O'REILLY®

Compliments of NGINX

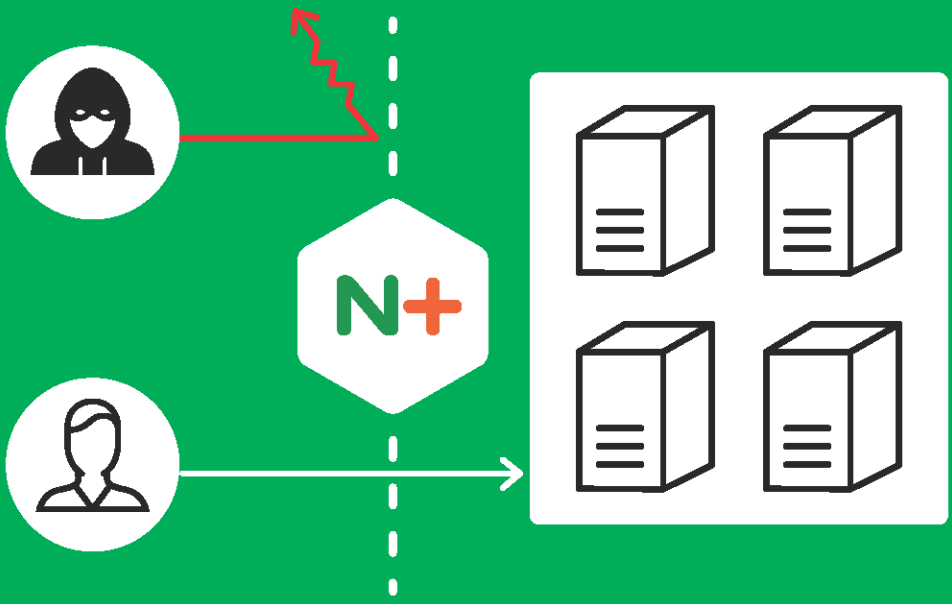
NGINX Cookbook

Advanced Recipes for Security

Derek DeJonghe

NGINX Plus with ModSecurity WAF

Protect your applications



- Layer 7 attack protection
- DDoS mitigation
- Real-time blacklists
- Data leakage protection
- Audit logging
- PCI-DSS 6.6 compliance

"Availability and scalability are incredibly important, but security is most important for us. With the ability to meet our security requirements and stay ahead of the curve, NGINX Plus is our vehicle for moving forward."

—Sean McElroy, Corporate Information Security Officer at Alkami Technology

Learn more at nginx.com/waf

NGINX+

NGINX Cookbook

Advanced Recipes for Security

Derek DeJonghe

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

NGINX Cookbook
by Derek DeJonghe

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Virginia Wilson
Acquisitions Editor: Brian Anderson
Production Editor: Shiny Kalapurakkel
Copyeditor: Amanda Kersey
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

Revision History for the First Edition

2016-09-19: Part 1
2017-01-23: Part 2

978-1-491-96893-2

[LSI]

Table of Contents

Foreword

Almost every day, you read headlines about another company being hit with a distributed denial-of-service (DDoS) attack, or yet another data breach or site hack. The unfortunate truth is that everyone is a target.

One common thread amongst recent attacks is that the attackers are using the same bag of tricks they have been exploiting for years: SQL injection, password guessing, phishing, malware attached to emails, and so on. As such, there are some common sense measures you can take to protect yourself. By now, these best practices should be old hat and ingrained into everything we do, but the path is not always clear, and the tools we have available to us as application owners and administrators don't always make adhering to these best practices easy.

To address this, the NGINX Cookbook Part 2 shows how to protect your apps using the open source NGINX software and our enterprise-grade product: NGINX Plus. This set of easy-to-follow recipes shows you how to mitigate DDoS attacks with request/connection limits, restrict access using JWT tokens, and protect application logic using the ModSecurity web application firewall (WAF).

We hope you enjoy this second part of the NGINX Cookbook, and that it helps you keep your apps and data safe from attack.

— Faisal Memon
Product Marketer, NGINX, Inc.

Introduction

This is the second of three installments of *NGINX Cookbook*. This book is about NGINX the web server, reverse proxy, load balancer, and HTTP cache. This installment will focus on security aspects and features of NGINX and NGINX Plus, the licensed version of the NGINX server. Throughout this installment you will learn the basics of controlling access and limiting abuse and misuse of your web assets and applications. Security concepts such as encryption of your web traffic and basic HTTP authentication will be explained as applicable to the NGINX server. More advanced topics are covered as well, such as setting up NGINX to verify authentication via third-party systems as well as through JSON Web Token Signature validation and integrating with single sign-on providers. This installment covers some amazing features of NGINX and NGINX Plus, such as securing links for time-limited access and security, as well as enabling web application firewall capabilities of NGINX Plus with the ModSecurity module. Some of the plug-and-play modules in this installment are only available through the paid NGINX Plus subscription. However, this does not mean that the core open source NGINX server is not capable of these securities.

CHAPTER 1

Controlling Access

1.0 Introduction

Controlling access to your web applications or subsets of your web applications is important business. Access control takes many forms in NGINX, such as denying it at the network level, allowing it based on authentication mechanisms, or HTTP instructing browsers how to act. In this chapter we will discuss access control based on network attributes, authentication, and how to specify *Cross-Origin Resource Sharing* (CORS) rules.

1.1 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 / {
    deny  10.0.0.1;
    allow 10.0.0.0/20;
    allow 2001:0db8::/32;
    deny  all;
}
```

Within the HTTP, server, and location contexts, `allow` and `deny` directives provide the ability to allow or block access from a given client, IP, CIDR range, Unix socket, or `all` keyword. 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 limit the 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.

1.2 Allowing Cross-Origin Resource Sharing

Problem

You're serving resources from another domain and need to allow CORS to enable browsers to utilize these resources.

Solution

Alter headers based on 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 information called a *preflight* request to the client about this server's CORS rules. As well as `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 cross-origin resource requests when the resource they're requesting is of a domain other than its own origin. When a request is considered cross origin, the browser is required to obey cross-origin resource sharing 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.

CHAPTER 2

Limiting Use

2.0 Introduction

Limiting use or abuse of your system can be important for throttling heavy users or stopping attacks. NGINX has multiple modules built in to help control the use of your applications. This chapter focuses on limiting use and abuse, the number of connections, the rate at which requests are served, and the amount of bandwidth used. It's important to differentiate between connections and requests: connections (TCP connection) are the networking layer on which requests are made and therefore are not the same thing. A browser may open multiple connections to a server to make multiple requests. However, in HTTP/1 and HTTP/1.1, requests can only be made one at a time on a single connection; where in HTTP/2, multiple requests can be made over a single TCP connection. This chapter will help you restrict usage of your service and mitigate abuse.

2.1 Limiting Connections

Problem

You need to limit the number of connections based on a predefined key, such as the client's IP address.

Solution

Construct a shared memory zone to hold connection metrics, and use the `limit_conn` directive to limit open connections:

```
http {
    limit_conn_zone $binary_remote_addr zone=limitbyaddr:10m;
    limit_conn_status 429;
    ...
    server {
        ...
            limit_conn limitbyaddr 40;
        ...
    }
}
```

This configuration creates a shared memory zone named `limitbyaddr`. The predefined key used is the clients IP address in binary form. The size of the shared memory zone is set to 10 megabytes. The `limit_conn` directive takes two parameters: a `limit_conn_zone` name, and the number of connections allowed. The `limit_conn_status` sets the response when the connections are limited to a status of 429, indicating too many requests.

Discussion

Limiting the number of connections based on a key can be used to defend against abuse and share your resources fairly across all your clients. It is important to be cautious of your predefined key. Using an IP address, as we are in the previous example, could be dangerous if many users are on the same network that originates from the same IP, such as when behind a *Network Address Translation* (NAT). The entire group of clients will be limited. The `limit_conn_zone` directive is only valid in the HTTP context. You can utilize any number of variables available to NGINX within the HTTP context in order to build a string on which to limit by. Utilizing a variable that can identify the user at the application level, such as a session cookie, may be a cleaner solution depending on the use case. The `limit_conn` and `limit_conn_status` directives are valid in the HTTP, server, and location context. The `limit_conn_status` defaults to 503, service unavailable. You may find it preferable to use a 429, as the service is available, and 500 level responses indicate error.

2.2 Limiting Rate

Problem

You need to limit the rate of requests by predefined key, such as the client's IP address.

Solution

Utilize the rate-limiting module to limit the rate of requests:

```
http {
    limit_req_zone $binary_remote_addr
        zone=limitbyaddr:10m rate=1r/s;
    limit_req_status 429;
    ...
    server {
        ...
            limit_req zone=limitbyaddr burst=10 nodelay;
        ...
    }
}
```

This example configuration creates a shared memory zone named `limitbyaddr`. The predefined key used is the client's IP address in binary form. The size of the shared memory zone is set to 10 megabytes. The zone sets the rate with a keyword argument. The `limit_req` directive takes two keyword arguments: `zone` and `burst`. `zone` is required to instruct the directive on which shared memory request limit zone to use. When the request rate for a given zone is exceeded, requests are delayed until their maximum burst size is reached, denoted by the `burst` keyword argument. The `burst` keyword argument defaults to zero. `limit_req` also optionally takes a third parameter, `nodelay`. This parameter enables the client to use its `burst` without delay before being limited. `limit_req_status` sets the status returned to the client to a particular HTTP status code; the default is 503. `limit_req_status` and `limit_req` are valid in the context of HTTP, server, and location. `limit_req_zone` is only valid in the HTTP context.

Discussion

The rate-limiting module is very powerful in protecting against abusive rapid requests while still providing a quality service to everyone. There are many reasons to limit rate of request, one being

security. You can deny a brute force attack by putting a very strict limit on your login page. You can disable the plans of malicious users that might try to deny service to your application or to waste resources by setting a sane limit on all requests. The configuration of the rate-limit module is much like the preceding connection-limiting module described in Recipe 2.1, and much of the same concerns apply. The rate at which requests are limited can be done in requests per second or requests per minute. When the rate limit is hit, the incident is logged. There's a directive not in the example: `limit_req_log_level`, which defaults to error, but can be set to info, notice, or warn.

2.3 Limiting Bandwidth

Problem

You need to limit download bandwidths per client for your assets.

Solution

Utilize NGINX's `limit_rate` and `limit_rate_after` directives to limit the rate of response to a client:

```
location /download/ {
    limit_rate_after 10m;
    limit_rate 1m;
}
```

The configuration of this location block specifies that for URIs with the prefix *download*, the rate at which the response will be served to the client will be limited after 10 megabytes to a rate of 1 megabyte per second. The bandwidth limit is per connection, so you may want to institute a connection limit as well as a bandwidth limit where applicable.

Discussion

Limiting the bandwidth for particular connections enables NGINX to share its upload bandwidth with all of the clients in a fair manner. These two directives do it all: `limit_rate_after` and `limit_rate`. The `limit_rate_after` directive can be set in almost any context: http, server, location, and `if` when the `if` is within a location. The `limit_rate` directive is applicable in the same contexts as

`limit_rate_after`, however, it can alternatively be set by setting a variable named `$limit_rate`. The `limit_rate_after` directive specifies that the connection should not be rate limited until after a specified amount of data has been transferred. The `limit_rate` directive specifies the rate limit for a given context in bytes per second by default. However, you can specify `m` for megabytes or `g` for gigabytes. Both directives default to a value of 0. The value 0 means not to limit download rates at all.

CHAPTER 3

Encrypting

3.0 Introduction

The internet can be a scary place, but it doesn't have to be. Encryption for information in transit has become easier and more attainable in that signed certificates have become less costly with the advent of Let's Encrypt and Amazon Web Services. Both offer free certificates with limited usage. With free signed certificates, there's little standing in the way of protecting sensitive information. While not all certificates are created equal, any protection is better than none. In this chapter, we discuss how to secure information between NGINX and the client, as well as NGINX and upstream services.

3.1 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 8083 ssl;
        ssl_protocols       TLSv1.2;
        ssl_ciphers         AES128-SHA:AES256-SHA;
        ssl_certificate     /usr/local/nginx/conf/cert.pem;
        ssl_certificate_key /usr/local/nginx/conf/cert.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, 8083. The server accepts the SSL protocol version TLSv1.2. AES encryption cipers are allowed and the SSL certificate and key locations are disclosed to the server for use. The SSL session cache and timeout allow for 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. Session cache options can be used in conjunction. 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. At the time of writing, the *Transport Layer Security* protocol (TLS) is the default over the *Secure Socket Layer* (SSL) protocol. That's because versions 1 through 3 of SSL are now considered insecure. While the protocol name may 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 is also able to 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.

3.2 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 available, 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.

CHAPTER 4

HTTP Basic Authentication

4.0 Introduction

Basic authentication is a simple way to protect private content. This method of authentication can be used to easily hide development sites or keep privileged content hidden. Basic authentication is pretty unsophisticated, not extremely secure, and, therefore, should be used with other layers to prevent abuse. It's recommended to set up a rate limit on locations or servers that require basic authentication to hinder the rate of brute force attacks. It's also recommended to utilize HTTPS, as described in Chapter 3 whenever possible, as the username and password are passed as a base64 encoded string to the server in a header on every authenticated request. The implications of basic authentication over an unsecured protocol such as HTTP means that the username and password can be captured by any machine the request passes through.

4.1 Creating a User File

Problem

You need an HTTP basic authentication user file to store usernames and passwords.

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. An optional third field can be used for comment on each user. NGINX can understand a few different formats for passwords, one of which is if 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 with the following command:

```
$ openssl passwd MyPassword1234
```

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

Discussion

Basic authentication passwords can be generated a few ways and in a few different formats to 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 LDAP and Dovecot use. NGINX supports more formats and hashing algorithms, however, many of them are considered insecure because they can be easily cracked.

4.2 Using Basic Authentication

Problem

You need basic authentication to protect an NGINX location or server.

Solution

Use the `auth_basic` and `auth_basic_user_file` directives to enable basic authentication:

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

The `auth_basic` directives can be used 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, which was just described in Recipe 4.1.

Discussion

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

CHAPTER 5

HTTP Authentication Subrequests

5.0 Introduction

With many different approaches to authentication, NGINX makes it easy to validate against a wide range of authentication systems by enabling a subrequest mid-flight to validate identity. The HTTP authentication request module is meant to enable authentication systems like LDAP or custom authentication microservices. The authentication mechanism proxies the request to the authentication service before the request is fulfilled. During this proxy you have the power of NGINX to manipulate the request as the authentication service requires. Therefore, it is extremely flexible.

5.1 Authentication Subrequests

Problem

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

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 access or not. 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_reqeust_body` directive, as demonstrated. This practice will reduce the request size and time. Because the response body is discarded, the `Content-Length` header can 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 services 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.

CHAPTER 6

Secure Links

6.0 Introduction

Secure links are a way to keep static assets secure with the `md5` hashing algorithm. With this module, you can also put a limit on the length of time to which the link is accepted. Using secure links enables your NGINX application server to serve static content securely while taking this responsibility off of the application server. This module is included in the free and open source NGINX. However, it is not built into the standard NGINX package but instead the `nginx-extras` package. Alternatively, it can be enabled with the `--with-http_secure_link_module` configuration parameter when building NGINX from source.

6.1 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 concatenation 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.

6.2 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 6.1, we will create the secured link that will work with the previous configuration example given 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 2.7 using the `md5` library that is included in the Python Standard Library:

```
import md5
md5.new('index.htmlmySecret').hexdigest()
'a53bee08a4bf0bbea978ddf736363a12'
```

Now that we have this hash digest, we can use it in a URL. Our example will be for 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.

6.3 Securing a Location with an Expire Date

Problem

You need to generate a link that expires at some future time.

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 6.1. 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.

6.4 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
1609390800
```

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 a example on a Unix system:

```
$ echo -n '1609390800/resources/index.html127.0.0.1 mySecret' \
  | openssl md5 -binary \
  | openssl base64 \
  | tr +/ -_ \
  | tr -d =
81CYyxXFADhLHaQD36_BK
```

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

```
/resources/index.html?md5=81CYyxXFADhLHaQD36_BK&expires=1609390
  800'
```

The following is a full example in Python 2.7 using the Python Standard Library:

```
from datetime import datetime
from base64 import b64encode
import md5

resource = '/resources/index.html'
remote_addr = '127.0.0.1'
host = 'www.example.com'
expire = datetime(2020,12,31,0,0).strftime('%s')
uncoded = expire + resource + remote_addr + ' mySecret'
md5hashed = md5.new(uncoded).digest()
b64 = b64encode(md5hashed)
hash = b64.replace('+', '-').replace('/', '_').replace('=', '')

linkformat = "{}{}?md5={}?expires{}"
securelink = linkformat.format(host,resource,hash,expire)
```

Discussion

With this pattern we're able to generate a secure link in a special format that is able to be used in URLs. The secret provides security 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.

CHAPTER 7

API Authentication Using JWT

7.0 Introduction

JSON Web Tokens (JWTs) are quickly becoming a widely used and preferred authentication method. These authentication tokens have the ability to store some information about the user as well as information about the user's authorization into the token itself. These tokens can also be validated asynchronously, which means load balancers and proxies are able to validate the token with a public key and do not need the private key that the token was signed with, thus enhancing security and flexibility. An advantage of offloading authentication verification to your NGINX Plus layer is that you're saving cycles on your authentication service, as well as speeding up your transactions. The JWT authentication module described in this chapter is available only with an NGINX Plus subscription.

7.1 Validating JWTs

Problem

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

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 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 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 time and resources of making 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

7.2 Creating JSON Web Keys

Problem

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

Solution

NGINX Plus utilizes the the JSON Web Key (JWK) format as specified in the RFC standard. The 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 to 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. Keeping them in memory on your host is best practice. You can do so by creating an in-memory file system like ramfs.

Also See

RFC standardization documentation of JSON Web Key

CHAPTER 8

OpenId Connect Single Sign On

8.0 Introduction

Single sign-on (SSO) authentication providers are a great way to reduce authentication requests to your application and provide your users with seamless integration into an application they already log into on a regular basis. As more authentication providers bring themselves to market, your application can be ready to integrate by using NGINX Plus to validate the signature of their JSON Web Tokens. In this chapter we'll explore using the NGINX Plus JWT authentication module for HTTP in conjunction with an existing OpenId Connect OAuth 2.0 provider from Google. As in Chapter 7, this chapter describes the JWT authentication module, which is only available with a NGINX Plus subscription.

8.1 Authenticate Users via Existing OpenId Connect Single Sign-On (SSO)

Problem

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

Solution

Use NGINX Plus's JWT module to secure a location or server and tell 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 tells 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, we must tell NGINX to look for the token in this cookie rather than the NGINX Plus Default location. The `auth_jwt_key_file` location is set to an arbitrary path, a step which we will cover in Recipe 8.2.

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 single sign-on 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 JSON Web Tokens 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.

Also See

Detailed NGINX Blog on OpenId Connect
OpenId Connect

8.2 Obtaining 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 to 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 interval 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 a regular interval. This cron solution is one way of doing so.

Also See

Cron

CHAPTER 9

ModSecurity Web Application Firewall

9.0 Introduction

ModSecurity is an open source web application firewall (WAF) that was first built for Apache web servers. It was made available to NGINX as a module in 2012 and added as an optional feature to NGINX Plus in 2016. This chapter will detail installing ModSecurity 3.0 with NGINX Plus through dynamic modules. It will also cover compiling and installing the ModSecurity 2.9 module and NGINX from source. ModSecurity 3.0 with NGINX Plus is far superior to ModSecurity 2.x in terms of security and performance. When running ModSecurity 2.9 configured from open-source, it's still wrapped in Apache and, therefore, requires much more overhead than 3.0, which was designed for NGINX natively. The plug-and-play ModSecurity 3.0 module for NGINX is only available with a NGINX Plus subscription.

9.1 Installing ModSecurity for NGINX Plus

Problem

You need to install the ModSecurity module for NGINX Plus.

Solution

Install the module from the NGINX Plus repository. The package name is `nginx-plus-module-modsecurity`. On an Ubuntu-based system, you can install NGINX Plus and the ModSecurity module through the advanced packaging tool, also known as `apt-get`:

```
$ apt-get update
$ apt-get install nginx-plus
$ apt-get install nginx-plus-module-modsecurity
```

Discussion

Installing NGINX Plus and the ModSecurity module is as easy as pulling it from the NGINX Plus repository. Your package management tool, such as `apt-get` or `yum`, will install NGINX Plus as well as the module and place the module in the *modules* directory within the default NGINX Plus configuration directory */etc/nginx/*.

9.2 Configuring ModSecurity in NGINX Plus

Problem

You need to configure NGINX Plus to use the ModSecurity module.

Solution

Enable the dynamic module in your NGINX Plus configuration, and use the `modsecurity_rules_file` to point to a ModSecurity rule file:

```
load_module modules/ngx_http_modsecurity.so;
```

The `load_module` directive is applicable in the main context, which means that this directive is to be used before opening the HTTP or Stream blocks.

Turn on ModSecurity and use a particular rule set:

```
modsecurity on;
location / {
    proxy_pass http://backend;
    modsecurity_rules_file rule-set-file;
 }
```

The `modsecurity` directive turns on the module for the given context when passed the `on` parameter. The `modsecurity_rules_file` instructs NGINX Plus to use a particular ModSecurity rule set.

Discussion

The rules for ModSecurity can prevent common exploits of web servers and applications. ModSecurity is known to be able to prevent application-layer attacks such as HTTP violations, SQL injection, cross-site scripting, application-layer, distributed-denial-of-service, and remote and local file-inclusion attacks. With ModSecurity, you're able to subscribe to real-time blacklists of malicious user IPs to help block issues before your services are affected. The ModSecurity module also enables detailed logging to help identify new patterns and anomalies.

Also See

OWASP ModSecurity Core Rule Set
TrustWave ModSecurity Paid Rule Set

9.3 Installing ModSecurity from Source for a Web Application Firewall

Problem

You need to run a web application firewall with NGINX using ModSecurity and a set of ModSecurity rules on a CentOS or RHEL-based system.

Solution

Compile ModSecurity and NGINX from source and configure NGINX to use the ModSecurity module.

First update security and install prerequisites:

```
$ yum --security update -y && \
   yum -y install automake \
   autoconf \
   curl \
   curl-devel \
   gcc \
   gcc-c++ \
```

```
    httpd-devel \
    libxml2 \
    libxml2-devel \
    make \
    openssl \
    openssl-devel \
    perl \
    wget
```

Next, download and install PERL 5 regular expression pattern matching:

```
$ cd /opt && \
    wget http://ftp.exim.org/pub/pcre/pcre-8.39.tar.gz && \
    tar -zxf pcre-8.39.tar.gz && \
    cd pcre-8.39 && \
    ./configure && \
    make && \
    make install
```

Download and install zlib from source:

```
$ cd /opt && \
    wget http://zlib.net/zlib-1.2.8.tar.gz && \
    tar -zxf zlib-1.2.8.tar.gz && \
    cd zlib-1.2.8 && \
    ./configure && \
    make && \
    make install
```

Download and install ModSecurity from source:

```
$ cd /opt && \
  wget \
  https://www.modsecurity.org/tarball/2.9.1/modsecurity-2.9.1.\
 tar.gz&& \
  tar -zxf modsecurity-2.9.1.tar.gz && \
  cd modsecurity-2.9.1 && \
  ./configure --enable-standalone-module && \
  make
```

Download and install NGINX from source and include any modules you may need with the configure script. Our focus here is the ModSecurity module:

```
$ cd /opt && \
    wget http://nginx.org/download/nginx-1.11.4.tar.gz && \
    tar zxf nginx-1.11.4.tar.gz && \
    cd nginx-1.11.4 && \
    ./configure \
        --sbin-path=/usr/local/nginx/nginx \
        --conf-path=/etc/nginx/nginx.conf \
        --pid-path=/usr/local/nginx/nginx.pid \
```

```
        --with-pcre=../pcre-8.39 \
        --with-zlib=../zlib-1.2.8 \
        --with-http_ssl_module \
        --with-stream \
        --with-http_ssl_module \
        --with-http_secure_link_module \
        --add-module=../modsecurity-2.9.1/nginx/modsecurity \
    && \
    make && \
    make install && \
    ln -s /usr/local/nginx/nginx /usr/bin/nginx
```

This will yield NGINX compiled from source with the ModSecurity version 2.9.1 module installed. From here we are able to use the `ModSecurityEnabled` and `ModSecurityConfig` directives in our configurations:

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    server_name _;
    location / {
      ModSecurityEnabled on;
      ModSecurityConfig modsecurity.conf;
    }
}
```

This configuration for an NGINX server turns on ModSecurity for the `location /` and uses a ModSecurity configuration file located at the base of the NGINX configuration.

Discussion

This section compiles NGINX from source with the ModSecurity for NGINX. It's advised when compiling NGINX from source to always check that you're using the latest stable packages available. With the preceding example, you can use the open source version of NGINX along with ModSecurity to build your own open source web application firewall.

Also See

ModSecurity Source
Updated and maintained ModSecurity Rules from SpiderLabs

CHAPTER 10

Practical Security Tips

10.0 Introduction

Security is done in layers, and much like an onion, there must be multiple layers to your security model for it to be truly hardened. In this installment, we've gone through many different ways to secure your web applications with NGINX and NGINX Plus. Many of these security methods can be used in conjunction to help harden security. The following are a few more practical security tips to ensure your users are using HTTPS and to tell NGINX to satisfy one or more security methods.

10.1 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 host name. 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*.

10.2 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 `HTTP_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 cost 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 costs. This is a handy trick to make sure that your HTTP traffic is secured.

10.3 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 Chapter 1, the

`http_auth_basic_module` detailed in Chapter 4, the `http_auth_request_module` detailed in Chapter 5, and the `http_auth_jwt_module` detailed in Chapter 7. 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.

About the Author

Derek DeJonghe has had a lifelong passion for technology. His background and experience in web development, system administration, and networking give him a well-rounded understanding of modern web architecture. Derek leads a team of site reliability engineers and produces self-healing, auto-scaling infrastructure for numerous applications. He specializes in Linux cloud environments. While designing, building, and maintaining highly available applications for clients, he consults for larger organizations as they embark on their journey to the cloud. Derek and his team are on the forefront of a technology tidal wave and are engineering cloud best practices every day. With a proven track record for resilient cloud architecture, Derek helps RightBrain Networks be one of the strongest cloud consulting agencies and managed service providers in partnership with AWS today.