The HAProxy Guide to Multi-Layered Security

Defense in Depth Using the Building Blocks of HAProxy

HAPROXY

The HAProxy Guide to Multi-Layered Security

Defense in Depth Using the Building Blocks of HAProxy

Chad Lavoie

Table of Contents

Our Approach to Multi-Layered Security **4**

Introduction to HAProxy ACLs **6**

- Formatting an ACL 7
- Fetches 11
- Converters 12
- Flags 13
- Matching Methods 14
- Things to do with ACLs 16
- Selecting a Backend 18
- Setting an HTTP Header 20
- Changing the URL 21
- Updating Map Files 21
- Caching 23
- Using ACLs to Block Requests 23
- Updating ACL Lists 26
- Conclusion 27

Introduction to HAProxy Stick Tables **28**

- Uses of Stick Tables 29
- Defining a Stick Table 31
- Making Decisions Based on Stick Tables 44
- Other Considerations 49
- Conclusion 55

Introduction to HAProxy Maps **56**

- The Map File 57
- Modifying the Values 61
- Putting It Into Practice 70

Conclusion 74

Application-Layer DDoS Attack Protection 75
HTTP Flood 76
Manning the Turrets 77
Setting Request Rate Limits 79
Slowloris Attacks 83
Blocking Requests by Static Characteristics 84
Protecting TCP (non-HTTP) Services 89
The Stick Table Aggregator 91
The reCAPTCHA and Antibot Modules 92
Conclusion 95

Bot Management with HAProxy 96
HAProxy Load Balancer 97
Bot Management Strategy 98
Beyond Scrapers 107
Allowlisting Good Bots 111
Identifying Bots By Their Location 113
Conclusion 116

The HAProxy Enterprise WAF 118
A Specific Countermeasure 119
Routine Scanning 120
HAProxy Enterprise WAF 127
Retesting with WAF Protection 129
Conclusion 132

Our Approach to Multi-Layered Security

Defending your infrastructure can involve a dizzying number of components: from network firewalls to intrusion-detection systems to access control safeguards. Wouldn't it be nice to simplify this? We always like to be the bearer of good news. So, do you know that the HAProxy load balancer—which you might already be using—is packed full of security features?

HAProxy is used all over the globe for adding resilience to critical websites and services. As a high-performance, open-source load balancer that so many companies depend on, making it reliable gets top billing and it's no surprise that that's what people know it for. However, the same components that you might use for sticking a client to a server, routing users to the proper backend, and mapping large sets of data to variables can be used to secure your infrastructure.

In this book, we decided to cast some of these battle-tested capabilities in a different light. To start off, we'll introduce you to the building blocks that make up HAProxy: ACLs, stick tables, and maps. Then, you will see how when combined they allow you to resist malicious bot traffic, dull the power of a DDoS attack, and other handy security recipes.

HAProxy Technologies, the company behind HAProxy, owns its mission to provide advanced protection for those who need it. Throughout this book, we'll highlight areas where HAProxy Enterprise, which combines the stable codebase of HAProxy with an advanced suite of add-ons, expert support and professional services, can layer on additional defenses.

At the end, you'll learn about the HAProxy Web Application Firewall, which catches application-layer attacks that are missed by other types of firewalls. In today's threat-rich environment, a WAF is an essential service.

This book is for those new to HAProxy, as well as those looking to learn some new tricks. In the end, if we've heightened your awareness to the attacks leveraged by hackers and the creative ways of shutting them down, then we'll feel like we've done our job.

Introduction to HAProxy ACLs

When IT pros add load balancers into their infrastructure, they're looking for the ability to scale out their websites and services, get better availability, and gain more restful nights knowing that their critical services are no longer single points of failure. Before long, however, they realize that with a full-featured load balancer like HAProxy Enterprise, they can add in extra intelligence to inspect incoming traffic and make decisions on the fly.

For example, you can restrict who can access various endpoints, redirect non-HTTPS traffic to HTTPS, and detect and block malicious bots and scanners; you can define conditions for adding HTTP headers, change the URL or redirect the user.

Access Control Lists, or ACLs, in HAProxy allow you to test various conditions and perform a given action based on those tests. These conditions cover just about any aspect of a request or response such as searching for strings or patterns, checking IP addresses, analyzing recent request rates (via stick tables), and observing TLS statuses. The action you take can include making routing decisions,

redirecting requests, returning static responses and so much more. While using logic operators (`AND`, `OR`, `NOT`) in other proxy solutions might be cumbersome, HAProxy embraces them to form more complex conditions.

Formatting an ACL

There are two ways of specifying an ACL—a **named ACL** and an **anonymous** or **in-line ACL**. The first form is a named ACL:

```
acl is_static path -i -m beg /static
```

We begin with the acl keyword, followed by a name, followed by the condition. Here we have an ACL named *is_static*. This ACL name can then be used with `if` and `unless` statements such as `use_backend be_static if is_static`. This form is recommended when you are going to use a given condition for multiple actions.

```
acl is_static path -i -m beg /static
use_backend be_static if is_static
```

The condition, `path -i -m beg /static`, checks to see if the URL starts with **/static**. You'll see how that works along with other types of conditions later in this chapter.

The second form is an anonymous or in-line ACL:

```
use_backend be_static if { path -i -m beg /static
}
```

This does the same thing that the above two lines would do, just in one line. For in-line ACLs, the condition is contained inside curly braces.

In both cases, you can chain multiple conditions together. ACLs listed one after another without anything in between will be considered to be joined with an and. The condition overall is only true if both ACLs are true. *(Note: ↪ means continue on same line)*

```
http-request deny if { path -i -m beg /api }
  ↪ { src 10.0.0.0/16 }
```

This will prevent any client in the **10.0.0.0/16** subnet from accessing anything starting with **/api**, while still being able to access other paths.

Adding an exclamation mark inverts a condition:

```
http-request deny if { path -i -m beg /api }
  ↪ !{ src 10.0.0.0/16 }
```

Now only clients in the **10.0.0.0/16** subnet are allowed to access paths starting with **/api** while all others will be forbidden.

The IP addresses could also be imported from a file:

```
http-request deny if { path -i -m beg /api }
  ↪ { src -f /etc/hapee-1.9/blacklist.acl }
```

Within **blacklist.acl** you would then list individual or a range of IP addresses using CIDR notation to block, as follows:

```
192.168.122.3
192.168.122.0/24
```

You can also define an ACL where either condition can be true by using ||:

```
http-request deny if { path -i -m beg /evil } ||
  ↪ { path -i -m end /evil }
```

With this, each request whose path starts with **/evil** (e.g. **/evil/foo**) or ends with **/evil** (e.g. **/foo/evil**) will be denied.

You can also do the same to combine named ACLs:

```
acl starts_evil path -i -m beg /evil
acl ends_evil path -i -m end /evil
http-request deny if starts_evil || ends_evil
```

With named ACLs, specifying the same ACL name multiple times will cause a logical OR of the conditions, so the last block can also be expressed as:

```
acl evil path_beg /evil
acl evil path_end /evil
http-request deny if evil
```

This allows you to combine ANDs and ORs (as well as named and in-line ACLs) to build more complicated conditions, for example:

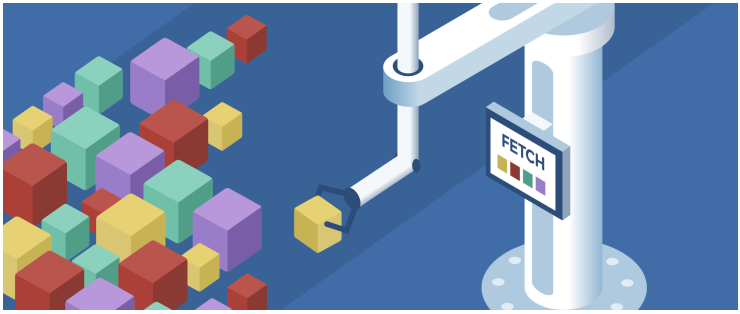
```
http-request deny if evil !{ src 10.0.0.0/16 }
```

This will block the request if the path starts or ends with **/evil**, but only for clients that are not in the **10.0.0.0/16** subnet.

Did you know? Innovations such as Elastic Binary Trees or EB trees have shaped ACLs into the high performing feature they are today. For example, string and IP address matches rely on EB trees that allow ACLs to process millions of entries while maintaining the best in class performance and efficiency that HAProxy is known for.

From what we've seen so far, each ACL condition is broken into two parts—the source of the information (or a fetch), such as `path` and `src`, and the string it is matching against. In the middle of these two parts, one can specify flags (such as `-i` for a case-insensitive match) and a matching method (`beg` to match on the beginning of a string, for example). All of these components of an ACL will be expanded on in the following sections.

Fetches



Now that you understand the basic way to format an ACL you might want to learn what sources of information you can use to make decisions on. A source of information in HAProxy is known as a **fetch**. These allow ACLs to get a piece of information to work with.

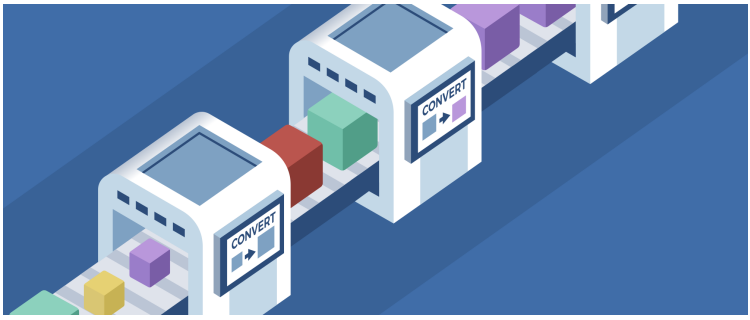
You can see the full list of fetches available in the documentation. The documentation is quite extensive and that is one of the benefits of having HAProxy Enterprise Support. It saves you time from needing to read through hundreds of pages of documentation.

Here are some of the more commonly used fetches:

src	Returns the client IP address that made the request
path	Returns the path the client requested

url_param(foo)	Returns the value of a given URL parameter
req.hdr(foo)	Returns the value of a given HTTP request header (e.g. User-Agent or Host)
ssl_fc	A boolean that returns true if the connection was made over SSL and HAProxy is locally deciphering it

Converters



Once you have a piece of information via a fetch, you might want to transform it. Converters are separated by commas from fetches, or other converters if you have more than one, and can be chained together multiple times.

Some converters (such as `lower` and `upper`) are specified by themselves while others have arguments passed to them. If an argument is required it is specified in parentheses. For example, to get the value of the path with **/static** removed from the start of it, you can use the

`regsub` converter with a regex and replacement as arguments:

```
path,regsub(^/static,/)
```

As with fetches, there are a wide variety of converters, but below are some of the more popular ones:

lower	Changes the case of a sample to lowercase
upper	Changes the case of a sample to uppercase
base64	Base64 encodes the specified string (good for matching binary samples)
field	Allows you to extract a field similar to awk. For example if you have “a|b|c” as a sample and run field(|,3) on it you will be left with “c”
bytes	Extracts some bytes from an input binary sample given an offset and length as arguments
map	Looks up the sample in the specified map file and outputs the resulting value

Flags

You can put multiple flags in a single ACL, for example:

```
path -i -m beg -f /etc/hapee/paths_secret.acl
```

This will perform a case insensitive match based on the beginning of the path and matching against patterns stored in the specified file. There aren't as many flags as there are fetch/converter types, but there is a nice variety.

Here are some of the commonly used ones:

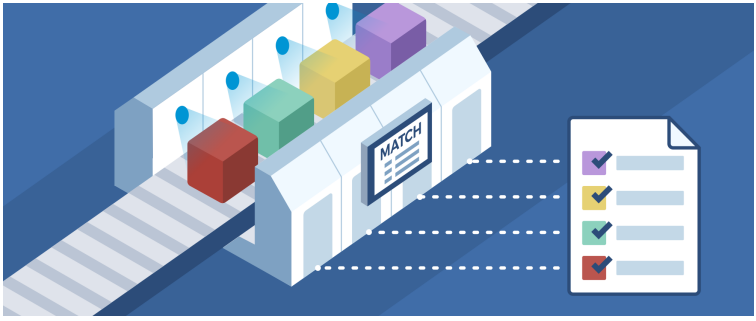
-i Perform a case-insensitive match (so a sample of FoO will match a pattern of Foo)

-f Instead of matching on a string, match from an ACL file. This ACL file can have lists of IP's, strings, regexes, etc. As long as the list doesn't contain regexes, then the file will be loaded into the b-tree format and can handle lookups of millions of items almost instantly

-m Specify the match type. This is described in detail in the next section.

You'll find a handful of others if you scroll down from the *ACL Basics* section of the documentation.

Matching Methods



Now you have a sample from converters and fetches, such as the requested URL path via `path`, and something to match against via the hardcoded path **/evil**. To compare the former to the latter you can use one of several matching methods. As before, there are a lot of matching methods and you can see the full list by scrolling down (further than the flags) in the *ACL Basics* section of the documentation. Here are some commonly used matching methods:

str	Perform an exact string match
beg	Check the beginning of the string with the pattern, so a sample of "foobar" will match a pattern of "foo" but not "bar".
end	Check the end of a string with the pattern, so a sample of foobar will match a pattern of bar but not foo.
sub	A substring match, so a sample of foobar will match patterns foo, bar, oba.
reg	The pattern is compared as a regular expression against the sample. Warning: This is CPU hungry compared to the other matching methods and should be avoided unless there is no other choice.
found	This is a match that doesn't take a pattern at all. The match is true if the sample is found, false otherwise. This can be used to (as a few common examples) see if a header (`req.hdr(x-foo) -m found`) is present, if a cookie is set (`cook(foo) -m found`), or if a sample is present in a map

(`src,map(/etc/hapee-1.9/ip_to_country.map) -m found`).

len	Return the length of the sample (so a sample of foo with `-m len 3` will match)

Up until this point, you may have noticed the use of `path -m beg /evil` for comparing our expected path **/evil** with the beginning of the sample we're checking. It uses the matching method `beg`. There are a number of places where you can use a shorthand that combines a sample fetch and a matching method in one argument. In this example `path_beg /foo` and `path -m beg /foo` are exactly the same, but the former is easier to type and read. Not all fetches have variants with built-in matching methods (in fact, most don't), and there's a restriction that if you chain a fetch with a converter you have to specify it using a flag (unless the last converter on the chain has a match variant, which most don't).

If there isn't a fetch variant of the desired matching method, or if you are using converters, you can use the `-m` flag noted in the previous section to specify the matching method.

Things to do with ACLs

Now that you know how to define ACLs, let's get a quick idea for the common actions in HAProxy that can be controlled by ACLs. This isn't meant to give you a complete list of all the conditions or ways that these rules can be

used, but rather provide fuel to your imagination for when you encounter something with which ACLs can help.

Redirecting a Request

The command `http-request redirect location` sets the entire URI. For example, to redirect non-www domains to their www variant you can use:

```
http-request redirect location
  ↪ http://www.%[hdr(host)]%[capture.req.uri]
  ↪ unless { hdr_beg(host) -i www }
```

In this case, our ACL, `hdr_beg(host) -i www`, ensures that the client is redirected unless their Host HTTP header already begins with www.

The command `http-request redirect scheme` changes the scheme of the request while leaving the rest alone. This allows for trivial HTTP-to-HTTPS redirect lines:

```
http-request redirect scheme https if !{ ssl_fc }
```

Here, our ACL `!{ ssl_fc }` checks whether the request did not come in over HTTPS.

The command `http-request redirect prefix` allows you to specify a prefix to redirect the request to. For example, the following line causes all requests that don't have a URL path beginning with **/foo** to be redirected to **/foo/{original URI here}**:

```
http-request redirect prefix /foo if
  ↪ !{ path_beg /foo }
```

For each of these a code argument can be added to specify a response code. If not specified it defaults to 302. Supported response codes are 301, 302, 303, 307, and 308. For example:

```
redirect scheme code 301 https if !{ ssl_fc }
```

This will redirect HTTP requests to HTTPS and tell clients that they shouldn't keep trying HTTP. Or for a more secure version of this, you could inject the Strict-Transport-Security header via `http-response set-header`.

Selecting a Backend

In HTTP Mode

The `use_backend` line allows you to specify conditions for using another backend. For example, to send traffic requesting the HAProxy Stats webpage to a dedicated backend, you can combine `use_backend` with an ACL that checks whether the URL path begins with **/stats**:

```
use_backend be_stats if { path_beg /stats }
```

Even more interesting, the backend name can be dynamic with `log-format` style rules (i.e. %[<fetch_method>]). In the following example, we put the path through a map and use that to generate the backend name:

```
use_backend
  ↪ be_%[path,map_beg(/etc/hapee-1.9/paths.map)]
```

If the file **paths.map** contains */api api* as a key-value pair, then traffic will be sent to *be_api*, combining the prefix *be_* with the string *api*. If none of the map entries match and you've specified the optional second parameter to the `map` function, which is the *default* argument, then that default will be used.

```
use_backend
  ↪ be_%[path,map_beg(/etc/hapee-1.9/paths.map,
  ↪ mydefault)]
```

In this case, if there isn't a match in the map file, then the backend *be_mydefault* will be used. Otherwise, without a default, traffic will automatically fall-through this rule in search of another `use_backend` rule that matches or the `default_backend` line.

In TCP Mode

We can also make routing decisions for TCP mode traffic, for example directing traffic to a special backend if the traffic is SSL:

```
tcp-request inspect-delay 10s
use_backend be_ssl if { req.ssl_hello_type gt 0 }
```

Note that for TCP-level routing decisions, when requiring data from the client such as needing to inspect the request, the `inspect-delay` statement is required to avoid HAProxy passing the phase by without any data from the client yet. It won't wait the full 10 seconds unless the client stays silent for 10 seconds. It will move ahead as soon as it can decide whether the buffer has an SSL hello message.

Setting an HTTP Header

There are a variety of options for adding an HTTP header to the request (transparently to the client). Combining these with an ACL lets you only set the header if a given condition is true.

add-header	Adds a new header. If a header of the same name was sent by the client this will ignore it, adding a second header of the same name.
set-header	Will add a new header in the same way as `add-header`, but if the request already has a header of the same name it will be overwritten. Good for security-sensitive flags that a client might want to tamper with.
replace-header	Applies a regex replacement of the named header (injecting a fake cookie into a cookie header, for example)

del-header	Deletes any header by the specified name from the request. Useful for removing an x-forwarded-for header before option forwardfor adds a new one (or any custom header name used there).

Changing the URL

This allows HAProxy to modify the path that the client requested, but transparently to the client. Its value accepts `log-format` style rules (i.e. *%[<fetch_method>]*) so you can make the requested path dynamic. For example, if you wanted to add **/foo/** to all requests (as in the redirect example above) without notifying the client of this, use:

```
http-request set-path /foo%[path] if
  ↪ !{ path_beg /foo }
```

There is also `set-query`, which changes the query string instead of the path, and `set-uri`, which sets the path and query string together.

Updating Map Files

These actions aren't used very frequently, but open up interesting possibilities in dynamically adjusting HAProxy maps. This can be used for tasks such as having a login server tell HAProxy to send a clients' (in this case by session cookie) requests to another backend from then on:

```
http-request set-var(txn.session_id)
  ↪ cook(sessionid)

use_backend
  ↪ be_%[var(txn.session_id),
  ↪ map(/etc/hapee-1.9/sessionid.map)]
  ↪ if { var(txn.session_id),
  ↪ map(/etc/hapee-1.9/sessionid.map) -m found }

http-response
  ↪ set-map(/etc/hapee-1.9/sessionid.map)
  ↪ %[var(txn.session_id)]
  ↪ %[res.hdr(x-new-backend)]
  ↪ if { res.hdr(x-new-backend) -m found }

default_backend be_login
```

Now if a backend sets the *x-new-backend* header in a response, HAProxy will send subsequent requests with the client's sessionid cookie to the specified backend. Variables are used as, otherwise, the request cookies are inaccessible by HAProxy during the response phase—a solution you may want to keep in mind for other similar problems that HAProxy will warn about during startup.

There is also the related `del-map` to delete a map entry based on an ACL condition.

Did you know? As with most actions, http-response set-map has a related action called http-request set-map. This is useful as a pseudo API to allow backends to add and remove map entries.

Caching

New to HAProxy 1.8 is small object caching, allowing the caching of resources based on ACLs. This, along with `http-response cache-store`, allows you to store select requests in HAProxy's cache system. For example, given that we've defined a cache named *icons*, the following will store responses from paths beginning with **/icons** and reuse them in future requests:

```
http-request set-var(txn.path) path
acl is_icons_path var(txn.path) -m beg /icons/
http-request cache-use icons if is_icons_path
http-response cache-store icons if is_icons_path
```

Using ACLs to Block Requests

Now that you've familiarized yourself with ACLs, it's time to do some request blocking!

The command `http-request deny` returns a 403 to the client and immediately stops processing the request. This

is frequently used for DDoS/Bot management as HAProxy can deny a very large volume of requests without bothering the web server.

Other responses similar to this include `http-request tarpit` (keep the request hanging until `timeout tarpit` expires, then return a 500—good for slowing down bots by overloading their connection tables, if there aren't too many of them), `http-request silent-drop` (have HAProxy stop processing the request but tell the kernel to not notify the client of this – leaves the connection from a client perspective open, but closed from the HAProxy perspective; be aware of stateful firewalls).

With both deny and tarpit you can add the `deny_status` flag to set a custom response code instead of the default 403/500 that they use out of the box. For example, using `http-request deny deny_status 429` will cause HAProxy to respond to the client with the error 429: Too Many Requests.

In the following subsections we will provide a number of static conditions for which blocking traffic can be useful.

HTTP Protocol Version

A number of attacks use HTTP 1.0 as the protocol version, so if that is the case it's easy to block these attacks using the built-in ACL `HTTP_1.0`:

```
http-request deny if HTTP_1.0
```

Contents of the user-agent String

We can also inspect the User-Agent header and deny if it matches a specified string.

```
http-request deny if { req.hdr(user-agent)
  ↪ -m sub evil }
```

This line will deny the request if the `-m sub` part of the user-agent request header contains the string *evil* anywhere in it. Remove the `-m sub`, leaving you with `req.hdr(user-agent) evil` as the condition, and it will be an exact match instead of a substring.

Length of the user-agent String

Some attackers will attempt to bypass normal user agent strings by using a random md5sum, which can be identified by length and immediately blocked:

```
http-request deny if { req.hdr(user-agent) -m
  ↪ len 32 }
```

Attackers can vary more with their attacks, so you can rely on the fact that legitimate user agents are longer while also being set to a minimum length:

```
http-request deny if { req.hdr(user-agent) -m
  ↪ len le 32 }
```

This will then block any requests which have a user-agent header shorter than 32 characters.

Path

If an attacker is abusing a specific URL that legitimate clients don't, one can block based on path:

```
http-request deny if { path /api/wastetime }
```

Or you can prevent an attacker from accessing hidden files or folders:

```
http-request deny if { path -m sub /. }
```

Updating ACL Lists

Using lb-update

ACL files are updated when HAProxy is reloaded to read the new configuration, but it is also possible to update its contents during runtime.
HAProxy Enterprise ships with a native module called **lb-update** that can be used with the following configuration:

```
dynamic update
    update id /etc/hapee-1.9/whitelist.acl
      ↪ url http://192.168.122.1/whitelist.acl
      ↪ delay 60s
```

HAPEE will now update the ACL contents every 60 seconds by requesting the specified URL. Support also exists for retrieving the URL via HTTPS and using client certificate authentication.

Using the Runtime API

To update the configuration during runtime, simply use the Runtime API to issue commands such as the following:

```
echo    "add    acl    /etc/hapee-1.9/whitelist.acl
1.2.3.4"
  ↪ | socat stdio /var/run/hapee-lb.sock
```

Conclusion

That's all folks! We have provided you with some examples to show the power within the HAProxy ACL system. The above list isn't exhaustive or anywhere near complete, but it should give you the building blocks needed to solve a vast array of problems you may encounter quickly and easily. Use your imagination and experiment with ACLs.

Introduction to HAProxy Stick Tables

HTTP requests are stateless by design. However, this raises some questions regarding how to track user activities, including malicious ones, across requests so that you can collect metrics, block users, and make other decisions based on state. The only way to track user activities between one request and the next is to add a mechanism for storing events and categorizing them by client IP or other key.

Out of the box, HAProxy Enterprise and HAProxy give you a fast, in-memory storage called **stick tables**. Released in 2010, stick tables were created to solve the problem of server persistence. However, StackExchange, the network of Q&A communities that includes Stack Overflow, saw the potential to use them for rate limiting of abusive clients, aid in bot management, and tracking data transferred on a per client basis. They sponsored further development of stick tables to expand the functionality. Today, stick tables are an incredibly powerful subsystem within HAProxy.

The name, no doubt, reminds you of sticky sessions used for sticking a client to a particular server. They do that, but also a lot more. Stick tables are a type of key-value storage where the key is what you track across requests, such as a client IP, and the values consist of counters that, for the most part, HAProxy takes care of calculating for you. They are commonly used to store information like how many

requests a given IP has made within the past 10 seconds. However, they can be used to answer a number of questions, such as:

- How many API requests has this API key been used for during the last 24 hours?
- What TLS versions are your clients using? (e.g. can you disable TLS 1.1 yet?)
- If your website has an embedded search field, what are the top search terms people are using?
- How many pages is a client accessing during a time period? Is it enough to signal abuse?

Stick tables rely heavily on HAProxy's access control lists, or ACLs. When combined with the Stick Table Aggregator that's offered within HAProxy Enterprise, stick tables bring real-time, cluster-wide tracking. Stick tables are an area where HAProxy's design, including the use of Elastic Binary Trees and other optimizations, really pays off.

Uses of Stick Tables

There are endless uses for stick tables, but here we'll highlight three areas: server persistence, bot detection, and collecting metrics.

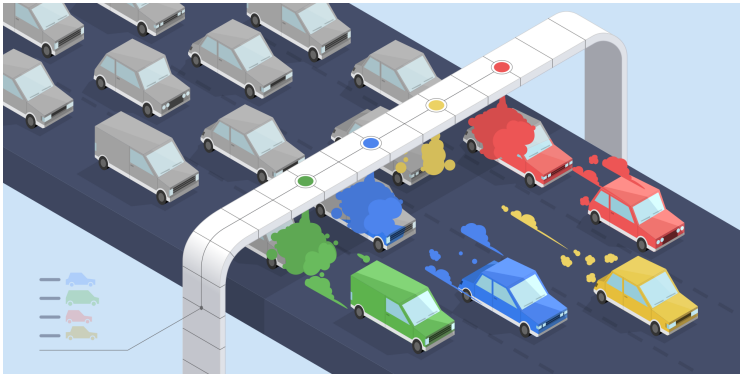
Server persistence, also known as sticky sessions, is probably one of the first uses that comes to mind when you hear the term "stick tables". For some applications, cookie-based or consistent hashing-based persistence methods aren't a good fit for one reason or another. With stick tables, you can have HAProxy store a piece of

information, such as an IP address, cookie, or range of bytes in the request body (a username or session id in a non-HTTP protocol, for example), and associate it with a server. Then, when HAProxy sees new connections using that same piece of information, it will forward the request to the same server. This is really useful if you're storing application sessions in memory on your servers.

Beyond the traditional use case of server persistence, you can also use stick tables for defending against certain types of bot threats. Request floods, login brute force attacks, vulnerability scanners, web scrapers, slow loris attacks—stick tables can deal with them all.

A third area we'll touch on is using stick tables for collecting metrics. Sometimes, you want to get an idea of what is going on in HAProxy, but without enabling logging and having to parse the logs to get the information in question. Here's where the power of the Runtime API comes into play. Using the API, you can read and analyze stick table data from the command line, a custom script or executable program. This opens the door to visualizing the data in your dashboard of choice. If you prefer a packaged solution, HAProxy Enterprise comes with a fully-loaded dashboard for visualizing stick table data.

Defining a Stick Table



A stick table collects and stores data about requests that are flowing through your HAProxy load balancer. Think of it like a machine that color codes cars as they enter a race track. The first step then is setting up the amount of storage a stick table should be allowed to use, how long data should be kept, and what data you want to observe. This is done via the `stick-table` directive in a `frontend` or `backend`.

Here is a simple stick table definition:

```
backend webfarm
    stick-table type ip size 1m expire 10s
      ↪ store http_req_rate(10s)
```

In this line we specify a few arguments: `type`, `size`, `expire` and `store`. The type, which is *ip* in this case, decides the classification of the data we'll be capturing. The size configures the number of entries it can store—in this case one million. The expire time, which is the time since a record in the table was last matched, created or refreshed, informs HAProxy when to remove data. The store argument declares the values that you'll be saving.

Did you know? If just storing rates, then the expire argument should match the longest rate period; that way the counters will be reset to 0 at the same time that the period ends.

Each `frontend` or `backend` section can only have one `stick-table` defined in it. The downside to that is if you want to share that storage with other frontends and backends. The good news is that you can define a frontend or backend whose sole purpose is holding a stick table. Then you can use that stick table elsewhere using the `table` parameter. Here's an example (we'll explain the `http-request track-sc0` line in the next section):

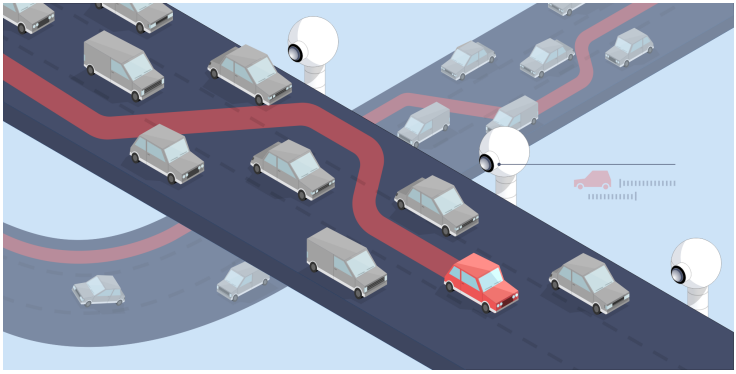
```
backend st_src_global
    stick-table type ip size 1m expire 10s
      ↪ store http_req_rate(10s)

frontend fe_main
    bind *:80
    http-request track-sc0 src table st_src_global
```

Two other stick table arguments that you'll want to know about are `nopurge` and `peers`. The former tells HAProxy to not remove entries if the table is full and the latter specifies a peers section for syncing to other nodes. We'll cover that interesting scenario a little later.

When adding a stick table and setting its size it's important to keep in mind how much memory the server has to spare after taking into account other running processes. Each stick table entry takes about 50 bytes of memory for its own housekeeping. Then the size of the key and the counters it's storing add up to the total.

Keep in mind a scenario where you're using stick tables to set up a DDoS defense system. An excellent use case, but what happens when the attacker brings enough IPs to the game? Will it cause enough entries to be added so that all of the memory on your server is consumed?
Memory for stick tables isn't used until it's needed, but even so, you should keep in mind the size that it could grow to and set a cap on the number of entries with the `size` argument.

Tracking Data



Now that you've defined a stick table, the next step is to track things in it. This is done by using `http-request track-sc0`, `tcp-request connection track-sc0`, or `tcp-request content track-sc0`. The first thing to consider is the use of a sticky counter, sc0. This is used to assign a slot with which to track the connections or requests. The maximum number that you can replace 0 with is set by the build-time variable `MAX_SESS_STKCTR`. In HAProxy Enterprise, it's set to 12, allowing sc0 through sc11.

This can be a bit of a tricky concept, so here is an example to help explain the nuances of it:

```
backend st_src_global
    stick-table type ip size 1m expire 10m
      ↪ store http_req_rate(10m)

backend st_src_login
    stick-table type ip size 1m expire 10m
      ↪ store http_req_rate(10m)

backend st_src_api
    stick-table type ip size 1m expire 10m
      ↪ store http_req_rate(10m)

frontend fe_main
    bind *:80
    http-request track-sc0 src table st_src_global
    http-request track-sc1 src table st_src_login
      ↪ if { path_beg /login }
    http-request track-sc1 src table st_src_api
      ↪ if { path_beg /api }
```

In this example, the line `http-request track-sc0` doesn't have an `if` statement to filter out any paths, so *sc0* is tracking all traffic. Querying the *st_src_global* stick table with the Runtime API will show the HTTP request rate per client IP. Easy enough.

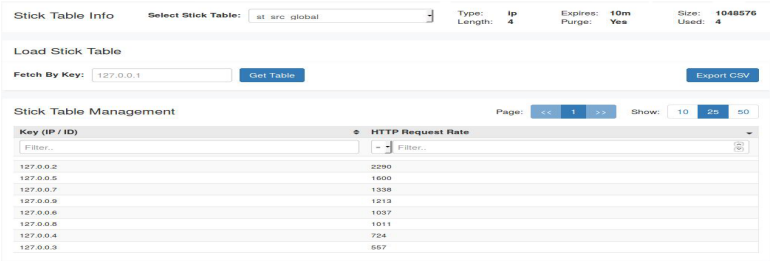
Sticky counter 1, *sc1*, is being used twice: once to track requests beginning with **/login** and again to track requests beginning with **/api**. This is okay because no request passing through this block is going to start with both **/login** and **/api**, so one sticky counter can be used for both tables.

Even though both tables are being tracked with *sc1* they are their own stick table definitions, and thus keep their data separate. So if you make a few requests and then query the tables via the Runtime API, you'll see results like the following:

```
$ echo "show table st_src_global"
  ↪ | socat stdio
  ↪ UNIX-CONNECT:/var/run/hapee-1.9/hapee-lb.sock

# table: st_src_global, type: ip, size:1048576,
used:1 0x18f907c: key=127.0.0.1 use=0 exp=3583771
http_req_rate(86400000)=3
```

You can see three total requests in the *st_src_global* table, two requests in the *st_src_api* table, and one in the *st_src_login* table. Even though the last two used the same sticky counter, the data was segregated. If I had made a mistake and tracked both *st_src_global* and *st_src_login* using *sc0*, then I'd find that the st_src_login table was empty because when HAProxy went to track it, *sc0* was already used for this connection.

In addition, this data can be viewed using HAProxy Enterprise's Real-Time Dashboard.



Using the dashboard can be quicker than working from the command-line and gives you options for filtering and sorting.

Types of Keys

A stick table tracks counters for a particular key, such as a client IP address. The key must be in an expected type, which is set with the `type` argument. Each `type` is useful for different things, so let's take a look at them:

Type	Size (b)	Description
ip	50	This will store an IPv4 address. It's primarily useful for tracking activities of the IP making the request and can be provided by HAProxy with the fetch method src. However, it can also be fed a sample such as `req.hdr(x-forwarded-for)` to get the IP from another proxy.
ipv6	60	This will store an IPv6 address or an IPv6 mapped IPv4 address. It's the same as the ip type otherwise.

integer	32	This is often used to store a client ID number from a cookie, header, etc. It's also useful for storing things like the frontend ID via `fe_id` or `int(1)` to track everything under one entry (reasons for which we will cover in a later section)
string	len	This will store a string and is commonly used for session IDs, API keys and similar. It's also useful when creating a dummy header to store custom combinations of samples. It requires a len argument followed by the number of bytes that can be stored. Larger samples will be truncated.
binary	len	This is used for storing binary samples. It's most commonly used for persistence by extracting a client ID from a TCP stream with the bytes converter. It can also be used to store other samples such as the base32 (IP+URL) fetch. It requires a len argument followed by the number of bytes that can be stored. Longer samples will be truncated.

The `type` that you choose defines the keys within the table. For example, if you use a `type` of *ip* then we'll be capturing IP addresses as the keys.

Types of Values

After the `store` keyword comes a comma delimited list of the values that should be associated with a given key. While some types can be set using ACLs or via the Runtime API, most are calculated automatically by built-in fetches in

HAProxy like `http_req_rate`. There can be as many values stored as you would like on a given key.

There are many values that a stick table can track. For a full list of values, see the stick-table section of the documentation, but here are some interesting highlights:

http_req_rate

This is likely the most frequently stored/used value in stick tables. As its name may imply, it stores the number of HTTP requests, regardless of whether they were accepted or not, that the tracked key (e.g. source IP address) has made over the specified time period. Using this can be as simple as the following:

```
stick-table type ip size 1m expire 10s
  ↪ store http_req_rate(10s)
tcp-request inspect-delay 10s
tcp-request content track-sc0 src
http-request deny if { sc_http_req_rate(0) gt 10 }
```

The first line defines a stick table for tracking IP addresses and their HTTP request rates over the last ten seconds. This is done by storing the `http_req_rate` value, which accepts the period as a parameter. Note that we've set the `expire` parameter to match the period of 10 seconds.

The second line is what inserts or updates a key in the table and updates its counters. Using the sticky counter *sc0*, it sets the key to the source IP using the `src` fetch method. You might wonder when to use `tcp-request`

`content track-sc0` instead of `http-request track-sc0`. It's mostly a matter of preference, but since the TCP phase happens before the HTTP phase, you should try to order tcp-* directives before http-* ones or else you'll get warnings when HAProxy starts up. Also, if you want the ability to deny connections in the earlier TCP phase, lean towards using the tcp-request variant. However, if you're capturing HTTP headers, cookies or other data encapsulated within the HTTP message, then to use `tcp-request content track-s0`, you must use an `inspect-delay` directive. We'll talk about that a little later on.

Finally the third line denies the request with a 403 Forbidden if the client has made more than 10 requests over the expiration period of 10 seconds. Notice that when deciding whether to deny the request, we check the value of `http_req_rate` with the `sc_http_req_rate` function, passing it 0, the number corresponding to our sticky counter, sc0.

Values that return a rate, like `http_req_rate`, all take an argument that is the time range that they cover. The maximum time that can be tracked is about 30 days (e.g. *30d*). For longer periods of time consider using the counter `http_req_cnt` and extrapolate from there.

conn_cur and conn_rate

Two closely related counters, `conn_cur` and `conn_rate`, track how many connections a given key has or is making. The `conn_cur` counter is automatically incremented or decremented when the `tcp-request content track-sc0`

`src` line is processed to reflect the number of currently open connections for the key, or source IP. The `conn_rate` counter is similar but is given a period of time and calculates an average rate of new connections over that period.

```
stick-table type ip size 1m expire 10s
  ↪ store conn_cur
tcp-request content track-sc0 src
tcp-request content reject if { sc_conn_cur(0)
  ↪ gt 10 }
```

One way to use this is to detect when a client has opened too many connections so you can deny any more connections from them. In this case the connection will be rejected and the connection closed if the source IP currently has more than 10 connections open at the moment.

These counters are primarily used to protect against attacks that involve a lot of new connections that originate from the same IP address. In the next section, you'll see HTTP counters, which are more effective at protecting against HTTP request floods. The HTTP counters track requests independently of whether HTTP keep-alive or multiplexing are used.

However in the case of floods of new connections, these counters can stop them best.

http_err_rate

This tracks the rate of HTTP requests that end in an error code (4xx) response. This has a few useful applications:

- You can detect vulnerability scanners, which tend to get a lot of error pages like 404 Not Found
- You can detect missing pages by using a URL path as the stick table key. For example:

```
stick-table type string len 128 size 2k expire 1d
  ↪ store http_err_rate(1d)

tcp-request content track-sc0 path
```

This will make a table that can be retrieved by the Runtime API and shows the error rate of various paths:

```
# table: fe_main, type: string, size:2048, used:2
0xbc929c: key=/ use=0 exp=86387441
http_err_rate(86400000)=0

0xbc99ac: key=/foobar use=0 exp=86390564
http_err_rate(86400000)=1
```

- You can detect login brute force attacks or scanners. If your login page produces an HTTP error code when a login fails, then this can be used to detect brute force attacks. For this you would

track on `src` rather than on `path` as in the previous example.

bytes_out_rate

The `bytes_out_rate` counter measures the rate of traffic being sent from your server for a given key, such as a path. Its primary use is to identify content or users who are creating the largest amounts of traffic. However, it has other interesting uses as well. It can help measure traffic by site or path, which you can use in capacity planning or to see which resources might need to be moved to their own cluster (e.g. If you operate a CDN, this could be used to select heavily trafficked content to move to other caching nodes).

We might also use `bytes_out_rate` as another data set to feed into an anomaly detection system (e.g. a web script that never sends much traffic all of a sudden begins sending 3gb might indicate a successful exfiltration of data).

Similar to `bytes_out_rate`, `bytes_in_rate` observes how much traffic a client is sending, which could be used to detect anomalous behavior, to factor into billing on a VPN system where client traffic is to be counted in both directions, and that type of thing.

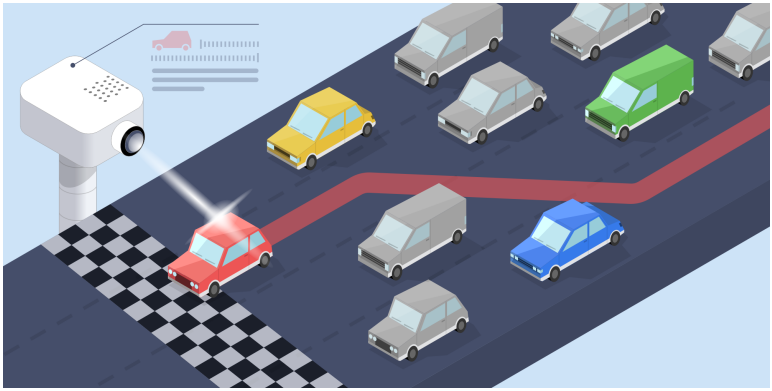
gpc0 / gpc1

The general purpose counters (gpc0 and gpc1) are special—along with gpt0 (general purpose tag)—for

defaulting to 0 when created and for not automatically updating. ACLs can be used to increase this counter via the `sc_inc_gpc0` fetch method so that you can track custom statistics with it.

If you track `gpc0_rate`, it will automatically give you a view of how quickly *gpc0* is being incremented. This can tell you how frequently this event is happening.

Making Decisions Based on Stick Tables



Now that you've seen how to create stick table storage and track data with it, you'll want to be able to configure HAProxy to take action based on that captured information. Going back to a common use case for stick tables, let's see how to use the data to persist a client to a particular server. This is done with the `stick on` directive

and is usually found in a `backend` section looking like the following:

```
stick-table type string len 32 size 100k expire
30m
stick on req.cook(sessionid)
```

In this example, notice that we don't use the `store` parameter on the `stick-table` directive. A server ID, which is an integer that HAProxy uses to identify each server, is unique in that you don't need to define it via a `store` keyword for it to be stored. If needed for a persistence setting, the server ID will be stored in the stick table in question automatically. After the `stick on` directive extracts the client's session ID from a cookie and stores it as the key in the table, the client will continue to be directed to the same server.

While on the topic of persistence, let us say we have a cluster of MySQL servers participating in master-master replication and we are worried that writing to one might cause a duplicate primary key if, at that moment, the primary master goes down and then comes back up. Normally this can make a rather complicated situation wherein both MySQL servers have some queries that the other doesn't and it requires a lot of fighting to get them back in sync. Suppose that instead we added the following to our MySQL backend?

```
backend mysql
    mode tcp
    stick-table type integer size 1 expire 1d
    stick on int(1)
    on-marked-down shutdown-sessions
    server primary 192.168.122.60:3306 check
    server backup 192.168.122.61:3306 check backup
```

With this configuration, we store only a single entry in the stick table, where the key is 1 and the value is the server_id of the active server. Now if the primary server goes down, the backup server's server_id will overwrite the value in the stick table and all requests will keep going to the backup even if the master comes back online. This can be undone by cycling the backup node into maintenance mode, or via the Runtime API, when you are ready to have the cluster resume normal operations.

Did you know? On-marked-down shutdown-sessions causes HAproxy to close all existing connections to a backend when it goes down. Normally HAProxy allows existing connections to finish which could result in duplicate primary keys if the connections kept working or query timeouts if it didn't.

Another way to use stick tables is for collecting information about traffic to your website so that you can make decisions based on it. Say that you wanted to know if it was safe to disable TLS 1.1. You could set up a stick table that tracks the TLS versions that people are using. Consider the following example:

```
backend st_ssl_stats
    stick-table type string len 32 size 200
      ↪ expire 24d store http_req_rate(24d)

frontend fe_main
    tcp-request inspect-delay 10s
    tcp-request content track-sc0
      ↪ ssl_fc_protocol table st_ssl_stats
```

Now you can query the server and see which TLS protocols have been used:

```
$ echo "show table st_ssl_stats" |
    ↪ socat stdio
                                                    ↪
UNIX-CONNECT:/var/run/hapee-1.9/hapee-lb.sock

# table: st_ssl_stats, type: string, size:200,
used:2
0xe4c62c: key=TLSv1 use=0 exp=2073596788
http_req_rate(2073600000)=1

0xe5a18c: key=TLSv1.2 use=0 exp=2073586582
http_req_rate(2073600000)=2
```

Or you could turn it around and track clients who have used TLSv1.1 by IP address:

```
backend st_ssl_stats
    stick-table type ip size 200 expire 1h
      ↪ store http_req_rate(1d)

frontend fe_main
    tcp-request inspect-delay 10s
    tcp-request content track-sc0 src
      ↪ table st_ssl_stats if
      ↪ { ssl_fc_protocol TLSv1.1 }
```

Now your stick table is a list of IPs that have used TLSv1.1. To learn more about the Runtime API, take a look at our blog post **Dynamic Configuration with the HAProxy Runtime API** (**bit.ly/2SpOPDX**).

If you look through the documentation you will see fetches specific to stick tables like `sc_http_req_rate` (one for each value you can store in a stick table) all starting with *sc_*. You will notice in the documentation that some of them have *sc0_*, *sc1_*, and *sc2_* aliases without arguments. These forms are deprecated as they don't let you access all of the sticky counters, but do the same thing. These fetches, in conjunction with ACLs, can be used to protect your website from malicious activity by returning the values in the stick table and giving you information needed to decide whether to deny the connection or request.

For example, to block requests that have made more than 100 requests over the time period defined on the stick table definition and by the key defined in the track line, you would use `sc_http_req_rate` in the following way:

```
http-request deny if { sc_http_req_rate(0) gt 100
}
```

If you aren't tracking the key that you want to look up, you can use the `table_http_req_rate` and similar fetches to retrieve a value without updating it. Using track-sc* will update `http_req_rate` and similar counters while looking up a value like this will not. These work like converters where they take the key as input, the table name as an argument, and output the value. For example we could do:

```
http-request deny if { src,
  ↪ table_http_req_rate(st_src_global) gt 100 }
```

These fetches are a small amount of extra work for the CPU if you are already tracking the key via an `http-request track-sc0` or `tcp-request content track-sc0` line elsewhere. However, there are a few good reasons for it:

1. You want to check if a request should be blocked without increasing the request counter by tracking it (so that a client can make 10 requests a second and everything above that gets blocked, rather than making 10 requests in a second and having future blocked requests keep them getting blocked until they cool down)

2. You want to pass another key; for example passing `req.arg(ip)` instead of `src` would allow an API of sorts where you could request

http://192.168.122.64/is_blocked?ip=192.168.122.1 and see if that IP is blocked (or what its request rate is).

3. You're using the Stick Table Aggregator and want to query data from the table that contains it creates (a new table is created that contains the aggregated data).

Other Considerations

inspect-delay

Let's talk about a line that is sometimes needed and ends up causing confusion:

```
tcp-request inspect-delay 10s
```

You only need to use this in a frontend or backend when you have an ACL on a statement that would be processed in an earlier phase than HAProxy would normally have the information. For example, `tcp-request content reject if { path_beg /foo }` needs a `tcp-request inspect-delay` because HAProxy won't wait in the TCP phase for the HTTP URL path data. In contrast `http-request deny if { path_beg /foo }` doesn't need an `tcp-request inspect-delay` line because HAProxy won't process `http-request` rules until it has an HTTP request.

When `tcp-request inspect-delay` is present, it will hold the request until the rules in that block have the data they need to make a decision or until the specified delay is reached, whichever is first.

nbproc

If you are using the `nbproc` directive in the `global` section of your configuration, then each HAProxy process has its own set of stick tables. The net effect is that you're not sharing stick table information among those processes. Also note that the peers protocol, discussed next, can't sync between processes on the same machine.

There are two ways to solve this. The first is to use the newer `nbthread` directive instead. This is a feature introduced in HAProxy Enterprise 1.8r1 and HAProxy 1.8 that enables multithreading instead of multiple processes and shares memory, thus sharing stick tables between threads running in a single process. See our blog post **Multithreading in HAProxy** (**http://bit.ly/2WGp160**) to learn more about it.

Another solution is to use a configuration like the following:

```
listen fe_main
    bind *:443 ssl crt /path/to/cert.pem
    bind *:80
    server local
      ↪ unix:/var/run/hapee-1.9/ssl_handoff.sock
      ↪ send-proxy-v2

frontend fe_secondary
    bind unix:/var/run/hapee-1.9/ssl_handoff.sock
      ↪ accept-proxy process 1
              #    Stick    tables,    use_backend,
default_backend...
```

The first proxy terminates TLS and passes traffic to a single server listed as `server local unix:/var/run/hapee-1.9/ssl_handoff.sock send-proxy-v2`. Then you add another frontend with `bind unix:/var/run/hapee-1.9/ssl_handoff.sock accept-proxy process 1` in it. Inside this frontend you can have all of your stick table and statistics collection without issue. Since TLS termination usually takes most of the CPU time, it's highly unusual to need more than one process for the backend work.

peers

Now that we've covered how to use stick tables, something to consider is setups that utilize HAProxy in active-active clusters, where a new connection from a client may end up at one of multiple HAProxy servers, such as by Route Health Injection or Amazon Elastic Load Balancer. One server has all of the stick table entries, but the other node

has its own set of stick table definitions. To solve that problem you can add a `peers` section to the top of your configuration:

```
peers mypeers
    peer centos7vert 192.168.122.64:10000
    peer shorepoint 192.168.122.1:10000
```

Then change your stick table definition to include a `peers` argument:

```
stick-table type string len 32 size 100k expire
30m
  ↪ peers mypeers
```

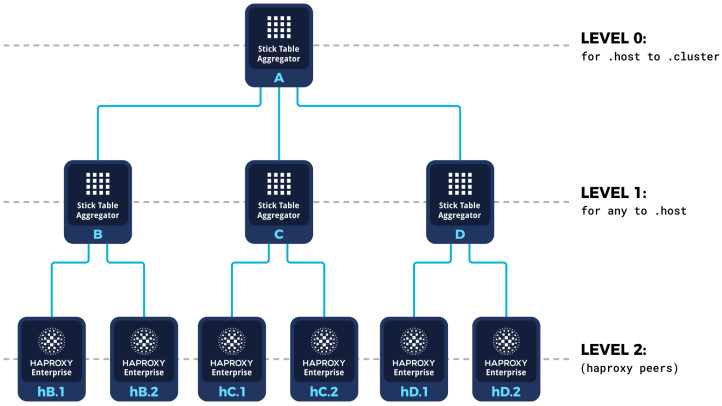
At least one of the peers needs to have a name that matches the server's host name or you must include a `setenv` *hostname* line in the `global` section of the configuration to inform HAProxy what it should see the host name as.

Now the two servers will exchange stick table entries; but there is a downside: they won't sum their individual counters, so `http_req_rate` on one will overwrite the value on the other, rather than both seeing a sum of the two.

Enter the Stick Table Aggregator. This is a feature of HAProxy Enterprise that watches for values coming in over the peers protocol, adds the values together, then returns the combined result back to each of the HAProxy

instances. The benefit of this is the ability to associate events that you wouldn't be able to otherwise, since the data resides on two or more different nodes.

For example, in an active-active cluster of HAProxy load balancers, an attacker will be hitting both instances. If you aren't combining the data, you're only seeing half of the picture. Getting an accurate representation of the state of your nodes is important to detecting and stopping attacks. Here's a representation of how the aggregator allows peers to exchange information:



Check out our webinar **DDoS Attack and Bot Protection with HAProxy Enterprise** (**bit.ly/2TzWo8I**) for a full example of using the Stick Table Aggregator.

Conclusion

In this chapter, you learned about HAProxy's in-memory storage mechanism, stick tables, that allow you to track client activities across requests, enable server persistence, and collect real-time metrics.

Introduction to HAProxy Maps

Dictionaries. Hashes. Associative arrays. Can you imagine life without these wonderful key-value data structures? The mad, dystopian world it would be? They're just sort of there when needed: a trusty tool, never too far out of reach.

So maybe you're not entirely shocked that they're counted among the HAProxy load balancer's extensive feature set. They're called **maps** and are built-in and ready to roll. What you're probably not expecting are the imaginative tasks that you can tackle with them. Want to set up blue-green deployments? Maybe you'd like to set up rate limits by URL path? How about just dynamically switching which backend servers are used for a domain? It's all done with maps!

In this chapter, you'll learn how to create a map file, store it on your system, reference it in your HAProxy configuration, and update it in real time. You'll also see some useful scenarios in which to put your new knowledge to good use.

The Map File

Before considering the fascinating things you can do with a map file, let's wrap our minds around what a map file is. Everything starts by creating a map file. Fire up your favorite text editor and create a file named **hosts.map**. Then add the following lines to it:

```
# A comment begins with a hash sign and must
# be on a new line
static.example.com be_static
www.example.com    be_static

# Empty lines are okay
example.com        be_static
api.example.com    be_api
```

A few things to note about the structure of this file:

- It's plain text
- A key begins each line (e.g. *static.example.com*)
- A value comes after a key, separated by at least one space (e.g. *be_static*)
- Empty lines and extra whitespace between words are ignored
- Comments must begin with a hash sign and must be on their own line

A map file stores key-value pairs. HAProxy uses them as a lookup table, such as to find out which backend to route a client to based on the value of the Host header. The benefit of storing this association in a file rather than in the HAProxy configuration itself is the ability to change those values dynamically.

Next, transfer this file to a server where you have an instance of HAProxy that you don't mind experimenting with and place it into a directory of your choice. In these examples, since we're using HAProxy Enterprise, we'll store it under **/etc/hapee-1.9/maps**. Map files are loaded by HAProxy when it starts, although, as you'll see, they can be modified during runtime without a reload.

Did you know? Map files are loaded into an Elastic Binary Tree format so you can look up a value from a map file containing millions of items without a noticeable performance impact.

Map Converters

To give you an idea of what you can do with map files, let's look at using one to find the correct backend pool of servers where users should be sent. You will use the **hosts.map** file that you created previously to look up which backend should be used based on a given domain name. Begin by editing your **haproxy.cfg** file. As you will see, you will add a map converter that reads the map file and returns a `backend` name.

A **converter** is a directive placed into your HAProxy configuration that takes in an input and returns an associated output. There are various types of converters. For example, you might use the `lower` converter to change a given string to lowercase or `url_dec` to URL decode it. In the following example, the input is a string literal *example.com* and the `map` converter looks up that key in the map file, **hosts.map**.

```
frontend fe_main
    bind :80
    use_backend %[str(example.com),
      ↪ map(/etc/hapee-1.9/maps/hosts.map)]
```

The first row in **hosts.map** that has *example.com* as a key will have its value returned. Notice how the input, `str(example.com)`, begins the expression and is separated from the converter with a comma.

When this expression is evaluated at runtime, it will be converted to the line `use_backend be_static`, which directs requests to the *be_static* pool of servers. Of course, rather than passing in a hardcoded string like *example.com*, you can send in the value of an HTTP header or a URL parameter. The next example uses the value of the Host header as the input.

```
use_backend
    %[req.hdr(host),lower,
      ↪ map(/etc/hapee-1.9/maps/hosts.map,
      ↪ be_static)]
```

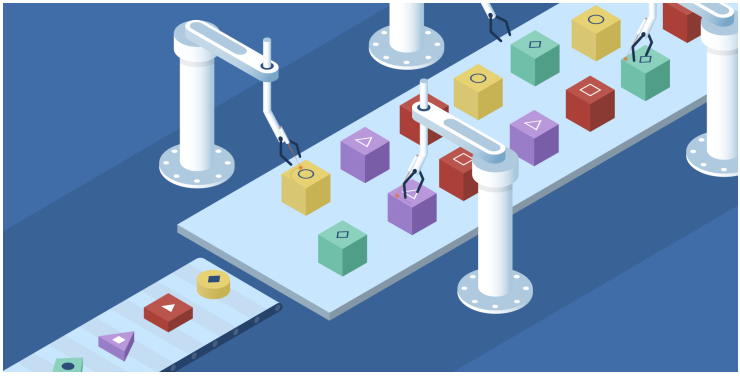
The `map` converter takes up to two arguments. The first is the path to your map file. The second, optional argument declares a default value that will be used if no matching key is found. So, in this case, if there's no match, *be_static* will be used. If the input matches multiple items in the map file, HAProxy will return the first one.

The `map` converter looks for an exact match in the file, but there are a few variants that provide opportunities for a partial match. The most commonly used are summarized here:

map_beg	Looks for entries in the map file that match the beginning of the input (e.g. an input of "abcd" would match "a" in the file).
map_end	Looks for entries in the map file that match the end of the input (e.g. an input of "abcd" would match "d" in the file). Unlike the other match modes, this doesn't perform ebtree lookups and instead checks each line.
map_sub	Looks for entries in the map file that make up a substring of the sample (e.g. an input of "abcd" would match "ab" or "c" in the file). Unlike the other match modes, this doesn't perform ebtree lookups and instead checks each line.

map_ip	This takes the input as an IP address and looks it up in the map. If the map has masks (such as **192.168.0.0/16**) in it then any IP in the range will match it.This is the default if the input type is an IP address.
map_reg	This reads the samples in the map as regular expressions and will match if the regular expression matches. Unlike the other match modes, this doesn't perform ebtree lookups and instead checks each line.
map_str	An alias for map. This takes a string as an input, matches on the whole key, and returns a value as a string.

Modifying the Values



Much of the value of map files comes from your ability to modify them dynamically. This allows you to, for example, change the flow of traffic from one backend to another, such as for maintenance.

There are four ways to change the value that we get back from a map file. First, you can change the values by editing the file directly. This is a simple way to accomplish the task, but does require a reload of HAProxy. This is a good choice if you're using a configuration management tool like Puppet or Ansible.

The second way is provided with HAProxy Enterprise via the **lb-update** module, which you'll really appreciate if you're running a cluster of load balancers. It allows you to update maps within multiple instances of HAProxy at once by watching a map file hosted at a URL at a defined interval.

A third way to edit the file's contents is by using the Runtime API. The API provides all of the necessary CRUD operations for creating, removing, updating, and deleting rows from the map in memory, without needing to reload HAProxy. There's also a simple technique for saving your changes to disk, which you'll see later in this chapter.

A fourth way is with the `http-request set-map` directive in your HAProxy configuration file. This gives you the opportunity to update map entries based on URL parameters in the request. It's easy to turn this into a convenient HTTP-based interface for making map file changes from a remote client.

In the next few sections, you'll get some guidance on how to use these techniques.

Editing the File Directly

A straightforward way to change the values you get back from your map file is to change the file itself. Open the file and make any modifications you need: adding rows, removing others, changing the values of existing rows. However, know that HAProxy only reads the file when it's starting up and then loads it into memory. Refreshing the file, then, means reloading HAProxy.

Thanks to hitless reloads introduced in HAProxy Enterprise 1.8r1 and HAProxy 1.8, you can trigger a reload without dropping active connections. Read our blog post **Hitless Reloads with HAProxy – HOWTO** (**http://bit.ly/2Uz0PAu**) for an explanation on how to use this feature.

This approach will work with configuration management tools like Puppet, which allow you to distribute changes to your servers at a set interval. Be sure to reload HAProxy to pick up the changes.

Editing With the lb-update Module

Although configuration management tools allow you to update the servers in your cluster, they can be a heavy solution that requires administration. An alternative is using the lb-update module to keep each replica of HAProxy within your cluster in sync. The **lb-update module** instructs HAProxy Enterprise to retrieve the contents of the map file from a URL at a defined interval. The module will automatically check for updates as frequently as configured. This is especially useful when

there are a lot of processes and/or servers in a cluster that need the updated files.

Did you know? The lb-update module can also be used to synchronize TLS ticket keys.

Below is a sample of a `dynamic-update` section that manages updating the **hosts.map** file from a URL. You'd add an `update` directive for each map file that you want to watch.

```
dynamic-update update id
  ↪ /etc/hapee-1.9/maps/sample.map
  ↪ url http://10.0.0.1/sample.map delay 300s
```

See the HAProxy Enterprise documentation for detailed usage instructions or contact us to learn more.

Editing With the Runtime API

There are several API methods available for updating an existing map file. The table below summarizes them.

API method	Description
show map	Lists available map files or displays a map file's contents.
get map	Reports the keys and values matching a given input.

set map	Modifies a map entry.
add map	Adds a map entry.
del map	Deletes a map entry.
clear map	Deletes all entries from a map file.

Without any parameters, `show map` lists the map files that are loaded into memory. If you give it the path to a particular file, it will display its contents. In the following example, we use it to display the key-value pairs inside **hosts.map**.

```
root@server1:~$ echo "show map
  ↪ /etc/hapee-1.9/maps/hosts.map" |
  ↪ socat stdio /var/run/hapee-1.9/hapee-lb.sock

0x1605c10 static.example.com be_static
0x1605c50 www.example.com be_static
0x1605c90 example.com be_static
0x1605cd0 api.example.com be_api
```

The first column is the location of the entry and is typically ignored. The second column is the key to be matched and the third is the value. We can easily add and remove entries via the Runtime API. To remove an entry from the map file, use `del map`. Note that this only removes it from memory and not from the actual file.

```
root@server1:~$ echo "del map
   ↪ /etc/hapee-1.9/hosts.map  static.example.com"
|
  ↪ socat stdio /var/run/hapee-1.9/hapee-lb.sock
```

You can also delete all entries with `clear map`:

```
root@server1:~$ echo "clear map
  ↪ /etc/hapee-1.9/maps/hosts.map" |
  ↪ socat stdio /var/run/hapee-1.9/hapee-lb.sock
```

Add a new key and value with `add map`:

```
root@server1:~$ echo "add map
  ↪ /etc/hapee-1.9/maps/hosts.map
  ↪ foo.example.com be_bar" |
  ↪ socat stdio /var/run/hapee-1.9/hapee-lb.sock
```

Change an existing entry with `set map`:

```
root@server1:~$ echo "set map
  ↪ /etc/hapee-1.9/maps/hosts.map
  ↪ foo.example.com be_baz" |
  ↪ socat stdio /var/run/hapee-1.9/hapee-lb.sock
```

Using `show map`, we can get the contents of the file, filter it to only the second and third columns with *awk*, and then save the in-memory representation back to disk:

```
root@server1:~$ echo "show map
  ↪ /etc/hapee-1.9/maps/hosts.map" |
   ↪ socat stdio /var/run/hapee-1.9/hapee-lb.sock
|
  ↪ awk '{print $2" "$3}' >
  ↪ /etc/hapee-1.9/maps/hosts.map
```

Actions can also be chained together with semicolons, which makes it easy to script changes and save the result:

```
root@server1:~$ echo "clear map
  ↪ /etc/hapee-1.9/maps/hosts.map;
  ↪ add map /etc/hapee-1.9/maps/hosts.map
  ↪ bar.example.com be_foo;
  ↪ add map /etc/hapee-1.9/maps/hosts.map
  ↪ foo.example.com be_baz" |
  ↪ socat stdio /var/run/hapee-1.9/hapee-lb.sock
```

Did you know? If you are forking HAProxy with multiple processes via nbproc, you'll want to configure one socket per process and then run a loop to update each process individually. This is not an issue when using multithreading.

Editing With http-request set-map

Suppose you didn't want to go about editing files by hand or using the Runtime API. Instead, you wanted to be able to make an HTTP request with a certain URL parameter

and have that update your map file. In that case, `http-request set-map` is your go-to.

This allows the use of fetches, converters, and ACLs to decide when and how to change a map during runtime. In addition to `set-map`, there's also `del-map`, which allows you to remove map entries in the same way. As with the runtime API, these changes also only apply to the process that the request ends up on.

Pass the map file's path to `set-map` and follow it with a key and value, separated by spaces, that you want to add or update. Both the key and value support the `log-format` notation, so you can specify them as plain strings or use fetches and converters. For example, to add a new entry to the **hosts.map** file, but only if the source address falls within the **192.168.122.0/24** range, you can use a configuration like this:

```
frontend fe_main
    bind :80
    acl in_network src 192.168.122.0/24
    acl is_map_add path_beg /map/add

    http-request
      ↪ set-map(/etc/hapee-1.9/maps/hosts.map)
                          ↪    %[url_param(domain)]
%[url_param(backend)]
      ↪ if is_map_add in_network

    http-request deny deny_status 200
      ↪ if { path_beg /map/ }
    use_backend %[req.hdr(host),lower,
      ↪ map(/etc/hapee-1.9/maps/hosts.map)]
```

This will allow you to make web requests such as **http://192.168.122.64/map/add?domain=example.com&backend=be_static** for a quick and easy way to update your maps. If the entry already exists, it will be updated. Notice that you can use `http-request deny deny_status 200` to prevent the request from going to your backend servers.

The `http-request del-map` command is followed by the key to remove from the map file.

```
acl is_map_del path_beg /map/delete
http-request
del-map(/etc/hapee-1.9/maps/hosts.map)
  ↪ %[url_param(domain)] if is_map_del in_network
```

Using the `show map` technique you saw earlier, you might schedule a cron job to save your map files every few minutes. However, if you need to replicate these changes across multiple instances of HAProxy, using one of the other approaches will be a better bet.

Did you know? Another way to control when to set or delete an entry is to check the method of the request and then set an entry if it's POST or PUT. If it's DELETE, delete an entry.

Putting It Into Practice

We've seen how to use the Host header to look up a key in a map file and choose a backend to use. Let's see some other ways to use maps.

A Blue-Green Deployment

Suppose you wanted to implement a blue-green deployment wherein you're able to deploy a new release of your web application onto a set of staging servers and then swap them with a set of production servers. You could create a file called **bluegreen.map** and add a single entry:

```
active be_blue
```

In this scenario, the *be_blue* `backend` contains your set of currently active, production servers. Here is your HAProxy configuration file:

```
frontend fe_main
    bind :80
    use_backend %[str(active),
      ↪ map(/etc/hapee-1.9/maps/bluegreen.map)]

backend be_blue
    server server1 10.0.0.3:80 check
    server server2 10.0.0.4:80 check

backend be_green
    server server1 10.0.0.5:80 check
    server server2 10.0.0.6:80 check
```

After you deploy a new version of your application to the *be_green* servers and test it, you can use the Runtime API to swap the active *be_blue* servers with the *be_green* servers, causing your *be_green* servers to become active in production.

```
root@server1:~$ echo
  ↪ "set map /etc/hapee-1.9/maps/bluegreen.map
  ↪ active be_green" |
  ↪ socat stdio /var/run/hapee-1.9/hapee-lb.sock
```

Now your traffic will be directed away from your *be_blue* servers and to your *be_green* servers. This, unlike a rolling deployment, ensures that all of your users are migrated to the new version of your application at the same time.

Rate Limiting by URL Path

For this example, you will set rate limits for your website. Using a map file lets you set different limits for different URLs. For example, URLs that begin with **/api/routeA** may allow a higher request rate than those that begin with **/api/routeB**.

Add a map file called **rates.map** and add the following entries:

```
/api/routeA 40
/api/routeB 20
```

Consider the following `frontend`, wherein the current request rate for each client is measured over 10 seconds. A URL path like **/api/routeA/some_function** would allow up to four requests per second (40 requests / 10 seconds = 4 rps).

```
frontend api_gateway
    bind :80
    default_backend api_servers

# Set up stick table to track request rates
stick-table type binary len 8 size 1m expire 10s
  ↪ store http_req_rate(10s)

# Track client by base32+src
# (Host header + URL path + src IP)
http-request track-sc0 base32+sr

# Check map file to get rate limit for path
http-request set-var(req.rate_limit) path,
  ↪ map_beg(/etc/hapee-1.9/maps/rates.map)

# Client's request rate is tracked
http-request set-var(req.request_rate) base32+src,
  ↪ table_http_req_rate(api_gateway)

# Subtract the current request rate from the limit
# If less than zero, set rate_abuse to true
acl rate_abuse var(req.rate_limit),
  ↪ sub(req.request_rate) lt 0

# Deny if rate abuse
http-request deny deny_status 429 if rate_abuse
```

Here, the `stick-table` definition records client request rates over ten seconds. Note that we are tracking clients using the `base32+src` fetch method, which is a combination of the Host header, URL path, and source IP

address. This allows us to track each client's request rate on a per-path basis. The `base32+src` value is stored in the stick table as binary data.

Then, two variables are set with `http-request set-var`. The first, req.rate_limit, is set to the predefined rate limit for the current path from the rates.map file. The second, req.request_rate, is set to the client's current request rate.

The ACL *rate_abuse* does a calculation to see whether the client's request rate is higher than the limit for this path. It does this by subtracting the request rate from the request limit and checking whether the difference is less than zero. If it is, the `http-request deny` directive responds with 429 Too Many Requests.

Conclusion

Now that you've seen a few of the possibilities, consider reaching for your trusty tool, maps, the next time you run into a problem where it can help.

Application-Layer DDoS Attack Protection

Put any website or application up these days and you're guaranteed to be the target of a wide variety of probes and attacks. Your website is a boat that must constantly weather the storm of various threats, including distributed denial-of-service (DDoS) attacks, malicious bots, and intrusion attempts. Over the years, HAProxy has evolved to living life in these perilous waters through the development of flexible building blocks that can be combined to mitigate nearly any type of threat. These building blocks include high-performance ACLs and maps, real-time tracking with stick tables, a performant SSL/TLS stack, a WAF, and much more. Even with all these added capabilities, it maintains the best-in-class performance that it's known for.

The spectrum of companies benefiting from HAProxy's advanced security capabilities range from small mom-and-pop shops to large enterprises, managed hosting companies and load balancer-as-a-service platforms serving millions of requests per second. Top websites include GitHub, which uses HAProxy to protect its network from application-layer DDoS attacks, and StackExchange, which uses it to detect and protect against

bot threats. Furthermore, Booking.com chose HAProxy as a core component in its edge infrastructure for its superior performance after comparing it with other software load balancers on the market.

In this chapter, we'll demonstrate how the HAProxy load balancer protects you from application-layer DDoS attacks that could, otherwise, render your web application dead in the water, unreachable by ordinary users. In particular, we'll discuss HTTP floods. An **HTTP flood** operates at the application layer and entails being immersed with web requests, wherein the attacker hopes to overwhelm your application's capacity to respond.

HTTP Flood

The danger of HTTP flood attacks is that they can be carried out by just about anyone. They don't require a large botnet and tools for orchestrating the attack are plentiful. This accessibility makes it especially important that you have defenses in place to repel these assaults.

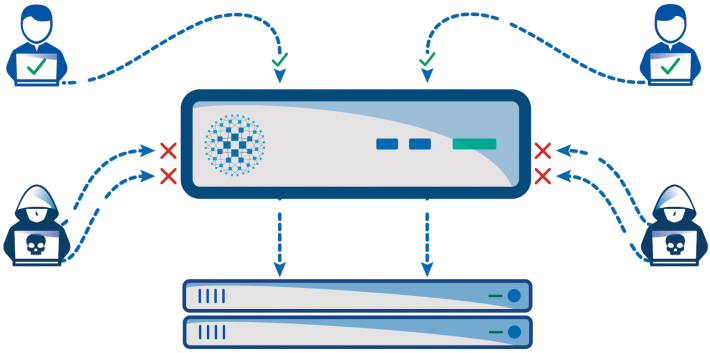
These attacks can come in a few different forms, but the most commonly seen pattern consists of attackers requesting one or more of your website's URLs with the highest frequency they are able to achieve. A shotgun approach will be to request random URLs, whereas more sophisticated attackers will profile your site first, looking for slow and uncached resources that are more vulnerable. For example, they may target search pages.

In order to evade detection for longer, the attack may consist of many different source IP addresses. It may be carried out by bots or by groups of real users working in unison to bring down your site. That was the case with the Low Orbit Ion Cannon (LOIC) and High Orbit Ion Cannon (HOIC) attacks carried out by the hacktivist collective Anonymous. The seemingly widespread range of source IPs is what characterizes the distributed nature of the attack.

HAProxy comes with features for mitigating HTTP floods and will play a vital part in your overall defense strategy. In this chapter, we will be using HAProxy Enterprise because of its additional security features, which we'll talk about later.

Manning the Turrets

The ideal place to stop an HTTP flood is at the edge of your network. Stopping threats here protects your upstream web applications by minimizing the traffic and system load that could impact them, as well as other sites and services running on those servers. It also prevents unnecessary confusion during attack identification by drawing a clear frontline to the battle.



The HAProxy load balancer receives requests from the Internet and passes them to your web servers. This lets you guard the perimeter. The other network devices that sit between HAProxy and the Internet, including routers and firewalls, are typically operating at too low a level to allow for request inspection.

Did you know? ALOHA, the HAProxy plug-and-play appliance, can protect you from low-level, protocol-based attacks, such as SYN floods, at line rate with our mitigation solution called PacketShield. PacketShield is powered by NDIV, an open-source network traffic processing framework that we've been working on since 2013. We have since been working closely with the XDP team to bring some NDIV features to XDP and make NDIV work on top of XDP.

With HAProxy, you have two methods that are very effective at classifying malicious requests. The first is to

monitor the rate at which a user is making requests. The second is to flag HTTP requests that have signatures you wouldn't expect to see from ordinary users. For the best results, you should combine the two methods. Setting request rate limits lets you block clients that access your website's resources too frequently, while denying requests that contain anomalous data narrows the field of possible attackers.

Setting Request Rate Limits

Tracking user activities across requests requires in-memory data storage that can identify returning clients and correlate their actions. This is key to setting rate-limiting thresholds—being able to track how many requests someone is making. HAProxy enables you to do this through an extremely flexible and high-performance data storage called stick tables, a feature that is unique to HAProxy.

Stick tables provide a generic key-value store and can be used to track various counters associated with each client. The key can be based on anything found within the request. Typically, it will be the user's IP address, although it can also be something more specific like the IP+UserAgent. Commonly tracked values are the request count and request rate over a period of time.

Stick tables were developed in collaboration with StackExchange, the network of Q&A communities that includes Stack Overflow, who initially approached HAProxy in 2010 about implementing rate limiting based on traffic

patterns. Stick tables are an extremely mature and proven technology within HAProxy, enabling many of its advanced features.

Defining the Storage

Create a stick table by adding a `stick-table` directive to a `backend` or `frontend`. In the following example, we use a placeholder backend named *per_ip_rates*. Dedicating a backend to holding just a `stick-table` definition allows you to reference it in multiple places throughout your configuration.

Consider the following example:

```
backend per_ip_rates
    stick-table type ip size 1m expire 10m
      ↪ store http_req_rate(10s)
```

This sets up the storage that will keep track of your clients by their IP addresses. It initializes a counter that tracks each user's request rate. Begin tracking a client by adding an `http-request track-sc0` directive to a `frontend` section, as shown:

```
frontend fe_mywebsite
    bind *:80
    http-request track-sc0 src table per_ip_rates
```

With this configuration in place, all clients visiting your website through HAProxy via the fe_mywebsite frontend

will be stored in the per_ip_rates stick table. All of the counters specified in the stick table definition will be automatically maintained and updated by HAProxy.

Next, let's see how to put this data to good use.

Limiting Request Rates

Let's say that you wanted to block any client making more than 10 requests per second. The `http_req_rate(10s)` counter that you added will report the number of requests over 10 seconds. So, to cap requests at 10 per second, set the limit to 100.

In the following example, we add the `http-request deny` directive to reject clients that have gone over the threshold:

```
frontend fe_mywebsite
    bind *:80
    http-request track-sc0 src table per_ip_rates
    http-request deny deny_status 429
      ↪ if { sc_http_req_rate(0) gt 100 }
```

This rule instructs HAProxy to deny all requests coming from IP addresses whose stick table counters are showing a request rate of over 10 per second. When any IP address exceeds that limit, it will receive an HTTP 429 Too Many Requests response and the request won't be passed to any HAProxy backend server.

These requests will be easy to spot in the HAProxy access log, as they will have a termination state of *PR–*, which means that the session was aborted because of a connection limit enforcement:

```
Feb 8 17:15:07 localhost hapee-lb[19738]:
192.168.1.2:49528       [08/Feb/2018:17:15:07.182]
fe_main fe_main/<NOSRV> 0/-1/-1/-1/0 429 188 - -
PR-- 0/0/0/0/0 0/0 "GET / HTTP/1.1"
```

If you'd like to define rate limit thresholds on a per URI basis, you can do so by adding a map file that pairs each rate limit with a URL path. See the previous chapter, **Introduction to HAProxy Maps** for an example.

Maybe you'd like to rate limit POST requests only? It's simple to do by adding a statement that checks the built-in ACL, `METH_POST`.

```
http-request track-sc0 src table per_ip_rates
   ↪ if METH_POST http-request deny deny_status
429
  ↪ if { sc_http_req_rate(0) gt 100 }
```

You can also tarpit abusers so that their requests are rejected with a HTTP 500 status code with a configurable delay. The duration of the delay is set with the `timeout tarpit` directive. Here, you're delaying any response for five seconds:

```
timeout tarpit 5s
http-request tarpit if { sc_http_req_rate(0)
  ↪ gt 100 }
```

When the timeout expires, the response that the client gets back after being tarpitted is 500 Internal Server Error, making it more likely that they'll think that their assault is working.

Slowloris Attacks

Before getting into our second point about DDoS detection, identifying odd patterns among users, let's take a quick look at another type of application-layer attack: Slowloris. **Slowloris** involves an attacker making requests very slowly to tie up your connection slots. Contrary to other types of DDoS, the volume of requests needed to make this attack successful is fairly low. However, as each request only sends one byte every few seconds, they can tie up many request slots for several minutes.

An HAProxy load balancer can hold a greater number of connections open without slowing down than most web servers. As such, the first step towards defending against Slowloris attacks is setting `maxconn` values. First, set a `maxconn` in the `global` section that leaves enough headroom so that your server won't run out of memory even if all the connections are filled. Then inside the `frontend` or a `defaults` section, set a `maxconn` value slightly under that so that if an attack saturates one frontend, the others can still operate.

Next, add two lines to your `defaults` section:

```
timeout http-request 5s
option http-buffer-request
```

The first line causes HAProxy to respond to any clients that spend more than five seconds from the first byte of the request to the last with an HTTP 408 Request Timeout error. Normally, this only applies to the HTTP request and its headers and doesn't include the body of the request. However, with `option http-buffer-request`, HAProxy will store the request body in a buffer and apply the `http-request timeout` to it.

Blocking Requests by Static Characteristics

You've seen how to block requests that surpass a maximum number of HTTP requests. The other way to identify and stop malicious behavior is by monitoring for messages that match a pattern. Patterns are set in HAProxy using access control lists (ACLs).

Let's see some useful ACLs for stopping DDoS attacks.

Using ACLs to Block Requests

A number of attacks use HTTP/1.0 as the protocol version because that's the version supported by some bots. It's

easy to block these requests using the built-in ACL, `HTTP_1.0`:

```
http-request deny if HTTP_1.0
```

You can also reject requests that have non-browser User-Agent headers, such as curl.

```
http-request deny if { req.hdr(user-agent) -i -m
  ↪ sub curl }
```

This line will deny the request if the `-m sub` part of the User-Agent request header contains the string *curl* anywhere in it. The `-i` makes it case-insensitive. You might also check for other strings such as *phantomjs* and *slimerjs*, which are two scriptable, headless browsers that could be used to automate an attack.

```
http-request deny if { req.hdr(user-agent) -i -m
  ↪ sub curl phantomjs slimerjs }
```

If you have many strings that you're checking, consider saving them to a file—one string per line—and referencing it like this:

```
http-request deny if { req.hdr(user-agent) -i -m
  ↪ sub -f /etc/hapee-1.9/badagents.acl }
```

At other times, an attacker who is using an automated tool will send requests that don't contain a User-Agent header at all. These can be denied too, as in the following example:

```
http-request deny unless { req.hdr(user-agent)
  ↪ -m found }
```

Even more common is for attackers to randomize the User-Agent strings that they send in order to evade detection for longer. Oftentimes, these come from a list of genuine values that a true browser would use and make it harder to identify malicious users.

This is where the HAProxy Enterprise **Fingerprint Module** comes in handy. It uniquely identifies clients across requests, even when they change their User-Agent string. It works by triangulating many data points about a client to form a signature specific to them. Using this information, you can then ID and dynamically block the abusers.

Denylisting and Greylisting

Another characteristic that you might use to filter out potentially dangerous traffic is the client's source IP address. Whether intentionally or unintentionally, China seems to be the origin of much DDoS traffic. You may decide to denylist all IPs coming from a particular country by researching which IP blocks are assigned to it and denying them en masse.

Use the `src` fetch method to get a client's source IP address. Then, compare it to a file containing all of the IP address ranges that you wish to block.

```
http-request deny if { src -f
  ↪ /etc/hapee-1.9/blacklist.acl }
```

Your **blacklist.acl** file might look like this:

```
1.0.1.0/2  4
1.0.2.0/2  3
1.0.8.0/2  1
1.0.32.0/1 9
1.1.0.0/2  4
1.1.2.0/2  3
1.1.4.0/2  2
# etc.
```

To streamline this, you can use a GeoIP database like MaxMind or Digital Element. Read our blog post, **Using GeoIP Database within HAProxy** (**http://bit.ly/2D5oqBU**) to see how to set this up. Alternatively, these lookups can happen directly from within HAProxy Enterprise using a native module that allows for live updates of the data and doesn't require extra scripts to translate to map files. The native modules also result in less memory consumption in cases where lookups need to be granular, for example, on a city basis.

If you don't like the idea of banning entire ranges of IP addresses, you might take a more lenient approach and

only greylist them. **Greylisting** allows those clients to access your website, but enforces stricter rate limits for them. The following example sets a stricter rate limit for clients that have IP addresses listed in **greylist.acl**:

```
http-request deny if { src -f
  ↪ /etc/hapee-1.9/greylist.acl }
  ↪ { sc_http_req_rate(0) gt 5 }
```

If you are operating two or more instances of HAProxy for redundancy, you'll want to make sure that each one has the list of the IP addresses that you've denylisted and that they are each updated whenever you make a change. Here's a place where using HAProxy Enterprise gives you an advantage. By using a module called **lb-update**, you can host your ACL file at a URL and have each HAProxy instance fetch updates at a defined interval.

In the next example, we're using lb-update to check for updates every 60 seconds:

```
dynamic-update
    update id /etc/hapee-1.9/blacklist.acl
      ↪ url https://192.168.122.1/blacklist.acl
      ↪ delay 60s
```

Protecting TCP (non-HTTP) Services

So far, we've primarily covered protecting web servers. However, HAProxy can also help in protecting other TCP-based services such as SSH, SMTP, and FTP. The first step is to set up a stick-table that tracks `conn_cur` and `conn_rate`:

```
frontend per_ip_connections
    stick-table type ip size 1m expire 1m
      ↪ store conn_cur,conn_rate(1m)
```

Next, create or modify a `frontend` to use this table by adding track and reject rules:

```
frontend fe_smtp
    mode tcp
    bind :25
    option tcplog
    timeout client 1m
    tcp-request content track-sc0 src
      ↪ table per_ip_connections
      ↪ tcp-request content reject
      ↪ if { sc_conn_cur(0) gt 1 } ||
      ↪ { sc_conn_rate(0) gt 5 }
    default_backend be_smtp
```

With the usual `backend`:

```
backend be_smtp
    mode tcp
    timeout server 1m
    option tcp-check
       server  smtp1  162.216.18.221:25  maxconn  50
check
```

Now, each client can establish one SMTP connection at a time. If they try to open a second one while the first is still open, the connection will be immediately closed again.

Delaying Connections

With e-mail and other server-speaks-first protocols (where the server sends a message as soon as a client connects instead of waiting for the client to say something, as with HTTP) we can delay connections as well by adding the following after the rules we added to block:

```
tcp-request inspect-delay 10s
tcp-request content accept if { sc_conn_rate(0)
  ↪ lt 2 }
tcp-request content reject if { req_len gt 0 }
```

This will immediately connect any client that has made only one connection within the last minute. A threshold of less than two is used so that we're able to accept one connection, but it also makes it easy to scale that threshold

up. Other connections from this client will be held in limbo for 10 seconds, unless the client sends data down that second pipe, which we check with `req_len`. In that case, HAProxy will close the connection immediately without bothering the backend.

This type of trick is useful against spam bots or SSH bruteforce bots, which will often launch right into their attack without waiting for the banner. With this, if they do launch right in, they get denied, and if they don't, they had to hold the connection in memory for an additional 10 seconds. If they open more connections to get around that rate limit, the *conn_cur* limits from the previous section will stop them.

The Stick Table Aggregator

Using active-active HAProxy load balancers in front of your websites increases your redundancy, protecting you in case one load balancer goes offline. It also provides extra capacity when weathering an application-based DDoS attack. You can learn how to set it up by watching our on-demand webinar, **Building Highly Scalable ADC Clusters Using BGP Equal-cost Multi-path Routing** (**http://bit.ly/2WGHr6v**).

In a standard HAProxy Community configuration, each individual instance of HAProxy only sees the requests coming to it. It does not see the requests that are received by other load balancer instances. This gives an attacker more time to stay under the radar. If you're using HAProxy Enterprise, enabling the Stick Table Aggregator module

solves this problem. It allows HAProxy servers to aggregate all of their request rates and statistics and make decisions based on the sum of data.

The illustration below depicts how multiple load balancers can be peered to share information. Note how by adding tiers of stick table aggregators, you can collect data from many instances of HAProxy. Contact us to learn how to set this up.

The reCAPTCHA and Antibot Modules

HAProxy isn't limited to just flat-out blocking a request. Sometimes, you'll deal with situations where things are less certain: Is it a bot or is it a bunch of visitors that appear with the same IP only because they are behind a NAT? More adaptable responses are in order.

Using a Lower Priority Backend

If you want to allow suspicious requests to your site normally, when loads are low, but restrict them when loads start increasing (or dedicate a cheap VM to suspicious requests, divert traffic to a static version of your site, etc), using another backend can help. To do this, create a backend with the new servers and then use a `use_backend` line to direct requests to it:

```
use_backend       be_website_bots       if       {
sc_http_req_rate(0)
  ↪ gt 100 }
```

This will typically go after the `http-request deny` rules, which would have a higher threshold like 200, so that an overly abusive bot will still get direct error responses, while ones with a lower request rate can get the *be_website_bots* backend instead. If returning errors even at the higher rates concerns you, you can add `{ be_conn(be_website) gt 3000 }` to only outright deny requests if there are more than 3,000 currently active connections to the backend.

Sending a Javascript Challenge

The HAProxy Enterprise **Antibot module** provides a way to make clients generate a key to enter the site, which will help identify individual users behind a NAT and seperate the clients that support Javascript from the ones that don't.

The Antibot module asks the client to solve a dynamically generated math problem. It works off of the idea that many automated DDoS bots aren't able to parse JavaScript. Or, if they are, doing so slows them down. Spending CPU time on solving the puzzle often consumes an attacker's resources that they're paying for by the minute and, frustrated, they will often go elsewhere in search of an easier target.

View our on-demand webinar, **DDoS Attack and Bot Protection with HAProxy Enterprise**

(**http://bit.ly/2TzWo8I**), to learn more and see a demo of the Antibot module in action.

Challenging a Visitor to Solve a Captcha

The reCAPTCHA module presents the client with a Google reCAPTCHA v2 challenge that a bot won't be able to complete. This is helpful for cases where a bot is taking advantage of a full-fledged browser such as headless Chrome or Selenium. This, like the Antibot module, weeds out illegitimate users, either stopping them in their tracks or slowing them down to the point where it's unfavorable for them to continue the assault.

Silently Dropping Requests

When your rules clearly indicate that a bot is a bot and it is just generating too much traffic, the best thing to do is to try and overload it.

In order to make requests, the bot needs to keep track of the TCP connections, and normally so does HAProxy. Thus, both are tied, except that HAProxy has to also answer other visitors at the same time. With `silent-drop` HAProxy will tell the kernel to forget about the connection and conveniently forget to notify the client that it did so. Now, HAProxy doesn't need to track that connection. This leaves the client waiting for a reply that will never come and it will have to keep the connection in its memory, using one of its source ports, until it times out. To do this, add `http-request silent-drop`, like so:

```
http-request silent-drop if { sc_http_req_rate(0)
  ↪ gt 100 }
```

The main downside to this, presuming that the rules are set such that no legitimate clients will get this treatment, is that any stateful network devices (namely firewalls) will be confused by this, as they too won't get a notification that the connection has closed. This will cause these devices to keep track of connections that HAProxy is no longer thinking about and, in addition, consume memory on the stateful firewall. Be mindful of this if you are using such a device.

Conclusion

In this chapter, you've learned how to defend your websites from application-layer attacks like HTTP floods and Slowloris by using features built into HAProxy for rate limiting and flagging suspicious clients. This safeguards your web servers and prevents malicious traffic from entering your network.

HAProxy Enterprise will give you some must-have features for aggregating stick table data and challenging suspicious clients with either JavaScript or reCAPTCHA puzzles. These extras will ensure that you're getting the full picture of your traffic and that regular users aren't locked out by false positives.

Bot Management with HAProxy

It is estimated that bots make up nearly half the traffic on the Internet. When we say bot, we're talking about a computer program that automates a mundane task. Typical bot activities include crawling websites for indexing, such as how Googlebot finds and catalogs your web pages. Or, you might sign up for services that watch for cheap airline tickets or aggregate price lists to show you the best deal. These types of bots are generally seen as beneficial.

Unfortunately, a large portion of bots are used for malicious reasons. Their intentions include web scraping, spamming, request flooding, brute forcing, and vulnerability scanning. For example, bots may scrape your price lists so that competitors can consistently undercut you or build a competitive solution using your data. Or they may try to locate forums and comment sections where they can post spam. At other times, they're scanning your site looking for security weaknesses.

HAProxy has best-in-class defense capabilities for detecting and protecting against many types of unwanted bot traffic. Its unique ACL, map, and stick table systems, as

well as its flexible configuration language, are the building blocks that allow you to identify any type of bot behavior and neutralize it. Furthermore, HAProxy is well known for maintaining its high performance and efficiency while performing these complex tasks. For those reasons, companies like StackExchange have used HAProxy as a key component in their security strategy.

In this chapter, you'll learn how to create an HAProxy configuration for bot management. As you'll see, bots typically exhibit unique behavior and catching them is a matter of recognizing the patterns. You'll also learn how to allowlist good bots.

HAProxy Load Balancer

To create an HAProxy configuration for bot management, you'll first need to install HAProxy and place it in front of your application servers. All traffic is going to be routed through it so that client patterns can be identified. Then, proper thresholds can be determined and response policies can be implemented.

In this chapter, we'll look at how many unique pages a client is visiting within a period of time and determine whether this behavior is normal or not. If it crosses the predetermined threshold, we'll take action at the edge before it gets any further. We'll also go beyond that and see how to detect and block bots that try to brute-force your login screen and bots that scan for vulnerabilities.

Bot Management Strategy

Bots can be spotted because they exhibit non-human behavior. Let's look at a specific behavior: web scraping. In that case, bots often browse a lot of unique pages very quickly in order to find the content or types of pages they're looking for. A visitor that's requesting dozens of unique pages per second is probably not human.

Our strategy is to set up the HAProxy load balancer to observe the number of requests each client is making. Then, we'll check how many of those requests are for pages that the client is visiting for the first time. Remember, web scraping bots want to scan through many pages in a short time. If the rate at which they're requesting new pages is above a threshold, we'll flag that user and either deny their requests or route them to a different backend.

You'll want to avoid blocking good bots like Googlebot though. So, you'll see how to define allowlists that permit certain IP addresses through.

Detecting Web Scraping

Stick tables store and increment counters associated with clients as they make requests to your website. To configure one, add a `backend` section to your HAProxy configuration file and then add a `stick-table` directive to it. Each `backend` can only have a single `stick-table` definition. We're going to define two stick tables, as shown:

```
backend per_ip_and_url_rates
     stick-table type binary len 8 size 1m expire
24h
      ↪ store http_req_rate(24h)

backend per_ip_rates
    stick-table type ip size 1m expire 24h
      ↪ store gpc0,gpc0_rate(30s)
```

The first table, which is defined within your *per_ip_and_url_rates* backend, will track the number of times that a client has requested the current webpage during the last 24 hours. Clients are tracked by a unique key. In this case, the key is a combination of the client's IP address and a hash of the path they're requesting. Notice how the stick table's `type` is *binary* so that the key can be this combination of data.

The second table, which is within a backend labeled *per_ip_rates*, stores a **general-purpose counter** called `gpc0`. You can increment a general-purpose counter when a custom-defined event occurs. We're going to increment it whenever a client visits a page for the first time within the past 24 hours.

The `gpc0_rate` counter is going to tell us how fast the client is visiting new pages. The idea is that bots will visit more pages in less time than a normal user would. We've arbitrarily set the rate period to thirty seconds. Most of the time, bots are going to be fast. For example, the popular Scrapy bot is able to crawl about 3,000 pages per minute!

On the other hand, bots can be configured to crawl your site at the same pace as a normal user would. Just keep in mind that you may want to change the rate period from thirty seconds to something longer, like 24 hours (*24h*), depending on how many pages a normal user is likely to look at within that amount of time.

Next, add a `frontend` section for receiving requests:

```
frontend fe_main
    bind :80

      # track client's IP in per_ip_rates stick
table
    http-request track-sc0 src table per_ip_rates

    # track client's source IP + URL accessed in
    # per_ip_and_url_rates stick table
    http-request track-sc1 url32+src
      ↪ table per_ip_and_url_rates unless
      ↪ { path_end .css .js .png .jpeg .gif }

    # Increment general-purpose counter in
    # per_ip_rates if client is visiting page
    # for the first time
    http-request sc-inc-gpc0(0) if
      ↪ { sc_http_req_rate(1) eq 1 }

    default_backend web_servers
```

The line `http-request track-sc1` adds the client to the `stick-table` storage. It uses a combination of their IP

address and page they're visiting as the key, which you get with the built-in fetch method `url32+src`. A fetch method collects information about the current request.

Web pages these days pull in a lot of supporting files: JavaScript scripts, CSS stylesheets, images. By adding an `unless` statement to the end of your `http-request track-sc1` line, you can exclude those file types from the count of new page requests. So, in this example, it won't track requests for JavaScript, CSS, PNG, JPEG and GIF files.

The `http-request track-sc1` line automatically updates any counters associated with the stick table, including the `http_req_rate` counter. So, in this case, the HTTP request count for the page goes up by one. When the count is exactly one for a given source IP address and page, it means the current user is visiting the page for the first time. When that happens, the conditional statement `if { sc_http_req_rate(1) eq 1 }` on the last line becomes true and the directive `http-request sc-inc-gpc0(0)` increments the `gpc0` counter in our second stick table.

Now that you're incrementing a general-purpose counter each time a client, identified by IP address, visits a new page, you're also getting the rate at which that client is visiting new pages via the `gpc0_rate(30s)` counter. How many unique page visits over thirty seconds denotes too many? Tools like Google Analytics can help you here with its *Pages / Session* metric. Let's say that 15 first-time page requests over that time constitutes bot-like behavior. You'll define that threshold in the upcoming section.

Setting a Threshold

Now that you're tracking data, it's time to set a threshold that will separate the bots from the humans. Bots will request pages much faster, over a shorter time. Your first option is to block the request outright. Add an `http-request deny` directive to your `frontend` section:

```
frontend fe_main
    bind :80
    http-request track-sc0 src table per_ip_rates
    http-request track-sc1 url32+src
      ↪ table per_ip_and_url_rates unless
      ↪ { path_end .css .js .png .jpeg .gif }

    # Set the threshold to 15 within the time period
    acl exceeds_limit sc_gpc0_rate(0) gt 15

    # Increase the new-page count if this is
    # the first time they've accessed this
    # page, unless they've already exceeded
    # the limit
    http-request sc-inc-gpc0(0) if
      ↪ { sc_http_req_rate(1) eq 1 } !exceeds_limit

    # Deny requests if over the limit
    http-request deny if exceeds_limit

    default_backend web_servers
```

With this, any user who requests more than 15 unique pages within the last thirty seconds will get a 403

Forbidden response. Optionally, you can use `deny_status` to pass an alternate code such as 429 Too Many Requests. Note that the user will only be banned for the duration of the rate period, or thirty seconds in this case, after which it will reset to zero. That's because we've added *!exceeds_limit* to the end of the `http-request sc-inc-gpc0(0)` line so that if the user keeps accessing new pages within the time period, it won't keep incrementing the counter.

To go even further, you could use a **general-purpose tag** (`gpt0`) to tag suspected bots so that they can be denied from then on, even after their new-page request rate has dropped. This ban will last until their entry in the stick table expires, or 24 hours in this case. Expiration of records is set with the `expire` parameter on the `stick-table`. Start by adding `gpt0` to the list of counters stored by the *per_ip_rates* stick table:

```
backend per_ip_rates
    stick-table type ip size 1m expire 24h
      ↪ store gpc0,gpc0_rate(30s),gpt0
```

Then, add `http-request sc-set-gpt0(0)` to your `frontend` to set the tag to 1, using the same condition as before. We'll also add a line that denies all clients that have this flag set.

```
http-request sc-set-gpt0(0) 1 if exceeds_limit
http-request deny if { sc_get_gpt0(0) eq 1 }
```

Alternatively, you can send any tagged IP addresses to a special backend by using the `use_backend` directive, as shown:

```
http-request sc-set-gpt0(0) 1 if exceeds_limit
use_backend be_bot_jail if { sc_get_gpt0(0) eq 1 }
```

This backend could, for example, serve up a cached version of your site or have `server` directives with a lower `maxconn` limit to ensure that they can't swamp your server resources. In other words, you could allow bot traffic, but give it less priority.

Observing the Data Collection

You can use the Runtime API to see the data as it comes in. If you haven't used it before, check out our blog post **Dynamic Configuration with the HAProxy Runtime API** (**http://bit.ly/2SpOPDX**) to learn about the variety of commands available. In a nutshell, the Runtime API listens on a UNIX socket and you can send queries to it using either *socat* or *netcat*.

The `show table [table name]` command returns the entries that have been saved to a stick table. After setting up your HAProxy configuration and then making a few requests to your website, take a look at the contents of the *per_ip_and_url_rates* stick table, like so:

```
$ echo "show table per_ip_and_url_rates" |
  ↪ socat stdio /var/run/hapee-1.9/hapee-lb.sock

#  table:  per_ip_and_url_rates,  type:  binary,
size:1048576, used:2
0x10ab92c:
key=203E97AA7F000001000000000000000000000000
use=0 exp=557590 http_req_rate(86400000)=1

0x10afd7c:
key=3CBC49B17F000001000000000000000000000000
use=0 exp=596584 http_req_rate(86400000)=5
```

I've made one request to **/foo** and five requests to **/bar**; all from a source IP of **127.0.0.1**. Although the key is in binary format, you can see that the first four bytes are different. Each key is a hash of the path I was requesting and my IP address, so it's easy to see that I've requested different pages. The *http_req_rate* tells you how many times I've accessed these pages.

Did you know? You can key off of IPv6 addresses with this configuration as well, by using the same url32+srcfetch method.

Use the Runtime API to inspect the *per_ip_rates* table too. You'll see the *gpc0* and *gpc0_rate* values:

```
# table: per_ip_rates, type: ip, size:1048576,
used:1
0x10ab878: key=127.0.0.1 use=0 exp=594039
gpc0=2 gpc0_rate(30000)=2
```

Here, the two requests for unique pages over the past 24 hours are reported as *gpc0=2*. The number of those that happened during the last thirty seconds was also two, as indicated by the *gpc0_rate(30000)* value.

If you're operating more than one instance of HAProxy, combining the counters that each collects will be crucial to getting an accurate picture of user activity. HAProxy Enterprise provides cluster-wide tracking with a feature called the **Stick Table Aggregator** that does just that. This feature shares stick table data between instances using the peers protocol, adds the values together, and then returns the combined results back to each instance of HAProxy. In this way, you can detect patterns using a fuller set of data.

Verifying Real Users

The risk in rate limiting is accidentally locking legitimate users out of your site. HAProxy Enterprise has the **reCAPTCHA module** that's used to present a Google reCAPTCHA v2 challenge page. That way, your visitors can solve a puzzle and access the site if they're ever flagged. In the next example, we use the reCAPTCHA Lua module so that visitors aren't denied outright with no way to get back in.

```
http-request use-service lua.request_recaptcha
  ↪ unless { lua.verify_solved_captcha "ok" }
  ↪ { sc_get_gpt0(0) eq 1 }
```

Now, once an IP is marked as a bot, the client will just get reCAPTCHA challenges until such time as they solve one, at which point they can go back to browsing normally.

HAProxy Enterprise has another great feature: the **Antibot module**. When a client behaves suspiciously by requesting too many unique pages, HAProxy will send them a JavaScript challenge. Many bots aren't able to parse JavaScript at all, so this will stop them dead in their tracks. The nice thing about this is that it isn't disruptive to normal users, so customer experience remains good.

Beyond Scrapers

So far, we've talked about detecting and blocking clients that access a large number of unique pages very quickly. This method is especially useful against scrapers, but similar rules can also be applied to detecting bots attempting to brute-force logins and scan for vulnerabilities. It requires only a few modifications.

Brute-force Bots

Bots attempting to brute force a login page have a couple of unique characteristics: They make POST requests and they hit the same URL (a login URL), repeatedly testing numerous username and password combinations. In the

previous section, we were tracking HTTP request rates for a given URL on a per-IP basis with the following line:

```
http-request track-sc1 base32+src
  ↪ table per_ip_and_url_rates
  ↪ unless { path_end .css .js .png .jpeg .gif }
```

We've been using `http-request sc-inc-gpc0(0)` to increment a general-purpose counter, `gpc0`, on the *per_ip_rates* stick table when the client is visiting a page for the first time.

```
http-request sc-inc-gpc0(0) if
  ↪ { sc_http_req_rate(1) eq 1 } !exceeds_limit
```

You can use this same technique to block repeated hits on the same URL. The reasoning is that a bot that is targeting a login page will send an anomalous amount of POST requests to that page. You will want to watch for POST requests only.

First, because the *per_ip_and_url_rates* stick table is watching over a period of 24 hours and is collecting both GET and POST requests, let's make a third stick table to detect brute-force activity. Add the following `stick-table` definition:

```
backend per_ip_and_url_brueforce
     stick-table type binary len 8 size 1m expire
10m
      ↪ store http_req_rate(3m)
```

Then add an `http-request track-sc2` and an `http-request deny` line to the `frontend`:

```
http-request track-sc2 base32+src
  ↪ table per_ip_and_url_bruteforce
  ↪ if METH_POST { path /login }
http-request deny if { sc_http_req_rate(2) gt 10 }
```

You now have a stick table and rules that will detect repeated POST requests to the **/login** URL, as would be seen when an attacker attempts to find valid logins. Note how the ACL *{ path /login }* restricts this to a specific URL. This is optional, as you could rate limit all paths that clients POST to by omitting it.

In addition to denying the request, you can also use any of the responses discussed in the *Unblocking Real Users* section above in order to give valid users who happen to get caught in this net another chance.

Vulnerability Scanners

Vulnerability scanners are a threat you face as soon as you expose your site or application to the Internet. Generic vulnerability scanners will probe your site for many

different paths, trying to determine whether you are running any known vulnerable, third-party applications.

Many site owners, appropriately, turn to a Web Application Firewall for such threats, such as the WAF that HAProxy Enterprise provides as a native module. However, many security experts agree that it's beneficial to have multiple layers of protection. By using a combination of stick tables and ACLs, you're able to detect vulnerability scanners before they are passed through to the WAF.

When a bot scans your site, it will typically try to access paths that don't exist within your application, such as **/phpmyadmin** and **/wp-admin**. Because the backend will respond with 404's to these requests, HAProxy can detect these conditions using the `http_err_rate` fetch. This keeps track of the rate of requests the client has made that resulted in a 4xx response code from the backend.

These vulnerability scanners usually make their requests pretty quickly. However, as high rates of 404's are fairly uncommon, you can add the `http_err_rate` counter to your existing *per_ip_rates* table:

```
backend per_ip_rates
    stick-table type ip size 1m expire 24h
                                        ↪       store
gpc0,gpc0_rate(30s),http_err_rate(5m)
```

Now, with that additional counter, and the `http-request track-sc0` already in place, you have—and can view via

the Runtime API—the 4xx rate for clients. Block them simply by adding the following line:

```
http-request deny if { sc_http_err_rate(0) gt 10 }
```

You can also use the `gpc0` counter that we are using for the scrapers to block them for a longer period of time:

```
http-request sc-inc-gpc0(0) if
  ↪ { sc_http_err_rate(0) eq 1 } !exceeds_limit
```

Now the same limits that apply to scrapers will apply to vulnerability scanners, blocking them quickly before they succeed in finding vulnerabilities. Alternatively, you can shadowban these clients and send their requests to a honeypot backend, which will not give the attacker any reason to believe that they have been blocked. Therefore, they will not attempt to evade the block. To do this, add the following in place of the `http-request deny` above. Be sure to define the backend *be_honeypot*:

```
use_backend be_honeypot if
  ↪ { sc_http_err_rate(0) gt 10 }
```

Allowlisting Good Bots

Although our strategy is very effective at detecting and blocking bad bots, it will also catch Googlebot, BingBot,

and other friendly search crawlers with equal ease. You will want to welcome these bots, not banish them.

The first step to fixing this is to decide which bots you want so that they don't get blocked. You'll build a list of good bot IP addresses, which you will need to update on a regular basis. The process takes some time, but is worth the effort! Google provides a helpful tutorial (**http://bit.ly/2BlCiYz**). Follow these steps:

1. Make a list of strings found in the User-Agent headers of good bots (e.g. GoogleBot).
2. Grep for the above strings in your access logs and extract the source IP addresses.
3. Run a reverse DNS query to verify that the IP is indeed a valid good bot. There are plenty of bad bots masquerading as good ones.
4. Check the forward DNS of the record you got in step 3 to ensure that it maps back to the bot's IP, as otherwise an attacker could host fake reverse DNS records to confuse you.
5. Use whois to extract the IP range from the whois listing so that you cover a larger number of IP's. Most companies are good about keeping their search bots and proxies within their own IP ranges.
6. Export this list of IP's to a file with one IP or CIDR netmask per line (e.g. **192.168.0.0/24**).

Now that you have a file containing the IP addresses of good bots, you will want to apply that to HAProxy so that these bots aren't affected by your blocking rules. Save the file as **whitelist.acl** and then change the `http-request track-sc1` line to the following:

```
http-request track-sc1 url32+src
  ↪ table per_ip_and_url_rates
   ↪ unless { path_end .css .js .png .jpeg .gif }
||
  ↪ { src -f /etc/hapee-1.9/whitelist.acl }
```

Now, search engines won't get their page views counted as scraping. If you have multiple files, such as another for allowlisting admin users, you can order them like this:

```
unless { src -f /etc/hapee-1.9/whitelist.acl
  ↪ -f /etc/hapee-1.9/admins.acl }
```

When using allowlist files, it's a good idea to ensure that they are distributed to all of your HAProxy servers and that each server is updated during runtime. An easy way to accomplish this is to purchase HAProxy Enterprise and use its **lb-update module**. This lets you host your ACL files at a URL and have each load balancer fetch updates at a defined interval. In this way, all instances are kept in sync from a central location.

Identifying Bots By Their Location

When it comes to identifying bots, using geolocation data to place different clients into categories can be a big help. You might decide to set a different rate limit for China, for

example, if you were able to tell which clients originated from there. In this section, you'll see how to read geolocation databases with HAProxy.

Geolocation with HAProxy Enterprise

HAProxy Enterprise provides modules that will read **MaxMind** and **Digital Element** geolocation databases natively. Let's see how to do this with MaxMind using HAProxy Enterprise.

First, load the database by adding the following directives to the `global` section of your configuration:

```
module-load hapee-lb-maxmind.so
maxmind-load COUNTRY
                                                    ↪
/etc/hapee-1.9/geolocation/GeoLite2-Country.mmdb
maxmind-cache-size 10000
```

Within your `frontend`, use `http-request set-header` to add a new HTTP header to all requests, which captures the client's country:

```
http-request set-header
  ↪ x-geoip-country
                                                    ↪
%[src,maxmind-lookup(COUNTRY,country,iso_code)]
```

Now, requests to the `backend` will include a new header that looks like this:

```
x-geoip-country: US
```

You can also add the line `maxmind-update url https://example.com/maxmind.mmdb` to have HAProxy automatically update the database from a URL during runtime.

If you're using Digital Element for geolocation, the same thing as we did for MaxMind can be done by adding the following to the `global` section of your configuration:

```
module-load hapee-lb-netacuity.so
netacuity-load 26
  ↪ /etc/hapee-1.9/geolocation/netacuity/
netacuity-cache-size 10000
```

Then, inside of your `frontend` add an `http-request set-header` line:

```
http-request set-header
  ↪ x-geoip-country %[src,netacuity-lookup-ipv4
  ↪ ("pulse-two-letter-country")]
```

This adds a header to all requests, which contains the client's country:

```
x-geoip-country: US
```

To have HAProxy automatically update the Digital Element database during runtime, add `netacuity-update url https://example.com/netacuity_db` to your `global` section.

Using the Location Information

You can now use the geolocation information to make decisions. For example, you could route clients that trigger too many errors to a special, honeypot backend. With geolocation data, the threshold that you use might be higher or lower for some countries.

```
use_backend be_honeypot if { sc_http_err_rate(0)
  ↪ gt 5 } { req.hdr(x-geoip-country) CN }
```

Since this information is stored in an HTTP header, your backend server will also have access to it, which gives you the ability to take further action from there. We won't get into it here, but HAProxy also supports device detection and other types of application intelligence databases.

Conclusion

In this chapter, you learned how to identify and ban bad bots from your website by using the powerful configuration language within the HAProxy load balancer. Placing this type of bot management in front of your servers will protect you from these crawlers as they attempt content scraping, brute forcing and mining for security vulnerabilities.

HAProxy Enterprise gives you several options in how you deal with these threats, allowing you to block them, send them to a dedicated backend, or present a challenge to them. Need help constructing an HAProxy configuration for bot detection and management that accommodates your unique environment? HAProxy Technologies' expert support team has many decades of experience mitigating many types of bot threats. They can help provide an approach tailored to your needs.

The HAProxy Enterprise WAF

Data breaches. Loss of consumer confidence. An endless cycle of companies being compromised. Not just fly-by-night, sketchy websites either. Large companies—companies that you'd think would do better—are being caught without reliable security measures in place.

There's a reason that one of the most consulted security guides, the OWASP Top 10, is a list of web application security risks. Applications that can be accessed online provide opportunities for attackers the most often. They typically listen on open ports like 80 and 443. Unlike traditional services like FTP and SSH, it's not as though you can just raise the drawbridge and shut out traffic to those ports. A network firewall, which operates at the TCP layer to restrict access to your networks, isn't the right tool to prevent these types of threats.

Web Application Firewalls were created specifically as a countermeasure to stop attacks against web applications. The **HAProxy Enterprise WAF** supports two modes: a zero-trust mode using a positive model and an OWASP CRS

mode. We will cover OWASP CRS mode in this chapter. The OWASP core rule sets can detect and stop the OWASP Top 10 threats—including SQL injection attacks (SQLi), cross-site scripting (XSS), remote file inclusion (RFI), remote code execution (RCE), and other hostile actions. WAFs are tools that don't just make the Internet safer for your customers. They make doing business online viable.

In this chapter, you'll learn more about the problems a WAF solves and get a look at how the HAProxy Enterprise WAF provides an essential layer of defense.

A Specific Countermeasure

We've enjoyed the benefits of network firewalls since the 1980s. They allow IT admins to filter traffic between networks based on any of the information in the TCP protocol: source IP, source port, destination IP, and destination port. Don't want someone directly accessing your database from the Internet? Put a firewall in front of it and close off access to the outside world. In fact, common practice is to block everything by default and only punch a hole through for specific applications.

Next-generation firewalls (NGFW) took this to the next level. They often include deep packet inspection (DPI) and intrusion detection systems (IDS) that allow the firewall to open up IP packets and look at their contents, even up to the application layer. For instance, an IDS might analyze packets to discover what type of messages they contain. Is this FTP? VoIP? HTTP traffic from video streaming or social

media websites? Or is it a virus, matched against a set of known signatures?

Traditional network firewalls and NGFWs don't adequately secure against the unique attacks aimed at web applications, though. For one thing, more and more online communication is being encrypted with SSL/TLS. An NGFW would have to decrypt this traffic as a man-in-the-middle to inspect it. Another problem is the level of sophistication of modern-day, application-layer attacks. What may seem like a reasonable HTTP request may actually be an attempt at SQL injection, for example.

Web application firewalls are built with the intent of recognizing and preventing attacks against websites and web applications. The HAProxy Enterprise WAF with OWASP core rule sets fills in the gaps left by other types of firewalls, protecting against the vulnerabilities listed in the OWASP Top 10. Really, network firewalls and WAFs complement each other well. It's always good to have multiple layers of security.

Routine Scanning

First things first. You need a way to assess the security of your application. There are a number of web security scanners out there including Acunetix, Nessus, and Burp Suite. We'll use one called **OWASP Zed Attack Proxy** (ZAP), which can be downloaded and installed onto Windows, Linux, and Mac. I've found ZAP to be one of the easier scanners to use and it's able to detect an impressive range of vulnerabilities. Also, go ahead and install **sqlmap**, which

is a pen testing tool laser-focused on finding web pages susceptible to SQL injection.

Routinely scanning your applications will help to make sure that flaws aren't slipping past you into production. It creates a baseline against which you can compare software releases. Injecting security into your regular development pipeline helps to keep everyone sharp. As you build out your product's features, you'll know early on when a vulnerability has been introduced.

We're going to demonstrate the types of threats that a scanner will detect and, ultimately, that a WAF will stop. To do that, we need an application that has some known flaws baked in. The **Damn Vulnerable Web Application** (DVWA) is perfect for this because it's been built to be, well, vulnerable.

Download the sample project (**http://bit.ly/2SdbQG3**) from Github. It uses Terraform to launch DVWA into a virtual machine running on AWS EC2. In front of it, we have an instance of HAProxy Enterprise that you can run as a free trial. The load balancer is exposed via a public IP address, which is assigned after Terraform has run. Remember to call `terraform destroy` to delete all resources from AWS afterwards so that you aren't billed for extra usage.

Note that when setting up the project with Terraform, you should set the *my_source_ip* variable to your own IP address. That way, the site is only accessible by you. More information can be found in the git repository's README file.

Once you have it up, open the site in a browser.

Log in with the credentials admin and password. Once in, click the *Create / Reset Database* button to initialize the site's MySQL database. At this point, there is no WAF protecting the site. It's wide open to security exploits.

Let's run sqlmap and see what it finds. When you log into DVWA, it places a cookie in your browser called PHPSESSID that tells the site that you're a logged-in user. So that sqlmap can bypass the login screen and scan the site, it needs the value of this cookie. Open your browser's Developers Tools and view the site's cookies on the Network tab. Then, find the PHPSESSID cookie and copy its value.

In the following command, the `--cookie` parameter is passed to sqlmap with the value of the PHPSESSID cookie. You should also give it the value of a cookie called *security*, which is set to *low*. This tells DVWA to not use its own built-in, practice WAF. Replace the session ID and IP address with your own values:

```
/usr/bin/python2 /usr/bin/sqlmap --random-agent
  ↪ --cookie="PHPSESSID={YOUR-SESSION-ID};
  ↪ security=low" --dbs
  ↪ --url="http://{IP}/vulnerabilities/sqli/?id=&
  ↪ Submit=Submit" -p id
```

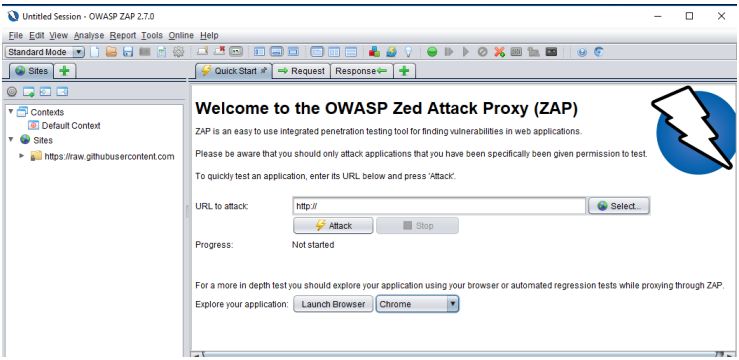
This command probes the **/vulnerabilities/sqli** page for SQL injection flaws, substituting various strings for the id parameter in the URL. When it's successful, it will gain access to the backend MySQL instance and enumerate the databases it finds:

```
[09:24:38] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian
web application technology: Apache 2.4.25
back-end DBMS: MySQL 5.0.12
[09:24:38] [INFO] fetching database names
available databases [2]:
[*] dvwa
[*] information_schema
```

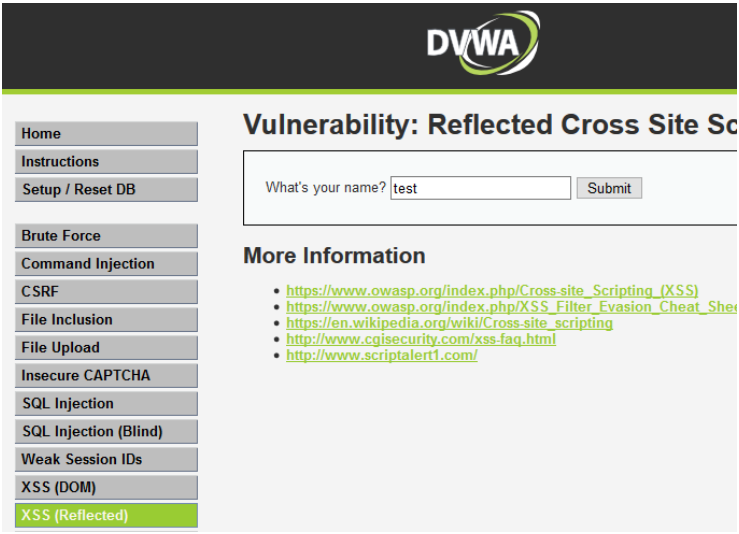
As you can see, sqlmap was able to find information about the website's databases and list out sensitive information. That's certainly a security weakness! You'll see in the next section how the HAProxy Enterprise WAF stops this from happening.

Next, let's use the ZAP scanner to find pages susceptible to cross-site scripting. You can use ZAP to scan for other sorts

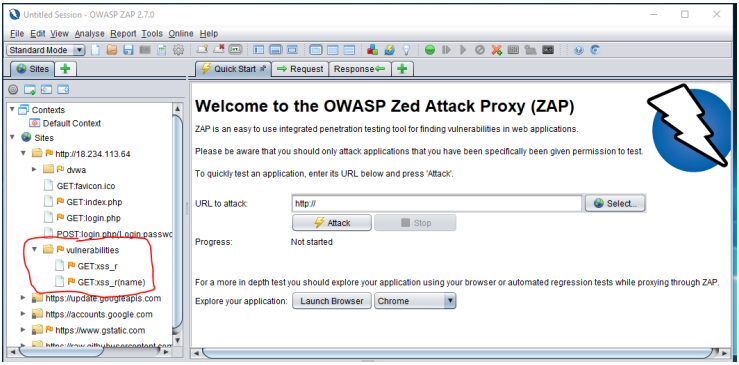
of vulnerabilities, too, if you like. Open ZAP and, from the right-hand panel, choose *Launch Browser*.



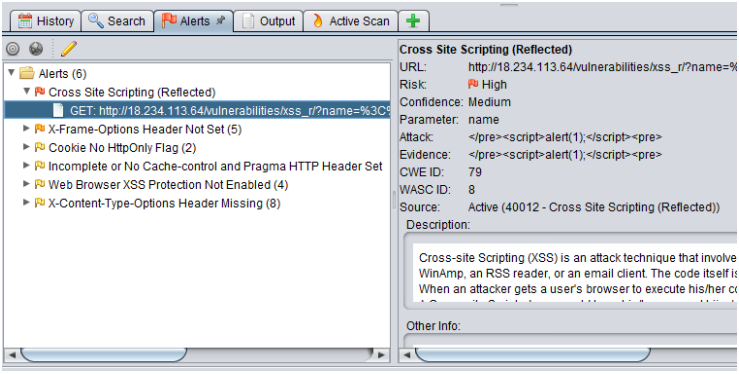
In the browser that opens, go to the site and log in. Using *Launch Browser* helps ZAP to learn the layout of the website. You can also have it crawl the site on its own, but that isn't as effective. To demonstrate a vulnerability, we'll focus on cross-site scripting (XSS) by going to the *XSS (Reflected)* page and typing a value into the *What's your name?* field. Then click Submit. After that, you can close the browser window.



When you go back to ZAP, you'll see that it has filled in the address of the DVWA website under Sites in the left-hand panel. Expand that folder and then expand the *vulnerabilities* folder. You should see that it captured two GET requests for the **/vulnerabilities/xss_r** page: *GET:xss_r* and *GET:xss_r(name)*.



Right-click on *GET:xss_r(name)* and choose **Attack > Active Scan**. ZAP will inspect that page, trying various strings for the name URL parameter. After it finishes, open the Alerts tab at the bottom and you should see that a *Cross Site Scripting (Reflected)* vulnerability was discovered.



We need to beef up our defenses so that sqlmap and ZAP don't find these vulnerabilities. In the next section, you'll see how to set up the WAF module in HAProxy Enterprise.

HAProxy Enterprise WAF

The WAF module utilizes the OWASP CRS by default to classify and detect malicious behavior. However, you can also add your own custom rules. The OWASP CRS WAF protects against many common threats.

Log into your HAProxy load balancer so that you can enable the WAF module. If you're following along with the sample project, then you can use SSH to log into the VM via its public IP address. Use the **haproxy_demo.pem** file as your SSH key:

```
ssh -i ./haproxy_demo.pem
  ↪ ubuntu@[HAPROXY_IP_ADDRESS]
```

You need to download the CRS. There's a script that will take care of this for you. Simply run the following command and the files will be downloaded to the **/etc/hapee-1.9/modsec.rules.d** directory:

```
sudo
/opt/hapee-1.9/bin/hapee-lb-modsecurity-getcrs
```

Next, go to **/etc/hapee-1.9** and edit the **hapee-lb.cfg** file with your favorite editor for these situations (vi, nano, etc.). Add the following `module-load` directive to the `global` section:

```
module-load hapee-lb-modsecurity.so
```

Also add a `filter` directive to your HAProxy `frontend` to enable protection for that proxy. Here's what it looks like:

```
frontend fe_main
    filter modsecurity owasp_crs rules-file
↪
/etc/hapee-1.9/modsec.rules.d/lb-modsecurity.conf
```

Then save the file and restart the load balancer services with the `hapee-1.9` command:

```
sudo hapee-1.9 restart
```

At this point, the WAF is in detection-only mode. That means that it will classify attacks as it sees them and write warnings to the file **/var/log/modsec_audit.log**. However, it will not block any requests. To turn on blocking, edit the file **/etc/hapee-1.9/modsec.rules.d/modsecurity.conf**. Near the beginning, change `SecRuleEngine DetectionOnly` to `SecRuleEngine On`. Then restart the load balancer services again.

Did you know? The modsec_audit.log file should be disabled in production use, since writing to disk will hinder performance.

Retesting with WAF Protection

Now that it is configured, a quick test with sqlmap shows that the WAF is working (remember to get the value of the PHPSESSID cookie):

```
/usr/bin/python2 /usr/bin/sqlmap --random-agent
              ↪       --cookie="PHPSESSID={SESSION
ID};security=low"
  ↪ --dbs
  ↪ --url="http://{IP}/vulnerabilities/sqli/?id=
  ↪ &Submit=Submit" -p id

[WARNING] GET parameter 'id' is not injectable
[CRITICAL] all tested parameters appear to be not
injectable.
[WARNING] HTTP error codes detected during run:
403 (Forbidden) - 1 times
```

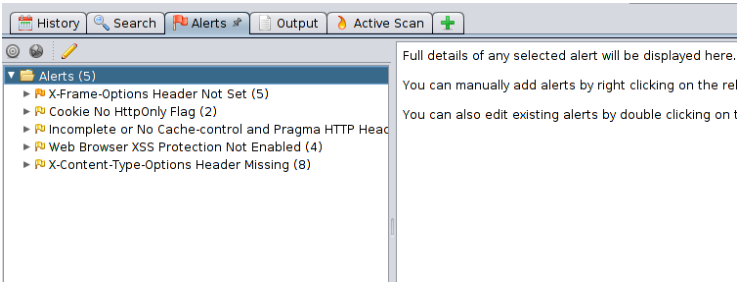
Here, even though we gave it a page that we know if susceptible to SQL injection, it wasn't able to find it. That's because the WAF is blocking requests that seem malicious with 403 Forbidden responses.

Did you know? When sqlmap runs it caches the results. So, if you ran it while the WAF was in detection-only mode, you'll want to delete the cache. It can be found under your user directory, ~/.sqlmap/output.

Stopping sqlmap from gaining access to the DVWA MySQL database is no small accomplishment! The tool scans for half a dozen types of relational databases and throws a barrage of injection attacks at its target. Yet, not a single one got through.

What you may find is that the OWASP CRS can be too assertive, triggering false positives and blocking legitimate users. If this is the first time you've used it, test it for a while in detection-only mode. Then you can determine which rules are right for your application and traffic, allowlisting those that are not or adjusting the severity levels of the rules. Just don't allowlist so much that the WAF loses its effectiveness!

Next, try running ZAP again, now that the WAF is enabled. Using the same steps as before, scan the *XSS (Reflected)* page for cross-site scripting vulnerabilities. Or, if you're feeling adventurous, try browsing around the rest of the site to map out more paths for ZAP. Then start an Active Scan against the *vulnerabilities* path.



The WAF rejects many of the suspicious requests with 403 Forbidden responses. This definitely strengthens your security posture. Remember, this was a website purposely built to be insecure. Your own applications will, no doubt, have more safeguards. However, it's never easy to catch all of the potential pitfalls and the HAProxy WAF module will create an essential layer of defense.

In an upcoming release of HAProxy Enterprise, you will be able to configure the OWASP CRS to defer its decision making to HAProxy. This will give you a wider range of options for how you deal with suspicious clients, beyond the blocking behavior of the WAF. The OWASP CRS will set variables, which the load balancer will be able to see, and action can be decided by ACL statements.

```
acl waf_blocked var(txn.owasp_crs.block) -m bool
http-request send-challenge ... if waf_blocked
```

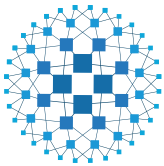
For example, you might show the client a Javascript challenge by using the Antibot module if they're flagged as potentially malicious. Subscribe to our blog to be alerted when this functionality becomes available!

Conclusion

In this chapter, we demonstrated the need for a web application firewall to protect you from threats like SQL injection and cross-site scripting. A WAF can filter out malicious behavior before it gets to your application, even defending against threats before you become aware of them. It's important to routinely scan for vulnerabilities and to share the responsibility for security with your entire team.

You've learned the building blocks of HAProxy: ACLs, stick tables and maps. Combined, they allow you to create countermeasures to a variety of threats including bots and DDoS. You also learned about the HAProxy Enterprise WAF. Where to go from here? Visit us online and contact us to learn how HAProxy can be used to solve your specific use case.

Want to know when content like this is published? Subscribe to our blog or follow us on Twitter **@HAProxy**. You can also join the conversation on Slack at **https://slack.haproxy.org**.



HAPROXY

The World's Fastest and Most Widely Used Software Load Balancer

www.haproxy.com