CHAPTER 4

Massively Scalable Content Caching

4.0 Introduction

Caching accelerates content serving by storing request responses to be served again in the future. Content caching reduces load to upstream servers, caching the full response rather than running computations and queries again for the same request. Caching increases performance and reduces load, meaning you can serve faster with fewer resources. Scaling and distributing caching servers in strategic locations can have a dramatic effect on user experience. It's optimal to host content close to the consumer for the best performance. You can also cache your content close to your users. This is the pattern of content delivery networks, or CDNs. With NGINX you're able to cache your content wherever you can place an NGINX server, effectively enabling you to create your own CDN. With NGINX caching, you're also able to passively cache and serve cached responses in the event of an upstream failure.

4.1 Caching Zones

Problem

You need to cache content and need to define where the cache is stored.

Solution

Use the `proxy_cache_path` directive to define shared memory cache zones and a location for the content:

```
proxy_cache_path /var/nginx/cache
                 keys_zone=CACHE:60m
                 levels=1:2
                 inactive=3h
                 max_size=20g;
proxy_cache CACHE;
```

The cache definition example creates a directory for cached responses on the filesystem at */var/nginx/cache* and creates a shared memory space named `CACHE` with 60 megabytes of memory. This example sets the directory structure levels, defines the release of cached responses after they have not been requested in 3 hours, and defines a maximum size of the cache of 20 gigabytes. The `proxy_cache` directive informs a particular context to use the cache zone. The `proxy_cache_path` is valid in the HTTP context, and the `proxy_cache` directive is valid in the HTTP, server, and location contexts.

Discussion

To configure caching in NGINX, it's necessary to declare a path and zone to be used. A cache zone in NGINX is created with the directive `proxy_cache_path`. The `proxy_cache_path` designates a location to store the cached information and a shared memory space to store active keys and response metadata. Optional parameters to this directive provide more control over how the cache is maintained and accessed. The `levels` parameter defines how the file structure is created. The value is a colon-separated value that declares the length of subdirectory names, with a maximum of three levels. NGINX caches based on the cache key, which is a hashed value. NGINX then stores the result in the file structure provided, using the cache key as a file path and breaking up directories based on the `levels` value. The `inactive` parameter allows for control over the length of time a cache item will be hosted after its last use. The size of the cache is also configurable with the use of the `max_size` parameter. Other parameters relate to the cache-loading process, which loads the cache keys into the shared memory zone from the files cached on disk.

4.2 Caching Hash Keys

Problem

You need to control how your content is cached and looked up.

Solution

Use the `proxy_cache_key` directive along with variables to define what constitutes a cache hit or miss:

```
proxy_cache_key "$host$request_uri $cookie_user";
```

This cache hash key will instruct NGINX to cache pages based on the host and URI being requested, as well as a cookie that defines the user. With this you can cache dynamic pages without serving content that was generated for a different user.

Discussion

The default `proxy_cache_key`, which will fit most use cases, is `"$scheme$proxy_host$request_uri"`. The variables used include the scheme, HTTP or HTTPS, the `proxy_host`, where the request is being sent, and the request URI. All together, this reflects the URL that NGINX is proxying the request to. You may find that there are many other factors that define a unique request per application, such as request arguments, headers, session identifiers, and so on, to which you'll want to create your own hash key.[1]

Selecting a good hash key is very important and should be thought through with understanding of the application. Selecting a cache key for static content is typically pretty straightforward; using the hostname and URI will suffice. Selecting a cache key for fairly dynamic content like pages for a dashboard application requires more knowledge around how users interact with the application and the degree of variance between user experiences. Due to security concerns you may not want to present cached data from one user to another without fully understanding the context. The `proxy_cache_key` directive configures the string to be hashed for the cache key. The

1 Any combination of text or variables exposed to NGINX can be used to form a cache key. A list of variables is available in NGINX: *http://nginx.org/en/docs/varindex.html*.

`proxy_cache_key` can be set in the context of HTTP, server, and location blocks, providing flexible control on how requests are cached.

4.3 Cache Bypass

Problem

You need the ability to bypass the caching.

Solution

Use the `proxy_cache_bypass` directive with a nonempty or nonzero value. One way to do this is by setting a variable within location blocks that you do not want cached to equal 1:

```
proxy_cache_bypass $http_cache_bypass;
```

The configuration tells NGINX to bypass the cache if the HTTP request header named `cache_bypass` is set to any value that is not `0`.

Discussion

There are a number of scenarios that demand that the request is not cached. For this, NGINX exposes a `proxy_cache_bypass` directive so that when the value is nonempty or nonzero, the request will be sent to an upstream server rather than be pulled from the cache. Different needs and scenarios for bypassing cache will be dictated by your applications use case. Techniques for bypassing cache can be as simple as a using a request or response header, or as intricate as multiple map blocks working together.

For many reasons, you may want to bypass the cache. One important reason is troubleshooting and debugging. Reproducing issues can be hard if you're consistently pulling cached pages or if your cache key is specific to a user identifier. Having the ability to bypass the cache is vital. Options include but are not limited to bypassing the cache when a particular cookie, header, or request argument is set. You can also turn off the cache completely for a given context such as a location block by setting `proxy_cache off;`.

4.4 Cache Performance

Problem

You need to increase performance by caching on the client side.

Solution

Use client-side cache control headers:

```
location ~* \.(css|js)$ {
  expires 1y;
  add_header Cache-Control "public";
}
```

This location block specifies that the client can cache the content of CSS and JavaScript files. The `expires` directive instructs the client that their cached resource will no longer be valid after one year. The `add_header` directive adds the HTTP response header `Cache-Control` to the response, with a value of `public`, which allows any caching server along the way to cache the resource. If we specify private, only the client is allowed to cache the value.

Discussion

Cache performance has many factors, disk speed being high on the list. There are many things within the NGINX configuration you can do to assist with cache performance. One option is to set headers of the response in such a way that the client actually caches the response and does not make the request to NGINX at all, but simply serves it from its own cache.

4.5 Purging

Problem

You need to invalidate an object from the cache.

Solution

Use the purge feature of NGINX Plus, the `proxy_cache_purge` directive, and a nonempty or zero-value variable:

```
map $request_method $purge_method {
    PURGE 1;
    default 0;
}
server {
    ...
    location / {
        ...
        proxy_cache_purge $purge_method;
    }
}
```

In this example, the cache for a particular object will be purged if it's requested with a method of `PURGE`. The following is a `curl` example of purging the cache of a file named `main.js`:

```
$ curl -XPURGE localhost/main.js
```

Discussion

A common way to handle static files is to put a hash of the file in the filename. This ensures that as you roll out new code and content, your CDN recognizes it as a new file because the URI has changed. However, this does not exactly work for dynamic content to which you've set cache keys that don't fit this model. In every caching scenario, you must have a way to purge the cache. NGINX Plus has provided a simple method of purging cached responses. The `proxy_cache_purge` directive, when passed a nonzero or nonempty value, will purge the cached items matching the request. A simple way to set up purging is by mapping the request method for `PURGE`. However, you may want to use this in conjunction with the `geo_ip` module or simple authentication to ensure that not anyone can purge your precious cache items. NGINX has also allowed for the use of *, which will purge cache items that match a common URI prefix. To use wildcards you will need to configure your `proxy_cache_path` directive with the `purger=on` argument.

4.6 Cache Slicing

Problem

You need to increase caching effiency by segmenting the file into fragments.

Solution

Use the NGINX `slice` directive and its embedded variables to divide the cache result into fragments:

```
proxy_cache_path /tmp/mycache keys_zone=mycache:10m;
server {
    ...
    proxy_cache mycache;
    slice 1m;
    proxy_cache_key $host$uri$is_args$args$slice_range;
    proxy_set_header Range $slice_range;
    proxy_http_version 1.1;
    proxy_cache_valid  200 206 1h;

    location / {
        proxy_pass http://origin:80;
    }
}
```

Discussion

This configuration defines a cache zone and enables it for the server. The `slice` directive is then used to instruct NGINX to slice the response into 1 MB file segments. The cache files are stored according to the `proxy_cache_key` directive. Note the use of the embedded variable named `slice_range`. That same variable is used as a header when making the request to the origin, and that request HTTP version is upgraded to HTTP/1.1 because 1.0 does not support byte-range requests. The cache validity is set for response codes of `200` or `206` for one hour, and then the location and origins are defined.

The Cache Slice module was developed for delivery of HTML5 video, which uses byte-range requests to pseudostream content to the browser. By default, NGINX is able to serve byte-range requests from its cache. If a request for a byte-range is made for uncached content, NGINX requests the entire file from the origin. When you use the Cache Slice module, NGINX requests only the necessary segments from the origin. Range requests that are larger than the slice size, including the entire file, trigger subrequests for each of the required segments, and then those segments are cached. When all of the segments are cached, the response is assembled and sent to the client, enabling NGINX to more efficiently cache and serve content requested in ranges. The Cache Slice module should be used only on large files that do not change. NGINX validates the ETag each time

it receives a segment from the origin. If the ETag on the origin changes, NGINX aborts the transaction because the cache is no longer valid. If the content does change and the file is smaller or your origin can handle load spikes during the cache fill process, it's better to use the Cache Lock module described in the blog listed in the following Also See section.

Also See

Smart and Efficient Byte-Range Caching with NGINX & NGINX Plus

CHAPTER 5

Programmability and Automation

5.0 Introduction

Programmability refers to the ability to interact with something through programming. The API for NGINX Plus provides just that: the ability to interact with the configuation and behavior of NGINX Plus through an HTTP interface. This API provides the ability to reconfigure NGINX Plus by adding or removing upstream servers through HTTP requests. The key-value store feature in NGINX Plus enables another level of dynamic configuration—you can utilize HTTP calls to inject information that NGINX Plus can use to route or control traffic dynamically. This chapter will touch on the NGINX Plus API and the key-value store module exposed by that same API.

Configuration management tools automate the installation and configuration of servers, which is an invaluable utility in the age of the cloud. Engineers of large-scale web applications no longer need to configure servers by hand; instead, they can use one of the many configuration management tools available. With these tools, engineers can write configurations and code one time to produce many servers with the same configuration in a repeatable, testable, and modular fashion. This chapter covers a few of the most popular configuration management tools available and how to use them to install NGINX and template a base configuration. These examples are extremely basic but demonstrate how to get an NGINX server started with each platform.

5.1 NGINX Plus API

Problem

You have a dynamic environment and need to reconfigure NGINX Plus on the fly.

Solution

Configure the NGINX Plus API to enable adding and removing servers through API calls:

```
upstream backend {
    zone http_backend 64k;
}
server {
    # ...
    location /api {
        api [write=on];
        # Directives limiting access to the API
        # See chapter 7
    }

    location = /dashboard.html {
        root   /usr/share/nginx/html;
    }
}
```

This NGINX Plus configuration creates an upstream server with a shared memory zone, enables the API in the `/api` location block, and provides a location for the NGINX Plus dashboard.

You can utilize the API to add servers when they come online:

```
$ curl -X POST -d '{"server":"172.17.0.3"}' \
  'http://nginx.local/api/3/http/upstreams/backend/servers/'

{
  "id":0,
  "server":"172.17.0.3:80",
  "weight":1,
  "max_conns":0,
  "max_fails":1,
  "fail_timeout":"10s",
  "slow_start":"0s",
  "route":"",
  "backup":false,
  "down":false
}
```

The `curl` call in this example makes a request to NGINX Plus to add a new server to the backend upstream configuration. The HTTP method is a POST, and a JSON object is passed as the body. The NGINX Plus API is RESTful; therefore, there are parameters in the request URI. The format of the URI is as follows:

```
/api/{version}/http/upstreams/{httpUpstreamName}/servers/
```

You can utilize the NGINX Plus API to list the servers in the upstream pool:

```
$ curl 'http://nginx.local/api/3/http/upstreams/backend/servers/'
[
  {
    "id":0,
    "server":"172.17.0.3:80",
    "weight":1,
    "max_conns":0,
    "max_fails":1,
    "fail_timeout":"10s",
    "slow_start":"0s",
    "route":"",
    "backup":false,
    "down":false
  }
]
```

The `curl` call in this example makes a request to NGINX Plus to list all of the servers in the upstream pool named backend. Currently, we have only the one server that we added in the previous `curl` call to the API. The request will return a upstream server object that contains all of the configurable options for a server.

Use the NGINX Plus API to drain connections from an upstream server, preparing it for a graceful removal from the upstream pool. You can find details about connection draining in Recipe 2.8:

```
$ curl -X PATCH -d '{"drain":true}' \
  'http://nginx.local/api/3/http/upstreams/backend/servers/0'
{
  "id":0,
  "server":"172.17.0.3:80",
  "weight":1,
  "max_conns":0,
  "max_fails":1,
  "fail_timeout":
  "10s","slow_start":
  "0s",
  "route":"",
```

```
    "backup":false,
    "down":false,
    "drain":true
}
```

In this `curl`, we specify that the request method is `PATCH`, we pass a JSON body instructing it to drain connections for the server, and specify the server ID by appending it to the URI. We found the ID of the server by listing the servers in the upstream pool in the previous `curl` command.

NGINX Plus will begin to drain the connections. This process can take as long as the length of the sessions of the application. To check in on how many active connections are being served by the server you've begun to drain, use the following call and look for the `active` attribute of the server being drained:

```
$ curl 'http://nginx.local/api/3/http/upstreams/backend'
{
   "zone" : "http_backend",
   "keepalive" : 0,
   "peers" : [
      {
         "backup" : false,
         "id" : 0,
         "unavail" : 0,
         "name" : "172.17.0.3",
         "requests" : 0,
         "received" : 0,
         "state" : "draining",
         "server" : "172.17.0.3:80",
         "active" : 0,
         "weight" : 1,
         "fails" : 0,
         "sent" : 0,
         "responses" : {
            "4xx" : 0,
            "total" : 0,
            "3xx" : 0,
            "5xx" : 0,
            "2xx" : 0,
            "1xx" : 0
         },
         "health_checks" : {
            "checks" : 0,
            "unhealthy" : 0,
            "fails" : 0
         },
         "downtime" : 0
      }
```

```
    ],
    "zombies" : 0
}
```

After all connections have drained, utilize the NGINX Plus API to remove the server from the upstream pool entirely:

```
$ curl -X DELETE \
  'http://nginx.local/api/3/http/upstreams/backend/servers/0'
[]
```

The `curl` command makes a `DELETE` method request to the same URI used to update the servers' state. The `DELETE` method instructs NGINX to remove the server. This API call returns all of the servers and their IDs that are still left in the pool. Because we started with an empty pool, added only one server through the API, drained it, and then removed it, we now have an empty pool again.

Discussion

The NGINX Plus exclusive API enables dynamic application servers to add and remove themselves to the NGINX configuration on the fly. As servers come online, they can register themselves to the pool, and NGINX will start sending it load. When a server needs to be removed, the server can request NGINX Plus to drain its connections, and then remove itself from the upstream pool before it's shut down. This enables the infrastructure, through some automation, to scale in and out without human intervention.

Also See

NGINX Plus API Swagger Documentation

5.2 Key-Value Store

Problem

You need NGINX Plus to make dynamic traffic management decisions based on input from applications.

Solution

Set up the cluster-aware key-value store and API, and then add keys and values:

```
keyval_zone zone=blacklist:1M;
keyval $remote_addr $num_failures zone=blacklist;

server {
    # ...
    location / {
        if ($num_failures) {
            return 403 'Forbidden';
        }
        return 200 'OK';
    }
}
server {
    # ...
    # Directives limiting access to the API
    # See chapter 6
    location /api {
        api write=on;
    }
}
```

This NGINX Plus configuration uses the `keyval_zone` directory to build a key-value store shared memory zone named `blacklist` and sets a memory limit of 1 MB. The `keyval` directive then maps the value of the key matching the first parameter `$remote_addr` to a new variable named `$num_failures` from the zone. This new variable is then used to determine whether NGINX Plus should serve the request or return a 403 Forbidden code.

After starting the NGINX Plus server with this configuration, you can `curl` the local machine and expect to receive a 200 OK response.

```
$ curl 'http://127.0.0.1/'
OK
```

Now add the local machine's IP address to the key-value store with a value of `1`:

```
$ curl -X POST -d '{"127.0.0.1":"1"}' \
  'http://127.0.0.1/api/3/http/keyvals/blacklist'
```

This `curl` command submits an HTTP POST request with a JSON object containing a key-value object to be submitted to the blacklist shared memory zone. The key-value store API URI is formatted as follows:

```
/api/{version}/http/keyvals/{httpKeyvalZoneName}
```

The local machine's IP address is now added to the key-value zone named `blacklist` with a value of 1. In the next request, NGINX

Plus looks up the `$remote_addr` in the key-value zone, finds the entry, and maps the value to the variable `$num_failures`. This variable is then evaluated in the `if` statement. When the variable has a value, the `if` evaluates to True and NGINX Plus returns the 403 Forbidden return code:

```
$ curl 'http://127.0.0.1/'
Forbidden
```

You can update or delete the key by making a `PATCH` method request:

```
$ curl -X PATCH -d '{"127.0.0.1":null}' \
  'http://127.0.0.1/api/3/http/keyvals/blacklist'
```

NGINX Plus deletes the key if the value is `null`, and requests will again return 200 OK.

Discussion

The key-value store, an NGINX Plus exclusive feature, enables applications to inject information into NGINX Plus. In the example provided, the `$remote_addr` variable is used to create a dynamic blacklist. You can populate the key-value store with any key that NGINX Plus might have as a variable—a session cookie, for example—and provide NGINX Plus an external value. In NGINX Plus R16, the key-value store became cluster-aware, meaning that you have to provide your key-value update to only one NGINX Plus server, and all of them will receive the information.

Also See

Dynamic Bandwidth Limits

5.3 Installing with Puppet

Problem

You need to install and configure NGINX with Puppet to manage NGINX configurations as code and conform with the rest of your Puppet configurations.

Solution

Create a module that installs NGINX, manages the files you need, and ensures that NGINX is running:

```
class nginx {
   package {"nginx": ensure => 'installed',}
   service {"nginx":
       ensure => 'true',
       hasrestart => 'true',
       restart => '/etc/init.d/nginx reload',
   }
   file { "nginx.conf":
       path    => '/etc/nginx/nginx.conf',
       require => Package['nginx'],
       notify  => Service['nginx'],
       content => template('nginx/templates/nginx.conf.erb'),
       user=>'root',
       group=>'root',
       mode='0644';
   }
}
```

This module uses the package management utility to ensure the NGINX package is installed. It also ensures NGINX is running and enabled at boot time. The configuration informs Puppet that the service has a restart command with the `hasrestart` directive, and we can override the `restart` command with an NGINX reload. The file resource will manage and template the *nginx.conf* file with the Embedded Ruby (ERB) templating language. The templating of the file will happen after the NGINX package is installed due to the `require` directive. However, the file resource will notify the NGINX service to reload because of the `notify` directive. The templated configuration file is not included. However, it can be simple to install a default NGINX configuration file, or very complex if using ERB or EPP templating language loops and variable substitution.

Discussion

Puppet is a configuration management tool based in the Ruby programming language. Modules are built in a domain-specific language and called via a manifest file that defines the configuration for a given server. Puppet can be run in a master-slave or masterless configuration. With Puppet, the manifest is run on the master and then sent to the slave. This is important because it ensures that the slave is only delivered the configuration meant for it and no extra configurations meant for other servers. There are a lot of extremely advanced public modules available for Puppet. Starting from these modules will help you get a jump-start on your configuration. A

public NGINX module from voxpupuli on GitHub will template out NGINX configurations for you.

Also See

Puppet Documentation
Puppet Package Documentation
Puppet Service Documentation
Puppet File Documentation
Puppet Templating Documentation
Voxpupuli NGINX Module

5.4 Installing with Chef

Problem

You need to install and configure NGINX with Chef to manage NGINX configurations as code and conform with the rest of your Chef configurations.

Solution

Create a cookbook with a recipe to install NGINX and configure configuration files through templating, and ensure NGINX reloads after the configuration is put in place. The following is an example recipe:

```
package 'nginx' do
  action :install
end

service 'nginx' do
  supports :status => true, :restart => true, :reload => true
  action   [ :start, :enable ]
end

template 'nginx.conf' do
  path   "/etc/nginx.conf"
  source "nginx.conf.erb"
  owner  'root'
  group  'root'
  mode   '0644'
  notifies :reload, 'service[nginx]', :delayed
end
```

The `package` block installs NGINX. The `service` block ensures that NGINX is started and enabled at boot, then declares to the rest of Chef what the `nginx` service will support as far as actions. The `template` block templates an ERB file and places it at */etc/nginx.conf* with an owner and group of root. The `template` block also sets the `mode` to `644` and notifies the `nginx` service to `reload`, but waits until the end of the Chef run declared by the `:delayed` statement. The templated configuration file is not included. However, it can be as simple as a default NGINX configuration file or very complex with ERB templating language loops and variable substitution.

Discussion

Chef is a configuration management tool based in Ruby. Chef can be run in a master-slave, or solo configuration, now known as Chef Zero. Chef has a very large community with many public cookbooks called the Supermarket. Public cookbooks from the Supermarket can be installed and maintained via a command-line utility called Berkshelf. Chef is extremely capable, and what we have demonstrated is just a small sample. The public NGINX cookbook in the Supermarket is extremely flexible and provides the options to easily install NGINX from a package manager or from source, and the ability to compile and install many different modules as well as template out the basic configurations.

Also See

Chef documentation
Chef Package
Chef Service
Chef Template
Chef Supermarket for NGINX

5.5 Installing with Ansible

Problem

You need to install and configure NGINX with Ansible to manage NGINX configurations as code and conform with the rest of your Ansible configurations.

Solution

Create an Ansible playbook to install NGINX and manage the *nginx.conf* file. The following is an example task file for the playbook to install NGINX. Ensure it's running and template the configuration file:

```
- name: NGINX | Installing NGINX
  package: name=nginx state=present

- name: NGINX | Starting NGINX
  service:
    name: nginx
    state: started
    enabled: yes

- name: Copy nginx configuration in place.
  template:
    src: nginx.conf.j2
    dest: "/etc/nginx/nginx.conf"
    owner: root
    group: root
    mode: 0644
  notify:
    - reload nginx
```

The `package` block installs NGINX. The `service` block ensures that NGINX is started and enabled at boot. The `template` block templates a *Jinja2* file and places the result at */etc/nginx.conf* with an owner and group of root. The `template` block also sets the `mode` to `644` and notifies the `nginx` service to `reload`. The templated configuration file is not included. However, it can be as simple as a default NGINX configuration file or very complex with Jinja2 templating language loops and variable substitution.

Discussion

Ansible is a widely used and powerful configuration management tool based in Python. The configuration of tasks is in YAML, and you use the Jinja2 templating language for file templating. Ansible offers a master named Ansible Tower on a subscription model. However, it's commonly used from local machines or to build servers directly to the client or in a masterless model. Ansible bulk SSHes into its servers and runs the configurations. Much like other configuration management tools, there's a large community of pub-

lic roles. Ansible calls this the Ansible Galaxy. You can find very sophisticated roles to utilize in your playbooks.

Also See

Ansible Documentation
Ansible Packages
Ansible Service
Ansible Template
Ansible Galaxy

5.6 Installing with SaltStack

Problem

You need to install and configure NGINX with SaltStack to manage NGINX configurations as code and conform with the rest of your SaltStack configurations.

Solution

Install NGINX through the package management module and manage the configuration files you desire. The following is an example state file (*sls*) that will install the `nginx` package and ensure the service is running, enabled at boot, and reload if a change is made to the configuration file:

```
nginx:
  pkg:
    - installed
  service:
    - name: nginx
    - running
    - enable: True
    - reload: True
    - watch:
      - file: /etc/nginx/nginx.conf

/etc/nginx/nginx.conf:
  file:
    - managed
    - source: salt://path/to/nginx.conf
    - user: root
    - group: root
    - template: jinja
    - mode: 644
```

```
    - require:
      - pkg: nginx
```

This is a basic example of installing NGINX via a package management utility and managing the *nginx.conf* file. The NGINX package is installed and the service is running and enabled at boot. With SaltStack you can declare a file managed by Salt as seen in the example and templated by many different templating languages. The templated configuration file is not included. However, it can be as simple as a default NGINX configuration file or very complex with the Jinja2 templating language loops and variable substitution. This configuration also specifies that NGINX must be installed prior to managing the file because of the `require` statement. After the file is in place, NGINX is reloaded because of the `watch` directive on the service and reloads as opposed to restarts because the `reload` directive is set to `True`.

Discussion

SaltStack is a powerful configuration management tool that defines server states in YAML. Modules for SaltStack can be written in Python. Salt exposes the Jinja2 templating language for states as well as for files. However, for files there are many other options, such as Mako, Python itself, and others. Salt works in a master-slave configuration as well as a masterless configuration. Slaves are called minions. The master-slave transport communication, however, differs from others and sets SaltStack apart. With Salt you're able to choose ZeroMQ, TCP, or Reliable Asynchronous Event Transport (RAET) for transmissions to the Salt agent; or you can not use an agent, and the master can SSH instead. Because the transport layer is by default asynchronous, SaltStack is built to be able to deliver its message to a large number of minions with low load to the master server.

Also See

SaltStack
Installed Packages
Managed Files
Templating with Jinja

5.7 Automating Configurations with Consul Templating

Problem

You need to automate your NGINX configuration to respond to changes in your environment through use of Consul.

Solution

Use the `consul-template` daemon and a template file to template out the NGINX configuration file of your choice:

```
upstream backend { {{range service "app.backend"}}
    server {{.Address}};{{end}}
}
```

This example is a Consul Template file that templates an upstream configuration block. This template will loop through nodes in Consul identified as `app.backend`. For every node in Consul, the template will produce a server directive with that node's IP address.

The `consul-template` daemon is run via the command line and can be used to reload NGINX every time the configuration file is templated with a change:

```
# consul-template -consul consul.example.internal -template \
 template:/etc/nginx/conf.d/upstream.conf:"nginx -s reload"
```

This command instructs the `consul-template` daemon to connect to a Consul cluster at `consul.example.internal` and to use a file named *template* in the current working directory to template the file and output the generated contents to */etc/nginx/conf.d/upstream.conf*, then to reload NGINX every time the templated file changes. The `-template` flag takes a string of the template file, the output location, and the command to run after the templating process takes place; these three variables are separated by a colon. If the command being run has spaces, make sure to wrap it in double quotes. The `-consul` flag tells the daemon what Consul cluster to connect to.

Discussion

Consul is a powerful service discovery tool and configuration store. Consul stores information about nodes as well as key-value pairs in

a directory-like structure and allows for restful API interaction. Consul also provides a DNS interface on each client, allowing for domain name lookups of nodes connected to the cluster. A separate project that utilizes Consul clusters is the `consul-template` daemon; this tool templates files in response to changes in Consul nodes, services, or key-value pairs. This makes Consul a very powerful choice for automating NGINX. With `consul-template` you can also instruct the daemon to run a command after a change to the template takes place. With this, we can reload the NGINX configuration and allow your NGINX configuration to come alive along with your environment. With Consul you're able to set up health checks on each client to check the health of the intended service. With this failure detection, you're able to template your NGINX configuration accordingly to only send traffic to healthy hosts.

Also See

Consul Home Page
Introduction to Consul Template
Consul Template GitHub

CHAPTER 6

Authentication

6.0 Introduction

NGINX is able to authenticate clients. Authenticating client requests with NGINX offloads work and provides the ability to stop unauthenticated requests from reaching your application servers. Modules available for NGINX Open Source include basic authentication and authentication subrequests. The NGINX Plus exclusive module for verfying JSON Web Tokens (JWTs) enables integration with third-party authentication providers that use the authentication standard OpenID Connect.

6.1 HTTP Basic Authentication

Problem

You need to secure your application or content via HTTP basic authentication.

Solution

Generate a file in the following format, where the password is encrypted or hashed with one of the allowed formats:

```
# comment
name1:password1
name2:password2:comment
name3:password3
```

The username is the first field, the password the second field, and the delimiter is a colon. There is an optional third field, which you can use to comment on each user. NGINX can understand a few different formats for passwords, one of which is whether the password is encrypted with the C function `crypt()`. This function is exposed to the command line by the `openssl passwd` command. With `openssl` installed, you can create encrypted password strings by using the following command:

```
$ openssl passwd MyPassword1234
```

The output will be a string that NGINX can use in your password file.

Use the `auth_basic` and `auth_basic_user_file` directives within your NGINX configuration to enable basic authentication:

```
location / {
    auth_basic          "Private site";
    auth_basic_user_file conf.d/passwd;
}
```

You can use the `auth_basic` directives in the HTTP, server, or location contexts. The `auth_basic` directive takes a string parameter, which is displayed on the basic authentication pop-up window when an unauthenticated user arrives. The `auth_basic_user_file` specifies a path to the user file.

Discussion

You can generate basic authentication passwords a few ways and in a few different formats with varying degrees of security. The `htpasswd` command from Apache can also generate passwords. Both the `openssl` and `htpasswd` commands can generate passwords with the `apr1` algorithm, which NGINX can also understand. The password can also be in the salted SHA-1 format that Lightweight Directory Access Protocol (LDAP) and Dovecot use. NGINX supports more formats and hashing algorithms; however, many of them are considered insecure because they can easily be defeated by brute-force attacks.

You can use basic authentication to protect the context of the entire NGINX host, specific virtual servers, or even just specific location blocks. Basic authentication won't replace user authentication for web applications, but it can help keep private information secure.

Under the hood, basic authentication is done by the server returning a 401 unauthorized HTTP code with the response header `WWW-Authenticate`. This header will have a value of `Basic realm="`*`your string`*`"`. This response causes the browser to prompt for a username and password. The username and password are concatenated and delimited with a colon, then base64-encoded, and then sent in a request header named `Authorization`. The `Authorization` request header will specify a `Basic` and `user:password` encoded string. The server decodes the header and verifies against the provided `auth_basic_user_file`. Because the username password string is merely base64-encoded, it's recommended to use HTTPS with basic authentication.

6.2 Authentication Subrequests

Problem

You have a third-party authentication system for which you would like requests authenticated.

Solution

Use the `http_auth_request_module` to make a request to the authentication service to verify identity before serving the request:

```
location /private/ {
    auth_request     /auth;
    auth_request_set $auth_status $upstream_status;
}

location = /auth {
    internal;
    proxy_pass              http://auth-server;
    proxy_pass_request_body off;
    proxy_set_header        Content-Length "";
    proxy_set_header        X-Original-URI $request_uri;
}
```

The `auth_request` directive takes a URI parameter that must be a local internal location. The `auth_request_set` directive allows you to set variables from the authentication subrequest.

Discussion

The `http_auth_request_module` enables authentication on every request handled by the NGINX server. The module makes a subrequest before serving the original to determine if the request has access to the resource it's requesting. The entire original request is proxied to this subrequest location. The authentication location acts as a typical proxy to the subrequest and sends the original request, including the original request body and headers. The HTTP status code of the subrequest is what determines whether or not access is granted. If the subrequest returns with an HTTP 200 status code, the authentication is successful and the request is fulfilled. If the subrequest returns HTTP 401 or 403, the same will be returned for the original request.

If your authentication service does not request the request body, you can drop the request body with the `proxy_pass_request_body` directive, as demonstrated. This practice will reduce the request size and time. Because the response body is discarded, the `Content-Length` header must be set to an empty string. If your authentication service needs to know the URI being accessed by the request, you'll want to put that value in a custom header that your authentication service checks and verifies. If there are things you do want to keep from the subrequest to the authentication service, like response headers or other information, you can use the `auth_request_set` directive to make new variables out of response data.

6.3 Validating JWTs

Problem

You need to validate a JWT before the request is handled with NGINX Plus.

Solution

Use NGINX Plus's HTTP JWT authentication module to validate the token signature and embed JWT claims and headers as NGINX variables:

```
location /api/ {
    auth_jwt          "api";
```

```
        auth_jwt_key_file conf/keys.json;
    }
```

This configuration enables validation of JWTs for this location. The `auth_jwt` directive is passed a string, which is used as the authentication realm. The `auth_jwt` takes an optional token parameter of a variable that holds the JWT. By default, the `Authentication` header is used per the JWT standard. The `auth_jwt` directive can also be used to cancel the effects of required JWT authentication from inherited configurations. To turn `off` authentication, pass the keyword to the `auth_jwt` directive with nothing else. To cancel inherited authentication requirements, pass the `off` keyword to the `auth_jwt` directive with nothing else. The `auth_jwt_key_file` takes a single parameter. This parameter is the path to the key file in standard JSON Web Key format.

Discussion

NGINX Plus is able to validate the JSON web signature types of tokens as opposed to the JSON web encryption type, where the entire token is encrypted. NGINX Plus is able to validate signatures that are signed with the HS256, RS256, and ES256 algorithms. Having NGINX Plus validate the token can save the time and resources needed to make a subrequest to an authentication service. NGINX Plus deciphers the JWT header and payload, and captures the standard headers and claims into embedded variables for your use.

Also See

RFC Standard Documentation of JSON Web Signature
RFC Standard Documentation of JSON Web Algorithms
RFC Standard Documentation of JSON Web Token
NGINX Embedded Variables
Detailed NGINX Blog

6.4 Creating JSON Web Keys

Problem

You need a JSON Web Key for NGINX Plus to use.

Solution

NGINX Plus utilizes the JSON Web Key (JWK) format as specified in the RFC standard. This standard allows for an array of key objects within the JWK file.

The following is an example of what the key file may look like:

```
{"keys":
  [
    {
        "kty":"oct",
        "kid":"0001",
        "k":"OctetSequenceKeyValue"
    },
    {
        "kty":"EC",
        "kid":"0002"
        "crv":"P-256",
        "x": "XCoordinateValue",
        "y": "YCoordinateValue",
        "d": "PrivateExponent",
        "use": "sig"
    },
    {
        "kty":"RSA",
        "kid":"0003"
        "n": "Modulus",
        "e": "Exponent",
        "d": "PrivateExponent"
    }
  ]
}
```

The JWK file shown demonstrates the three initial types of keys noted in the RFC standard. The format of these keys is also part of the RFC standard. The `kty` attribute is the key type. This file shows three key types: the Octet Sequence (`oct`), the EllipticCurve (`EC`), and the `RSA` type. The `kid` attribute is the key ID. Other attributes to these keys are specified in the standard for that type of key. Look to the RFC documentation of these standards for more information.

Discussion

There are numerous libraries available in many different languages to generate the JSON Web Key. It's recommended to create a key service that is the central JWK authority to create and rotate your JWKs at a regular interval. For enhanced security, it's recommended

to make your JWKs as secure as your SSL/TLS certifications. Secure your key file with proper user and group permissions. Keeping them in memory on your host is best practice. You can do so by creating an in-memory filesystem like ramfs. Rotating keys on a regular interval is also important; you may opt to create a key service that creates public and private keys and offers them to the application and NGINX via an API.

Also See

RFC standardization documentation of JSON Web Key

6.5 Authenticate Users via Existing OpenID Connect SSO

Problem

You want to offload OpenID Connect authentication validation to NGINX Plus.

Solution

Use the JWT module that comes with NGINX Plus to secure a location or server, and instruct the `auth_jwt` directive to use `$cookie_auth_token` as the token to be validated:

```
location /private/ {
    auth_jwt "Google Oauth" token=$cookie_auth_token;
    auth_jwt_key_file /etc/nginx/google_certs.jwk;
}
```

This configuration directs NGINX Plus to secure the */private/* URI path with JWT validation. Google OAuth 2.0 OpenID Connect uses the cookie `auth_token` rather than the default bearer token. Thus, you must instruct NGINX to look for the token in this cookie rather than in the NGINX Plus default location. The `auth_jwt_key_file` location is set to an arbitrary path, which is a step that we cover in Recipe 6.6.

Discussion

This configuration demonstrates how you can validate a Google OAuth 2.0 OpenID Connect JSON Web Token with NGINX Plus. The NGINX Plus JWT authentication module for HTTP is able to

validate any JSON Web Token that adheres to the RFC for JSON Web Signature specification, instantly enabling any SSO authority that utilizes JSON Web Tokens to be validated at the NGINX Plus layer. The OpenID 1.0 protocol is a layer on top of the OAuth 2.0 authentication protocol that adds identity, enabling the use of JWTs to prove the identity of the user sending the request. With the signature of the token, NGINX Plus can validate that the token has not been modified since it was signed. In this way, Google is using an asynchronous signing method and makes it possible to distribute public JWKs while keeping its private JWK secret.

NGINX Plus can also control the Authorization Code Flow for OpenID Connect 1.0, enabling NGINX Plus as a Relay Party for OpenID Connect. This capability enables integration with most major identity providers, including CA Single Sign On (formerly SiteMinder), ForgeRock OpenAM, Keycloak, Okta, OneLogin, and Ping Identity. For more information and a reference implementation of NGINX Plus as a relaying party for OpenID Connect authentication, check out the NGINX Inc OpenID Connect GitHub Repository.

Also See

Detailed NGINX Blog on OpenID Connect
OpenID Connect

6.6 Obtaining the JSON Web Key from Google

Problem

You need to obtain the JSON Web Key from Google to use when validating OpenID Connect tokens with NGINX Plus.

Solution

Utilize Cron to request a fresh set of keys every hour to ensure your keys are always up-to-date:

```
0 * * * * root wget https://www.googleapis.com/oauth2/v3/ \
  certs-O /etc/nginx/google_certs.jwk
```

This code snippet is a line from a crontab file. Unix-like systems have many options for where crontab files can live. Every user will

have a user-specific crontab, and there's also a number of files and directories in the */etc/* directory.

Discussion

Cron is a common way to run a scheduled task on a Unix-like system. JSON Web Keys should be rotated on a regular basis to ensure the security of the key, and in turn, the security of your system. To ensure that you always have the most up-to-date key from Google, you'll want to check for new JWKs at regular intervals. This Cron solution is one way of doing so.

Also See

Cron

CHAPTER 7

Security Controls

7.0 Introduction

Security is done in layers, and there must be multiple layers to your security model for it to be truly hardened. In this chapter, we go through many different ways to secure your web applications with NGINX and NGINX Plus. You can use many of these security methods in conjunction with one another to help harden security. The following are a number of security sections that explore features of NGINX and NGINX Plus that can assist in strengthening your application. You might notice that this chapter does not touch upon one of the largest security features of NGINX, the ModSecurity 3.0 NGINX module, which turns NGINX into a Web Application Firewall (WAF). To learn more about the WAF capabilities, download the ModSecurity 3.0 and NGINX: Quick Start Guide.

7.1 Access Based on IP Address

Problem

You need to control access based on the IP address of the client.

Solution

Use the HTTP access module to control access to protected resources:

```
location /admin/ {
    deny  10.0.0.1;
```

```
        allow 10.0.0.0/20;
        allow 2001:0db8::/32;
        deny  all;
    }
```

The given location block allows access from any IPv4 address in 10.0.0.0/20 except 10.0.0.1, allows access from IPv6 addresses in the `2001:0db8::/32` subnet, and returns a 403 for requests originating from any other address. The `allow` and `deny` directives are valid within the HTTP, server, and location contexts. Rules are checked in sequence until a match is found for the remote address.

Discussion

Protecting valuable resources and services on the internet must be done in layers. NGINX provides the ability to be one of those layers. The `deny` directive blocks access to a given context, while the `allow` directive can be used to allow subsets of the blocked access. You can use IP addresses, IPv4 or IPv6, CIDR block ranges, the keyword `all`, and a Unix socket. Typically when protecting a resource, one might allow a block of internal IP addresses and deny access from all.

7.2 Allowing Cross-Origin Resource Sharing

Problem

You're serving resources from another domain and need to allow cross-origin resource sharing (CORS) to enable browsers to utilize these resources.

Solution

Alter headers based on the `request` method to enable CORS:

```
map $request_method $cors_method {
  OPTIONS 11;
  GET  1;
  POST 1;
  default 0;
}
server {
  ...
  location / {
    if ($cors_method ~ '1') {
        add_header 'Access-Control-Allow-Methods'
           'GET,POST,OPTIONS';
```

```
            add_header 'Access-Control-Allow-Origin'
               '*.example.com';
            add_header 'Access-Control-Allow-Headers'
                       'DNT,
                        Keep-Alive,
                        User-Agent,
                        X-Requested-With,
                        If-Modified-Since,
                        Cache-Control,
                        Content-Type';
        }
        if ($cors_method = '11') {
            add_header 'Access-Control-Max-Age' 1728000;
            add_header 'Content-Type' 'text/plain; charset=UTF-8';
            add_header 'Content-Length' 0;
            return 204;
        }
      }
    }
```

There's a lot going on in this example, which has been condensed by using a `map` to group the `GET` and `POST` methods together. The `OPTIONS` request method returns a *preflight* request to the client about this server's CORS rules. `OPTIONS`, `GET`, and `POST` methods are allowed under CORS. Setting the `Access-Control-Allow-Origin` header allows for content being served from this server to also be used on pages of origins that match this header. The preflight request can be cached on the client for 1,728,000 seconds, or 20 days.

Discussion

Resources such as JavaScript make CORS when the resource they're requesting is of a domain other than its own. When a request is considered cross origin, the browser is required to obey CORS rules. The browser will not use the resource if it does not have headers that specifically allow its use. To allow our resources to be used by other subdomains, we have to set the CORS headers, which can be done with the `add_header` directive. If the request is a `GET`, `HEAD`, or `POST` with standard content type, and the request does not have special headers, the browser will make the request and only check for origin. Other request methods will cause the browser to make the preflight request to check the terms of the server to which it will obey for that resource. If you do not set these headers appropriately, the browser will give an error when trying to utilize that resource.

7.3 Client-Side Encryption

Problem

You need to encrypt traffic between your NGINX server and the client.

Solution

Utilize one of the SSL modules, such as the `ngx_http_ssl_module` or `ngx_stream_ssl_module` to encrypt traffic:

```
http { # All directives used below are also valid in stream
    server {
        listen 8433 ssl;
        ssl_protocols TLSv1.2 TLSv1.3;
        ssl_ciphers HIGH:!aNULL:!MD5;
        ssl_certificate /etc/nginx/ssl/example.pem;
        ssl_certificate_key /etc/nginx/ssl/example.key;
        ssl_certificate /etc/nginx/ssl/example.ecdsa.crt;
        ssl_certificate_key /etc/nginx/ssl/example.ecdsa.key;
        ssl_session_cache shared:SSL:10m;
        ssl_session_timeout 10m;
    }
}
```

This configuration sets up a server to listen on a port encrypted with SSL, 8443. The server accepts the SSL protocol versions TLSv1.2 and TLSv1.3. Two sets of certificate and key pair locations are disclosed to the server for use. The server is instructed to use the highest strength offered by the client while restricting a few that are insecure. The Elliptic Curve Cryptopgraphy (ECC) ciphers are prioritized as we've provided an ECC certificate key pair. The SSL session cache and timeout allow workers to cache and store session parameters for a given amount of time. There are many other session cache options that can help with performance or security of all types of use cases. You can use session cache options in conjunction with one another. However, specifying one without the default will turn off that default, built-in session cache.

Discussion

Secure transport layers are the most common way of encrypting information in transit. As of this writing, the TLS protocol is preferred over the SSL protocol. That's because versions 1 through 3 of

SSL are now considered insecure. Although the protocol name might be different, TLS still establishes a secure socket layer. NGINX enables your service to protect information between you and your clients, which in turn protects the client and your business. When using a signed certificate, you need to concatenate the certificate with the certificate authority chain. When you concatenate your certificate and the chain, your certificate should be above the chain in the file. If your certificate authority has provided many files in the chain, it can also provide the order in which they are layered. The SSL session cache enhances performance by not having to negotiate for SSL/TLS versions and ciphers.

In testing, ECC certificates were found to be faster than the equivalent-strength RSA certificates. The key size is smaller, which results in the ability to serve more SSL/TLS connections, and with faster handshakes. NGINX allows you to configure multiple certificates and keys, and then serve the optimal certificate for the client browser. This allows you to take advantage of the newer technology but still serve older clients.

Also See

Mozilla Server Side TLS Page
Mozilla SSL Configuration Generator
Test Your SSL Configuration with SSL Labs SSL Test

7.4 Upstream Encryption

Problem

You need to encrypt traffic between NGINX and the upstream service and set specific negotiation rules for compliance regulations or if the upstream is outside of your secured network.

Solution

Use the SSL directives of the HTTP proxy module to specify SSL rules:

```
location / {
    proxy_pass https://upstream.example.com;
    proxy_ssl_verify on;
    proxy_ssl_verify_depth 2;
```

```
        proxy_ssl_protocols TLSv1.2;
    }
```

These proxy directives set specific SSL rules for NGINX to obey. The configured directives ensure that NGINX verifies that the certificate and chain on the upstream service is valid up to two certificates deep. The `proxy_ssl_protocols` directive specifies that NGINX will only use TLS version 1.2. By default, NGINX does not verify upstream certificates and accepts all TLS versions.

Discussion

The configuration directives for the HTTP proxy module are vast, and if you need to encrypt upstream traffic, you should at least turn on verification. You can proxy over HTTPS simply by changing the protocol on the value passed to the `proxy_pass` directive. However, this does not validate the upstream certificate. Other directives, such as `proxy_ssl_certificate` and `proxy_ssl_certificate_key`, allow you to lock down upstream encryption for enhanced security. You can also specify `proxy_ssl_crl` or a certificate revocation list, which lists certificates that are no longer considered valid. These SSL proxy directives help harden your system's communication channels within your own network or across the public internet.

7.5 Securing a Location

Problem

You need to secure a location block using a secret.

Solution

Use the secure link module and the `secure_link_secret` directive to restrict access to resources to users who have a secure link:

```
location /resources {
    secure_link_secret mySecret;
    if ($secure_link = "") { return 403; }

    rewrite ^ /secured/$secure_link;
}

location /secured/ {
    internal;
```

```
        root /var/www;
    }
```

This configuration creates an internal and public-facing location block. The public-facing location block `/resources` will return a 403 Forbidden unless the request URI includes an `md5` hash string that can be verified with the secret provided to the `secure_link_secret` directive. The `$secure_link` variable is an empty string unless the hash in the URI is verified.

Discussion

Securing resources with a secret is a great way to ensure your files are protected. The secret is used in conjunction with the URI. This string is then `md5` hashed, and the hex digest of that `md5` hash is used in the URI. The hash is placed into the link and evaluated by NGINX. NGINX knows the path to the file being requested as it's in the URI after the hash. NGINX also knows your secret as it's provided via the `secure_link_secret` directive. NGINX is able to quickly validate the `md5` hash and store the URI in the `$secure_link` variable. If the hash cannot be validated, the variable is set to an empty string. It's important to note that the argument passed to the `secure_link_secret` must be a static string; it cannot be a variable.

7.6 Generating a Secure Link with a Secret

Problem

You need to generate a secure link from your application using a secret.

Solution

The secure link module in NGINX accepts the hex digest of an `md5` hashed string, where the string is a concatenation of the URI path and the secret. Building on the last section, Recipe 7.5, we will create the secured link that will work with the previous configuration example given that there's a file present at */var/www/secured/index.html*. To generate the hex digest of the `md5` hash, we can use the Unix `openssl` command:

```
$ echo -n 'index.htmlmySecret' | openssl md5 -hex
(stdin)= a53bee08a4bf0bbea978ddf736363a12
```

Here we show the URI that we're protecting, *index.html*, concatenated with our secret, `mySecret`. This string is passed to the `openssl` command to output an `md5` hex digest.

The following is an example of the same hash digest being constructed in Python using the `hashlib` library that is included in the Python Standard Library:

```
import hashlib
hashlib.md5.(b'index.htmlmySecret').hexdigest()
'a53bee08a4bf0bbea978ddf736363a12'
```

Now that we have this hash digest, we can use it in a URL. Our example will be www.example.com making a request for the file */var/www/secured/index.html* through our */resources* location. Our full URL will be the following:

```
www.example.com/resources/a53bee08a4bf0bbea978ddf736363a12/\
index.html
```

Discussion

Generating the digest can be done in many ways, in many languages. Things to remember: the URI path goes before the secret, there are no carriage returns in the string, and use the hex digest of the `md5` hash.

7.7 Securing a Location with an Expire Date

Problem

You need to secure a location with a link that expires at some future time and is specific to a client.

Solution

Utilize the other directives included in the secure link module to set an expire time and use variables in your secure link:

```
location /resources {
    root /var/www;
    secure_link $arg_md5,$arg_expires;
    secure_link_md5 "$secure_link_expires$uri$remote_addr
   mySecret";
    if ($secure_link = "") { return 403; }
    if ($secure_link = "0") { return 410; }
}
```

The `secure_link` directive takes two parameters separated with a comma. The first parameter is the variable that holds the `md5` hash. This example uses an HTTP argument of `md5`. The second parameter is a variable that holds the time in which the link expires in Unix epoch time format. The `secure_link_md5` directive takes a single parameter that declares the format of the string that is used to construct the `md5` hash. Like the other configuration, if the hash does not validate, the `$secure_link` variable is set to an empty string. However, with this usage, if the hash matches but the time has expired, the `$secure_link` variable will be set to `0`.

Discussion

This usage of securing a link is more flexible and looks cleaner than the `secure_link_secret` shown in Recipe 7.5. With these directives, you can use any number of variables that are available to NGINX in the hashed string. Using user-specific variables in the hash string will strengthen your security as users won't be able to trade links to secured resources. It's recommended to use a variable like `$remote_addr` or `$http_x_forwarded_for`, or a session cookie header generated by the application. The arguments to `secure_link` can come from any variable you prefer, and they can be named whatever best fits. The conditions around what the `$secure_link` variable is set to returns known HTTP codes for Forbidden and Gone. The HTTP 410, Gone, works great for expired links as the condition is to be considered permanent.

7.8 Generating an Expiring Link

Problem

You need to generate a link that expires.

Solution

Generate a timestamp for the expire time in the Unix epoch format. On a Unix system, you can test by using the date as demonstrated in the following:

```
$ date -d "2020-12-31 00:00" +%s --utc
1609372800
```

Next, you'll need to concatenate your hash string to match the string configured with the `secure_link_md5` directive. In this case, our string to be used will be `1293771600/resources/index.html127.0.0.1 mySecret`. The `md5` hash is a bit different than just a hex digest. It's an `md5` hash in binary format, base64-encoded, with plus signs (+) translated to hyphens (-), slashes (/) translated to underscores (_), and equal (=) signs removed. The following is an example on a Unix system:

```
$ echo -n '1609372800/resources/index.html127.0.0.1 mySecret' \
  | openssl md5 -binary \
  | openssl base64 \
  | tr +/ -_ \
  | tr -d =
TG6ck3OpAttQ1d7jW3JOcw
```

Now that we have our hash, we can use it as an argument along with the expire date:

```
/resources/index.html?md5=TG6ck3OpAttQ1d7jW3JOcw&expires=1609372
  800'
```

The following is a more practical example in Python utilizing a relative time for the expiration, setting the link to expire one hour from generation. At the time of writing this example works with Python 2.7 and 3.x utilizing the Python Standard Library:

```
from datetime import datetime, timedelta
from base64 import b64encode
import hashlib

# Set environment vars
resource = b'/resources/index.html'
remote_addr = b'127.0.0.1'
host = b'www.example.com'
mysecret = b'mySecret'

# Generate expire timestamp
now = datetime.utcnow()
expire_dt = now + timedelta(hours=1)
expire_epoch = str.encode(expire_dt.strftime('%s'))

# md5 hash the string
uncoded = expire_epoch + resource + remote_addr + mysecret
md5hashed = hashlib.md5(uncoded).digest()

# Base64 encode and transform the string
b64 = b64encode(md5hashed)
unpadded_b64url = b64.replace(b'+', b'-')\
    .replace(b'/', b'_')\
```

```
        .replace(b'=', b'')

    # Format and generate the link
    linkformat = "{}{}?md5={}?expires={}"
    securelink = linkformat.format(
        host.decode(),
        resource.decode(),
        unpadded_b64url.decode(),
        expire_epoch.decode()
    )
    print(securelink)
```

Discussion

With this pattern we're able to generate a secure link in a special format that can be used in URLs. The secret provides security through use of a variable that is never sent to the client. You're able to use as many other variables as you need to in order to secure the location. `md5` hashing and base64 encoding are common, lightweight, and available in nearly every language.

7.9 HTTPS Redirects

Problem

You need to redirect unencrypted requests to HTTPS.

Solution

Use a rewrite to send all HTTP traffic to HTTPS:

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

This configuration listens on port 80 as the default server for both IPv4 and IPv6 and for any hostname. The `return` statement returns a 301 permanent redirect to the HTTPS server at the same host and request URI.

Discussion

It's important to always redirect to HTTPS where appropriate. You may find that you do not need to redirect all requests but only those

with sensitive information being passed between client and server. In that case, you may want to put the `return` statement in particular locations only, such as */login*.

7.10 Redirecting to HTTPS where SSL/TLS Is Terminated Before NGINX

Problem

You need to redirect to HTTPS, however, you've terminated SSL/TLS at a layer before NGINX.

Solution

Use the standard `X-Forwarded-Proto` header to determine if you need to redirect:

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    server_name _;
    if ($http_x_forwarded_proto = 'http') {
        return 301 https://$host$request_uri;
    }
}
```

This configuration is very much like HTTPS redirects. However, in this configuration we're only redirecting if the header `X-Forwarded-Proto` is equal to HTTP.

Discussion

It's a common use case that you may terminate SSL/TLS in a layer in front of NGINX. One reason you may do something like this is to save on compute costs. However, you need to make sure that every request is HTTPS, but the layer terminating SSL/TLS does not have the ability to redirect. It can, however, set proxy headers. This configuration works with layers such as the Amazon Web Services Elastic Load Balancer, which will offload SSL/TLS at no additional cost. This is a handy trick to make sure that your HTTP traffic is secured.

7.11 HTTP Strict Transport Security

Problem

You need to instruct browsers to never send requests over HTTP.

Solution

Use the HTTP Strict Transport Security (HSTS) enhancement by setting the `Strict-Transport-Security` header:

```
add_header Strict-Transport-Security max-age=31536000;
```

This configuration sets the `Strict-Transport-Security` header to a max age of a year. This will instruct the browser to always do an internal redirect when HTTP requests are attempted to this domain, so that all requests will be made over HTTPS.

Discussion

For some applications a single HTTP request trapped by a man in the middle attack could be the end of the company. If a form post containing sensitive information is sent over HTTP, the HTTPS redirect from NGINX won't save you; the damage is done. This opt-in security enhancement informs the browser to never make an HTTP request, and therefore the request is never sent unencrypted.

Also See

RFC-6797 HTTP Strict Transport Security
OWASP HSTS Cheat Sheet

7.12 Satisfying Any Number of Security Methods

Problem

You need to provide multiple ways to pass security to a closed site.

Solution

Use the `satisfy` directive to instruct NGINX that you want to satisfy any or all of the security methods used:

```
location / {
    satisfy any;

    allow 192.168.1.0/24;
    deny  all;

    auth_basic           "closed site";
    auth_basic_user_file conf/htpasswd;
}
```

This configuration tells NGINX that the user requesting the `location /` needs to satisfy one of the security methods: either the request needs to originate from the *192.168.1.0/24* CIDR block or be able to supply a username and password that can be found in the *conf/htpasswd* file. The `satisfy` directive takes one of two options: `any` or `all`.

Discussion

The `satisfy` directive is a great way to offer multiple ways to authenticate to your web application. By specifying `any` to the `satisfy` directive, the user must meet one of the security challenges. By specifying `all` to the `satisfy` directive, the user must meet all of the security challenges. This directive can be used in conjunction with the `http_access_module` detailed in Recipe 7.1, the `http_auth_basic_module` detailed in Recipe 6.1, the `http_auth_request_module` detailed in Recipe 6.2, and the `http_auth_jwt_module` detailed in Recipe 6.3. Security is only truly secure if it's done in multiple layers. The `satisfy` directive will help you achieve this for locations and servers that require deep security rules.

7.13 Dynamic DDoS Mitigation

Problem

You need a dynamic Distributed Denial of Service (DDoS) mitigation solution.

Solution

Use NGINX Plus to build a cluster-aware rate limit and automatic blacklist:

```
limit_req_zone   $remote_addr zone=per_ip:1M rate=100r/s sync;
                 # Cluster-aware rate limit
limit_req_status 429;

keyval_zone zone=sinbin:1M timeout=600 sync;
              # Cluster-aware "sin bin" with
              # 10-minute TTL
keyval $remote_addr $in_sinbin zone=sinbin;
              # Populate $in_sinbin with
              # matched client IP addresses

server {
    listen 80;
    location / {
        if ($in_sinbin) {
            set $limit_rate 50; # Restrict bandwidth of bad clients
        }

        limit_req zone=per_ip;
              # Apply the rate limit here
        error_page 429 = @send_to_sinbin;
              # Excessive clients are moved to
              # this location
        proxy_pass http://my_backend;
    }

    location @send_to_sinbin {
        rewrite ^ /api/3/http/keyvals/sinbin break;
              # Set the URI of the
              # "sin bin" key-val
        proxy_method POST;
        proxy_set_body '{"$remote_addr":"1"}';
        proxy_pass http://127.0.0.1:80;
    }

    location /api/ {
        api write=on;
        # directives to control access to the API
    }
}
```

Discussion

This solution uses a synchronized rate limit and a synchronized key-value store to dynamically respond to DDoS attacks and mitigate their effects. The `sync` parameter provided to the `limit_req_zone` and `keyval_zone` directives synchronizes the shared memory zone with other machines in the active-active NGINX Plus cluster. This example identifies clients that send more than 100 requests per sec-

ond, regardless of which NGINX Plus node receives the request. When a client exceeds the rate limit, its IP address is added to a "sin bin" key-value store by making a call to the NGINX Plus API. The sin bin is synchronized across the cluster. Further requests from clients in the sin bin are subject to a very low bandwidth limit, regardless of which NGINX Plus node receives them. Limiting bandwidth is preferable to rejecting requests outright because it does not clearly signal to the client that DDoS mitigation is in effect. After 10 minutes the client is automatically removed from the sin bin.