.003s			
level 8		0m0.003s			
level 9		0m0.003s			

gzip_http_version 1.0 allows the server to send compressed data to HTTP/1.0 clients. HTTP/1.1 clients use the proper headers so they can always ask for compressed data. Most new browsers use SPDY and compression headers are ignored on SPDY connections.

gzip_min_length set to 0 means that nginx should compress all files no matter what the

size. The value is the size in bytes. You can always set this value to something higher if you do not wish to compress small files.

gzip_types text/plain text/html text/css image/bmp are the only files types to be compressed. For example, JPG's are already compressed so it would be useless for us to try to compress them again. TXT and BMP files on the other hand compress well at an average of 250% smaller. Smaller files mean less bandwidth used and less time to transmit the same amount of data. This makes your site "feel" significantly faster.

gzip_vary on enables the response header "Vary: Accept-Encoding". This way clients know that our server has the ability to send out compressed data. Again, SPDY connections ignore compression headers.

log_format main : is the log format of the web logs. This format is assigned to the variable "main" and can be used later in the http section. This format is fully compatible with standard log analyzing tools like [Awstats](#), [Webalizer](#) and custom tools like the [Calomel.org Web Log Sentry](#). We also have added two(2) more fields at the end of each log line. "\$request_time" logs how much time the server took to generate the content and "\$gzip_ratio" shows what X-factor the file was compressed by. A value of 2.50 means the file was compressed 250%.

access_log and error_log : are the locations you want the logs to be placed in. In the access_log directive you can also use the buffer command. This will buffer the access log activity into ram and once the limit has been reached Nginx will then write the logs. This can save I/O stress and bandwidth on your hard drive. You will want to remove "buffer=32k" while testing else you will not see any log output until at least 32 kilobytes of data are ready to be written to the access_log file. 32K of logs input is approximately 150 lines. The notice directive on the error_log will increase the verbosity of the logs to include the reasons that clients were denied access.

expires 31d: says we want our pages to be expired from the clients cache in 31 days. We also highly suggest setting the value to "max" if your pages do not change that often. Time in the Expires header is obtained as the sum of the current system time added to the time assigned in this directive. In effect, we are saying that pages are to be expired 31 days after they were accessed by the client. You can also specify a time in hours using "h". In the Nginx v0.7.0 release you can use the format "expires modified +1d" to set the expires header based on the modified time of a file. The expire header tag will tell clients they should keep a copy of the object they already downloaded for the specified amount of time. This saves a significant amount of upload bandwidth for you. Instead of clients going from page to page downloading the same picture banner over and over again, they can keep a copy locally and just get the changes on your site. Imagine a client getting 5 pages from your site. Each page has a banner that is 15KB. With expires headers enabled that client will only download the banner once instead of 5 times (15KB compared to 75KB) saving your upload bandwidth and making your site "feel" quicker responding.

limit_req zone=gulag burst=1000 nodelay; : limits remote clients to no more than 1000 concurrently "open" connections per remote ip address being processed by Nginx. See the complimentary directive **limit_req_zone** above for more information about defining the "gulag" table.

listen 127.0.0.1:80 default rcvbuf=8K sndbuf=128k backlog=128 :

- listen 127.0.0.1:80 tells nginx to listen to localhost (127.0.0.1) port 80.
- rcvbuf=8K buffers incoming data (sysctl net.inet.tcp.sendspace). rcvbuf can be decreased to as little as 1K, possibly decreasing the probability of overflow during a DDoS attack. We recommend decrease the rcvbuf to the smallest size you expect a valid client to send to you at any one time. For example, if you serve a static site with no file uploads and only expect client to request files, you can safely set the receive buffer to as low as one(1) kilobyte for http connections and two(2) kilobytes for https connections. rcvbuf is the system call SO_RCVBUF which is the size of the buffer the kernel allocates to hold the data arriving into the given socket during the time between it arrives over the network and when it is read by the program that owns this socket. With TCP, if data arrives and you aren't reading it, the buffer will fill up, and the sender will be told to slow down (using TCP window adjustment mechanism). For UDP, once the buffer is full, new packets will just be discarded.
- send buffer directive (sndbuf=128k) tells nginx to buffer up to 128 kilobytes in ram for returned file requests read from the filesystem. The default is 32 kilobytes. SO_SNDBUF only matters for TCP (in UDP data is immediately sent out to the network). For TCP, a program could fill the buffer either if the remote side is not reading or receiving (so that remote buffer becomes full, then TCP communicates this fact to your kernel, and your kernel stops sending data, instead accumulating it in the local buffer until it fills up). Or it could fill up if there is a network problem, and the kernel is not getting acknowledgments for the data it sends (CWND size). The kernel will then slow down sending data on the network until, eventually, the outgoing buffer fills up. If so, future write() calls to this socket by the application will block (or return EAGAIN if you've set the O_NONBLOCK option). Using the sndbuf call can reduce constant small reads and allow the system to buffer up data for slow client sends. You may want to test setting up a send buffer size to around double your largest static resource. If you have a 500 kilobyte file you are sending to the client and the buffer is 128K then nginx will need to read at least four(4) times from the filesystem (500KB / 128 buffer = 3.9 reads). In this case you may want to increase the send buffer to 512K or even 1M. This way nginx will read all 500K of the file and put it all into the 512K buffer reducing your disk reads to only one(1). Very efficient. Note that since you using more ram to fulfill client requests you want to make sure you have enough system resources for the amount of clients you expect to serve at any one time. So, what kind of speed increases can you see by adding a larger buffer? We saw an average decrease in random disk IO right away and reduced latency in fulfilling client requests by at least 62%. You may see an even bigger advantage if your files are on slow hard disks or your OS is a bit slow at IO. For example OpenBSD's UFS filesystem is glacial compared to FreeBSD's ZFS filesystem.
- accept_filter=httptready -- is for HTTP only listening servers. The httptready accept filter buffers the entire HTTP request at the kernel level. Once an entire request is received, the kernel then sends it to the nginx server. This filter is an excellent way to filter out SYN scans and illegal connections and keep them away from the web server. Only FreeBSD's Accept Filters (accf_http) is currently supported. Make sure you have the FreeBSD accf_http kernel module loaded.
- accept_filter=dateready -- is used for HTTPS listening servers since HTTPS requests are encrypted. The accf_data(9) filter buffers incoming connections until data arrives and then the nginx server is notified. Filtering data connections is a great way to stop a lot of the attacks and scans. Make sure have the FreeBSD accf_data kernel module loaded.
- deferred -- indicates to use that postponed accept(2) on Linux with the aid of the TCP option TCP_DEFER_ACCEPT. This is a simplified method similar to FreeBSD's accf_data as it does not buffer data but will wait for a completed TCP connection. Again, using TCP_DEFER_ACCEPT can stop many of the more obvious scans or SYN attacks from getting to the nginx server.

- backlog (sysctl kern.somaxconn) are the max number of backlogged client requests Nginx will process. If you server is quite busy you will want to increase this value.

BTW, we listen on 127.0.0.1:8080 in order to use the redirection rules in iptables or in [OpenBSD's pf packet filter firewall](#). The argument "default" says that this server {...} function should handle any client request sent to this port no matter the hostname (not used in the example).

root /var/www/html : is the location of document root on your server. This is where nginx will look for all files to be served out to clients.

server_name mydomain.com www.mydomain : means this server {...} block will only answer requests that have "mydomain.com" or "www.mydomain" host headers. By default the hostname of the machine is used. We are expecting the client to ask for the correct hostname with the Host header, if not, the default server block with "server_name _;" returns an error 444 to the client. BTW, the server_name_in_redirect and server_name directives work in conjunction with each other.

SSL Options (only enable if you use a SSL certificate) If you are interested in setting up a SSL certificate for encrypted traffic on your site then we highly suggest reading our [Guide to Webserver SSL Certificates](#). Once you understand the details of SSL certs then you must build Nginx from source and enable the argument **"./configure --with-http_ssl_module"**.

ssl on; Enables the use of the ngx_http_ssl_module once it has been built into the Nginx binary.

ssl_certificate /ssl_keys/mydomain.com_ssl.crt; This file is the combined certificate which contains both of the "crt" files signed and sent to you by your certificate authority. See "How to setup a SSL cert from Comodo through NameCheap for Nginx" below for details.

ssl_certificate_key /ssl_keys/mydomain_ssl.key; Specifies the location of the file with the secret key in PEM format for this server. This file is the public certificate secret key you made using the OpenSSL binary.

ssl_ciphers lists the ciphers in the order our server will negotiate. The server decides which cipher the client and server will use, so always list the strongest ciphers first. Notice in our string of ciphers we list the Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) Perfect Forward Secrecy (PFS) ciphers first. Then, since some Redhat Linux machines do not support Elliptic Curve (EC) ciphers we included the TLSv1.2 Galois Counter Mode (GCM) ciphers and some AES 256 bit, SHA 256 bit ciphers afterward. We DO NOT recommend regular RC4-SHA, but your business may not want to deny access to your services for ancient clients like original smart phones or IE8 on Windows XP. The choice is yours, but we suggest never using RC4-SHA or ECDHE-RSA-RC4-SHA equivalent.

The command "openssl ciphers -v 'ALL:@STRENGTH'" will show you all of the ciphers your version of OpenSSL supports. Our [Guide to Webserver SSL Certificates](#) explains many of the details about ciphers and compatibility models. Also, notice we did NOT include any Diffie-Hellman Ephemeral (DHE) or Triple DES (3DES) ciphers as they are six(6) to ten(10) times slower than ECDHE. Every millisecond counts.

ssl_prefer_server_ciphers on; just means that our server will use the ciphers specified in the "ssl_ciphers" directive over the ciphers preferred by remote clients. It is never a good security practice to trust remote clients.

ssl_protocols TLSv1 TLSv1.1 TLSv1.2; tells the server to only allow TLS version 1.0 or greater (TLSv1). It is highly recommended never to use SSL version 2 (SSLv2) or SSL version 3 (SSLv3) as they have vulnerabilities due to weak key strength. In order to be [FIPS 140-2 compliant](#) only TLSv1 (which stands for 1.0 or higher) can be used. Please check out the section lower down on this page where we explain how to build nginx with the latest version of OpenSSL for TLS v1.0, 1.1 and 1.2 support.

ssl_session_cache shared:SSL:1m; allows Nginx to cache the SSL session keys in its own cache structure instead of using OpenSSL slower, single threaded cache. This means Nginx can now take advantage of multiple worker_processes and separate the SSL jobs between them. The result is an impressive speed boost (2x or more) over the slower OpenSSL cache depending on your OS. The format of the line "shared:SSL:1m" is as follows: "shared" is the internal caching function, "SSL" is just an arbitrary name of this SSL cache (you can name it anything you want) and "1m" is the size of the cache (1 megabyte can hold around 4000 SSL cache sessions). It is recommended to set this value larger than the amount of clients connecting in the "ssl_session_timeout" amount of time. If you have ten thousand clients connecting and your "ssl_session_timeout 5m" (5 minutes) then set this "ssl_session_cache shared" to 4m (4 megabytes max). $(10,000 \text{ clients} \times 5 \text{ minutes}) / 4000 = 12.5$; so 1 megabyte of ram can hold 4000 cache entries and we set the cache to 4m so it can hold sixteen thousand entries. Also note, if the cache ever fills up the oldest, unused cache entry is removed automatically and the new client entry is made. According to the developers you should not rely on automatic removal as it is considered an emergency function and the client may be denied connecting in the mean time. It is a good idea to make your cache size larger than you expect to use, similar to our example above.

ssl_session_timeout 5m; is the cache session timeout between the client and the server set at 5 minutes or 300 seconds. When this time runs out the clients ssl session information is removed from the "ssl_session_cache". The reason this number was chosen is it also the default amount of time many client browsers will cache an ssl session. If you expect the client to stay on your site longer and go to multiple pages you can always increase this default value. Depending on the client browser, they may or may not respect your ssl_session_timeout value if it larger than 5 minutes.

ssl_ecdh_curve secp384r1; Elliptic curve Diffie-Hellman (ECDH) is a key agreement protocol allowing two parties, each having an elliptic curve public-private key pair, to establish a shared secret over an insecure channel. secp384r1 is a 384 bit Elliptic curve which most efficiently supports ssl_ciphers up to a SHA-384 hash and is compatible with all desktop and mobile clients. Using a larger hash value, like secp521r1, and then truncating down to 384 or 256 wastes around one(1) millisecond of extra cpu time. On the other hand, calomel.org uses secp521r1 due to its more difficult cracking potential and the fact that older OS's like XP can not solve the equation; i.e. those old compromised machines can not connect to us. Take a look at the NSA's paper titled, [The Case for Elliptic Curve Cryptography](#) for more details.

ssl_stapling on; enables Online Certificate Status Protocol (OCSP) certificate revocation status to be sent to the client directly by our server saving the client the time of checking the OCSP server themselves. The reason for stapling the OCSP data is to speed up the client's load time of our site by saving them the time of check the OCSP server themselves.

The OCSP check by our server adds just a bit of load on the server side, but can speed up the client significantly and puts less load on the certificate authority servers. Once Nginx receives a valid OCSP response from the CA's OCSP server, Nginx will cache it for about an hour before check the OCSP server again. The OCSP check only takes 100 milliseconds or so and is of negligible load to the server.

WARNING: Due to limitations in OpenSSL, browser implementations and the way virtual hosts work in Nginx SSL Stapling will **ONLY** work if it is in your default SSL server. For example, if we had a `server{}` block of code and we put in `"server_name calomel.org;"` then ssl stapling will **_NOT_** work. We would have to use `"server_name _;"` to make the `server{}` block the default server and then ssl stapling will work as expected. The reason this limitation comes up is some clients do not set the correct Host header even if they are SNI compatible.

resolver 8.8.8.8; (optional) This is the DNS server nginx will query the OCSP server domain against. If you have a default DNS server for your machine you will not need a "resolver" line. If you can run the `host` command and get an ip address returned, like `"host calomel.org"`, then you do not need this resolver directive. BTW, 8.8.8.8 is Google's public DNS server.

ssl_stapling_file /ssl_keys/example.com_staple; (optional) The staple file is a speed optimization of OCSP stapling. It allows the nginx server to send out a trusted OCSP response instead of checking the OCSP server over and over again. This directive is completely optional and **_not_** needed to get OCSP Stapling to work. The following is how to make your own `ssl_stapling_file` using our domain as an example:

```
## to make a OCSP ssl_stapling_file file use the following openssl commands.

# this will collect all of the ssl publicly available certificates
# from your site. Replace calomel.org with your domain.
openssl s_client -showcerts -connect calomel.org:443 < /dev/null | awk -v c=

# print out all of the cert information in human readable format. This is ju
# each level file looks ok. Take a look at the "CN" domain names.
for i in level?.crt; do openssl x509 -noout -serial -subject -issuer -in "$i

# cat all of the level certs into a bundle. Calomel.org has 3 levels of cert
# if you have more then just add to them to "{0,1,2,3,4,5,etc}". Make sure to
# the certs in order, order is important.
cat level{0,1,2}.crt > CABundle.crt

# print out the OCSP server for your certificate's domain. You will need
# the OCSP server which your certificate can be validated against.
for i in level?.crt; do echo "$i:"; openssl x509 -noout -text -in "$i" | gre

# validate our certs against the OCSP server from the previous command and
# make the example.com_staple DER encoded OCSP response file. You will need
# to change ocp.comodoca.com to your certs OCSP server.
openssl ocp -text -no_nonce -issuer level1.crt -CAfile CABundle.crt -cert l

# Done. Now place the example.com_staple where you have the
# ssl_stapling_file directive pointing to.
```

Once you get OCSP Stapling configured you can test and verify the server's OCSP stapling response using openssl client.

```

## a successful OCSP result will have a large code block after
## "OCSP Response Data:" similar to our server's OCSP response.
## Your results may be longer or shorter, but it is important to
## at least see the following:

$ openssl s_client -connect calomel.org:443 -tls1 -tlsextdebug -status
...
OCSP response:
=====
OCSP Response Data:
  OCSP Response Status: successful (0x0)
  Response Type: Basic OCSP Response
...
  Cert Status: good
...

## A ssl server that does not support, or is incorrectly configured for
## OCSP stapling will instead show "OCSP response: no response sent"

$ openssl s_client -connect google.com:443 -tls1 -tlsextdebug -status
...
OCSP response: no response sent
...

```

Why are if statements bad ?

If statements (`if{ }`) are expensive to process and they can be unpredictable. Every object a client requests will trigger the if statement and this takes cpu time to process in a time when sites are expected to come up within tenths of a second. We are not saying to never use an if statement as there are some times where it can be avoided. Just be aware of the way if statements work, how much of a performance hit you may take on and what alternatives exist.

Take a look at these two pages on the Nginx wiki site for more information and insight:

- [Nginx: Pitfalls and Common Mistakes](#)
- [Nginx: "If" Is Evil](#)

Setting up an ECDSA or RSA SSL certificate for https access

Certificates can use either the RSA or ECDSA public-key cryptosystem. RSA is most commonly used by 99% of https web sites. The advantage of RSA is backwards compatibility with even the most ancient browsers, but RSA does not scale as well with regards to performance when key size increases. Larger bit keys take more processing on the client and server. Compared to RSA, ECDSA signatures are ten times faster on the server and

more secure for the same key size in bits. For example, an ECDSA 256 bit key is more than ten thousand times harder to crack compared an RSA 2048 bit key. For more information about ECDSA please read Symantec's [Elliptic Curve Cryptography \(ECC\) Certificates Performance Analysis](#).

So, which algorithm should I use? If you want the most compatibility with ancient clients and good security then RSA is a fine choice. On our site, calomel.org, we prefer using ECDSA to take advantage of the reduced server load (efficiency), smaller key exchange sizes (speed) and increased encryption strength per in bit (security). Who else uses ECDSA? The entire Bitcoin infrastructure is based on ECDSA SHA-256 and the banking financial sector uses ECDSA to sign the images of customer deposited checks. What about client compatibility with ECDSA? All modern clients on the desktop and mobile platform connect to our site fine. You are welcome to point your browsers to our site to test. Googlebot also connects without issue when indexing our pages. Older clients like IE3-7 and some of the Bing search bot cluster do fail to connect though.

NOTE: after you receive the certificate from the certificate authority, your server needs to be configured with the type of ciphers your system will negotiate with. If you set up an RSA certificate then only use ciphers with RSA like, ECDHE-**RSA**-AES128-GCM-SHA256 . If you instead set up an ECDSA signed certificate you need to only use ciphers with ECDSA like, ECDHE-**ECDSA**-AES128-GCM-SHA256 . A cipher can only be used if the cipher uses the same certificate signing type as your certificate. Take a look at the Nginx server configurations at the top of this page for syntax examples.

The market for SSL certificates is highly concentrated, despite the large number of certificate issuers. Approximately 75 percent of SSL certificates in use on the public internet are issued by just three companies: Symantec, GoDaddy, and Comodo. Symantec, the largest commercial certificate authority, owns multiple brands including Verisign, GeoTrust, Thawte, RapidSSL, and TC TrustCenter. The distribution is heavily skewed, with smaller CAs having little or no presence on the public Internet. The result of just a few companies owning all of the SSL signing rights with minimal competition means one certificate authority is no better than another so you can simply choose a CA with the lowest price. The paper titled, [Security Collapse in the HTTPS Market](#) explains more about the general corruption of the SSL authorities.

For this example we are going to look at [Comodo's PositiveSSL](#) for both RSA and ECDSA certificates purchased through NameCheap.com for as little as \$9 US per year. Comodo offers ECDSA certificates signed by the Elliptical Curve DSA all the way up to the built in, browser trusted, Comodo ECC Root Certificate. This makes Comodo one the only CA's to offer a pure ECC certificate chain. Comodo also offers standard RSA certificates on a separate RSA chaining infrastructure. Both ECDSA and RSA certificates support Online Certificate Status Protocol (OCSP), Certificate Revocation List (CRL) and Nginx's OCSP stapling option for speed.

Once registered with the NameCheap site, find the SSL section and ask to issue a new certificate or re-issue your current cert. Once your email address has been verified you will be brought to a page where you will cut and paste your certificate signing request (CSR) into the web page box.

The following two(2) sections will explain how to create a 384 bit ECDSA (Elliptic Curve Digital Signature Algorithm) SHA-256 certificate signing request or a 2048 bit RSA SHA-256 CSR. Option one(1) is the ECDSA CSR and Option two(2) is the RSA CSR; the steps are the

same and only the openssl commands differ. The certificate type, ECDSA or RSA, Comodo returns to you depends on the certificate signing request (CSR) type you submit.

Option 1: setup an ECDSA SSL cert from Comodo for Nginx

Generate your site's 384 bit ECDSA SHA-256 Certificate Signing Request (CSR) to be given to Comodo:

```
# First, generate a ECC key using a prime 384 bit curve
openssl ecparam -out mydomain_ssl.key -name secp384r1 -genkey

# Generate a SHA-256 ECDSA hashed certificate signing request
openssl req -new -sha256 -key mydomain_ssl.key -nodes -out mydomain.com_ss
```

You are about to be asked to enter information that will be incorporated into your certificate request.

What you are about to enter is what is called a Distinguished Name or a DN

There are quite a few fields but you can leave some blank

For some fields there will be a default value,

If you enter '.', the field will be left blank.

Country Name (2 letter code) [AU]:US

State or Province Name (full name) [Some-State]:MD

Locality Name (eg, city) []:Washington

Organization Name (eg, company) [Internet Widgits Pty Ltd]:MyDomain

Organizational Unit Name (eg, section) []:

Common Name (e.g. server FQDN or YOUR name) []:mydomain.com

Email Address []:

Please enter the following 'extra' attributes to be sent with your certificate request

A challenge password []:

An optional company name []:

```
# Understand what is in the certificate signing request. Look for the EC
```

Comodo will have you copy and paste the CSR into a text box on their web page and hit submit. Cat the mydomain.com_ssl.csr file and make sure to include the "BEGIN CERTIFICATE REQUEST" and "END CERTIFICATE REQUEST" lines.

When you provide Comodo with your CSR it is **IMPORTANT** to make sure to choose the "APACHE OPENSSL" certificate type when asked what format to download in so we can use the correct format for Nginx.

Comodo will then email you a compressed ZIP file called something like mydomain_com.zip . In the zip file will be four(4) files. We will need to combine only two(2) files in the correct order for the site crt file to work with Nginx. Here is an example of the files in the zip archive:

- mydomain_com.crt
- COMODOECCDomainValidationSecureServerCA.crt
- COMODOECCAddTrustCA.crt
- AddTrustExternalCARoot.crt

Now we need to combine most of the Comodo crt files into a single site crt file Nginx will use. The AddTrustExternalCARoot.crt and COMODOECCAddTrustCA.crt files are not needed because the client's browser, like Firefox or Chrome, already has a list of root certs they will trust including Comodo's ECC Root certificate. When combining the crt files the order is important to Nginx:

```
# Combine certs for Nginx. Start with your domain and work upward:
cat mydomain_com.crt COMODOECCDomainValidationSecureServerCA.crt > mydomain.

# Again, familiarize yourself with the format of the site crt file.
openssl x509 -noout -text -in mydomain.com_ssl.crt

# Now the mydomain.com_ssl.crt is your Nginx "ssl_certificate" file.
```

Finally, make a secure directory outside of the web document tree and limit access to the directory to root (read and write) and NO ONE ELSE. In our example we will make a directory called /ssl_keys. Both your public "key" file generated by OpenSSL (ssl_certificate_key /ssl_keys/mydomain_ssl.key) and the combined certificate "crt" file (ssl_certificate /ssl_keys/mydomain.com_ssl.crt) are copied to /ssl_keys. It is a good idea to make both files read only by root. Now you can start your SSL server and make sure clients can connect.

```
# This is what the /ssl_keys directory should look like

root@calomel: ls -al /ssl_keys
total 30
drwx-----  2 root  wheel    4 Jan  1 01:23 .
drwx----- 10 root  wheel  125 Jan  1 01:23 ..
-rw-----  1 root  wheel 2705 Jan  1 01:23 mydomain.com_ssl.crt
-rw-----  1 root  wheel  359 Jan  1 01:23 mydomain_ssl.key

# And this is an example configuration in nginx.conf

ssl_certificate /ssl_keys/mydomain.com_ssl.crt;
ssl_certificate_key /ssl_keys/mydomain_ssl.key;
```

Option 2: setup an RSA SSL cert from Comodo for Nginx

Generate your site's 2048 bit RSA SHA-256 Certificate Signing Request (CSR) to be given to Comodo:

```
# Generate a SHA-256 at 2048 bit hashed certificate signing request
openssl req -nodes -sha256 -newkey rsa:2048 -keyout mydomain_ssl.key -out

Generating a 2048 bit RSA private key
.....+++
....+++
writing new private key to 'mydomain_ssl.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MD
Locality Name (eg, city) []:Washington
Organization Name (eg, company) [Internet Widgits Pty Ltd]:MyDomain
Organizational Unit Name (eg, section) []: -hit enter, leave empty-
Common Name (e.g. server FQDN or YOUR name) []:mydomain.com
Email Address []: -hit enter, leave empty-

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []: -hit enter, leave empty-
An optional company name []: -hit enter, leave empty-
```

Upload the mydomain.com_ssl.csr file or copy and paste the contents of the CSR file into the website. The certificate authorities' site should tell you the methods allowed. For Comodo, they prefer we copy and paste the CSR into a text box on their web page and hit submit. Cat the mydomain.com_ssl.csr file and make sure to include the "BEGIN CERTIFICATE REQUEST" and "END CERTIFICATE REQUEST" lines.

When you provide Comodo with your CSR it is **IMPORTANT** to make sure to choose the "APACHE OPENSSL" certificate type when asked what format to download in so we can use the correct format for Nginx.

Comodo will then email you a compressed ZIP file called something like mydomain_com.zip . In the zip file will be four(4) files. We will need to combine the files in the correct order for the site crt file to work with Nginx. Here is an example of the files in the zip archive:

- mydomain_com.crt
- COMODORSADomainValidationSecureServerCA.crt
- COMODORSAAddTrustCA.crt
- AddTrustExternalCARoot.crt

Now we need to combine most of the Comodo crt files into a single site crt file Nginx will use. The AddTrustExternalCARoot.crt file is not needed because the client's browser, like Firefox or Chrome, already has a list of root certs they will trust. When combining the crt files the order is important to Nginx:

```
# Combine certs for Nginx. Start with your domain and work upward:
cat mydomain_com.crt COMODORSADomainValidationSecureServerCA.crt COMODORSAAddTrustExternalCARoot.crt

# Again, familiarize yourself with the format of the site crt file.
openssl x509 -noout -text -in mydomain.com_ssl.crt

# Now the mydomain.com_ssl.crt is your Nginx "ssl_certificate" file.
```

Finally, make a secure directory outside of the web document tree and limit access to the directory to root (read and write) and NO ONE ELSE. In our example we will make a directory called /ssl_keys. Both your public "key" file generated by OpenSSL (ssl_certificate_key /ssl_keys/mydomain_ssl.key) and the combined certificate "crt" file (ssl_certificate /ssl_keys/mydomain.com_ssl.crt) are copied to /ssl_keys. It is a good idea to make both files read only by root. Now you can start your SSL server and make sure clients can connect.

```
# This is what the /ssl_keys directory should look like

root@calomel: ls -al /ssl_keys
total 30
drwx-----  2 root  wheel    4 Jan  1 01:23 .
drwx----- 10 root  wheel  119 Jan  1 01:23 ..
-rw-----  1 root  wheel  5994 Jan  1 01:23 mydomain.com_ssl.crt
-rw-----  1 root  wheel  1704 Jan  1 01:23 mydomain_ssl.key

# And this is an example configuration in nginx.conf

ssl_certificate /ssl_keys/mydomain.com_ssl.crt;
ssl_certificate_key /ssl_keys/mydomain_ssl.key;
```

How can I build Nginx with Google's BoringSSL ?

BoringSSL is Google's modified version of OpenSSL v1.0.2 including hundreds of [Google's patches](#) minus much of the code bloat from OpenSSL. For example, Google removed the dependency of the heartbeat code which led to the HeartBleed exploit as well as much of the legacy OpenSSL code.

Nginx with BoringSSL supports SSL, SPDY 3.1, SNI, ECDSA and RSA certificates and all major ciphers including ChaCha20 Poly1305. Our servers at calomel.org are running a similar build

in production.

The only function this build of Nginx and BoringSSL does not support OCSP certificate pinning also called OCSP stapling. Google believes [OCSP stapling is flawed](#) so there is no support in BoringSSL at this time, but OCSP stapling might be included in the future. Make sure you comment out the directive "ssl_stapling" in your nginx.conf to avoid errors on startup. Other than the lack of OCSP stapling support, Nginx and BoringSSL is fully featured, rapidly patched, stable and fast.

NOTE: Please use Clang v3.4.x and CMake v3.x to build BoringSSL on FreeBSD 10.1 ; check Clang and Cmake with "clang --version;cmake --version". We found building BoringSSL with Clang v3.3.x and CMake v2.x on FreeBSD causes an Nginx worker daemon to randomly use 100% cpu time when illegal SSL requests are made. It seems that BoringSSL is getting stuck somehow and spin locking the cpu core at 100% load. The issue is caused by Internet clients sending a random string of binary data to the server instead of a standard GET or POST formatted request. We are still investigating with the hope of reporting the issue to Google.

The following are our notes to build Nginx v1.7.5 with the latest git checkout of Google's BoringSSL. The install is not difficult and only takes a few steps.

```
##### BoringSSL build

# install git to collect the BoringSSL repository and install cmake to build
# the BoringSSL libraries
pkg install cmake cmake-modules          # freebsd 10.1
-OR-
portmaster devel/cmake; pkg install git   # freebsd 10.1
-OR-
apt-get install cmake git                 # ubuntu 12.04

# make or change to a directory you can work in. /tmp should be fine for now
cd /tmp

# download a clone of the git repo of BoringSSL. git will make a directory
# named, "boringssl" in the current directory. To update your git clone use
# "git reset --hard && git pull" while in the "boringssl" directory. The first
# checkout the repo the first time run the following:
git clone https://boringssl.googlesource.com/boringssl

# Download our patch of BoringSSL to remove the debug build options and add
# amd64 (64bit) and i386 (32bit) build identifiers.
wget --no-check-certificate https://calomel.org/boringssl_freebsd10_calomel

# change to the BoringSSL directory created by git
cd boringssl

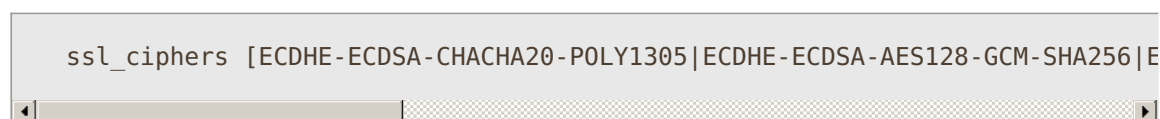
# patch BoringSSL. Make sure to re-patch after a git reset and pull.
patch < ../boringssl_freebsd10_calomel.org.patch
```

Equal Preference Cipher Groups in BoringSSL

Once Nginx is built against BoringSSL setup your Nginx ssl ciphers with equal-preference cipher groups.

"... new concept of an equal-preference group: a set of cipher suites in the server's preference order which are all "equally good". When choosing a cipher suite using the server preferences, the server finds its most preferable cipher suite that the client also supports and, if that is in an equal preference group, picks whichever member of the group is the client's most preferable. For example, Google servers have a cipher suite preference that includes AES-GCM and ChaCha20-Poly1305 cipher suites in an equal preference group at the top of the preference list. So if the client supports any cipher suite in that group, then the server will pick whichever was most preferable for the client." -- Google

The ciphers which are considered to be of similar strength are put into square brackets separated by a pipe. This is the standard regular expression format meaning "OR". For example, if we say X or Y the format is "[X|Y]". The following ssl_cipher list shows ECDHE-ECDSA-CHACHA20-POLY1305 and ECDHE-ECDSA-AES128-GCM-SHA256 and ECDHE-ECDSA-AES256-GCM-SHA384 as equal-preference high end ciphers just as Google explained.



```
ssl_ciphers [ECDHE-ECDSA-CHACHA20-POLY1305|ECDHE-ECDSA-AES128-GCM-SHA256|E
```

With equal preference cipher groups calomel.org can now respect the requests of clients who wish to use certain ciphers within the restrictions we allow. With mobile devices, the cpu is low powered and does not have AES hardware acceleration. Most mobile clients will prefer the CHACHA20-POLY1305 stream cipher because it is three(3) times faster than AES. Server and workstations with newer cpus have AES-NI; accelerated Advanced Encryption Standard (AES) in the hardware CPU. For these machines, AES is at least twice as fast as CHACHA20-POLY1305. Equal preference cipher groups allow our server to use the cipher the client considers most beneficial. Check out our [AES-NI SSL Performance Study](#) for more information.

Why setup a "default blank SSL server" ?

When a client browser asks for your server they should request the hostname you registered your SSL certificate for. In our example, we registered www.example.com and example.com. If a remote client asks for anything other than these they should not be allow to get to our site. The reason being that if they did not ask for our site by name then the SSL certificate is going to be invalid. The second reason is security. If you are not asking for our site they you should not be able to get to our site.

A "default blank SSL server" is the catch all for any client not specifically asking for our site by name.

In order to allow multiple SSL certificates to be served from a single IP address we need to use virtual hosting with [Server Name Indication](#). This means that your nginx build must report that it is supporting TLS with SNI like this:

```
user@machine$ nginx -V
nginx version: nginx/0.8.45
TLS SNI support enabled
```

If your server supports SNI you are good to go. If not, you will probably need to upgrade your version of Nginx or possibly get a newer version of OpenSSL. The example above used Nginx v0.8.45 and OpenSSL v1.0.0a for example. Now you can setup a second server block in Nginx.conf.

The following code from our "SSL only webserver" example above will tell Nginx to serve out the blank, self-signed SSL certificate for any clients not using the hostname www.example.com or example.com. This includes any scanners looking at the ip address of the server or any bad clients using false "Host:" headers.

```
## https ... default blank SSL server
server {
    listen          127.0.0.1:443 default;
    server_name     _;
    ssl             on;
    ssl_certificate  ssl_keys/default_blank.crt;
    ssl_certificate_key ssl_keys/default_blank.key;
    return          403;
}
```

We need to generate the certificate "crt" and public "key" files for Nginx. The following commands will make a self-signed certificate and key file without a pass phrase. The ssl cert is blank will not give any information to anyone looking at it.

First we need to make a new key. This will be a 4096 bit key signed using AES at 256 bits. Put in any pass phrase you want because we are going to remove it in the next step.

```
openssl genrsa -aes256 4096 > default_blank.key
```

Next, this is a dummy key we really do not care about so this command will remove the pass phrase.

```
openssl rsa -in default_blank.key -out default_blank.key
```

Third, create a certificate signing request. The only question you need to answer if the first one for Country. Put any two(2) letters in like "US" and hit enter for the rest. Make sure to keep the "hostname" entry blank too.


```
openssl req -new -key default_blank.key -out default_blank.csr
```

self sign your own certificate.

```
openssl x509 -req -days 1460 -in default_blank.csr -signkey default_blank.ke
```



Finally, copy the default_blank.crt and default_blank.key into your "ssl keys" directory so Nginx can find them.

Testing the setup

Using openssl and the s_client directive we can query the server for the proper hostname and server name using SNI. For this example we will use our real hostname calomel.org. What you are looking for in the output is the "Certificate chain" information. The real SSL cert will show multiple entries for the CA like GoDaddy. The blank SSL cert will show zero certificates in the chain and no real information. The following line will properly access the server.

```
openssl s_client -servername calomel.org -connect calomel.org:443
```

The second test should be to test a failure. Here we connect to the proper hostname calomel.org on port 443, but we ask for the wrong hostname in the SSL certificate with the "-servername" directive.

```
openssl s_client -servername NotTheRightHost.com -connect calomel.org:443
```

NOTE: there are some clients that can not use SNI enabled servers. Very OLD browsers like IE5 and Firefox v1.0 are not compatible. Strangely, some modern versions of Wget are also not compatible, but Curl and Elinks are. Modern OS's and browsers do support SNI as well as search bots like Googlebot, MSN and Yahoo. As an added bonus we find that a lot of the spammer bots do not negotiate SNI well.

Why use the error code "return 444" ?

Error 444 or "return 444;" is a custom error code understood by the Nginx daemon to mean, "Close the connection using a tcp reset with the client without sending any headers." In essence, we are closing the connection with the client without sending them any web page data. This definitely saves bandwidth and may be necessary in DDOS situation. This error code should be used with great care.

If you have a private site and wish to just close the connection, i.e. slam the door on them the error 444 is great. On the other hand search bots like Googlebot will punish a site which does not send proper error codes back in a timely fashion. To Googlebot an error 444 looks like a misconfigured server or a bad connection and will mark down your page rank as such. Please take care when using this code as it is useful in certain situations and harmful in others.

So what does the client see when receiving an error 404 compared to an 444 ? Lets take a look at the header results using the cURL command to a server sending a 404 error. FYI: cURL will be sending just the HEAD request to the server. Notice the server sends back a set of headers with the error code and some information about the server.

```
user@machine: curl -I http://www.somedomain.com/
HTTP/1.1 404 Not Found
Server: Nginx
Date: Mon, 10 Jan 2020 20:10:30 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: keep-alive
Keep-Alive: timeout=5
```

This following is what cURL says about a Nginx server returning an error 444. The server sent nothing back to the client and closed the connection. The client did not get any useful information about the server. If you are paranoid about security or just do not want to provide any data to clients who cause errors, this is a good response for them. Of course, if the client does not get a standard error reply the client might just try connecting over and over again.

```
user@machine: curl -I http://www.somedomain.com/
curl: (52) Empty reply from server
```

If you wish to change the return code to those found in the `error_codes` directive then your error page will be sent out instead. For example, instead of using code 444 you could send a 403 (Forbidden). For a full list of the error codes and their official definitions check out the [w3.org Error Status Code Definitions](http://w3.org/Status/Code/Definitions).

Directive explanation and random insight

Strict Transport Security add_header : Strict Transport Security (STS or HSTS) is a HTTPS only response header that will require the user agent (such as a web browser) to access the website using secure connections only (such as HTTPS). The header specifies a period of time during which the user agent is not allowed to access the site insecurely. We use 2592000 seconds or 30 days.

When Strict-Transport-Security is active for a website, a complying user agent like Firefox, Internet Explorer, Opera or Chrome does the following: It automatically turns any insecure links to the website into secure links. (For instance, <http://www.example.com/page/> will be modified to <https://www.example.com/page/> before accessing the server.) Secondly, if the security of the connection cannot be ensured (e.g. the server's TLS certificate is self-signed), show an error message and do not allow the user to access the site despite the error. Strict-Transport-Security helps protect website users against some passive (eavesdropping) and active network attacks. A man-in-the-middle attacker will not be able to intercept any request to a website while the user's browser has Strict-Transport-Security active for that site. -[Wikipedia \(STS\)](#)

Only allow GET and HEAD request methods : Request Method restrictions allow you to filter on GET, HEAD, POST, SEARCH, etc. We will be limiting access to our example server to GET and HEAD requests only as we do not allow uploads or any other options due to security concerns. All other request methods will get an error defined by "return 405". Understand that Nginx looks at the files and file permissions you want to serve. If all of the files in document root are read only then Nginx knows this is a static only site. Thus only GET and HEAD will be accepted. Other requests will get an error 405. If you have a dynamically generate site or if nginx is setup as a reverse proxy then you may want to take some time and restrict the request methods.

How do I test each type of request method ?

Curl is an excellent testing tool. These are some examples of the different request methods and how to get a response from a server. Test each of these against your server and make sure the header responses are what you expect. For example, only the GET and HEAD should work against a site setup to serve only static files.

```
GET
curl -HAccept:text/plain http://example.com

HEAD
curl -I -X HEAD http://example.com

PUT
curl -X PUT -HContent-type:text/plain --data "value1:value2" http://example.com

DELETE
curl -X DELETE http://example.com

POST
curl -d "param1=value1&param2=value2" http://example.com/

TRACE
curl -v -A "Curl" -X TRACE http://example.com
```

Deny certain User-Agents : You may want to list out some user-agents you do not want connecting to your server. They can be scanners, bots, spammers or any one else you find is abusing your server.

Deny certain Referers : Referrer spam is more of an annoyance than a problem. A web site or bot will connect to your server with the referer field referencing their web site. The idea is that if you publish your web logs or statistics then their hostname will show up on your page. When a search bot like Google comes by it will see the link from your site to theirs and give the spammers more PageRank credit. First, never make your weblogs public. Second, block access to referer spammers with these lines.

Redirect from www to non-www : is if you prefer clients who connect to your site to instead use the non-www domain. For example, if a browser connects to `www.mydomain.com` they will be redirected to the URL `mydomain.com` with a code 301. If they then save your site location in a bookmark it will show up as the preferred non-www domain.

Stop Image and Document Hijacking : Image hijacking is when someone makes a link to your site to one of your pictures or videos, but displays it on their site as their own content. The reason this is done is to send a browser to your server to use your bandwidth and make the content look like part of the hijacker's site. This is most common as people make links to pictures and add them to a public forum or blog listing. They get to use your picture in their content and not have to use their bandwidth or server to host the file. In order to keep your bandwidth usage low you should block access to images from those clients who are not referring the content from a page on your site. Note, this function can be used for any kind on content. Just add the file types to the list. If would like more ideas on lowering bandwidth usage check out our [Saving Webserver Bandwidth \(Tips\)](#).

Restricted Access directory : This area is to limit access to a private or content sensitive directory. We will be limiting access to it by ip address (first check) and if that passes then ask for a password (second check). Both must match before access is granted.

access control list : This is a way you can define a directory and only allow clients coming

from the specified ips to have access. Use this function to allow internal LAN clients access to the status pages or employee contact information and deny other clients. In our example we will allow the clients coming from localhost (127.0.0.1/32) and internal LAN ips 10.10.10.0/24 to access the protected "secure" directory. BTW, if you use [OpenBSD's pf packet filter firewall](#) we highly suggest enabling "synproxy" in your pf.conf for all connections to your web server. Normally when a client initiates a TCP connection to a server, PF will pass the handshake packets between the two endpoints as they arrive. PF has the ability, however, to proxy the handshake. With the handshake proxied, PF itself will complete the handshake with the client, initiate a handshake with the server, and then pass packets between the two. The benefit of this process is that no packets are sent to the server before the client completes the handshake. This eliminates the threat of spoofed TCP SYN floods affecting the server because a spoofed client connection will be unable to complete the handshake.

password protected area : If you are coming from an authorized ip address then we will ask for a username and password. If you have an area of the web site you only want authorized personnel to see then you should protect it. This set of directives will password protect the directory "/secure" and all files and directories under it. We will use the basic method which is the authors best choice to use especially for non-https sites. It will not send any of the passwords in clear text. Check "man htdigest" for details.

To make the "access_list" password file use the binary htdigest in the following form. Supply the "username" and "password" pair for access. Remember that our configuration file will look for this file at /var/www/htdocs/secure/access_list :

```
htpasswd -b -c access_list username password
```

Only allow these file types to document root : We want to restrict access to our server to clients who are actually looking for data we serve out. For example, if a client is asking for a PHP file and we do not serve that type of file then we want to deny them access.

"(^V|\.html|\.css|\.jpg|favicon\.ico|robots\.txt|\.png)\$" matches the incoming request. If a request fails to match than this service will be skipped and the client will get an error. If all services fail to match Nginx returns a generic error (next section). The example URL string specifies the file types we expect a client to want to retrieve. The dollar sign (\$) says that all the strings listed must be located at the end of the request URL. This line will allow:

- ^V allows the root request http://mydomain.com/ to be accepted. / is expanded into /index.html by the web server
- \.html HTML page files
- \.css Cascading Style Sheets
- \.jpg JPG pictures
- favicon\.ico is the only .ico file
- robots\.txt is the only text file
- \.png PNG pictures

- \$ says that each of these strings have to be located at the end of the line

Serve an empty 1x1 gif _OR_ an error 204 (No Content) for favicon.ico : Using either of these lines simply direct the Nginx daemon to serve out an empty 1x1 pixel (43 byte) favicon.ico file to the client or send back a return code 204, meaning "No content". When either option is in place you do not need a file called favicon.ico in your document root. This is perfect for anyone who sees the favicon.ico as useless and does not want to waste any bandwidth on it, but also do not want to see "file not found" errors in their logs. For more information make sure to read the section titled, "The Cursed favicon.ico" on our [Webserver Optimization and Bandwidth Saving Tips](#) page.

System Maintenance : This function will look for the file /system_maintenance.html in the document root. If the file exists then _ALL_ client requests will be redirected to this file with an error code 503 (Service Unavailable). If the file does not exist then your pages will be served as normal. The purpose is so that you can work on your site and still keep the Nginx daemon up to show helpful information to your users. The system_maintenance.html file can contain something as simple as "Site Temporarily Down. Be back up soon." It is vitally important to send clients an error code 503 in order to notify them that the page they are looking for has NOT changed, but that the site is temporally down. Google for example understands this error and will return to index your page later. If you were to redirect Google to the /system_maintenance.html page and send them a code 200 for example, Google might replace the indexing information they have with this page. Your site would then have to be completely re-indexed once you got your site back up. Definitely not what you want to happen.

All other errors get the generic error page : If the client fails the previous tests then they will receive our error_page.html. This error page will be sent out for all error codes listed (400-417, 500-505). Note the use of the **internal** directive? This means that an external client will not be able to access the /error_page.html file directly, only Nginx can serve this file to a client.

Starting the daemon

Make sure that the user and group Nginx is going to run as exists and can access the files in document root. Our example file will run the child daemons as the user "nginx" and the group "nginx".

Now that you have the config file installed in /etc/nginx.conf and configured you can start the daemon by hand with "**usr/local/sbin/nginx**" or add the following into /etc/rc.local to start the Nginx daemon on boot.

```
#### Nginx start in /etc/rc.local
if [ -x /usr/local/sbin/nginx ]; then
    echo -n ' Nginx'; /usr/local/sbin/nginx
fi
```

In Conclusion

Nginx has many more options not covered in this how to. We highly suggest taking some time to read the [Nginx English Wiki](#) if you need more web functionality. If you are happy with the options we have started out with then at this point all that is left is finding some content to serve and setup your web tree.

Strip Unnecessary White space like spaces, tabs and new line characters

How much bandwidth can you expect to save by stripping out white space? On average , one could expect to save 2% of the total size of the HTML pages written by hand or previously un-optimized. If your average page size is 100 kilobytes you could save around 2 kilobytes every time they were served. If you served a hundred thousand pages per day you could reduce your bandwidth usage by 200 megabytes per day. Not a lot, but every bit makes a difference.

This is a simple perl script called "strip_whitespace.pl". It will read in any html file and output the stripped version. Use this script to make a published copy of your html docs while keeping the human readable versions in a private directory. BTW, we use this code with the pre-compression option in Nginx to serve out pre-stripped, pre-compressed files to save on bandwidth and CPU time.

```

#!/usr/bin/perl -w
#
## Calomel.org -- strip_whitespace.pl
#
## PURPOSE: This program will strip out all
##    whitespace from a HTML file except what is
##    between the pre and /pre and tags.
#
## DEPENDANCIES: p5-HTML-Parser which you can get by CPAN or
##    installing a package from your OS supporters.
#
## USAGE: ./strip_whitespace.pl < input.html > output.html
#

use HTML::Parser;
my $preserve = 0;

# Ignore any test between the /pre tags
sub process_tag
{
    my ($tag, $text) = @_;
    if ($tag eq 'pre') { $preserve = 1; }
    elsif ($tag eq '/pre') { $preserve = 0; }
    print $text;
}

# Replace all white space with a single space except what
# is between the pre tags. This includes all tabs (\t),
# returns (\r) and new line characters (\n).
sub process default

```

Speed tips for an optimized, low latency, high speed and efficient web server

Whether your website is new or old everyone wants to send out information to their users quickly. A fast serving website is known to keep users around as people do not like to wait for a page to load. Google even includes the speed your webserver loads through Googlebot in its calculations of PageRank. In essence, the faster your page loads the higher your Google rank and the happier your users are. Lets take a look at a few ways to get your pages load quickly.

Network speed and capacity: First and foremost you have to take in account your connection speed and packet rate to your users. If you have a lot of data to send and your are limited by your upload speed of your connection then your server might be fast enough to send out the data, but you simply can not get the bits uploaded fast enough to the user. Also be aware of the packet switching rate of your connection is usually rated in packets per seconds or pps. Make sure that your network is up to the task of delivering your data.

Lets say you have a main web page and it contains 10 objects,(i.e. pictures, text, css and so

on) and the total size of all the objects are 300 kilobytes. 300 KB by the way is the average size of web page these days according to Google. If you want to serve one person per second then you will need at least 400 KB/sec of upload bandwidth including the TCP stack over head. Truthfully, this would be doable with most home DSL, Cable or FIOS connections. If you want to serve that same page to 10 people per second then you will need 10x the bandwidth or 4 megabytes per second. You need to make sure your upload bandwidth can handle the site you are trying to serve.

Workers and worker_processes: ideally you want to setup nginx to use one(1) less worker then you have physical cores in your machine. A four(4) core machine would have three(3) workers. The last free core is left to be used by the OS and interrupts for the network interface. Setup the worker_processes to as low a number as you feel can support the maximum number of concurrent users connecting to your site. When you multiply workers, three(3) in this case, against worker_processes you get to total amount of client connections accepted by nginx. Lets say our webserver will never expect to receive more then 50 concurrent users at any one time. We can set worker_process to as little as 25 since workers times worker_processes would equal 75. The advantage of a low worker_process count is fast reaction times when a client connects. When a client issues a request there is some serial contention when Nginx assigns a processes to fulfilling a request. By keeping worker_processes low you gain fast reaction speed.

Google SPDY: Support for Google's SPDY implementation has just been added as a patch to the newest version of nginx. SPDY is already supported in the newest versions of Google Chrome, Mozilla Firefox and Opera. The SPDY project defines and implements an application-layer protocol for the web which greatly reduces latency. The high-level goals for SPDY are:

- To target a 50% reduction in page load time. Our preliminary results have come close to this target
- To minimize deployment complexity. SPDY uses TCP as the underlying transport layer, so requires no changes to existing networking infrastructure
- To avoid the need for any changes to content by website authors. The only changes required to support SPDY are in the client user agent and web server applications

At this time the only way to get SPDY in nginx is to build from source since you need OpenSSL 1.0.1 with the Next Protocol Negotiation TLS extension. Take a look at the announcement for the [SPDY draft 2 module for nginx](#) for more information. Calomel.org is currently running SPDY and we have noticed reduced client load times by at least 63%. Just incredible.

Keep-alives enabled: make sure you use Nginx's capability to use keep-alive requests for all non-SPDY connections. Keep-alive requests allow the client to request multiple items through a single TCP connection. This is efficient delivery method compared to opening a TCP connection for every item. For example, it might take a client 0.5 second to open a new TCP connection when they could have just asked for another object in a current TCP stream which is practically instant. If you had 50 objects on the page and had to open 50 connections that could mean the page would load in 25 seconds. If you allowed the client to request the items through keep-alive the page could have loaded in just a few seconds. Its that big of a difference.

Static data is faster: make sure the method you are serving pages at can handle the load of your clients. For example, static content can be delivered faster then dynamically generated content. Try to have as many static files as possible on your site if you can. You

will see a big increase in page load times if you do.

Cache dynamic content: If your content is dynamically generated then take a look at caching responses to the clients. Caching will increase the speed of your dynamically generated content significantly. If a page is the same for all clients there is no reason to generate it for every request. Cache those pages and decrease your response times to milliseconds.

High speed RAM disk: setup your htdocs directory in a [RAM disk or RAM drive](#) and content delivery will be almost instantaneous. A ram disk is simply a directory which points to a section of ram instead of the normal hard drive. Ram is thousands of times faster than the hard drive or even a SSD drive so it makes sense to use it for a web server. The idea is to have your normal web files on the hard drive as your primary copy since ram disks go away when the system is rebooted. Then setup a ram drive as a directory and copy or rsync the files from the hard drive htdocs to the high speed ram disk. Then point nginx to the mount which reads from the ram disk.

Open_file_cache: Once you have setup your ram drive or hard drive make sure you use Nginx's open_file_cache directive like we have in the examples above. This will allow nginx to access the files by inode number and bypass many of the slow lookups the OS needs to do to find a file. The cache will reduce access times by critical tenths of a second.

Buffer log writing: When a client accesses your server Nginx will log the action. Every time the log is written there is an I/O call to the hard drive and this is expensive. Use the "buffer=32k" directive in the access_log line to reduce the server load. This will wait until at least 32 kilobytes of log data is ready before writing it out to the log file. In fact, increase this number even larger if you have lots of traffic. It will reduce the load on your box considerably.

Strip out white space: White space is any extra lines, new line characters, extra spaces or extraneous data in the HTML page. Taking this junk out can reduce your web page size by just a few percent, but every bit counts. The smaller the page is the faster it can be sent. Right above this section we have a script called "strip_whitespace.pl" to help you out.

Make your pictures small and efficient: If you do not need a big JPG on your page then make it smaller. Crop out any extra sections of the picture that are not needed or even blur the background to allow the compression algorithm to be more efficient. The majority of bandwidth used today is for pictures according to Yahoo and Google. You can save a lot of bandwidth and increase response time by serving optimized pictures. Use a program like pngcrush to make your PNG files smaller. If you prefer a web based tool, Yahoo offers [Yahoo Smush.it](#) to compress your images for you and download the result. Take a look at Google's [Optimizing web graphics](#) for more ideas on making your pictures smaller.

To quantify the speed you can serve data at you need to test the webserver under extreme load and your real world busiest conditions.

Stress testing your system and web server

One of the tools we have found to be the best at stress testing a system is httpperf. It does a

really good job at opening connections quickly and sending requests. We noticed that many of the problems with tools is consistency and httpperf does a significantly better job than "ab" also called apache benchmark. Also note that using httpperf, apache bench or any other testing tool which just looks at single files is considered a "micro-benchmark". microbenchmark is really not a true measure of the server's performance in the real world, but a test on a specific set of circumstances.

Here is an example of a FreeBSD 9.x system we setup to serve out static images for a web site. There are a few security tweaks to the system, but mainly it is a default install with `_no_` `/boot/loader.conf` or `/etc/sysctl.conf` changes. The only purpose of this machine is to support the main web server by delivering buttons, favicons, signature images and such. We found that the majority of calls were made for these support objects and it was desired to split this function off to another box. Many sites split off their static data like Google's `gstatic.com` domain name.

To run the test you will need at least two separate boxes. One will be the client you will run httpperf on and the other will be the web server, in this case it is the ip `192.168.1.100` we are connecting to. You need at least two boxes so one box is not the limiting factor of the test. For example, you could run out of cpu time, PCI bus bandwidth or hard drive bandwidth if serving and requesting was done on the same machine. If you have more clients to run httpperf from that is even better. The limit should always be on the webserver side.

The webserver box is a simple 4 core (2.4GHz) machine with 4 gig or ram, a 320 gig SSD hard drive and a gigabit card. Using Nginx with "worker_processes 8", "worker_connections 500;" and keepalive_requests 100;" we served out an average of 74,908 images at a 12.3ms response time per request. Does the live site actually serve out this much data? No, the current average user load on the system is 6,00 sessions per second so we have room to grow. We could also add more machines and load balancer too, but to keep costs (power, rack space) down we limit this function to one machine.

```
user@machine:~$ httpperf --hog --server 192.168.1.100 --num-conn 10000 --rate
Total: connections 3444 requests 1725444 replies 1722000 test-duration 23.04

Connection rate: 149.4 conn/s (6.7 ms/conn, <=1022 concurrent connections)
Connection time [ms]: min 529.2 avg 6161.9 max 9756.6 median 6409.5 stddev 1
Connection time [ms]: connect 1.2
Connection length [replies/conn]: 500.000

Request rate: 74867.6 req/s (0.0 ms/req)
Request size [B]: 65.0

Reply rate [replies/s]: min 74683.7 avg 74908.1 max 75211.1 stddev 222.6 (4
Reply time [ms]: response 12.3 transfer 0.0
Reply size [B]: header 215.0 content 151.0 footer 0.0 (total 366.0)
Reply status: 1xx=0 2xx=1722000 3xx=0 4xx=0 5xx=0

CPU time [s]: user 4.03 system 18.91 (user 17.5% system 82.1% total 99.5%)
Net I/O: 31530.5 KB/s (258.3*10^6 bps)

Errors: total 10000 client-timo 0 socket-timo 0 connrefused 0 connreset 3444
Errors: fd-unavail 6556 addrunavail 0 ftab-full 0 other 0
```

Real world load, not just stress testing

Stress testing is only good to find out what your system can do in the worst possible scenario (DDOS ?). You do not want to run your server at 100% utilization in production so stress testing is not good way to find out what the server will look like in production. For real world testing look at what your normal traffic load looks like at your busiest times of the day. This will show you how the system will behave when it is being used normally.

In our real world test case the web server was busiest when it was serving 192 concurrent sessions with 47 requests from each client (ip address). Every client can request each of those 47 items (pictures) as fast as they can to keep page load times down to a minimum. This peak load was only seen for a few minutes per hour for a total of 4 hours in the later part of the day. In our test we want to see what the webserver would look like if it had to serve out the peak load for a full 45 minutes.

The results of the test were promising; both top and httpperf results are in the following window. While running httpperf for 47 minutes straight, top showed that the web-server's load was steady at 0.42, each of the 8 nginx processes were at around 4%, system interrupts at 7% and the system as a whole was 85% idle for the entire testing time. Network load on the webserver was 1.2 megabytes per second incoming and 3.8 megabytes per second outgoing. We received requests for and served 25,627,681 small static objects total or an average of 9,024 requests per second.

```
##
#### top on the webserver box
##
last pid: 2655; load averages: 0.42, 0.45, 0.33
28 processes: 5 running, 23 sleeping
CPU 0: 1.6% user, 0.0% nice, 6.7% system, 6.7% interrupt, 85.0% idle
CPU 1: 4.7% user, 0.0% nice, 8.7% system, 0.0% interrupt, 86.6% idle
CPU 2: 1.6% user, 0.0% nice, 13.0% system, 0.0% interrupt, 85.4% idle
CPU 3: 2.8% user, 0.0% nice, 10.2% system, 0.0% interrupt, 87.0% idle
Mem: 15M Active, 11M Inact, 125M Wired, 28K Cache, 18M Buf, 3791M Free
Swap: 4096M Total, 4096M Free

  PID USERNAME  THR PRI NICE   SIZE    RES STATE  C  TIME  WCPU COMMAND
2599 nobody      1  26   0 22892K 4636K CPU0   0   0:56  5.27% nginx
2601 nobody      1  20   0 22892K 4636K kqread  0   1:00  4.20% nginx
2600 nobody      1  25   0 22892K 4624K CPU3   3   1:04  3.37% nginx
2597 nobody      1  25   0 22892K 4624K RUN     0   0:59  3.37% nginx
2598 nobody      1  20   0 22892K 4632K kqread  1   0:59  3.27% nginx
2602 nobody      1  20   0 22892K 4616K kqread  1   0:49  2.98% nginx
2595 nobody      1  25   0 22892K 4624K CPU2   2   1:10  2.20% nginx
2596 nobody      1  20   0 22892K 4624K kqread  0   1:13  0.00% nginx

##
#### httpperf on the client box (stopped after ~47 minutes)
##
user@machine:~$ httpperf --hog --server=192.168.1.100 --wsess=100000,47,0 -

Total: connections 545279 requests 25627681 replies 25627681 test-duration
```

These tests are just a guide. Your results will be different as the speed and latency of your network, machine speed and object sizes will influence the results greatly. Take some time to familiarize yourself with the testing methods and tools. Then you can objectively see what your server can do and how you can make delivery of every single item on your web site as fast as possible. The best result will be an increase in your Google PageRank and increased traffic and at least your users will be thankful your site is quick to respond.

Check the [Webserver Optimization and Bandwidth Saving Tips](#) for ideas on how to serve your data even more efficiently.

Questions?

How can I test access times from the command line or cli ?

An easy way to test your site from the command line is using curl. The following line will connect to the remote site and download the the page you specify. Curl will time how long it takes to connect, then how long the server takes to start transferring the first packet of data and finally the time the server takes to complete the entire transfer. All times are in seconds. This example runs curl against our site. It took 16 milliseconds to connect and 86 milliseconds to start and complete the transfer of the static index.html. Not too bad for a SSL only site.

```
$ curl -o /dev/null -w "Connect: %{time_connect} Start Transfer: %{time_star  
Connect: 0.016 Start Transfer: 0.086 Total time: 0.086
```

Can Nginx be setup as a Forward Proxy ?

Yes, Nginx can be used as a forward proxy for LAN connected devices similar to squid. The following nginx.conf server block will have nginx listen on port 8080 of all interfaces. When you point your LAN connected machine to to proxy and ask for a web page, nginx will collect the page for you. Nginx will then forward the web page to your device. If you have slow wireless devices using a proxy might help speed up their connections.

```
# forward caching proxy for wireless LAN devices
server {
    access_log /var/log/nginx/access_proxy.log main;
    error_log /var/log/nginx/error.log error;
    listen 8080 default_server ;

    proxy_http_version 1.1;
    proxy_hide_header Etag;
    proxy_hide_header Set-Cookie;

    location / {
        resolver 8.8.8.8;
        proxy_pass http://$http_host$uri$is_args$args;
    }
}
```

Can Nginx use an external authentication script like mod_auth_external with Apache ?

Yes. With Apache you need to use mod_authnz_external or mod_auth_external to call a script which will authenticate a user in any way you wish. So, you can authenticate a username and password against a custom database server, flat text file or even some one off XML web site. You do not have to rely on limited authentication support in the webserver. Nginx has the same functionality.

To allow Nginx to run an external auth script you will need to build Nginx from source and include Maxim Dounin's ngx_http_auth_request_module. The build is really easy so do not let this cause any concern. Then start the fcgiwrap daemon which nginx will use to call the external script. Next, we will setup a sample nginx.conf calling the external auth script. The final step is writing the actual authentication script itself. Our fully working example script is written in Perl which you can easily modify. The script authenticates a user using a single username and password. If you do not like Perl then you can write in any language you are happy with as long as it can be executed in a shell like bash, sh or tcsh.

First, we need to build nginx and include the ngx_http_auth_request_module add on in the build. The following lines are just an example, but should give you some direction. Make sure you always have the latest version of Nginx in case there are any bug fixes. You will also need "git" to download the nginx-auth-request-module from github and a daemon called fcgiwrap to spawn the authentication script.

```
## Ubuntu 12.04 example install
sudo apt-get install git fcgiwrap
cd /tmp
wget http://nginx.org/download/nginx-1.3.1.tar.gz
git clone git://github.com/perusio/nginx-auth-request-module.git
tar zxvf nginx-1.3.1.tar.gz
cd nginx-1.3.1
make clean; ./configure --add-module=/tmp/nginx_http_auth_request_module-a29d7
```

The fcgiwrap daemon is used to listen on a UNIX socket for Nginx calls. Nginx itself will not execute any commands outside of itself. For this reason we need another method to allow Nginx to call an external script. Edit the /etc/fcgiwrap.conf file and make sure the user fcgiwrap is going to run as is the same as the username Nginx is running as. Next make sure that you increase the amount of child processes fcgiwrap will spawn. fcgiwrap child process can only do ONE task a time. So, if you have 50 clients authorizing concurrently you should have at least 51 fcgiwrap child processes started. The only use 84bytes or ram each so you are not wasting many resources by starting extra servers. When you have the configuration set start the fcgiwrap daemon. Make sure "/var/run/fcgiwrap.socket" is writable by the Nginx users as well.

Now that Nginx is built and fcgiwrap is listening we need to add some methods to the nginx.conf configuration file. Any of the configurations found at the beginning of this how to will work. We are just going to list the methods needed to get the external authorization script works.

The first "location" looks for calls to the /secure/ URI path. Clients who ask for this path will be prompted with a username and password box. This happens because of the auth_request directive is calling our second "location" "/auth". In the /auth method we are calling a fastcgi script called /web/scripts/authentication.pl. authentication.pl is the name of the authentication perl script.

```
# Path to secure data that need authentication access
location ~ ^/secure/[\w.-]+$ {
    auth_request /auth;
    auth_request_set $username $upstream_http_x_username;
    auth_request_set $my_error_page $upstream_http_x_error_page;
    add_header X-Set-Username $username;
}

# External Authentication
location = /auth {
    gzip off;
    fastcgi_max_temp_file_size 0;
    fastcgi_split_path_info ^((?U).+auth)(/?.+)$;
    fastcgi_intercept_errors off;
    fastcgi_pass unix:/var/run/fcgiwrap.socket;
    include /usr/local/nginx/conf/fastcgi_params;
    fastcgi_param DOCUMENT_ROOT $document_root;
    fastcgi_param PATH_INFO $fastcgi_path_info;
    fastcgi_param SCRIPT_NAME /scripts/authentication.pl;
    fastcgi_param SCRIPT_FILENAME /web/scripts/authentication.pl;
    internal;
}
```

The last part is to make the authentication script itself. The following perl script is quite long only because of all the comments we put in. Take some time to look through the script. Basically, the script will look for an "Authorization" header sent by the client. This header contains a base64 encode username and password. We decode the user/pass pair and see if it matches any known pairs. If so the script returns a code 200 header to Nginx saying that the client is allowed to get to the /secure/ directory. If the user/pass the client gave us is wrong then they are paused for 5 seconds and sent a code 401 to attempt to authenticate again.


```
#!/usr/bin/perl
#
## Nginx external authentication script called from
## Maxim Dounin's ngx_http_auth_request_module add on.
#

use strict;
use HTTP::Headers;
use MIME::Base64;

# Return codes to Nginx can be one of the following:
# 200 - access allowed
# 401 - unauthorized or prompt for username password
# 403 - forbidden or no permission
# Any other codes are considered critical errors when sent back the serve

#####

## Authorization Sent by the Client -- Check and Allow access

# The client is sending the "Authorization:" header to the server with the
# request for a file. Here we check if they are allowed to get the file by
# looking at the base64 encoded username and password. The client will ALW
# send the authorization header if they want to get in. The format of the
# is - "Authorization: Basic dXNlcjE6cGFzc2E="

# Does the "Authorization" header exist? If not, they go down to the UNAUT
if(defined($ENV{HTTP_AUTHORIZATION})) {
```

How can I protect my server from a DOS or DDOS with nginx ?

DDOS or Distributed Denial of Service attacks are primarily used to take down a server or individual service. Attacks are usually initiated for a reason. They may be from a competitor trying to disrupt your business or from someone who feels you have wronged them in some way. Attacks do cost money and time so you can be assured they will stop eventually. The question is what price will you have to pay in time and technical services to stay on line. Plan ahead and have a disaster plan in place; you may have to deploy it at the most inopportune time like the day of a product launch or even in the middle of a holiday weekend.

There are primarily three types of attacks; hitting a CPU or database intensive page which uses up server resources, using up all your network bandwidth or hitting the server with a small packet payload, high rate of packets per second. The high packet per second attack is mostly used due to the low bandwidth needed from the attacking bot net and how easy it is to resource starve the target machine if it is not setup with this type of attack in mind.

We would like to give you specific settings to fend off a DDOS, but the configuration really depends on the type of attack you are receiving. Start with the settings we used in the examples above. On top of that you will probably want to have nginx setup with short connection timeouts and make sure the server is using rate limiting as well to mitigate

aggressive clients. You may also want to take a look at a perl script we have called, [Web Server Abuse Detection Script \(perl\)](#) which will watch the logs in real time for errors and use the firewall to block those ips.

Truthfully, there is only so much you can do to weather the storm of a DOS on a single server. You may need to setup a machine in front of the main webserver to be a reverse proxy. Something like a FreeBSD box using Nginx reverse proxy and acting as a caching server plus the use of Pf (packet filter) as a real time firewall.

If the DDOS is big enough you will want to look at setting up a cluster of machines at many different data centers. You only need to rent the servers for a month or so because most DDOS attacks do not last more than a few days. The costs for the servers might be as low as \$25 per machine per data center. Each of these machines can be a Nginx reverse proxy pointing to your main server. Your main server only accepts traffic from the proxy servers and you change your domain's DNS to round robin to the rented proxy servers. Each reverse proxy will be filtering out DDOS traffic as needed, cache static data and serving clients on their own dedicated network. What is great about this approach is you control all the reverse proxies so you can setup scripting to block the attack as needed. You also have the full bandwidth of that data center's system to absorb the attack. If you need more bandwidth just sign up for another machine in another data center. It might cost you as little as a few hundred dollars per month to rent 10 servers at ten different data centers to hold off the attack and keep your customers traffic flowing.

Being hit by a big DDOS is not something you can just flip a switch and hold off unless you are ready for it and understand it. You need to plan for worst case scenarios and understand what steps you are going to take to stop an attack. Setup your servers securely, limit access to your services to sane values, try to use caching as much as possible and understand what each anti-dos mitigating step will accomplish. You may find you can hold off the script kiddies with your normal web server, a larger DOS with a reverse proxy cluster and a large DDOS with a distributed reverse proxy constellation.

In the end, can you win against a DDOS attack? Sadly, it really depends on your attacker's resources, how much money and time they have and how much business you are willing to loose if you can not out match your attackers. You might be able to withstand the onslaught if the attack is small compared to your infrastructure like when Anonymous attacked Amazon.com. It is also possible you or your company are no match for your attackers and they make you pay an "extortion" fee because it is less than the business you are losing. It is also possible that the attack is just not worth your time and you take yourself offline till the attacks go away. Take a look at the WIRED link below for a story of a gambling site with plenty of money who actually lost the good fight.

Here are some good references you may want to start your research with:

- [WIRED: Attack of the Bots](#)
- [Tuning FreeBSD to serve 100-200 thousands of connections](#)
- [Under a DDoS Attack? Here is what you can do...](#)
- [The penultimate guide to stopping a DDoS attack](#)
- [FreeBSD and Internet Attacks: Protecting yourself from denial of service](#)

How can I build Nginx with TLS 1.2 (TLS 1.0 to 1.2) support and the newest OpenSSL ?

More then likely the current OS's revision of openssl is 0.9.x or similar. Check your version of Openssl by running, "openssl version". OpenSSL before version 1.0.1 will only support TLS 1.0 and this is the root of the issue. Only the latest build of OpenSSL will support TLS 1.1 and 1.2. Also, there are many vulnerabilities to TLS 1.0 which 1.1 and 1.2 fix. Building Nginx against the newest version of OpenSSL will allow you to support TLS 1.2 clients like Opera and IE9 (Firefox and Chrome use NSS and do not support TLS 1.2 yet).

You will need to build Nginx from source and use the OpenSSL 1.0.1 (stable snapshot). The build is quite easy and this is one way to do it.

```
## make a directory to build the source of both Nginx and OpenSSL
mkdir /tmp/build
cd /tmp/build/

## wget the tar files for openssl stand snapshot. Goto this
## ftp server and get the latest 1.0.1 tar file.
ftp://ftp.openssl.org/snapshot/

## Now wget the latest Nginx source from here.
http://nginx.org/en/download.html

## Untar both files in /tmp/build/

## cd directory into the nginx source tree and execute the following line.
## SURE to change the "--with-openssl" directive in this line to point to
## path of the openssl 1.0.1 source you downloaded. This is a long line
## since we cut out a lot of the modules not needed for a static file serv
## copy and paste it somewhere you can look at it more closely.

make clean; ./configure --with-http_ssl_module --with-http_gzip_static_mod

## that's about it. When you start nginx you can test out the build
## using your new copy of openssl. For example, You can see that calomel.o
## supports TLS 1.2 with the strongest Elliptic curve Diffie Hellman key a
## and SHA384 hash.
echo 'GET HTTP/1.0' | /tmp/build/openssl-1.0.1-stable-SNAP-20111220/apps/o
...
```

How do I rotate the logs in Nginx ?

There are two option regarding Nginx log rotation: you can use logrotate or you can write your own script. Using logrotate is good if you are on a Linux system and it is available. If you are running on OpenBSD or FreeBSD using the simple script method might be better. Let's look at both.

Logrotate: This is the config for the logrotate daemon. Just make a file called "/etc/logrotate.d/nginx" with the following in it. This will rotate the log files daily and keep 12 archived copies. The Nginx daemon will then be HUP'd and start writing to the new log files. Please make sure the kill line points to the location the nginx.pid is in on your machine.

```
user@machine:~# cat /etc/logrotate.d/nginx
/var/log/nginx/*.log {
    daily
    missingok
    rotate 12
    compress
    delaycompress
    notifempty
    create 644 nobody root
    sharedscripts
    postrotate
        kill -USR1 `cat /usr/local/nginx/logs/nginx.pid` > /dev/null
    endscript
}
```

Script method log rotation: This is a simple script to move the files to another directory and date them. The script is run daily through cron.

```
## First make this script and make sure it is executable by the root user,
## "chmod 700 /root/nginx_logrotate.sh" and "chown root /root/nginx_logrotate.sh"

root@machine:~# cat /root/nginx_logrotate.sh
#!/bin/bash
#
## Nginx log rotation

## move the logs
mv /var/log/nginx/access.log /log_archives/access.log_`date +%F`
mv /var/log/nginx/error.log /log_archives/error.log_`date +%F`
mv /var/log/nginx/cache.log /log_archives/cache.log_`date +%F`

## HUP nginx to start new log files
kill -USR1 `cat /usr/local/nginx/logs/nginx.pid`

## clear out old log files after 90 days
find /log_archives/ -type f -mtime 90 -exec ls -la {} \;

## Second, add a cron job to execute the log rotation script every night at

root@machine:~# crontab -l
## nginx log rotation
59 23 * * * /root/nginx_logrotate.sh
```

How can Nginx log to a central log server ?

One way to get Nginx to log to a central log server or loghost is to use rsyslog. The newest Linux distributions should already have rsyslog installed by default. Rsyslog will simply log any lines found in the access log to the central loghost facility. Once this config is setup just restart rsyslog and look in /var/log/syslog for the webserver logs.

```
## First, add this line to the /etc/rsyslog.conf . This will add support for
$ModLoad imfile

## Second, make a file called /etc/rsyslog.d/nginx.conf and put the followin
## in it. This will tell rsyslog to look at the Nginx access log located at
## /var/log/nginx/access.log and log at the "info" level. This method is
## efficient.

root@machine:~# cat /etc/rsyslog.d/nginx.conf
$InputFileName /var/log/nginx/access.log
$InputFileTag nginx:
$InputStateFile stat-nginx-access
$InputFileSeverity info
$InputRunFileMonitor
```

How can I time the latency of multiple TCP and SSL handshakes ?

Curl is the tool of choice. Here we make two(2) connections to encrypted.google.com and time the responses from both the 3 way TCP handshake and the SSL negotiation of Google's 1024 bit rsa certificate key. We see that the first connection completes the 3 way TCP handshake in 32ms and the SSL handshake finishes 95ms after that. The second connection on the next line is all 0.00's because Google allows keepalive connections. So, the second request went over the same TCP connection as the first and thus saved us time. Keepalive's are quite useful when used correctly.

```
### bash shell
$ export URL=https://encrypted.google.com
$ curl -w "tcp: %{time_connect} ssl:%{time_appconnect} .. $URL\n" -sk -o /dev/null
tcp: 0.032 ssl:0.095 .. https://encrypted.google.com
tcp: 0.000 ssl:0.000 .. https://encrypted.google.com
```

How can I best optimize Nginx for HTTPS connections ?

SSL negotiations consume a bit more CPU resources than a standard http connection. Understand that the amount of CPU used for HTTPS is not excessive, but the negotiation process does add a bit of delay (latency) to the initial connection process. On multi-processor systems you want to run several worker processes which are no less than the

number of available CPU cores. By cores we mean real cores and not hyperthread Intel virtual cores.

The most CPU-intensive operation in HTTPS is the SSL handshake. There are a few ways to minimize the number of operations per client:

- First, enable keepalive connections to send several requests via one connection.
- Second, reuse SSL session parameters to avoid SSL handshakes for parallel and subsequent connections.
- Third, decide how many clients will connect per second and figure out if your cipher and hardware can handle the load

Enable keepalives

Enable the "keepalive" directive to allow a remote client to send multiple queries per TCP connections. Take a look at our examples above concerning `keepalive_requests` and `keepalive_timeout`. You will want to look at the average amount of objects per page you serve. If we have 10 objects (pictures, html, css, html, ect.) then set the `keepalive_requests` to something like 50 or 5 times the average. This means a client could load 5 full pages if they did not have local caching enabled before they would need to open another connection. A `keepalive_timeout` of 300 seconds is good to support long lived connections and the default 300 second timeout of a negotiated ssl connection.

```
keepalive_requests    50;
keepalive_timeout     300 300;
```

Setting up SSL session cache

The sessions stored in an SSL session cache are shared between workers and configured by the `ssl_session_cache` directive. One megabyte of the cache contains around four thousand (4000) sessions. The default cache timeout is 5 minutes and this can be increased by using the `ssl_session_timeout` directive. Here is a sample of the "Option 2" configuration from above. Just like in the example Nginx is optimized for a quad core system with 10M shared session cache:

```

worker_processes 4;

http {
    ## Global SSL options
    ssl_ciphers HIGH:!ADH:!MD5;
    ssl_prefer_server_ciphers on;
    ssl_protocols TLSv1;
    ssl_session_cache shared:SSL:10m;
    ssl_session_timeout 5m;

    server {
        listen          443;
        server_name      example.com www.example.com;
        keepalive_timeout 300 300;

        ssl              on;
        ssl_certificate  /ssl_keys/mydomain.com_ssl.crt;
        ssl_certificate_key /ssl_keys/mydomain_ssl.key;
        ...
    }
}

```

How many ssl clients and how fast can we encrypt data?

You need to know how many SSL enable clients will be connecting to your servers per second and how fast you can encrypt the data flow. As we stated before the majority of time with SSL connections is taken up by the SSL handshake. The cipher you choose and the CPU in your machine are going to be the determining factors on how fast you negotiate with clients and how fast you encrypt data.

For these tests we will be using an Intel Quad core Xeon CPU L5630 @ 2.13GHz with 4gig of 1333MHz ram. The test OS is OpenBSD v5.1 (stable) and OpenSSL 1.0.0a. AES-NI (AES New Instructions) or the Advanced Encryption Standard (AES) Instruction Set can be enabled in our BIOS and is supported by our CPU. AES-NI is an extension to the x86 instruction set architecture for microprocessors from Intel and AMD with the purpose to improve the speed of applications performing encryption and decryption using the Advanced Encryption Standard (AES). The AES standard is comprised of three block ciphers, AES-128, AES-192 and AES-256, adopted from a larger collection originally published as Rijndael.

UPDATE: OpenSSL 1.0.1 now has AES-NI support enabled by default in the (rsax) RSAX engine. You may not see an engine listed as aesni anymore. If you have an Intel cpu and BIOS enabled AES-NI support then OpenSSL 1.0.1 will use aesni. On average, aesni hardware support is 10x faster for cipher sizes of 256K or smaller and 4x faster for cipers 1024k and larger. If you run a highly loaded SSL site make sure to check for AES-NI support.

The first test will show us how fast our system can sign our ssl certificate during a handshake and how many ssl clients we can handshake with per second. A handshake is the action of the client and server opening up an encrypted connection between each other and negotiating with the site's SSL certificate. The common size of a SSL certificate is 1024, 2048 or 4096 bits. For example we sign calomel.org with a rsa 4096 bit key. So, when a client connects to our site they must negotiate with us with a rsa 4096 bit certificate.

The "sign" and "sign/s" are the values we want to examine. Make special note that the first

results are for a single core and Nginx can work with multiple cores depending on your "worker_processes" directive. At rsa 4096, like what calomel.org is using, openssl specifies this machine can handshake with 44.9 clients per second per core and complete each certificate signing in 22ms (0.022). At rsa 2048 openssl can handle 300.1 SSL handshakes per second per core and sign in 0.3ms (0.003).

Note: AES-NI does NOT increase the speed of handshakes at all.

```
#### Intel Quad core Xeon CPU L5630 @ 2.13GHz
user@machine: openssl speed rsa
      sign      verify      sign/s verify/s
rsa 4096 bits 0.022267s 0.000336s    44.9   2977.1 (1 core)
rsa 2048 bits 0.003332s 0.000092s   300.1  10814.8 (1 core)
rsa 1024 bits 0.000535s 0.000030s  1867.7  33674.9 (1 core)

user@machine: openssl speed -multi 4 rsa
      sign      verify      sign/s verify/s
rsa 4096 bits 0.005498s 0.000084s   181.9  11922.5 (4 cores)
rsa 2048 bits 0.000831s 0.000023s  1203.6  43244.5 (4 cores)
rsa 1024 bits 0.000134s 0.000007s  7435.1 134482.8 (4 cores)
```

The second test shows how much block cipher data our system can encrypt in real time. This simulates a bulk data transfer like uploading or downloading a large file encrypted over SSL after the handshake is completed. We see that our machine and openssl can process over (145855.83k / 1024) 142 megabytes per second per core of AES 256bit, 8192 byte chunked encrypted data. This is where AES-NI can help the most. If we enable AES-NI we can almost quadruple the encryption / decryption speed as a single core can processes (402330.62k / 1024) 392 megabytes per second per core up from 142 megabytes per second per core. If we use all four(4) cores we could saturate a 10 gigabit link. For more information see our [AES-NI SSL Performance Study](#).

```
### single core AES 256bit encryption
user@machine: openssl speed -engine aesni aes-256-cbc
The 'numbers' are in 1000s of bytes per second processed.
type          16 bytes      64 bytes      256 bytes      1024 bytes      8192 by
aes-256 cbc   51537.86k     53793.62k     54669.65k     143895.81k     145855.
aes-256-cbc   269689.64k    328777.67k    401489.65k    400522.80k    402330.

### four(4) cores AES 256bit encryption
user@machine: openssl -multi 4 -engine aesni speed aes-256-cbc
The 'numbers' are in 1000s of bytes per second processed.
type          16 bytes      64 bytes      256 bytes      1024 bytes      8192 by
evp           197800.42k    212958.32k    218247.91k     574456.18k     578834.
evp          1078184.98k    1222598.23k    1574396.94k    1610181.83k    1609346.
```

SSL optimization conclusions

How do I know if my machine supports AES-NI? The new range of Intel and AMD CPU's have support for AES-NI. You should see a section in your BIOS called "AES-NI" under the

CPU processor options. You can also check to see if your OS supports AES-NI. We used OpenBSD v5.1 stable and this does. You can use the command "openssl engine -t" and it should report "(aesni) Intel AES-NI engine [available]". If you see "(no-aesni)" anywhere on the line then AES-NI is NOT supported for your hardware even though the engine is available. Again, you need to make sure you enable the AES-NI option in the BIOS if available before the OS can support the option. Be aware that in order for you to take advantage of AES-NI the hardware (Intel L5630 CPU), the BIOS, the OS (OpenBSD v5.1 stable) and the program (OpenSSL 1.0.0.a) you use must all support AES-NI instructions.

What if you need more speed than a single machine can offer? In most cases it is more cost effective to buy a weaker SSL certificate (1024 bit or 2048 bit compared to 4096 bit) than it is to buy SSL hardware. Looking at the results of the test above we see our machine can sign a 4096bit key 181.9 handshakes per second using all four(4) cores. What if you need to support 500 handshakes per second? You can buy more machines or you could just use a weaker rsa key if your business plan allows it. Our tests show this machine can support 1203.6 handshakes per second when using a rsa 2048 certificate key.

You can also take a look at some hardware SSL accelerator cards. They are beyond the scope of this article. What we have seen is hardware cards add too much overhead to the process to be truly viable and they normally do not help to speed up the handshake process. The hardware cards normally just speed up the AES block cipher encryption like AES-NI did. We prefer to add more machines to our cluster as this is the best price to performance option. If you have a few moderately powered front end SSL web proxies this will be a lot less expensive to run than one huge machine. Also, if one of the small front end goes down the others can take over. If the huge machine dies then you are offline. We prefer the smaller modular design.

Can Nginx support many HTTPS domains on a single ip address ?

Yes. A solution for running several HTTPS servers on a single IP address is TLSv1.1 [Server Name Indication](#) extension (SNI, RFC3546), which allows a browser to pass a requested server name during the SSL handshake and, therefore, the server will know which certificate it should use for the connection. Support is available in all modern browsers. Note: regardless of server SNI support, older browsers always get the certificate of default server and they complain if a server name does not match a certificate's server name. Theoretically, after this step you may redirect them to an other server, but it's too late from user point of view.

In order to use SNI in Nginx, SNI must be supported in both the OpenSSL library with which the Nginx binary has been built as well as the library to which it is being dynamically linked at run time. OpenSSL supports SNI since 0.9.8f version if it was built with config option --enable-tlsex. Since OpenSSL 0.8.9j this option is enabled by default. If nginx was built with SNI support, then nginx will show this when run with the -V switch:

```
$ nginx -V
...
TLS SNI support enabled
...
```

However, if the SNI-enabled nginx is linked dynamically to an OpenSSL library without SNI support, nginx displays the warning:

```
nginx was built with SNI support, however, now it is linked
dynamically to an OpenSSL library which has no tlsext support,
therefore SNI is not available
```

When using SSL, where does Nginx get its entropy ?

Nginx uses OpenSSL's default entropy source. On Unix systems OpenSSL will try to use `/dev/urandom`, `/dev/random`, `/dev/srandom` one after another. On FreeBSD `/dev/urandom` is symlink to `/dev/random`. On OpenBSD `/dev/arandom` is used. We highly advise using `/dev/arandom` On OpenBSD or FreeBSD if possible as it is extremely fast and uses high entropy. `/dev/arandom` uses the `arc4random()` method.

The `arc4random()` function provides a high quality 32-bit pseudo-random number quickly. `arc4random()` seeds itself on a regular basis from the kernel strong random number subsystem described in `random(4)`. On each call, an ARC4 generator is used to generate a new result. The `arc4random()` function uses the ARC4 cipher key stream generator, which uses 8*8 8-bit S-Boxes. The S-Boxes can be in about (2^{1700}) states.

The entropy source can be redefined by using the "SSLRandomSeed" directive and pointing to the new device. For example you can use the [Simtec Electronics Entropy USB Key](#) for high-quality random number generation.

For more detailed information about entropy check out our [Entropy and random number generators](#) page.

Is Nginx susceptible to the Slowloris DoS attack like Apache ?

It can be. Slowloris (`slowloris.pl`) holds connections open by sending partial HTTP requests. It continues to send subsequent headers at regular intervals to keep the sockets from closing. In this way web servers can be quickly tied up. In particular, servers that have threading enabled will tend to be more vulnerable, by virtue of the fact that they attempt to limit the amount of threading they'll allow.

Slowloris will wait for all the sockets to become available before it's successful at consuming them, so if it's a high traffic website, it may take a while for the site to free up its sockets. So while you may be unable to see the website from your client, others may still be able to see it until all sockets are freed by them and consumed by Slowloris. This is because other users of the system must finish their requests before the sockets become available for

Slowloris to consume.

Though it is difficult to be completely immune to this type of attack, Nginx is quite resilient. You can practice good web server security by doing the following:

1. Limit the amount of connections and how fast those connections can be made from any ip address to your web server by use of your firewall. This can be done with [PF \(we have a "how to"\)](#) or Iptables.
2. Time out clients who take too long to perform any action.
3. Drop clients immediately who send invalid data.
4. Limit the amount of memory, cpu time and system resources the webserver can use. This is so the webserver can not take down the machine if the site is attacked.

The following Nginx directives will timeout clients who take too long to communicate their intentions to the server. The ignore_invalid_headers directive will drop any client trying to send invalid headers to the server. The explanations of each one is listed above on this page.

- client_header_timeout 60;
- client_body_timeout 60;
- keepalive_timeout 300 300;
- ignore_invalid_headers on;
- send_timeout 60;

How do I setup log rotation for Nginx logs on OpenBSD?

Add the following two lines to the bottom of the /etc/newsyslog.conf file. This will rotate the logs if they are larger then 512KB and gzip the old file. We will keep a total of 4 log files.

```
root@machine# vi /etc/newsyslog.conf

# logfile          owner:group  mode count size when flags
/var/log/nginx/access.log  root:wheel  644  4   512  *    Z    /va
/var/log/nginx/error.log   root:wheel  644  4   512  *    Z    /va
```

Is it possible to ask Nginx to look in Memcached, but if not found to look on the local file system before passing to the back end? Would this make things more efficient?

Yes, it's possible, but it would NOT be more efficient.

Sending the file from the local file system is most efficient way if kernel VM has cached that file: then `sendfile()` will be a zero-copy operation: you save memory and CPU time. Working with memcached or a back end server requires:

- copy operations to the kernel by memcached or back end systems,
- then copy operations from the kernel by nginx,
- then copy operations to the kernel by nginx

If memcached or a back end server are on the same computer as nginx, then all these operations involve context switches between nginx and memcached or back end. -Igor Sysoev

In what circumstances would Nginx take advantage of multiple CPUs or cores?

From my experience nginx needs more CPUs in 3 cases:

- nginx does a lot of gzip'ing
- nginx handles many SSL connections
- the kernel processes a lot of TCP connections of around 3,000 requests/s.

For example, one could use a quad core for a PHP back end and a dual core for the nginx proxy to handle the connections and compressing the output. -Igor Sysoev

I am interested in setting up a SSL encrypted server (https port 443). Can your help me?

Yes. Please check out our [Guide to Webserver SSL Certificates](#). Setting up SSL through Nginx is quite easy. What is most important is understanding how SSL encryption works and what your expectations should be.

Do you have a listing of all of the nginx directives?

The nginx wiki site has all of the config directives listed on the [Nginx Wiki Configuration File Options](#) page.

Can you suggest a good web page optimization reporting site?

We highly suggest taking a look at [PageTest - Web Page Optimization and Performance Test](#). It is free and will analyze a page and show graphs pertaining to your sites performance and

speed of delivery.

How can I test to see if my server is actually compressing/gzip'ing served files?

Look at the logs and take a look at the end of each line. In the log format directive from our example above we defined the \$gzip_ratio to show if a file is compressed. If you would rather use an external site for verification then check out [WhatIsMyIp Gzip Test](#) page. Put in the full URL in question and it will display the results. Finally, you can use the utility cURL, "curl -I --compressed http://localhost". This will show you the headers returned from the server. Look for a header "Content-Encoding: gzip".

I am looking for a simpler web server. Can your help me?

If you want a dead simple web server then check out the [tHttpd "how to"](#). It is a webserver that is ready to use by executing one line and it works really well too.

