Tutorial 9 - Setting up a reverse proxy server

What are we doing?

We are configuring a *reverse proxy*, or *gateway server*, protecting access to the application and shielding the application server from the internet. In doing so, we'll become familiar with several configuration methods and will be working with *ModRewrite* for the first time.

Why are we doing this?

A modern application architecture has multiple layers. Only the *reverse proxy* is exposed to the internet. It conducts a security check on the application layer and forwards the requests found to be good to the application server in the second layer. This in turn is connected to a database server located in yet another layer. This is referred to as a *three-tier model*. In a staggered defense spanning three levels, the *reverse proxy* or to be technically correct, the *gateway server*, provides the first look into the encrypted requests. On the way back it is in turn the last instance in which the responses can be checked one last time.

There are a number of ways for converting an Apache server into a *reverse proxy*. More importantly, there are multiple ways of communicating with the application server. In this tutorial we will be restricting ourselves to the normal *HTTP*-based *mod_proxy_http*. We will not be discussing other methods of communication such as *FastCGI proxy* or *AJP* here. There are also several ways of getting the proxy process going in Apache. We will start by looking at the normal setup using *ProxyPass* and will afterwards discuss other options using *mod_rewrite*.

Requirements

- An Apache web server, ideally one created using the file structure shown in Tutorial 1 (Compiling an Apache web server).
- Understanding of the minimal configuration in Tutorial 2 (Configuring a minimal Apache server).
- An Apache web server with SSL/TLS support as in Tutorial 4 (Configuring an SSL server)
- An Apache web server with extended access log as in Tutorial 5 (Extending and analyzing the access log)
- An Apache web server with ModSecurity as in Tutorial 6 (Embedding ModSecurity)
- An Apache web server with Core Rules installation as in Tutorial 7 (Embedding Core Rules)

Step 1: Preparing the backend

The purpose of a reverse proxy is to shield an application server from direct internet access. As somewhat of a prerequisite for this tutorial, we'll be needing a backend server like this. In principle, any HTTP application can be used for such an installation and we could very well use the application server from the third tutorial. However, it seems appropriate for me to demonstrate a very simple approach. We'll be using the tool *socat*, short for *SOcket CAt*.

```
$> socat -vv TCP-LISTEN:8000,bind=127.0.0.1,crlf,reuseaddr,fork SYSTEM:"echo HTTP/1.0 200;\
echo Content-Type\: text/plain; echo; echo 'Server response, port 8000.'"
```

Using this complex command we instruct *socat*, to install a *listener* on local port 8000 and to use several *echoes* to return an HTTP response when a connection occurs. The additional parameters make sure that the listener stays permanently open and error output works.

```
$> curl -v http://localhost:8000/
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 8000 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost:8000
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200
< Content-Type: text/plain
```

```
<
Server response, port 8000
* Closing connection 0
```

We have set up a backup system with the simplest of means. So easy, that in the future we will might again be happy to be know about this method, when we want to verify that a proxy server is working before the right backend is running.

Step 2: Linking the proxy module

Several modules are required to use Apache as a *proxy server*. We compiled them in the first tutorial and can now simply link them.

```
LoadModule              proxy_module            modules/mod_proxy.so
LoadModule              proxy_http_module       modules/mod_proxy_http.so
```

The *proxying* feature set is provided by a basic proxy module and a proxy HTTP module. *Proxying* actually means receiving a request and forwarding it to another server. In our case we will be defining the backend system from the beginning and then accept requests from different clients for this backend service. It's a different situation if you set up a proxy server that accepts requests from a group of clients and sends then onward to any server on the internet. This is referred to as a *forward proxy*. This is useful for when you don't want to directly expose clients on a corporate network to the internet since the proxy server appears as a client to the servers on the internet.

This mode is also possible in Apache, even if for historical reasons. Alternative software packages offering these features have become well established, e.g. *squid*. The case is relevant insofar as a faulty configuration may have fatal consequences, particularly if the *forward proxy* accepts requests from any client and sends them onward to the internet in anonymized form. This is referred to as an *open proxy*. It's essential to prevent this since we don't want to operate Apache in this mode. That requires a directive in the past used to reference the risky default `on` value, but is now correctly predefined as `off`:

```
ProxyRequests Off
```

This directive actually means forwarding requests to servers on the internet, even if the name indicates a general setting. As mentioned before, the directive is correctly set for Apache 2.4 and it is only being mentioned here to guard against questions or incorrect settings in the future.

Step 3: ProxyPass

This brings us to the actual *proxying* settings: There are many ways for instructing Apache to forward a request to a backend application. We'll be looking at each option in turn. The most common way of proxying requests is based on the *ProxyPass* directive. It is used as follows:

```
ProxyPass       /service1   http://localhost:8000/service1
ProxyPassReverse    /service1   http://localhost:8000/service1

<Proxy http://localhost:8000/service1>

    Require all granted

    AllowOverride  None
    Options None

</Proxy>
```

The most important directive is *ProxyPass*. It defines a `/service1` path and specifies how it is mapped to the backend: To the service defined above running on our own host, localhost, on port 8000. The path to the application server is again `/service1`. We are proxying symmetrically, because the paths never change. This mapping is however not absolutely required. Technically, it would be entirely possible to proxy `service1` to `/`, but this results to administrative difficulties and misunderstandings, if a path in the log file no longer maps to the path on the *reverse proxy* and the requests can no longer be correlated.

On the next line comes a related directive that despite having a similar name performs only a small auxiliary function.

Redirect responses from the backend are fully qualified in *http-compliant* form. Such as `https://backend.example.com/service1`, for example. The address is however not accessible by the client. For this reason, the *reverse proxy* has to rewrite the backend's *location header*, `backend.example.com`, replacing it with its own name and thus mapping it back to its own *namespace*. *ProxyPassReverse*, with such a great name, only has a simple search and replace feature touching the *location headers*. As already seen in the *ProxyPass* directive, *proxying* is symmetric: the paths are rewritten 1:1. We are free to ignore this rule, but I urgently recommend keeping it, because misunderstandings and confusion lie beyond. In addition to accessing *location headers*, there is a series of further *reverse directives* for handling things like cookies. It can be useful from case to case.

Step 4: Proxy stanza

Continuing on in the configuration: now comes the *Proxy block* where the connection to the backend is more precisely defined. Specifically, this is where requests are authenticated and authorized. Further below in the tutorial we will also be adding a *load balancer* to this block.

The *Proxy bock* is similar to the *location* and the *directory block* we have previously become familiar with in our configuration. These are called *containers*. *Containers* specify to the web server how to structure the work. When a *container* appears in the configuration, it prepares a processing structure for it. In the case of *mod_proxy* the backend can also be accessed without a *Proxy container*. However, access protection is not taken into account and other directives no longer have any place where it can be inserted. Without the *Proxy block* the processing of complex servers remains a bit haphazard and it would do us well to configure this part as well. Using the *ProxySet* directive enables us to intervene even more here and specify things like the connection behavior. *min*, *max* and *smax* can be used to specify the number of threads assigned to the proxy connection pool. This can impact performance from case to case. The *keep-alive behavior* of the proxy connection can be influenced and a variety of different *timeouts* defined for it. Additional information is available in the Apache Project documentation.

Step 5: Defining exceptions when proxying and making other settings

The *ProxyPass* directive we are using has forwarded all requests for `/service1` to the backend. However, in practice it is often the case that you don't want to forward everything. Let's suppose there's a path `/service1/admin` that we don't want to expose to the internet. This can also be prevented by the appropriate *ProxyPass* setting, where the exception is initiated by using an exclamation mark. What's important is to define the exception before configuring the actual proxy command:

```
ProxyPass       /service1/admin !
ProxyPass       /service1   http://localhost:8000/service1
ProxyPassReverse    /service1   http://localhost:8000/service1
```

You often see configurations that forward the entire namespace below `/` to the backend. Then a number of exceptions to the pattern above are often defined. I think this is the wrong approach and I prefer to forward only what is actually being processed. The advantage is obvious: scanners and automated attacks looking for their next victim from a pool of IP addresses on the internet make requests for a lot of non-existent paths on our server. We can now forward them to the backend and may overload the backend or even put it in danger. Or we can just block these requests on the reverse proxy server. The latter is clearly preferable for security-related reasons.

An essential directive which may optionally be part of the proxy concerns the timeout. We defined our own *timeout* value for our server. This *timeout* is also used by the server for the connection to the backend. But this is not always wise, because while we can expect from the client that it will quickly make its request and not take its sweet time, depending on the backend application, it can take a while until the response is processed. For a short, general *timeout* which is wise to have for the client for defensive reasons, the *reverse proxy* would interrupt access to the backend too quickly. For this reason, there is a *ProxyTimeout* directive which affects only the connection to the backend. By the way, time measurement is not the total processing time on the backend, but the duration of time between IP packets: When the backend sends part of the response the clock is reset.

```
ProxyTimeout    60
```

Now comes time to fix the *host header*. Via the HTTP request host header the client specifies which of a server's *VirtualHosts* to use for the request. If there are multiple *VirtualHosts* being operated using the same IP address, this value is important. However, when forwarding the *reverse proxy* normally sets a new host header, specifically the one from the backend system. This is often undesired, because in many cases the backend system sets its links based on the host header. Fully qualified links for a backend application may be bad practice, but we avoid conflicts if we make clear from the beginning that the host

header should be preserved and forwarded as is by the backend.

```
ProxyPreserveHost   On
```

Backend systems often pay less attention to security than a reverse proxy. Error messages are one place this is obvious. Detailed error messages are often desirable since they enable the developer or backend administrator to get at the root of the problem. But we don't want to distribute them over the internet, because without authentication on the *reverse proxy* an attacker could always be lurking behind the client. It's better to hide error messages from the backend application or to replace it with an error message from the *reverse proxy*. The *ProxyErrorOverride* directive intervenes in the HTTP response body and replaces it if a status code greater than or equal to 400 is present. Requests with normal statuses below 400 are not affected by this directive.

```
ProxyErrorOverride  On
```

Step 6: ModRewrite

In addition to the *ProxyPass* directive, the *Rewrite module* can be used to enable *reverse proxy* features. Compared to *ProxyPass*, it enables more flexible configuration. We have not seen *ModRewrite* up to this point. Since this is a very important module, we should take a good look at it.

ModRewrite defines its own *rewrite engine* used to manipulate, or change, HTTP requests; This *rewrite engine* can run in the server or *VirtualHost* context. Strictly speaking, we are using two separate *rewrite engines*. The *rewrite engine* in the *VirtualHost context* can also be configured from the *Proxy container* that we learned about above. If we define a *rewrite engine* in the server context, then it may be circumvented if there is an *engine* in the *VirtualHost* context. in this case we have to manually ensure that the rewrite rules are being inherited. We are therefore setting up a *rewrite engine* in the server context, configuring an example rule and initiating the inheritance.

```
LoadModule              rewrite_module          modules/mod_rewrite.so
LoadModule              headers_module          modules/mod_headers.so

...

RewriteEngine           On
RewriteOptions          InheritDownBefore

RewriteRule         ^/$ %{REQUEST_SCHEME}://%{HTTP_HOST}/index.html  [redirect,last]
```

We initialize the engine on the server level. We then instruct the engine to pass on our rules to other rewrite engines. Specifically, so that our rules are performed before the rules further down. Then comes the actual rule. We tell the server to instruct the client to send a new request to */index.html* for a request without a path or a request for "/". This is a *redirect*. What's important is for the *redirect* to indicate the schema of the request, *http* or *https* as well as the host name. Relatives paths won't work. But because we are outside the *VirtualHost*, we don't see the type. And we don't want to hard code the host name, but prefer to take the host names from client requests. Both of these values are available as variables as you can see in the example above.

Then appearing within square brackets come the flags influencing the behavior of the rewrite rule. As previously mentioned, we want a *redirect* and tell the *engine* that this is the last rule to process (*last*).

Let's have a look at a request like this and the redirect returned:

```
$> curl -v http://localhost/
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost
> Accept: */*
>
< HTTP/1.1 302 Found
< Date: Thu, 10 Dec 2015 05:24:42 GMT
* Server Apache is not blacklisted
< Server: Apache
```

```
< Location: http://localhost/index.html
< Content-Length: 211
< Content-Type: text/html; charset=iso-8859-1
<
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>302 Found</title>
</head><body>
<h1>Found</h1>
<p>The document has moved <a href="http://localhost/index.html">here</a>.</p>
</body></html>
* Connection #0 to host localhost left intact
```

The server now responds with HTTP status code *302 Found*, corresponding to the typical *redirect status code*. Alternatively, *301*, *303*, *307* or very rarely *308* also appear. The differences are subtle, but influence the behavior of the browser. What's important then is the *location header*. It tells the client to make a new request, specifically for the fully qualified URL with a schema specified here. This is required for the *location header*. Only returning the path here, assuming that the client would then correctly conclude that it's the same server name, would be incorrect and prohibited according to the specification.

In the body part of the response the redirect is included as a link in HTML text. This is provided for users to click manually, if the browser does not initiate the redirect. This is however very unlikely and in my opinion, is for historical reasons only.

You could now ask yourself why we are opening a *rewrite engine* in the server context and not dealing with everything on the *VirtualHost* level. In the example I chose you see that this would result in redundancy, because the redirect from "/" to "index.html" should take place on port 80 and not on encrypted port 443. This is the rule of thumb: It's best for us to define and inherit everything being used on all *VirtualHosts* in the server context. We also deal with individual rules for a single *VirtualHost* on this level. Typical is the following rule we can use to redirect all requests from port 80 to port 443:

```
<VirtualHost 127.0.0.1:80>

    RewriteEngine       On

    RewriteRule     ^/(.*)$ https://%{HTTP_HOST}/$1 [redirect,last]

    ...

</VirtualHost>
```

The schema we want is now clear. But to the left of it comes a new item. We don't suppress the path as quickly as above. We instead put it in brackets and use *$1* to reference the content of the brackets again in the *redirect*. This means that we are forwarding the request on port 80 using the same URL on port 443.

ModRewrite has been introduced. For further examples refer to the documentation or the sections of this tutorial below where will become familiar with yet more recipes.

Step 7: ModRewrite [proxy]

We have seen how a *rewrite engine* is initialized and how you can easily trigger somewhat more complex redirects. Using these means we will now be configuring a *reverse proxy*. We do this as follows:

```
<VirtualHost 127.0.0.1:443>

    ...

    RewriteEngine   On

    RewriteRule     ^/service1/(.*)     http://localhost:8000/service1/$1 [proxy,last]
    ProxyPassReverse    /                   http://localhost:8000/

    <Proxy http://localhost:8000/service1>

    Require all granted

    AllowOverride None
    Options None

    </Proxy>
```

The instruction follows a pattern similar to the variation using ProxyPass. Here however, the last part of the path has to be explicitly intercepted by using a bracket and again indicated by "$1" as we saw above. Instead of the suggested *redirect flag* a *proxy* is used here. *ProxyPassReverse* and the proxy stanza remain identical to the setup using *ProxyPass*.

So much for the simple configuration using a rewrite rule. There is no real advantage over *ProxyPass* syntax. Referencing parts of paths by using *$1*, *$2*, etc. does provide a bit of flexibility. But if we are working with rewrite rules anyway, then by rewrite rule proxying we ensure that RewriteRule and ProxyPass don't come into conflict by touching the same request and impacting one another.

However, it may now be that we want to use a single reverse proxy to combine multiple backends or to distribute the load over multiple servers. This calls for our own load balancer. We'll be looking at it in the next section:

Step 8: Balancer [proxy]

We first have to link the Apache load balancer module:

```
LoadModule        proxy_balancer_module          modules/mod_proxy_balancer.so
LoadModule        lbmethod_byrequests_module     modules/mod_lbmethod_byrequests.so
LoadModule        slotmem_shm_module             modules/mod_slotmem_shm.so
```

Besides the load balancer module itself we also need a module that can help us distribute the requests to the different backends. We'll take the easiest route and load the *lbmethod_byrequests* module. It's the oldest module from a series of four modules and distributes requests evenly across backends by counting them sequentially. Once to the left and once to the right for two backends.

Here is a list of available algorithms:

- mod_lbmethod_byrequests (counts requests)
- mod_lbmethod_bytraffic (totals sizes of requests and responses)
- mod_lbmethod_bybusyness (Load balancing based on active threads in an connection established with the backend. The backend with the lowest number of threads is given the next request)
- mod_lbmethod_heartbeat (the backend can even communicate via a heartbeat on network and use it to inform the reverse proxy whether it has any free capacity).

The different modules are well documented online so this brief description will have to suffice for now. And finally, we still need a module to help us manage shared segments of memory. These features are required by the proxy balancer module and provided by `mod_slotmem_shm.so` .

We are now ready to configure the load balancer. We can now set It up via RewriteRule. This modification of RewriteRule also affects the proxy stanza, where the balancer just defined must be referenced and resolved:

```
    RewriteRule     ^/service1/(.*)     balancer://backend/service/$1   [proxy,last]
    ProxyPassReverse    /               balancer://backend/

    <Proxy balancer://backend>
        BalancerMember http://localhost:8000 route=backend-port-8000
        BalancerMember http://localhost:8001 route=backend-port-8001

        Require all granted

        AllowOverride None
        Options None

    </Proxy>
```

We are also defining two backends, one on previously configured port 8000 and a second on port 8001. I recommend using socat to quickly set up this service on a second port and then to try it out. I have defined a number of different responses so we will be able to see from the HTTP response which backend has processed the request. This then looks like this:

```
$> curl -v -k https://localhost/service1/index.html https://localhost/service1/index.html
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
```

```
* Connected to localhost (127.0.0.1) port 443 (#0)
* successfully set certificate verify locations:
*   CAfile: none
  CApath: /etc/ssl/certs
* SSLv3, TLS handshake, Client hello (1):
* SSLv3, TLS handshake, Server hello (2):
* SSLv3, TLS handshake, CERT (11):
* SSLv3, TLS handshake, Server key exchange (12):
* SSLv3, TLS handshake, Server finished (14):
* SSLv3, TLS handshake, Client key exchange (16):
* SSLv3, TLS change cipher, Client hello (1):
* SSLv3, TLS handshake, Finished (20):
* SSLv3, TLS change cipher, Client hello (1):
* SSLv3, TLS handshake, Finished (20):
* SSL connection using ECDHE-RSA-AES256-GCM-SHA384
* Server certificate:
*    subject: CN=lubuntu.fritz.box
*    start date: 2013-10-26 18:00:21 GMT
*    expire date: 2023-10-24 18:00:21 GMT
*    issuer: CN=lubuntu.fritz.box
*    SSL certificate verify ok.
> GET /service1/index.html HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost
> Accept: */*
>
< HTTP/1.1 200
< Date: Thu, 10 Dec 2015 05:42:14 GMT
* Server Apache is not blacklisted
< Server: Apache
< Content-Type: text/plain
< Content-Length: 28
<
Server response, port 8000
* Connection #0 to host localhost left intact
* Found bundle for host localhost: 0x24e3660
* Re-using existing connection! (#0) with host localhost
* Connected to localhost (127.0.0.1) port 443 (#0)
> GET /service1/index.html HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost
> Accept: */*
>
< HTTP/1.1 200
< Date: Thu, 10 Dec 2015 05:42:14 GMT
* Server Apache is not blacklisted
< Server: Apache
< Content-Type: text/plain
< Content-Length: 28
<
Server response, port 8001
* Connection #0 to host localhost left intact
```

In this somewhat unusual request two identical request are being initiated via a single curl command. What's also interesting is the fact that with this method curl can use HTTP keep-alive. The first request lands on the first backend, and the second one on the second backend. Let's have a look at the entries for this in the server's access log:

```
127.0.0.1 - - [2015-12-10 06:42:14.390998] "GET /service1/index.html HTTP/1.1" 200 28 "-" "curl/7.35.0"\
localhost 127.0.0.1 443 proxy-server backend-port-8000 + "-" VmkQtn8AAQEAAH@M3zAAAAAN TLSv1.2 \
ECDHE-RSA-AES256-GCM-SHA384 538 1402 -% 7856 1216 3708 381 0 0
127.0.0.1 - - [2015-12-10 06:42:14.398995] "GET /service1/index.html HTTP/1.1" 200 28 "-" "curl/7.35.0"\
localhost 127.0.0.1 443 proxy-server backend-port-8001 + "-" VmkQtn8AAQEAAH@M3zEAAAAN TLSv1.2 \
ECDHE-RSA-AES256-GCM-SHA384 121 202 -% 7035 1121 3752 354 0 0
```

Besides the keep-alive header, the request handler is also of interest. The request was thus processed by the *proxy server handler*. We also see entries in the route, specifically the values defined as *backend-port-8000* and *backend-port-8001*. This makes it possible to determine from the server's access log the exact route a request took.

In a subsequent tutorial we will be seeing that the proxy balancer can also be used in other situations. For the moment we will however be content with what is happening and will now be turning to RewriteMaps. RewriteMaps is a auxiliary structure

which again increases the power of ModRewrite. Combined with the proxy server, flexibility rises substantially.

Step 9: RewriteMap [proxy]

RewriteMaps comes in a number of different variations. It works by assigning a value to a key parameter at every request. A hash table is a simple example. But it is then also not possible to configure external scripts as a programmable RewriteMap. The following types of maps are possible:

- txt : A key value pair in a text file is searched for here.
- rnd : Several values can be specified for each key here. They are then selected at random.
- dbm : This variation works like the txt variation, but provides a big speed advantage as a binary hash table.
- int : This abbreviation stands for *internal function* and refers to a function from the following list: *toupper*, *tolower*, *escape* and *unescape*.
- prg : An external script is invoked in this variation. The script is started along with the server and each time the RewriteMap is accessed receives new input via STDIN.
- dbd und fastdbd : The response value is searched for in a database request.

This list makes clear that RewriteMaps are extremely flexible and can be used in a variety of situations. Determining the backend for proxying is only one of many possible applications. In our example we want to ensure that the request from a specific client always goes to the same backend. There are a number of different ways of doing this, specifically, by setting a cookie. But we don't want to intervene in the requests and at the same time prevent a large number of clients from a specific network range from all being taken to the same backend. Some kind of distribution should thus take place. To do so, we combine ModSecurity using ModRewrite and a RewriteMap. Let's have a look at it step by step.

First, we calculate a hash value from the client's IP address. This means that we are converting the IP address into a random hexadecimal string:

```
SecRule REMOTE_ADDR "^(.)" \
    "phase:1,id:50001,capture,nolog,t:sha1,t:hexEncode,setenv:IPHashChar=%{TX.1}"
```

We have used hexEncode to convert the binary hash value we generated using sha1 into readable characters. We then apply the regular expression to this value. "^(.)" means that we want to find a match on any of the first characters. Of the ModSecurity flags that follow *capture* is of interest. It indicates the value of the *TX.1* transaction variable in brackets. We then extract the value of this variable and put it into the IPHashChar environment variable.

If there is any uncertainty as to whether this will really work, then the content of the variable *IPHashChar* can be mapped and checked using *%{IPHashChar}e* on the server's access log. This brings us to RewriteMap and the request itself:

```
RewriteMap hashchar2backend "txt:/apache/conf/hashchar2backend.txt"

RewriteCond     "%{ENV:IPHashChar}" ^(.)
RewriteRule     ^/service1/(.*)     \
    http://${hashchar2backend:%1|localhost:8000}/service1/$1 [proxy,last]

<Proxy http://localhost:8000/service1>

    Require all granted

    AllowOverride None
    Options None

</Proxy>

<Proxy http://localhost:8001/service1>

    Require all granted

    AllowOverride None
    Options None

</Proxy>
```

We introduce the map by using the RewriteMap command. We assign it a name, define its type and the path to the file. RewriteMap is invoked in a RewriteRule. Before we really access the map, we enable a rewrite condition. This is done using

the *RewriteCond* directive. There we reference the *IPHashChar* environment variable and determine the first byte of the variable. We know that only a single byte is included in the variation, but this won't put a stop to our plans. On the next line then the typical start of the Proxy directive. But instead of now specifying the backend, we reference RewriteMap by the name previously assigned. After the colon comes the parameter for the request. Interestingly, we use *%1* to communicate with the rewrite conditions captured in brackets. The RewriteRule variable is not affected by this and continues to be referenced via *$1*. After the *%1* comes the default value separated by a pipe character. Should anything go wrong when accessing the map, then communication with *localhost* takes place over port 8000.

All we need now is the RewriteMap. In the code sample we specified a text file. Better performance is provided by a dbm hash, but this is not the focus at the present. Here's the `/apache/conf/hashchar2backend.txt` map file:

```
##
## RewriteMap linking hex characters with one of two backends
##
1   localhost:8000
2   localhost:8000
3   localhost:8000
4   localhost:8000
5   localhost:8000
6   localhost:8000
7   localhost:8000
8   localhost:8000
9   localhost:8001
0   localhost:8001
a   localhost:8001
b   localhost:8001
c   localhost:8001
d   localhost:8001
e   localhost:8001
f   localhost:8001
```

We are differentiating between two backends and can perform the distribution any way we want. All in all, this is more indicative of a complex recipe that we put together forming a hash for each IP address and using the first character in order to determine one of two backends in the hash tables we just saw. If the client IP address remains constant (which does not always have to be the case in practice) the result of this lookup will always be the same. This means that the client will always end up on the same backend. This is called IP stickiness. However, since this entails is a hash operation and not a simple IP address lookup, two clients with a similar IP address will be given a completely different hash and will not necessarily end up on the same backend. This gives us a somewhat flat distribution of the requests yet we can still be sure that specific clients will always end up on the same backend until the IP address changes.

Step 10: Forwarding information to backend systems

The *reverse proxy* server shields the application server from direct client access. However, this also means that the application server is no longer able to see certain types of information about the client and its connection to the *reverse proxy*. To compensate for this loss, the Proxy module sets three HTTP request header lines that describe the *reverse proxy:

- X-Forwarded-For : The IP address of the reverse proxy
- X-Forwarded-Host : The original HTTP host header in the client request
- X-Forwarded-Server : The name of the *reverse proxy* server

If multiple reverse proxies are staggered behind one another then the additional IP addresses and server names are comma separated. In addition to this information about the connection, it is also a good idea to pass along yet more information. This would of course include the unique ID, uniquely identifying the request. A well-configured backend server will create a key value similar to our *reverse proxy* in the log rile. Being able to easily correlate the different log file entries simplifies debugging in the future.

A reverse proxy is frequently uses to perform authentication. Although we haven't set that up yet, it is still wise to add this value to an expanding basic configuration. If authentication is not defined, this value simply remains empty. And finally, we want to tell the backend system about the type of encryption the client and *reverse proxy* agreed upon. The entire block looks like this:

```
RequestHeader set "X-RP-UNIQUE-ID"  "%{UNIQUE_ID}e"
RequestHeader set "X-RP-REMOTE-USER"    "%{REMOTE_USER}e"
RequestHeader set "X-RP-SSL-PROTOCOL"   "%{SSL_PROTOCOL}s"
```

```
RequestHeader set "X-RP-SSL-CIPHER"     "%{SSL_CIPHER}s"
```

Let's see how this affects the request between the *reverse proxy* and the backend:

```
GET /service1/index.html HTTP/1.1
Host: localhost
User-Agent: curl/7.35.0
Accept: */*
X-RP-UNIQUE-ID: VmpSwH8AAQEAAG@hXBcAAAAC
X-RP-REMOTE-USER: (null)
X-RP-SSL-PROTOCOL: TLSv1.2
X-RP-SSL-CIPHER: ECDHE-RSA-AES256-GCM-SHA384
X-Forwarded-For: 127.0.0.1
X-Forwarded-Host: localhost
X-Forwarded-Server: localhost
Connection: close
```

The different extended header lines are listed sequentially and are filled in with values where present.

Step 11 (Goodie): Configuration of the complete reverse proxy server including the preceding tutorials

This small extension brings us to the end of this tutorial and also to the end of the basic block of different tutorials. Over several tutorials we have seen how to set up an Apache web server, from compiling it to the basic configuration, and ModSecurity tuning to reverse proxies, gaining deep insight into the how the server and its most important modules work.

Now, here's the complete configuration for the *reverse proxy server* that we worked out in the last tutorials.

```
ServerName        localhost
ServerAdmin       root@localhost
ServerRoot        /apache
User              www-data
Group             www-data
PidFile           logs/httpd.pid

ServerTokens      Prod
UseCanonicalName  On
TraceEnable       Off

Timeout           10
MaxClients        100

Listen            127.0.0.1:80
Listen            127.0.0.1:443

LoadModule        mpm_event_module        modules/mod_mpm_event.so
LoadModule        unixd_module            modules/mod_unixd.so

LoadModule        log_config_module       modules/mod_log_config.so
LoadModule        logio_module            modules/mod_logio.so
LoadModule        rewrite_module          modules/mod_rewrite.so
LoadModule        headers_module          modules/mod_headers.so

LoadModule        authn_core_module       modules/mod_authn_core.so
LoadModule        authz_core_module       modules/mod_authz_core.so

LoadModule        ssl_module              modules/mod_ssl.so

LoadModule        unique_id_module        modules/mod_unique_id.so
LoadModule        security2_module        modules/mod_security2.so

LoadModule        proxy_module            modules/mod_proxy.so
LoadModule        proxy_http_module       modules/mod_proxy_http.so
LoadModule        proxy_balancer_module   modules/mod_proxy_balancer.so
LoadModule        lbmethod_byrequests_module modules/mod_lbmethod_byrequests.so
LoadModule    slotmem_shm_module      modules/mod_slotmem_shm.so


ErrorLogFormat          "[%{cu}t] [%-m:%-l] %-a %-L %M"
```

```
LogFormat "%h %{GEOIP_COUNTRY_CODE}e %u [%{%Y-%m-%d %H:%M:%S}t.%{usec_frac}t] \"%r\" %>s %b \
\"%{Referer}i\" \"%{User-Agent}i\" %v %A %p %R %{BALANCER_WORKER_ROUTE}e %X \"%{cookie}n\" \
%{UNIQUE_ID}e %{SSL_PROTOCOL}x %{SSL_CIPHER}x %I %O %{ratio}n%% \
%D %{ModSecTimeIn}e %{ApplicationTime}e %{ModSecTimeOut}e \
%{ModSecAnomalyScoreIn}e %{ModSecAnomalyScoreOut}e" extended

LogFormat "[%{%Y-%m-%d %H:%M:%S}t.%{usec_frac}t] %{UNIQUE_ID}e %D \
PerfModSecInbound: %{TX.perf_modsecinbound}M \
PerfAppl: %{TX.perf_application}M \
PerfModSecOutbound: %{TX.perf_modsecoutbound}M \
TS-Phase1: %{TX.ModSecTimestamp1start}M-%{TX.ModSecTimestamp1end}M \
TS-Phase2: %{TX.ModSecTimestamp2start}M-%{TX.ModSecTimestamp2end}M \
TS-Phase3: %{TX.ModSecTimestamp3start}M-%{TX.ModSecTimestamp3end}M \
TS-Phase4: %{TX.ModSecTimestamp4start}M-%{TX.ModSecTimestamp4end}M \
TS-Phase5: %{TX.ModSecTimestamp5start}M-%{TX.ModSecTimestamp5end}M \
Perf-Phase1: %{PERF_PHASE1}M \
Perf-Phase2: %{PERF_PHASE2}M \
Perf-Phase3: %{PERF_PHASE3}M \
Perf-Phase4: %{PERF_PHASE4}M \
Perf-Phase5: %{PERF_PHASE5}M \
Perf-ReadingStorage: %{PERF_SREAD}M \
Perf-WritingStorage: %{PERF_SWRITE}M \
Perf-GarbageCollection: %{PERF_GC}M \
Perf-ModSecLogging: %{PERF_LOGGING}M \
Perf-ModSecCombined: %{PERF_COMBINED}M" perflog

LogLevel                        debug
ErrorLog                        logs/error.log
CustomLog                       logs/access.log extended
CustomLog                       logs/modsec-perf.log perflog env=write_perflog


# == ModSec Base Configuration

SecRuleEngine                   On

SecRequestBodyAccess            On
SecRequestBodyLimit             10000000
SecRequestBodyNoFilesLimit      64000

SecResponseBodyAccess           On
SecResponseBodyLimit            10000000

SecPcreMatchLimit               15000
SecPcreMatchLimitRecursion      15000

SecTmpDir                       /tmp/
SecDataDir                      /tmp/
SecUploadDir                    /tmp/

SecDebugLog                     /apache/logs/modsec_debug.log
SecDebugLogLevel                0

SecAuditEngine                  RelevantOnly
SecAuditLogRelevantStatus       "^(?:5|4(?!04))"
SecAuditLogParts                ABIJEFHKZ

SecAuditLogType                 Concurrent
SecAuditLog                     /apache/logs/modsec_audit.log
SecAuditLogStorageDir           /apache/logs/audit/

SecDefaultAction                "phase:1,pass,log,tag:'Local Lab Service'"


# == ModSec Rule ID Namespace Definition
# Service-specific before Core-Rules:    10000 -  49999
# Service-specific after Core-Rules:     50000 -  79999
# Locally shared rules:                  80000 -  99999
# - Performance:                        90000 -  90199
# Recommended ModSec Rules (few):       200000 - 200010
# OWASP Core-Rules:                     900000 - 999999


# === ModSec timestamps at the start of each phase (ids: 90000 - 90009)
```

```
SecAction "id:'90000',phase:1,nolog,pass,setvar:TX.ModSecTimestamp1start=%{DURATION}"
SecAction "id:'90001',phase:2,nolog,pass,setvar:TX.ModSecTimestamp2start=%{DURATION}"
SecAction "id:'90002',phase:3,nolog,pass,setvar:TX.ModSecTimestamp3start=%{DURATION}"
SecAction "id:'90003',phase:4,nolog,pass,setvar:TX.ModSecTimestamp4start=%{DURATION}"
SecAction "id:'90004',phase:5,nolog,pass,setvar:TX.ModSecTimestamp5start=%{DURATION}"

# SecRule REQUEST_FILENAME "@beginsWith /" "id:'90005',phase:5,t:none,nolog,noauditlog,pass,setenv:write_pe


# === ModSec Recommended Rules (in modsec src package) (ids: 200000-200010)

SecRule REQUEST_HEADERS:Content-Type "text/xml" \
    "id:'200000',phase:1,t:none,t:lowercase,pass,nolog,ctl:requestBodyProcessor=XML"

SecRule REQBODY_ERROR "!@eq 0" \
    "id:'200001',phase:2,t:none,deny,status:400,log,msg:'Failed to parse request body.',\
logdata:'%{reqbody_error_msg}',severity:2"

SecRule MULTIPART_STRICT_ERROR "!@eq 0" \
"id:'200002',phase:2,t:none,log,deny,status:403, \
msg:'Multipart request body failed strict validation: \
PE %{REQBODY_PROCESSOR_ERROR}, \
BQ %{MULTIPART_BOUNDARY_QUOTED}, \
BW %{MULTIPART_BOUNDARY_WHITESPACE}, \
DB %{MULTIPART_DATA_BEFORE}, \
DA %{MULTIPART_DATA_AFTER}, \
HF %{MULTIPART_HEADER_FOLDING}, \
LF %{MULTIPART_LF_LINE}, \
SM %{MULTIPART_MISSING_SEMICOLON}, \
IQ %{MULTIPART_INVALID_QUOTING}, \
IP %{MULTIPART_INVALID_PART}, \
IH %{MULTIPART_INVALID_HEADER_FOLDING}, \
FL %{MULTIPART_FILE_LIMIT_EXCEEDED}'"

SecRule TX:/^MSC_/ "!@streq 0" "id:'200004',phase:2,t:none,deny,status:500,\
msg:'ModSecurity internal error flagged: %{MATCHED_VAR_NAME}'"


# === ModSecurity Rules (ids: 900000-999999)

# === ModSec Core Rules Base Configuration (ids: 900001-900021)

SecAction "id:'900001',phase:1,t:none, \
   setvar:tx.critical_anomaly_score=5, \
   setvar:tx.error_anomaly_score=4, \
   setvar:tx.warning_anomaly_score=3, \
   setvar:tx.notice_anomaly_score=2, \
   nolog, pass"
SecAction "id:'900002',phase:1,t:none,\
setvar:tx.inbound_anomaly_score_level=10000,setvar:tx.inbound_anomaly_score=0,nolog,pass"
SecAction "id:'900003',phase:1,t:none,\
setvar:tx.outbound_anomaly_score_level=10000,setvar:tx.outbound_anomaly_score=0,nolog,pass"
SecAction "id:'900004',phase:1,t:none,setvar:tx.anomaly_score_blocking=on,nolog,pass"

SecAction "id:'900006',phase:1,t:none,setvar:tx.max_num_args=255,nolog,pass"
SecAction "id:'900007',phase:1,t:none,setvar:tx.arg_name_length=100,nolog,pass"
SecAction "id:'900008',phase:1,t:none,setvar:tx.arg_length=400,nolog,pass"
SecAction "id:'900009',phase:1,t:none,setvar:tx.total_arg_length=64000,nolog,pass"
SecAction "id:'900010',phase:1,t:none,setvar:tx.max_file_size=10000000,nolog,pass"
SecAction "id:'900011',phase:1,t:none,setvar:tx.combined_file_sizes=10000000,nolog,pass"
SecAction "id:'900012',phase:1,t:none, \
  setvar:'tx.allowed_methods=GET HEAD POST OPTIONS', \
  setvar:'tx.allowed_request_content_type=application/x-www-form-urlencoded\
|multipart/form-data|text/xml|application/xml|application/x-amf|application/json', \
  setvar:'tx.allowed_http_versions=HTTP/0.9 HTTP/1.0 HTTP/1.1', \
  setvar:'tx.restricted_extensions=.asa/ .asax/ .ascx/ .axd/ .backup/ .bak/ .bat/ \
.cdx/ .cer/ .cfg/ .cmd/ .com/ .config/ .conf/ .cs/ .csproj/ .csr/ .dat/ .db/ \
.dbf/ .dll/ .dos/ .htr/ .htw/ .ida/ .idc/ .idq/ .inc/ .ini/ .key/ .licx/ .lnk/ \
.log/ .mdb/ .old/ .pass/ .pdb/ .pol/ .printer/ .pwd/ .resources/ .resx/ .sql/ \
.sys/ .vb/ .vbs/ .vbproj/ .vsdisco/ .webinfo/ .xsd/ .xsx/', \
  setvar:'tx.restricted_headers=/Proxy-Connection/ /Lock-Token/ /Content-Range/ \
/Translate/ /via/ /if/', \
  nolog,pass"
```

```
SecRule REQUEST_HEADERS:User-Agent "^(.*)$" \
    "id:'900018',phase:1,t:none,t:sha1,t:hexEncode,setvar:tx.ua_hash=%{matched_var}, \
  nolog,pass"
SecRule REQUEST_HEADERS:x-forwarded-for "^\b(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})\b" \
    "id:'900019',phase:1,t:none,capture,setvar:tx.real_ip=%{tx.1},nolog,pass"
SecRule &TX:REAL_IP "!@eq 0" \
    "id:'900020',phase:1,t:none,initcol:global=global,initcol:ip=%{tx.real_ip}_%{tx.ua_hash}, \
  nolog,pass"
SecRule &TX:REAL_IP "@eq 0" \
    "id:'900021',phase:1,t:none,initcol:global=global,initcol:ip=%{remote_addr}_%{tx.ua_hash},\
    setvar:tx.real_ip=%{remote_addr},nolog,pass"


# === ModSecurity Ignore Rules Before Core Rules Inclusion; order by id of ignored rule (ids: 10000-49999)

# ...


# === ModSecurity Core Rules Inclusion

Include  /modsecurity-core-rules/*.conf


# === ModSecurity Ignore Rules After Core Rules Inclusion; order by id of ignored rule (ids: 50000-59999)

# ...


# === ModSec Timestamps at the End of Each Phase (ids: 90010 - 90019)

SecAction "id:'90010',phase:1,pass,nolog,setvar:TX.ModSecTimestamp1start=%{DURATION}"
SecAction "id:'90011',phase:2,pass,nolog,setvar:TX.ModSecTimestamp2end=%{DURATION}"
SecAction "id:'90012',phase:3,pass,nolog,setvar:TX.ModSecTimestamp3end=%{DURATION}"
SecAction "id:'90013',phase:4,pass,nolog,setvar:TX.ModSecTimestamp4end=%{DURATION}"
SecAction "id:'90014',phase:5,pass,nolog,setvar:TX.ModSecTimestamp5end=%{DURATION}"


# === ModSec performance calculations and variable export (ids: 90100 - 90199)

SecAction "id:'90100',phase:5,pass,nolog,setvar:TX.perf_modsecinbound=%{PERF_PHASE1}"
SecAction "id:'90101',phase:5,pass,nolog,setvar:TX.perf_modsecinbound=+%{PERF_PHASE2}"
SecAction "id:'90102',phase:5,pass,nolog,setvar:TX.perf_application=%{TX.ModSecTimestamp3start}"
SecAction "id:'90103',phase:5,pass,nolog,setvar:TX.perf_application=-%{TX.ModSecTimestamp2end}"
SecAction "id:'90104',phase:5,pass,nolog,setvar:TX.perf_modsecoutbound=%{PERF_PHASE3}"
SecAction "id:'90105',phase:5,pass,nolog,setvar:TX.perf_modsecoutbound=+%{PERF_PHASE4}"
SecAction "id:'90106',phase:5,pass,nolog,setenv:ModSecTimeIn=%{TX.perf_modsecinbound}"
SecAction "id:'90107',phase:5,pass,nolog,setenv:ApplicationTime=%{TX.perf_application}"
SecAction "id:'90108',phase:5,pass,nolog,setenv:ModSecTimeOut=%{TX.perf_modsecoutbound}"
SecAction "id:'90109',phase:5,pass,nolog,setenv:ModSecAnomalyScoreIn=%{TX.inbound_anomaly_score}"
SecAction "id:'90110',phase:5,pass,nolog,setenv:ModSecAnomalyScoreOut=%{TX.outbound_anomaly_score}"


# === ModSec finished

RewriteEngine           On
RewriteOptions          InheritDownBefore

RewriteRule             ^/$  %{REQUEST_SCHEME}://%{HTTP_HOST}/index.html  [redirect,last]



SSLCertificateKeyFile   /etc/ssl/private/ssl-cert-snakeoil.key
SSLCertificateFile      /etc/ssl/certs/ssl-cert-snakeoil.pem

SSLProtocol             All -SSLv2 -SSLv3
SSLCipherSuite          'kEECDH+ECDSA kEECDH kEDH HIGH +SHA !aNULL !eNULL !LOW !MEDIUM !MD5 !EXP !DSS \
!PSK !SRP !kECDH !CAMELLIA !RC4'
SSLHonorCipherOrder     On

SSLRandomSeed           startup file:/dev/urandom 2048
SSLRandomSeed           connect builtin

DocumentRoot            /apache/htdocs
```

```
<Directory />

      Require all denied

      Options SymLinksIfOwnerMatch
      AllowOverride None

</Directory>

<VirtualHost 127.0.0.1:80>

    RewriteEngine     On

    RewriteRule       ^/(.*)$  https://%{HTTP_HOST}/$1  [redirect,last]

    <Directory /apache/htdocs>

        Require all granted

        Options None
        AllowOverride None

    </Directory>

</VirtualHost>

<VirtualHost 127.0.0.1:443>

    SSLEngine On

    ProxyTimeout              60
    ProxyErrorOverride        On

    RewriteEngine             On

    RewriteRule               ^/service1/(.*)   http://localhost:8000/service1/$1 [proxy,last]
    ProxyPassReverse          /                 http://localhost:8000/


    <Proxy http://localhost:8000/service1>

        Require all granted

        AllowOverride None
        Options None

    </Proxy>

    <Directory /apache/htdocs>

        Require all granted

        Options None
        AllowOverride None

    </Directory>

</VirtualHost>
```

References

- Apache mod_proxy https://httpd.apache.org/docs/2.4/mod/mod_proxy.html)
- Apache mod_rewrite https://httpd.apache.org/docs/2.4/mod/mod_rewrite.html)

License / Copying / Further use