

POLITECNICO DI TORINO

Web Application Firewalls

Daniele Bringhenti



April 2, 2025

License

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

You are free:

- **to Share:** to copy, distribute and transmit the work
- **to Remix:** to adapt the work

Under the following conditions:

- **Attribution:** you must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- **Noncommercial:** you may not use this work for commercial purposes.
- **Share Alike:** if you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

More information on the Creative Commons website.



Acknowledgments

The author would like to thank all the people who contributed to this document.

Contents

1	Introduction	4
2	Squid	5
2.1	Squid installation and setup	6
2.2	Understanding Squid configuration for web filtering	9
2.3	Filtering web traffic by blocking specific domains	10
2.4	Filtering web traffic by blocking URLs using regular expressions	11
2.5	Filtering web traffic by blocking specific media types	12
2.6	Filtering web traffic depending on the time	13
3	ModSecurity	15
3.1	ModSecurity installation and setup	16
3.2	Understanding ModSecurity configuration for web filtering	17
3.3	Protection against SQL Injection	19
3.4	Protection against Cross Site Scripting	22

1 Introduction

This lab aims at practicing with Web Application Firewalls (WAFs). Differently from a packet filtering firewall, which operates at the network and transport layer (i.e., layers 3 and 4 of the ISO/OSI model), a WAF works at the application layer (i.e., layer 7). Specifically, a WAF is a firewalling module, installed on a proxy (forward and/or reverse), to filter HTTP traffic, depending on parameters such as HTTP commands, HTTP request/response header, and the content of HTTP requests and responses.

You will practice with two implementations of the web application filtering capability: Squid and ModSecurity. You will use the former to block traffic selectively (e.g., to block web access to specific websites). Instead, you will use the latter to block more advanced attacks, such as SQL injection and Cross-Side Scripting (XSS), thanks to attack detection rules defined by the OWASP foundation.

All the exercises about WAFs will be carried out by using a Virtual Machine on Crownlabs (<https://crownlabs.polito.it/>), running the Kali Linux OS. In fact, Kali Linux simplifies the execution of attacks, in a more controlled environment.

2 Squid



Figure 2.1: Squid logo.

Squid (<http://www.squid-cache.org/>) is an open-source caching and forwarding web proxy server that operates as an intermediary between clients and servers on a network. It acts as a gateway, enabling clients to access various internet resources such as websites, files, and other content from servers. Squid Proxy is highly flexible and customizable, offering a wide range of configuration options and support for various protocols such as HTTP, HTTPS, FTP, and more. Squid optimizes the data flow between client and server to improve performance and caches frequently-used content to save bandwidth. Squid can also route content requests to servers in a wide variety of ways to build cache server hierarchies which optimize network throughput. In view of these characteristics, it is widely used in enterprise networks, educational institutions, ISPs, and other environments where efficient caching, content control, and network optimization are desired.

Squid offers multiple functionalities: caching, forwarding, web filtering, access control, authentication, traffic optimizations. In this lab activity, we are mostly interested in the web filtering feature.

In order to provide the web filtering feature, Squid is often installed on a forward proxy. A forward proxy is the most common form of a proxy server and is generally used to pass requests from an isolated (private) network to the Internet through a firewall. Using a forward proxy, requests from an isolated network, or intranet, can be rejected or allowed to pass through a firewall. Requests may also be fulfilled by serving from cache rather than passing through the Internet. This allows a level of network security and lessens network traffic. A forward proxy server will first check to make sure a request is valid. If a request is not valid, or not allowed (blocked by the proxy), it will reject the request resulting in the client receiving an error or a redirect. If a request is valid, a forward proxy may check if the requested information is cached. If it is, the forward proxy serves the cached information. If it is not, the request is sent through a firewall to an actual content server which serves the information to the forward proxy. The proxy, in turn, relays this information to the client and may also cache it, for future requests.

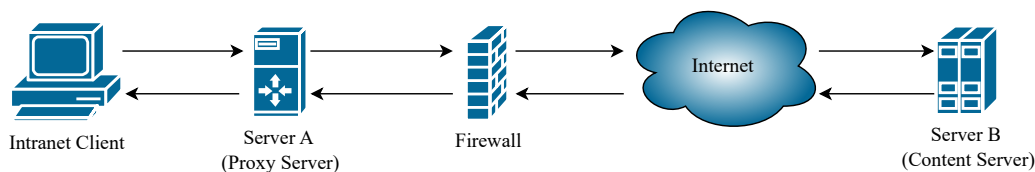


Figure 2.2: Example of network topology with a forward proxy.

Figure 2.2 shows a forward proxy configuration. An intranet client initiates a request that is valid but is not cached on Server A (Proxy Server). The request is sent through the firewall to the Internet server, Server B (Content Server), which has the information the client is requesting. The information is sent back through the firewall, it is cached on Server A and served to the client. Future requests for the same information will be fulfilled by the cache, lessening network traffic (proxy caching is optional and not necessary for the forward proxy to function on the HTTP Server).

For sake of simplicity, in this lab activity you will install the Squid firewall directly on the Virtual Machine running Kali Linux and working as a Intranet client. You will then use the client to try to establish communications with remote servers on the Internet. In this exercise, you will try to make some communications blocked, while other ones allowed, depending on the required security behavior.

2.1 Squid installation and setup

When you instantiate a Kali Linux Virtual Machine on Crownlabs, Squid is not installed by default. Therefore, you are in charge of installing it every time, with the following procedure.

First, you must update the APT cache (the APT acronym stands for Advanced Package Tool, and it is a package manager tools for Debian-based Linux distributions):

```
sudo apt update
```

You can then proceed to the actual installation of Squid, by executing the following command:

```
sudo apt install squid
```

At this point, Squid can already be executed. In order to do so, you can use the `systemctl` tool:

```
sudo systemctl restart squid
```

The initial Squid configuration is stored in the `/etc/squid/squid.conf` file. You can have a look at it by using an editor, such as nano, vim or gedit:

```
sudo gedit /etc/squid/squid.conf
```

As you can see, there are several commented lines. The reason is that, by reading this annotated configuration examples, users may find many features that can be later enabled by removing the comments. However, if you prefer, for sake of conciseness, you can directly remove all commented lines, so as to work with a shorter configuration file in the reminder of the exercise. If you do so, the configuration file will appear as follow (a more detailed explanation about its basic structure will be provided later in the document):

```
acl localnet src 0.0.0.1-0.255.255.255 # RFC 1122 "this" network (LAN)
acl localnet src 10.0.0.0/8 # RFC 1918 local private network (LAN)
acl localnet src 100.64.0.0/10 # RFC 6598 shared address space (CGN)
acl localnet src 169.254.0.0/16 # RFC 3927 link-local(directly plugged)machines
acl localnet src 172.16.0.0/12 # RFC 1918 local private network (LAN)
acl localnet src 192.168.0.0/16 # RFC 1918 local private network (LAN)
acl localnet src fc00::/7 # RFC 4193 local private network range
```

```

acl localnet src fe80::/10          # RFC 4291 link-local(directly plugged)machines

acl SSL_ports port 443
acl Safe_ports port 80  # http
acl Safe_ports port 21  # ftp
acl Safe_ports port 443  # https
acl Safe_ports port 70  # gopher
acl Safe_ports port 210  # wais
acl Safe_ports port 1025-65535 # unregistered ports
acl Safe_ports port 280  # http-mgmt
acl Safe_ports port 488  # gss-http
acl Safe_ports port 591  # filemaker
acl Safe_ports port 777  # multiling http

http_access deny !Safe_ports
http_access deny CONNECT !SSL_ports
http_access allow localhost manager
http_access deny manager
http_access allow localhost
http_access deny to_localhost
http_access deny to_linklocal
include /etc/squid/conf.d/*.conf
http_access deny all

http_port 3128
coredump_dir /var/spool/squid
refresh_pattern ^ftp:  1440 20% 10080
refresh_pattern -i (/cgi-bin/|\?) 0 0% 0
refresh_pattern . 0 20% 4320

```

Every time you will make a change to this configuration will, you will have to launch the *restart* command to make the change active:

```
sudo systemctl restart squid
```

After these preliminary setup operations, Squid should already be functional, but you are recommended to perform a pair of checks, so as to identify any possible issue in the installation or setup.

First, you can directly check if Squid is active with the same the `systemctl` tool you used to start it: At this point, Squid can already be executed. In order to do so, you can use the `systemctl` tool:

```
sudo systemctl status squid
```

In the command line, you should see that Squid is declared active (running):

```


squid.service - Squid Web Proxy Server
Loaded: loaded (/lib/systemd/system/squid.service; disabled; preset: disabled)
Active: active (running) since Wed 2024-01-24 07:34:01 EST; 14s ago

```

Second, you can try to send an HTTP request through the proxy, with the `curl` command. By default, the Squid proxy is listening to port 3128:

```
curl -O -L "https://www.polito.it" -x "localhost:3128"
```

If the page is locally downloaded, it means Squid is properly working.

Having checked that Squid works, you can now set up your browser to use it as a proxy. To do so, open Mozilla Firefox, click on the  button, and select Settings. In the General page, scroll down until you reach the Network Settings section, which allows you to configure how Firefox connects to the Internet. In that section, click on the Settings... button. A new window, entitled Connection Settings will open. You should configure it as shown in Figure 2.3. Specifically, You must configure the manual proxy configuration, so that Squid, listening to localhost:3128, is used as HTTP/HTTPS proxy. Clicking on the OK button, the connection settings will be applied to Firefox.

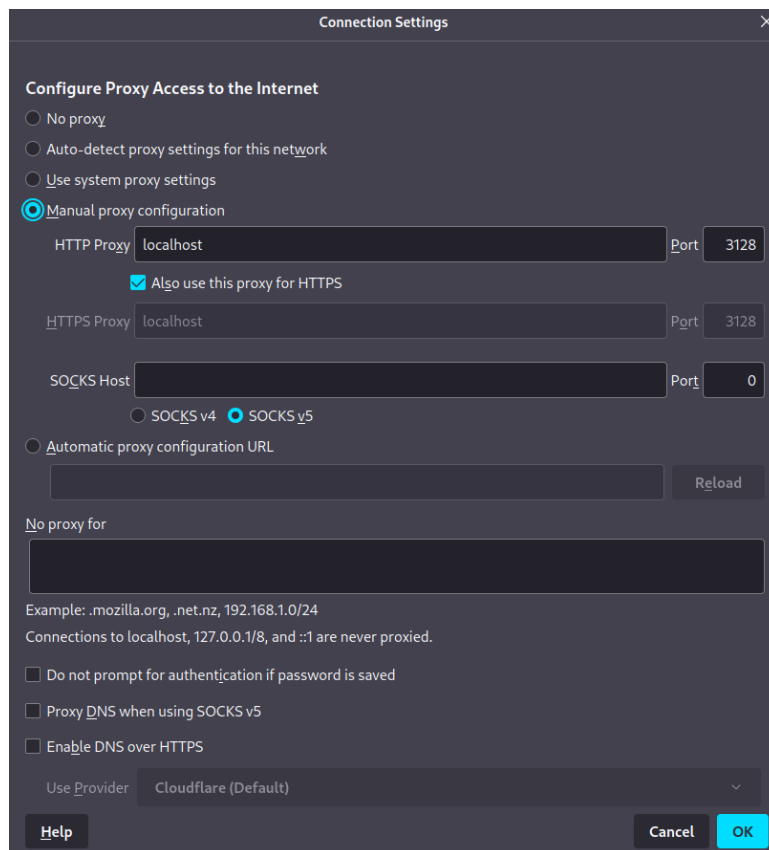


Figure 2.3: Connection settings for Firefox.

As a final check, you are invited to try to connect to any website, e.g., to *https://www.polito.it* with Firefox. If the connection is successfully established, it means that Firefox is correctly using the proxy to connect to the Internet.

This concludes the Squid setup. We will now look at how Squid is configured to provide web filtering in a forward proxy.

2.2 Understanding Squid configuration for web filtering

Squid configuration scheme is relatively comprehensive and difficult for the users to understand, as it envisions a large number of directives that can be executed by this firewalling proxy. However, in this lab activity, we will focus only on two directive types: *acl* and *http_access*.

The *acl* directive allows the creation of variables, internally named access list, which may represent entities such as packet classes or time frames, and which may be used in the context of other directives. In fact, when loading the configuration file, Squid processes all the *acl* directives into memory, so that it can use them to process other directive type. For this reason, a Squid configuration file should first have *acl* directives.

Each *acl* directive definition must begin with an *aclname* and *acltype*, followed by either type-specific arguments or a quoted filename that they are read from. When using “file”, the file should contain one item per line.

```
acl aclname acltype argument ...  
acl aclname acltype "file" ...
```

The *aclname* is a simple identifier of the *acl* element. Instead, *acltype* may be assigned with different values, depending on what the *acl* element must actually represent. You can find all the possible values (with examples of arguments) for this parameter in the official Squid documentation, at the following link: <http://www.squid-cache.org/Doc/config/acl/>. Here, the most relevant ones are reported, including the ones useful for the lab activity itself:

- *src*: source (client) IP addresses;
- *dst*: destination (server) IP addresses;
- *srcdomain*: source (client) domain name;
- *dstdomain*: destination (server) domain name;
- *time*: time of day, and day of week;
- *url_regex*: URL regular expression pattern matching.

The *http_access* directive allows the creation of access list rules, used to allow or deny access to the Internet based on the previously defined *acl* elements. The syntax of this directive is the following:

```
http_access allow|deny acl1 [acl2 acl3 ... acln]
```

If a list of *http_access* directives is defined, then each access to the Internet is matched with them sequentially, and the action that is applied to manage that access (allow—deny) is the one of the first matching access list rule, according to the FMR (First Matching Rule) resolution method of firewalls. Instead, if a list of *acl* elements are defined in a single *http_access* directive instead of one *acl* elements, all ACL elements of the rule must be a match in order for the rule to be a match. In other words, to summarize, the ACL logics can be described as follows (note that the AND/OR operators used below are just for illustration, so they are not part of the syntax):

```
http_access allow|deny acl AND acl AND ...  
OR  
http_access allow|deny acl AND acl AND ...  
OR  
...
```

Now you are finally ready to practice with Squid configurations for web filtering.

2.3 Filtering web traffic by blocking specific domains

The first activity is about filtering web traffic by blocking specific domains. For this purpose, let us suppose that, in the scenario initially depicted in Figure 2.2, the Intranet Client is used by a company employee, while you are the company network administrator, in charge of configuring Squid so as to control the Internet access from the employee. In particular, you will try to modify the basic default configuration (Note: as you can see from the *http_access* directive list, it is not a pure whitelisting or blacklisting configuration. In real scenarios, it may be modified to be one of them. However, it is useful to keep it as it is now, to understand better how Squid actually works).

Your employer requested you to block Internet access from Intranet clients towards specific web domains, e.g., a list of domains related to social networks and a list of domains related to video streaming.

First, you need to introduce two *acl* directives in the Squid configuration file `/etc/squid/squid.conf`. There are two alternative ways to do so. The first way is to introduce the domain in line, i.e., as direct arguments of the directive.

```
acl social_networks dstdomain .facebook.com .twitter.com .instagram.com
    .linkedin.com .discord.com
acl streaming dstdomain .youtube.com .netflix.com .crunchyroll.com
    .funimation.com .primevideo.com
```

A second way is to include the domains in two files, with one domain for each line, and then include the file names as arguments of the *acl* directives. For example, you may create a *social_networks.acl* file and a *streaming.acl* file in `/etc/squid/`, and then define the following directives:

```
acl social_networks dstdomain "/etc/squid/social_networks.acl"
acl streaming dstdomain "/etc/squid/streaming.acl"
```

The expected behavior is the same. The only difference is that the second solution is more flexible, so appreciated for real long Squid configurations – but for this lab activity, both are fine.

Second, you need to introduce two *http_access* directives in the Squid configuration files. Indeed, the *acl* directives only create internal variables, but do not have a direct impact to web filtering decisions. As we have previously seen, we need two *http_access* directives instead of a single one, because they must be linked with an OR logic (i.e., a connection must be blocked if it is towards social networks or streaming services). The two directives can be easily defined as follows:

```
http_access deny social_networks
http_access deny streaming
```

So far, everything may seem easy, including the definition of these two *http_access* directives. Nevertheless, the main problem may be their positioning in the Squid configuration files.

You may try to insert these two directives in different positions, and then you may check the outcome by trying to connect to a blocked domain by using Firefox. Be aware that, every time you make a change to the Squid configuration file, you have to restart Squid with the usual command:

```
sudo systemctl restart squid
```

When Squid is blocking the request, Firefox will show you the window reported in Figure 2.4.

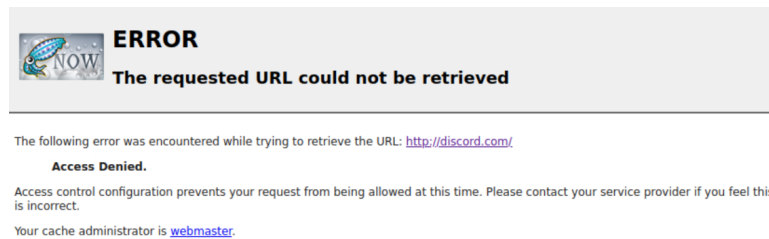


Figure 2.4: Firefox window when the proxy refuses a connection.

In particular, you may try to insert these two directives before and after the following line:

```
http_access allow localhost
```

You will see that the behavior is different. You are recommended to see it experimentally. If you put the `http_access` directives in charge of blocking social networks and streaming websites *before* the `http_access` directive to allow localhost, the proxy refuses the connection (but it does not refuse connections to other domains, e.g., to `.google.com`). Instead, if you put the `http_access` directives in charge of blocking social networks and streaming websites *after* the `http_access` directive to allow localhost, the proxy does not refuse the connection.

What is the reason of this behavior?

As previously explained, Squid sequentially processes the `http_access` directives. Keeping this in mind:

- if the `http_access allow localhost` directive appears *before* the blocking one, as your connection request actually comes from localhost, than directive is executed... and the blocking ones are ignored;
- if the `http_access allow localhost` directive appears *after* the blocking one, one of the blocking ones is enforced if the connection request is towards a blocked domain, even if the request comes from localhost.

2.4 Filtering web traffic by blocking URLs using regular expressions

The second activity is about filtering web traffic by using regular expressions. For this purpose, let us suppose that, referring again to the scenario initially depicted in Figure 2.2, your employer requests you to block any connection request to websites which allow the employees to play with online games.

Through a preliminary Web search, you notice that there exist several websites offering the possibility to play free games (e.g., by emulating old Flash games, which were really popular in the 2000s). Clearly, as a network administrator, you must block them, as they may distract your colleagues from working on their tasks. In your search, you also notice that many of these websites include the “game” word in their domain name.

At this point, instead of enumerating all of them, you may want to use a regular expression to express all of them concisely. However, be aware that Squid understands GNUregex (Extended Regular Expressions, i.e., ERE REGEXP), but it does not fully understand Perl Regular Expressions, i.e., PCRE).

You can find more information about regular expressions at the following links: https://www.gnu.org/software/gnulib/manual/html_node/Regular-expressions.html, https://www.gnu.org/software/grep/manual/html_node/Regular-Expressions.html.

Specifically, you may want to define a regular expression representing any URL with the “game” word in it. The regular expression that you can use is the following:

```
^.*game.*$
```

At this point, you again have to include the proper *acl* and *http_access* directives in the Squid configuration file `/etc/squid/squid.conf...` but one of each will be enough. The *acl* directive that must be included has *url_regex* type, as that allows to check if the regular expression matches with the whole URL, including the domain and the URL path (i.e., the URL part that follows the domain). Besides, as for the previous exercise, remember that the *http_access* directive must be put before the one allowing localhost, otherwise the connection will not be blocked.

```
acl games url_regex ^.*game.*$
http_access deny games
```

Then, restart Squid:

```
sudo systemctl restart squid
```

At this point, every connection attempt towards URLs including the “game” word in it will be blocked, even if the actual website does not exist, i.e., even if the domain to which the URL belongs has not been bought by anyone on the Web. You are recommended to try it experimentally, by trying to connect to existing and non-existing URLs with the the “game” word in them.

2.5 Filtering web traffic by blocking specific media types

The third activity is about filtering web traffic by blocking specific media types. For this purpose, let us suppose that, referring again to the scenario initially depicted in Figure 2.2, your employer requests you to block any connection request to torrent file, to avoid that your colleagues use the Internet connection to download illegal videos.

Also for this activity, you are invited to use regular expressions. However, you do not need to use an *acl* directive of *url_regex* type, because the file extension would not be present in the domain, but in the URL path. Instead, you should use an *acl* directive of *urlpath_regex* type, so that Squid will also be able to perform the matching operation in faster times.

Specifically, you can include the following directives in the Squid configuration file `/etc/squid/squid.conf`:

```
acl torrent urlpath_regex \.torrent$
http_access deny torrent
```

Then, restart Squid:

```
sudo systemctl restart squid
```

At this point, every connection attempt towards URLs whose path ends with “.torrent” should be blocked. You are recommended to try it experimentally, e.g., by trying to connect to the torrent file to download the most recent version of the Kali Linux Virtual Machine (of course, your employer was instead interested in preventing the employees from downloading other file types...):

```
https://cdimage.kali.org/kali-2024.4/kali-linux-2024.4-virtualbox-amd64.7z.torrent
```

You will thus see that..., actually, the torrent file is downloaded, because Squid did not block the connection?

How is it possible?

The reason is that, in HTTPS provides end-to-end security. Squid is not at the end of the TLS channel, therefore it can only see the CONNECT header of an HTTPS Request and the only the base domain part of an URL. In view of this reason, Squid will never be able to block the access to specific media types on HTTPS...

At least, you can try to see what happens if you try to download the same previous torrent file on HTTP (even if it is not actually served on HTTP):

```
http://cdimage.kali.org/kali-2024.4/kali-linux-2024.4-virtualbox-amd64.7z.torrent
```

On Firefox, you will see the window reported in Figure 2.5, telling you that access to the torrent file is denied, as expected.

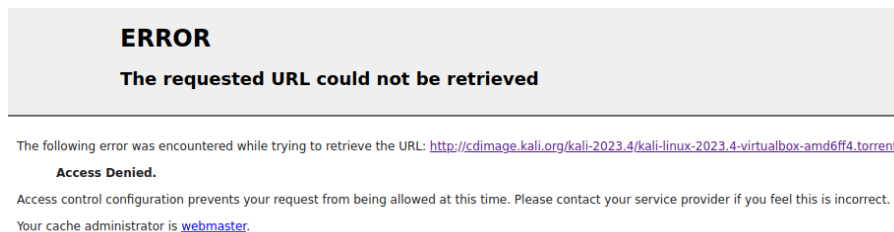


Figure 2.5: Access denied to torrent tile.

2.6 Filtering web traffic depending on the time

The forth activity is about filtering web traffic depending on the time. For this purpose, let us suppose that, referring again to the scenario initially depicted in Figure 2.2, your employer requests you to block any connection request to streaming websites only in the working hours 08:00-12:00 and 14:00-18:00 of the weekdays, so that the employees could use them to relax during their lunch break.

Squid offers a specific *acl* directive type, named *time*, to define day and time variables related to when web traffic must be blocked. The structure of this directive is the following:

```
acl aclname time [day-abbrevs] [h1:m1-h2:m2]
# [fast]
# day-abbrevs:
# S - Sunday
# M - Monday
```

```
# T - Tuesday
# W - Wednesday
# H - Thursday
# F - Friday
# A - Saturday
# h1:m1 must be less than h2:m2
```

For simplicity, you can open the Squid configuration file `/etc/squid/squid.conf` and remove all the new *acl* and *http_access* directives (i.e., all the ones not belonging to the initial default configuration) that you have introduced in the previous exercises except for the *acl* directive about streaming websites.

Then, you can define the following two *acl* directives:

```
acl morning time MTWHF 03:00-12:00
acl afternoon time MTWHF 14:00-18:00
```

Unfortunately, you cannot use a single one to define two non-contiguous time intervals, as reported in the except of documentation above.

At this point, you can proceed to include the following two *http_access* directives:

```
http_access deny social_networks morning
http_access deny social_networks afternoon
```

Then, restart Squid:

```
sudo systemctl restart squid
```

At this point, every connection attempt towards domains of the listed streaming websites in the working hours of weekdays should be blocked. You are recommended to try it experimentally, by trying to connect to those websites. of course, you can change the actual values of the time intervals, depending on the time you are actually carrying out the lab activity.

Finally, you are invited to stop Squid before continuing the activity with the next exercise:

```
sudo systemctl stop squid
```

You should also restore the previous connection settings in Firefox.

3 ModSecurity



Figure 3.1: ModSecurity logo.

ModSecurity (<https://github.com/owasp-modsecurity/ModSecurity>) is an open-source cross-platform web application firewall engine for Apache, IIS and Nginx that is developed by Trustwave's SpiderLabs. It has a robust event-based programming language which provides protection from a range of attacks against web applications and allows for HTTP traffic monitoring, logging and real-time analysis.

A main feature of this WAF with respect to other ones such as Squid is that it supports a configuration based on OWASP rules. The OWASP (Open Worldwide Application Security Project) CRS (Core Rule Set) (<https://coreruleset.org/>) is a free and open-source collection of rules that work with ModSecurity and some other WAFs. These rules are designed to provide easy to use, generic attack detection capabilities, with a minimum of false positives (false alerts), to web applications as part of a well balanced defense-in-depth solution. The CRS thus provides protection against many common attack categories, including SQL Injection (SQLi), Cross Site Scripting (XSS), Local File Inclusion (LFI), Remote File Inclusion (RFI), PHP Code Injection, Java Code Injection, HTTPoxy, Shellshock, Unix/Windows Shell Injection, Session Fixation, Scripting/Scanner/Bot Detection, Metadata/Error Leakages.

In order to provide the web filtering feature based on the OWASP CRS, ModSecurity is often installed on a reverse proxy. A reverse proxy is another common form of a proxy server and is generally used to pass requests from the Internet, through a firewall to isolated (private) networks. It is used to prevent Internet clients from having direct, unmonitored access to sensitive data residing on content servers on an isolated network, or intranet. If caching is enabled, a reverse proxy can also lessen network traffic by serving cached information rather than passing all requests to actual content servers. Reverse proxy servers may also balance workload by spreading requests across a number of content servers. One advantage of using a reverse proxy is that Internet clients do not know their requests are being sent to and handled by a reverse proxy server. This allows a reverse proxy to redirect or reject requests without making Internet clients aware of the actual content server (or servers) on a protected network. A reverse proxy server will first check to make sure a request is valid. If a request is not valid, or not allowed (blocked by the proxy), it will not continue to process the request resulting in the client receiving an error or a redirect. If a request is valid, a reverse proxy may check if the requested information is cached. If it is, the reverse proxy serves the cached information. If it is not, the reverse proxy will request the information from the content server and serve it to the requesting client. It also caches the information for future requests.

Figure 3.2 shows a reverse proxy configuration. An Internet client initiates a request to Server A (Proxy Server) which, unknown to the client, is actually a reverse proxy server. The request is allowed to pass through the firewall and is valid but is not cached on Server A. The reverse proxy (Server

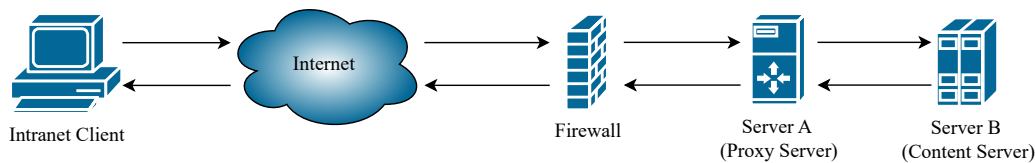


Figure 3.2: Example of network topology with a reverse proxy.

A) requests the information from Server B (Content Server), which has the information the Internet client is requesting. The information is served to the reverse proxy, where it is cached, and relayed through the firewall to the client. Future requests for the same information will be fulfilled by the cache, lessening network traffic and load on the content server (proxy caching is optional and not necessary for proxy to function on your HTTP Server). In this example, all information originates from one content server (Server B).

For sake of simplicity, in this lab activity you will install the ModSecurity firewall directly on the Virtual Machine running Kali Linux and working as a Content Server. You will then use Kali Linux itself as a client to try to establish communications with that Content Server. In this exercise, you will try to block two kinds of attacks: an SQL Injection attack and a Cross Site Scripting attack.

3.1 ModSecurity installation and setup

When you instantiate a Kali Linux Virtual Machine on Crownlabs, ModSecurity is not installed by default. Therefore, you are in charge of installing it every time. However, for this lab activity, you will simply install ModSecurity as a security module of an Apache HTTP server, already available in Kali Linux by default. This will allow a faster setup of this lab activity section.

First, you are recommended to update the APT cache, if you have not already done it when installing Squid:

```
sudo apt update
```

After this preliminary operation, you can directly execute the Apache server:

```
sudo systemctl restart apache2
```

You can then proceed to the actual installation of ModSecurity as a module of Apache, by executing the following command:

```
sudo apt install libapache2-mod-security2
```

At this point you can enable ModSecurity with the following command:

```
sudo a2enmod security2
```

You are recommended to verify if the ModSecurity module was loaded with the following command:

```
apachectl -M | grep --color security
```


You should see a module named `security2_module` (shared) which indicates that the module was loaded:

```
(kali@kali)-[~]
$ sudo apachectl -M | grep --color security
security2_module (shared)
```

Figure 3.3: Confirmation that ModSecurity has been successfully loaded to Apache.

After getting this confirmation, you must restart the Apache server for changes to take effect:

```
sudo systemctl restart apache2
```

Before analyzing how ModSecurity is configured, you should check that, by accessing `http://localhost/` with Firefox, you see that Apache is active, as shown in Figure 3.4.

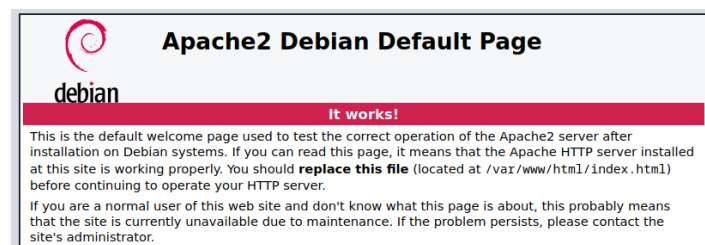


Figure 3.4: Confirmation that the Apache server is active.

3.2 Understanding ModSecurity configuration for web filtering

The initial ModSecurity configuration is stored in the `/etc/apache2/mods-enabled/security2.conf` file. You can have a look at it by using an editor, such as nano, vim or gedit:

```
sudo gedit /etc/apache2/mods-enabled/security2.conf
```

This short file will appear as follows:

```
<IfModule security2_module>
# Default Debian dir for modsecurity's persistent data
SecDataDir /var/cache/modsecurity

# Include all the *.conf files in /etc/modsecurity.
# Keeping your local configuration in that directory
# will allow for an easy upgrade of THIS file and
# make your life easier
IncludeOptional /etc/modsecurity/*.conf

# Include OWASP ModSecurity CRS rules if installed
IncludeOptional /usr/share/modsecurity-crs/*.load
</IfModule>
```

As you can intuitively see, except for the initial section about the definition of the directory for storing persistent data, this configuration file is just composed of two main sections: one for including all other configuration files, and one for including the OWASP CRS.

For what concerns the former, that means Apache will include all the `*.conf` files present in `/etc/modsecurity/` directory. You should go to that directory to see them:

```
cd /etc/modsecurity/
```

There, you should rename the `modsecurity.conf-recommended` file to `modsecurity.conf`:

```
sudo mv modsecurity.conf-recommended modsecurity.conf
```

Then, you can open it with an editor:

```
sudo gedit modsecurity.conf
```

In that file, find the following line:

```
SecRuleEngine DetectionOnly
```

This such configured line tells ModSecurity to log HTTP transactions, but takes no action when an attack is detected. In order to request ModSecurity to detect and block web attacks, this line should be changed to the following:

```
SecRuleEngine On
```

However, for now you can keep this line as it was, i.e., you can keep ModSecurity in `DetectionOnly` mode. In this way, you will start the lab activity about ModSecurity without any actual protection against attacks.

Then, find the following line, which tells ModSecurity what information should be included in the audit log:

```
SecAuditLogParts ABDEFHIJZ
```

However, this default setting is wrong. The setting should be changed to the following:

```
SecAuditLogParts ABCEFHKZ
```

Save and close the file. Then restart Apache for the change to take effect. (Reloding the web server is not enough.)

```
sudo systemctl restart apache2
```

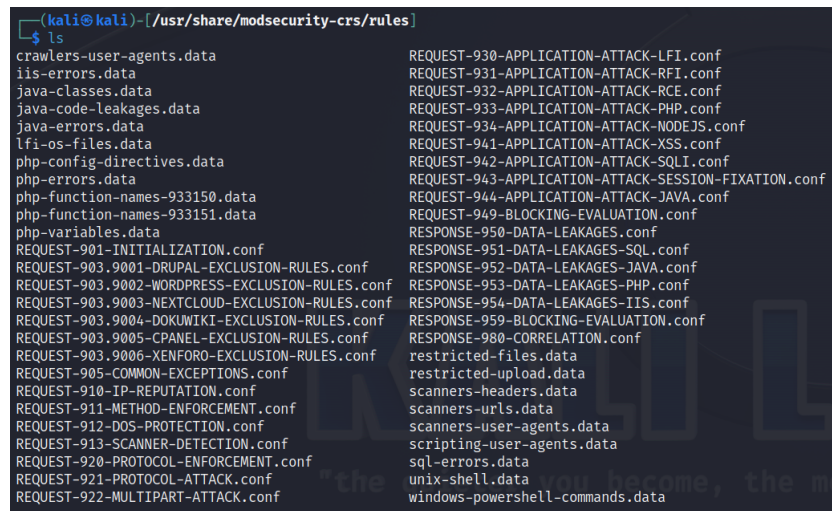
For what concerns the latter (i.e., the section about OWASP CRS), that means Apache will include all the `*.load` files present in `/usr/share/modsecurity-crs/` directory. You should go to that directory to see them:

```
cd /usr/share/modsecurity-crs/
```

There, you can find the `owasp-crs.load` file, which in turn loads all the `*.conf` files present in `/usr/share/modsecurity-crs/rules/` directory. So, go there:

```
cd /usr/share/modsecurity-crs/rules/
```

With the `ls` command, you will see all the already installed OWASP CRS. You can see a screenshot of what you should see in Figure 3.5.



```
(kali@kali)-[/usr/share/modsecurity-crs/rules]
$ ls
crawlers-user-agents.data      REQUEST-930-APPLICATION-ATTACK-LFI.conf
iis-errors.data               REQUEST-931-APPLICATION-ATTACK-RFI.conf
java-classes.data            REQUEST-932-APPLICATION-ATTACK-RCE.conf
java-code-leakages.data      REQUEST-933-APPLICATION-ATTACK-PHP.conf
java-errors.data             REQUEST-934-APPLICATION-ATTACK-NODEJS.conf
lfi-os-files.data            REQUEST-941-APPLICATION-ATTACK-XSS.conf
php-config-directives.data    REQUEST-942-APPLICATION-ATTACK-SQLI.conf
php-errors.data              REQUEST-943-APPLICATION-ATTACK-SESSION-FIXATION.conf
php-function-names-933150.data REQUEST-944-APPLICATION-ATTACK-JAVA.conf
php-function-names-933151.data REQUEST-949-BLOCKING-EVALUATION.conf
php-variables.data           RESPONSE-950-DATA-LEAKAGES.conf
REQUEST-901-INITIALIZATION.conf RESPONSE-951-DATA-LEAKAGES-SQL.conf
REQUEST-903.9001-DRUPAL-EXCLUSION-RULES.conf RESPONSE-952-DATA-LEAKAGES-JAVA.conf
REQUEST-903.9002-WORDPRESS-EXCLUSION-RULES.conf RESPONSE-953-DATA-LEAKAGES-PHP.conf
REQUEST-903.9003-NEXTCLOUD-EXCLUSION-RULES.conf RESPONSE-954-DATA-LEAKAGES-IIS.conf
REQUEST-903.9004-DOKUWIKI-EXCLUSION-RULES.conf RESPONSE-959-BLOCKING-EVALUATION.conf
REQUEST-903.9005-CPANEL-EXCLUSION-RULES.conf RESPONSE-980-CORRELATION.conf
REQUEST-903.9006-XENFORO-EXCLUSION-RULES.conf restricted-files.data
REQUEST-905-COMMON-EXCEPTIONS.conf restricted-upload.data
REQUEST-910-IP-REPUTATION.conf scanners-headers.data
REQUEST-911-METHOD-ENFORCEMENT.conf scanners-urls.data
REQUEST-912-DOS-PROTECTION.conf scanners-user-agents.data
REQUEST-913-SCANNER-DETECTION.conf scripting-user-agents.data
REQUEST-920-PROTOCOL-ENFORCEMENT.conf sql-errors.data
REQUEST-921-PROTOCOL-ATTACK.conf unix-shell.data
REQUEST-922-MULTIPART-ATTACK.conf windows-powershell-commands.data
```

Figure 3.5: List of already installed OWASP CRS.

In particular, check that there are the two about SQL Injection and Cross Site Scripting attacks:

```
REQUEST-941-APPLICATION-ATTACK-XSS.conf
REQUEST-942-APPLICATION-ATTACK-SQLI.conf
```

Now you are finally ready to practice with ModSecurity, so as to see how it can protect the Apache server against an SQL Injection attack and a Cross Site Scripting attack.

3.3 Protection against SQL Injection

In this first exercise with ModSecurity, you will use it to protect the Apache server against an SQL Injection attack. An SQL Injection attack consists in the insertion or “injection” of an SQL query via the input data from the client to the application, so as to read sensitive data from the database, modify database data, execute administration operations on the database, or to perform other similar operations on the database.

First of all, you need to set up a vulnerable page on the website hosted by the Apache server. For this purpose, create a php file named `login.php` in the Apache document root (`/var/www/html/`).

```
sudo gedit /var/www/html/login.php
```

The content of this php file must be the same as the `login.php` file that has been uploaded as additional material on the `didattica.polito.it` website, alongside with this document. This script will display a login form, as shown in Figure 3.6. Entering the right credentials will display the message: “You logged in with valid credentials!”. Entering the wrong credentials will display the message: “Invalid username or password!”.

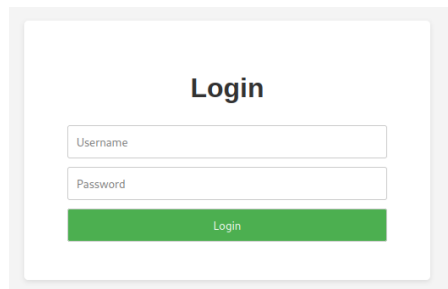
The image shows a web form titled "Login" in bold black text. Below the title are two white input fields with thin grey borders. The first field is labeled "Username" and the second is labeled "Password". Below these fields is a solid green rectangular button with the word "Login" in white text. The entire form is centered within a light grey rectangular frame.

Figure 3.6: Login form.

In order to have successful logins, however, you need to have a functional database. For this activity, you can use a simple `mysql` database.

You can run `mysql` with the following command:

```
sudo service mysql start
```

Then, you need to set the password authentication method for the `root` user in `mysql`, instead of the default `unix_socket` authentication. To do so, open the `mysql>` prompt with `sudo` privileges with the following command:

```
sudo mysql
```

In the prompt, you can alter the authentication method with the following command, which sets the `secret` password for the `root` user:

```
ALTER USER 'root'@'localhost' IDENTIFIED BY 'secret';
```

You can then exit from that procedure:

```
quit;
```

At this point, you need to have some credentials in the database. So, you must create a MySQL database and a table, then insert usernames and passwords. To do so, open the `mysql>` prompt without `sudo` privileges with the following command:

```
mysql -u root -p
```

The requested password is `secret`. Then, in the prompt, you can create the database and connect to it:

```
create database db;
connect db;
```

Next, you can create a users database, and you introduce a pair of credentials:

```
create table users(username VARCHAR(100),password VARCHAR(100));
insert into users values('beatrice','seagulls ');
insert into users values('ai','45510');
```

You can then quit the mysql> prompt:

```
quit;
```

At this point, you are recommended to check that the php script works. Open your browser, navigate to <http://localhost/login.php> and enter a right pair of credentials:

```
Username: beatrice
Password: seagulls
```

You will see the message that indicates successful login. Now come back and enter a wrong pair of credentials – you will see the message Invalid username or password. These two messages confirm you that the script works right.

Now you will try to authenticate to the server without using a right pair of credentials, but by exploiting an SQL Injection attack.

Return to the login page, and enter the following for the username field:

```
' or 1=1 --
```

Note that there should be a space after `--`. Otherwise, this injection attack will not work without that space. Insert any string in the password field empty (e.g., “unknown”) and hit the login button. You will see that the script shows the message meant for authenticated users.

How was it possible? Let us have a look at the SQL query that is included in the php script:

```
SELECT * FROM 'users' WHERE username='$username' AND password='$password'
```

The “username” and “password” variables are replaced at run time with the values the user introduce in the form. Replacing them with the values used in the attack, we have the following:

```
SELECT * FROM 'users' WHERE username='' or 1=1 -- ' AND password='unknown'
```

The `--` symbol is used to comment everything that follows. So, the SQL query that is actually executed is the following:

```
SELECT * FROM 'users' WHERE username='' or 1=1
```

Basically, any entry of the `users` table is retrieved, if the `username` field is empty *or* if `1=1`. However, the `1=1` identity is always true... so all the table entries are retrieved.

There are clearly many faults in this login page: for example, there is no check on the exact number of retrieved entries (i.e., the page only checks if that number is 0 or not), and there is no sanitization of the inputs (which is always fundamental). But the purpose was to have such a web page, subject to SQL Injection attacks.

Now, it is finally time to use ModSecurity to block this attack. Open again its configuration file with an editor:

```
sudo gedit /etc/modsecurity/modsecurity.conf
```

In that file, modify the `SecRuleEngine` mode as follows:

```
SecRuleEngine On
```

After saving the file, restart the Apache server for the change to take effect:

```
sudo systemctl restart apache2
```

Try again the SQL attack after refreshing the login page... and now the attack is blocked, as shown in Figure 3.7.

Forbidden

You don't have permission to access this resource.

Apache/2.4.57 (Debian) Server at localhost Port 80

Figure 3.7: Confirmation that the SQL Injection attack is blocked.

Differently from the activity with Squid, here in ModSecurity you did not have to define any specific rule, it was enough to have the OWASP CRS rule for SQL Injection to block the attack. Indeed, ModSecurity offers much more powerful filtering features with respect to Squid.

3.4 Protection against Cross Site Scripting

In this second exercise with ModSecurity, you will use it to protect the Apache server against a Cross Site Scripting (XSS) attack. XSS attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.

Before starting this exercise, you are recommended to bring ModSecurity back to the configuration where it only logs attacks without stopping them. Therefore, open its configuration file with an editor:

```
sudo gedit /etc/modsecurity/modsecurity.conf
```

In that file, modify the `SecRuleEngine` mode as follows:

```
SecRuleEngine DetectionOnly
```

After saving the file, restart the Apache server for the change to take effect:

```
sudo systemctl restart apache2
```

Now you need to set up a vulnerable page on the website hosted by the Apache server. For this purpose, create a php file named `welcome.php` in the Apache document root (`/var/www/html/`).

```
sudo gedit /var/www/html/welcome.php
```

The content of this php file must be the same as the `welcome.php` file that has been uploaded as additional material on the `didattica.polito.it` website, alongside with this document. This script will display a simple “Welcome” page. That page prints the “Welcome `<string>!`” sentence, where `<string>` is the string passed as value of the *name* query parameter. For example, if you connect to `http://localhost/welcome.php?name=Emilia`, then the “Welcome” page will print “Welcome Emilia!”, as also shown in Figure 3.8.

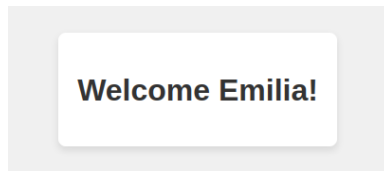


Figure 3.8: Welcome page.

Let us have a look at the php code of the “Welcome” page:

```
<?php
$name = $_GET['name'];
echo "<h1>Welcome $name!</h1>";
?>
```

Unfortunately, an attacker can easily inject executable code as value of the *name* query parameter. There is no check about what a user may pass as that query parameter, and this is immediately evaluated with the `$` operator in the php code.

Here, for simplicity, we suppose to carry out a “benign” attack, where the attacker only makes an alert box pop up in the “Welcome” page. However, you may imagine that other types of real, worse attacks may instead be executed, in the very same way...

For example, try to connect to:

```
http://localhost/welcome.php?name=Emilia<script>alert('Attacked!')</script>
```

You will see that an alert box pops up, telling you that you have been attacked, because external code has been executed in your own web server, as shown in Figure 3.9.

Now, it is again time to use ModSecurity, through OWASP CSR, to block this other attack. Open again its configuration file with an editor:



Figure 3.9: “Attacked!” alert box.

```
sudo gedit /etc/modsecurity/modsecurity.conf
```

In that file, modify the `SecRuleEngine` mode as follows:

```
SecRuleEngine On
```

After saving the file, restart the Apache server for the change to take effect:

```
sudo systemctl restart apache2
```

Try again the XSS attack after refreshing the “Welcome” page... and now the attack is blocked, as shown in Figure 3.10.

Forbidden

You don't have permission to access this resource.

Apache/2.4.57 (Debian) Server at localhost Port 80

Figure 3.10: Confirmation that the XSS attack is blocked.

Again, in ModSecurity you did not have to define any specific rule, it was enough to have the OWASP CRS rule for XSS to block the attack.