Table of Contents

Overview

This tutorial is not on how to secure a web server using fancy configuration files, but is primarily intended in documenting how you can stop hackers and bots dead in the water before they can query too much of your website simply.

It is intended for advanced Linux users that are comfortable with manually editing firewall rules; not a requirement but you will have difficulty in following the document if you are not familiar with firewall rules.

It is intended for Linux users that run apache and use either iptables or firewalld on their web server. ***In this iteration of the document the focus is on iptables as that is what I use***.

Anyone who has run their own web server for more than a few weeks will already have seen in their web server access logs attempts to access resources such as ‘phpmyadmin’ searching for packages you may have installed, and binary or large query strings trying to cause buffer overflows on their webserver. The goal of this document is to show you an easy way to trap them and blacklist the requesters ip-address.

Apache ReWrite rules, why not to use them

There are many tutorials out there on the internet on how to handle the first, a search for installed applications. The large majority of them focus on using apache ReWrite rules to direct such requests to custom error response pages or to cgi scripts to add firewall blacklist rules for the source of the request.

The issue I have with this approach is that it requires you to know in advance what the query url will be in order to create a ReWrite rule for it so this approach is not a valid protection.

These days bots and hacker toolkits use a wide range of url requests, you need a way to trap them all which is not possible with apache rewrite rules that need a known string to match on. How I do that is discussed a little below.

Pre-requisites

A local firewall on your system

You will need either iptables or firewalld installed and running on your web server in order to blacklist machines that are running hacking attempts against your webserver.

You will also need to be comfortable with the firewall rules already in place, as you will need to modify them.

A stable website with no broken links in it

This is a requirement as the method discussed here will blacklist any source ip-address that tries to access a page that does not exist; and you will end up blacklisting search engines if you have broken links in your website.

As noted above the method I discuss here will blacklist any source that tries to obtain a web page that does not exist.

One issue with this is if you deliberately remove a page from your website it will affect search engines. Search engines such as google and bing when re-indexing a site do not start at the top and work their way through the site again, they will go directly to a page they ‘remember’ to re-index it and if the page has been deleted they will trigger the blacklist scripts.

So you must regularly review your logs and for reasonable urls where nslookup shows the ip-address belongs to a search engine you must manually un-blacklist that address.

Also see the “Things you must take into consideration” section when iptables are covered, as those considerations apply to any automated blacklisting method you choose.

Note: For checking your website for broken links it is probably easiest to use a tool such as OWASP. The easiest way to use that is to obtain Kali Linux where it is an option in the web testing section as owasp-zap and will just work.

An overview of how automated blacklisting can be implemented for an apache web server

An overview of how dynamic blacklisting using apache can be done

As anyone who has run their own web server will have seen from their access logs there are machines out there that will request many different pages that do not exist over a few seconds; likewise fuzzing attempts from script kiddies using pre-canned scripts to test for buffer overflows will send requests for A*100 them A*200 then A*300 etc and you want to stop them at the first request.

To stop them all simply treat any request for a page that does not exist on your web server as a attempt to hack into your web server and blacklist the address in your firewall in real time. This will catch all attempts to access unknown applications and buffer overflow/fuzzing attempts on the first request they make.

The trigger for capturing these attempts is already provided by apache with its custom error page facility. While normally used for throwing up a custom page with pretty pictures and a ‘sorry page not found’ message rather than have a static error page you can create an error page that is either a php page or a less safe cgi-script and use those to handle the error. On sites that use php already you are probably already using php pages to build your static response page.

As these error pages have all environment variables needed to perform blacklisting, namely the source ip-address and url that was requested, these error pages as programmed pages rather than the static pages can be used to dynamically add a blacklist rule for the source ip-address on the first trigger; so instead of seeing many requests over a few seconds in the logs from an ip-address you will see just the first request with all other requests blocked.

A special note for users of firewalld

Up until CentOS7/RHEL7/Fedora31 firewalld used iptables as the back-end for managing its rules, allowing the iptables commands to be used for blacklisting regardless of whether you used native iptables or firewalld.

From CentOS8/RHEL8/Fedora32 onward firewalld uses netfilter instead of iptables as the back-end, as such the procedures documented for use using iptables will not work on those releases. Users have a choice of learning how to use complicated firewall-cmd commands or even more complicated native nft commands to achieve the same thing.

The details, for iptables

Creating the blacklist chain

Most iptables rules are setup to pass traffic from the local network down one chain, and from external networks down another chain.

While it may therefore seem the best option is to place the blacklist rules in the chain used for external traffic I prefer to place the my blacklist chain where it handles all input traffic. The benefit of doing so is that the rules can be tested from the local network.

Iptables are generally loaded at boot time from either a static/saved configuration or from a script you manually maintain that is run at server boot time. While either method achieves the same result

I prefer to use the rc.firewall script method as

1. I deploy it using puppet to make it easy to deploy as needed
2. it is easy to restore from backups of the system should it be needed
3. and the main reason for me, it is possible to *include a lot of comments* in the script documenting the flow of traffic through the chains and what each port permitted is actually used for

If you rely on a saved configuration method I hope you have a lot of documentation lying around as depending on complexity dumping out the rules from a saved configuration may not be much help in determining what your firewall is doing.

Dynamically blacklisting hackers attacking an Apache web server

In my iptables configuration script I have added code as below

```
iptables -N logdrop3                        # see note3 below
iptables -A logdrop3 -j LOG --log-prefix "BLACKLIST " \
   --log-level 3 --log-ip-options --log-tcp-options \
   --log-tcp-sequence
iptables -A logdrop3 -j DROP

iptables -N blacklist
if [ -f /some/dir/blacklist.sh ];          # See note2 below
then
   bash /some/dir/blacklist.sh
fi
iptables -A blacklist -j RETURN
iptables -A INPUT -j blacklist             # see Note1 below
```

The code above creates a chain named “blacklist”, ensures the last line of the blacklist chain is a command to return from that chain if no rules match to resume normal rule processing, and then direct all input through the blacklist chain.

Note1: the -A option is used to append this to any existing INPUT chain rules; I obviously have some rules above this code in my rc.firewall script to allow local loopback traffic.

Note2: the code here checks to see if we have a list of blacklist commands that need to be loaded already predefined and runs the command file if found. How this file is created is covered in the custom error page section.

Note3: the logdrop3 chain I added simply to log the attempts to access the server from blacklisted addresses, you will see later on that when I blacklist an ip-address I send it to that chain to log and drop, you may prefer to just drop it and avoid the extra chain. That is discussed in the custom error page script section.

Once this rule is in place we have a clearly defined place to store the blacklist rules.

The advantages of using a separate chain for the blacklist entries are

- self documenting as to what the rules are for
- rules can be inserted/removed from that chain without affecting any other complicated chaining rules you may have
- in the case of issues the rules for that one chain can be flushed without affecting any other rules with a simple “iptables -F blacklist;iptables -A blacklist -j RETURN” to remove all the rules and re-add the return rule.

To use something similar if you only use a static configuration you will need to identify where in your configuration to **insert** a INPUT rule to jump to a blacklist chain and define an empty blacklist chain with a return. After which you will be able to dynamically insert entries into the blacklist chain as I discuss here (remembering to save your changes a lot during server operation) .

You would not need to redefine the blacklist chain or load any separately saved blacklist entries at server boot time as your static saved configuration will contain them.

Creating the custom error pages

It is important to note that when creating custom scripted error pages the apache server may (and in most cases will not) send the expected response code for a failed url request.

For example is a page not found (404) error should be sent if you return a perfectly formatted scripted page the requester will actually get a OK (200) response in most cases. For static html error pages apache can insert the correct headers, in a scripted page whether PHP or BASH where you are creating the entire response page and inserting your own headers the default will be OK.

This may not be an issue for you, as some sites such as “null-byte” explicitly return OK (200) for pages that are not found in order to confuse hackers that are trying to investigate their site (but it must play havoc with search engines).

However I believe a custom error page should return the correct code in most cases, and it is extremely easy to do in both php and bash as follows

For PHP

```
http_response_code(404);
```

For BASH shell script before any html code is sent (yes you do need two blank lines)

```
cat << EOF
Content-Type: text/html
Status: 404 Not Found

EOF
   fi
```

It is probably preferable to use PHP for a custom error page to take benefit of its functions to escape any data encoded in the url string. However if you wish to write your error page entirely in BASH shell code it is worth noting that all the server environment variables available to PHP are also available to BASH (for example $_SERVER[‘REQUEST_URI’] in PHP is available to BASH as the environment variable $REQUEST_URI) so if you prefer coding in bash to PHP it is simple to do so.

For simplicity, as more people are able to code in BASH than in PHP most of the code here will be for bash.

However as I strongly recommend if you intend to do anything with the URL requested it should as least be passed through a few functions to sanitise the input the examples here are actually a PHP front-end which then passes that data to a bash script.

Dynamically blacklisting hackers attacking an Apache web server

This is a PHP front-end than can be used for a not found (404) page. The key points are that it escapes characters in the url before risking passing then to exec and the bash script.

```
<?php
/*
 This is a wrapper that sanity checks the input;
 then pass it to the shell script.
*/
http_response_code(404);
/* header("HTTP/1.0 404 Not Found"); */
$xx = escapeshellcmd( $_SERVER['REQUEST_URI'] );
$xx = escapeshellarg( $xx );

/* note the below must be on one line */
$cmd = "/var/www/cgi-bin/error_404_handler.sh '".$xx."' '".
$_SERVER['REMOTE_ADDR']."' '".$_SERVER['REQUEST_METHOD']."'";

exec( $cmd, $output );
foreach( $output as &$xx ) {
   echo $xx;
}
?>
```

This is a shell script that can be invoked by the above front-end. The key points to note in the script itself will be covered below the script. ***This is an extremely cut-down version of a script that can be used, see the notes in the section "things you must take into consideration" on things I do that you also must do to make the script usable without adverse effect***.

```
#!/bin/bash
#
# Called by a redirect rule when a website user tries to access
# something we do not have installed (ie: phpMyAdmin and Mail seem
# to be targeted by people trawling the site).
#
# (1) if not in the blacklist file
#     (a) add to the blacklist file  (with a datestamp at the
#         end of the command so we can clear old entries monthly
#         the blacklist file can be used to append the iptables
#         rules to the custom rule table during server reboots
#     (b) log that we have blacklisted it and what they were
#         trying to do
# (2) if already in the blacklist file
#     (a) we should not have triggered, log the event for
#         investigation
#
# Note: this script will issue the iptables command to add the
#       block rule immediately, so the webserver user needs to
#       be authorised to sudo the iptables command.

BLACKLIST="/some/dir/blacklist.sh" # This is embedded in rc.firewall startup
LOGFILE="/some/dir/blacklist.log"  # This must be checked for bad blacklists
myname=`basename $0`               # for log messages
datenow=`date`                     # for log messages
```

```
datestamp=`date +"%Y%m%d"`    # datestamp iptable rules (for easy cleanup)

# If called from the php wrapper with the variables we want
# then use those as we will not have server globals availabe
# as we would if called directly by apache.
PHP_WRAPPER="NO"
test1="$1"
test2="$2"
test3="$3"
if [ "${test2}." != "." ];
then
   REQUEST_URI="${test1}"
   REMOTE_ADDR="${test2}"
   REQUEST_METHOD="${test3}"
   PHP_WRAPPER="YES"
fi

# ---------------------------------------------------------------
# helper proc: write a message to the logger and logfile
# ---------------------------------------------------------------
log_proc() {
   msg="$*"
   logger "${myname}:${msg}"
   echo "${datenow}:${msg}" >> ${LOGFILE}
} # end log_proc

# ---------------------------------------------------------------
# helper proc: blacklist the ipaddr in iptables, save the command
#              in the blacklist file, so it can be re-applied each
#              system restart
# ---------------------------------------------------------------
do_blacklist() {
   # record into the blacklist file to use on system restarts
   echo "/sbin/iptables -w3 -A blacklist -j logdrop3 -s ${REMOTE_ADDR}" >> $
{BLACKLIST}
   # log that we have blacklisted this ipaddr
   log_proc "ipaddr ${REMOTE_ADDR} requested ${REQUEST_URI}, method $
{REQUEST_METHOD}, blacklisted (404)"
   # can be run by apache with no password (configured in sudoers file)
   /usr/bin/sudo /sbin/iptables -w3 -I blacklist -j logdrop3 -s $
{REMOTE_ADDR}
} # end do_blacklist

# ---------------------------------------------
# A hacking attempt then.
# Display a web page back to the requesting user
# Yes it needs the blank line after the content type.
# Do this before adding the firewall rule or
# the hacker will never see it.
# ---------------------------------------------
if [ "${PHP_WRAPPER}." == "NO." ];
then
cat << EOF
Content-Type: text/html
Status: 404 NotFound
```

```
EOF
fi
cat << EOF
<html>
<head><title>Page not found on this server</title></head>
<body bgcolor="yellow">
<h1>Page not found on this server</h1>
<hr>
<p>
You have requested a page not provided by this website. Unfortunately this
generally indicates a hacking attempt.
</p>
<p>
You requested: ${REQUEST_URI}
</p>
<p>
Your ip-address <b>${REMOTE_ADDR}</b> has been automatically added to this
sites blacklist.
<br />
You can no longer view any content from this site.
</p>
</body>
</html>
EOF


# ---------------------------------------------
# Now block all traffic to and from the ipaddress
# that triggered this script.
# If the blacklist file exists
#   - check if the entry exists, we do not want duplicates
#   - if the entry does not exist add it
#   - if the entry does exist we should not have triggered, log warning
# ---------------------------------------------
if [ -f ${BLACKLIST} ];
then
   isfound=`grep \'${REMOTE_ADDR}\' ${BLACKLIST}`
   if [ "${isfound}." = "." ];
   then
      do_blacklist
   else
      log_proc "*WARN* blacklisted ipaddr ${REMOTE_ADDR} reached webserver,
investigate"
   fi
else
   do_blacklist
fi

# All done
exit 0
```

The above script explained

- constants, BLACKLIST is where we store the iptables commands to use, LOGFILE is where we log activity for historical review
- The start of the script is a quick test to see if we were called with parameters (such as from the php script which passed the details we need) or if we were called directly by apache and have the environment variables we need. The PHP_WRAPPER flag is used is we were called by the php script to indicate we do not need to write the page headers
- note in the do_blacklist routine when writing the iptables command to the script file we use -A as when the script file is run from the rc.firewall startup script it adds a lot of entries in sequence before we finally add the -j RETURN command; but when we actually dynamically add the drop command to the blacklist rule we use -I to insert the rule (as if we appended we would add the drop after the return statement where it would never be triggered)
- for apache to run the iptables command it must be configured in the sudoers file
- we write the response page to the requester before adding the drop rule, if we added it earlier they would never see the response
- we check if the ip-address is already in our blacklist file before adding it, as if it was already there we have an issue to investigate as the request should have been dropped
- the iptables command jumps to the logdrop3 chain to log and drop any future requests, if you do not want to log just “-j DROP” instead of “-j logdrop3” and you do not need to define the logdrop3 chain at all

The above example script is not production-ready. A lot of important logic has been removed in order to make the script as clear as possible to follow. See the section on “Things you must take into consideration” for the additional things you should be adding into this script.

Sudoers requirements

For commands using iptables the apache user must be able to sudo to run the commands. This needs to be limited as much as possible to the exact command actually needed to add the iptables rule.

Also with the webserver running the command it must be able to be done without a prompt for a password.

This is an example of the sudoers entry needed is the webserver runs under the apache userid.

```
Defaults:apache !requiretty
apache vosprey4=NOPASSWD: /sbin/iptables -w3 -I blacklist -j DROP -s *
```

Using firewalld instead of iptables

Firewalld is not as friendly to configure as iptables, and is not easy to implement internal/external traffic splitting rules within a zone. So if using firewalld it is best to use the predefined zones.

First you need to determine what zones you are using, and how your interfaces are laid out.

First use “ifconfig -a” (in the net-tools package) or “ip a” to determine what interfaces are in your system.

Then for each use “firewall-cmd --get-zone-of-interface ifacename” to see what zone each interface is attached to.

Then use “firewall-cmd –get-active-zone” to determine what zones are in use.

Also use “firewall-cmd --get-default-zone” to determine the default zone for commands; although to be safe you should always specify a target zone for all update commands to avoid issues.

At this point you may want to adjust the zone in which interfaces belong to and change the active zone; or just use the already active zone, totally up to you. You may even want to create a custom zone. All are easy to do but as noted at the start of this document this is not for novices and the steps to do so are not covered here.

It is also important to note you cannot add a “blacklist” zone, as any drop rules can only be added to zones that contain network interfaces.

The same method as described for using iptables can be used to deny traffic from a specific ip-address on a firewalld system using a “rich” firewalld rule using the command below.

```
firewall-cmd --permanent –nameofzone \
  --add-rich-rule="rule family='ipv4' \
  source address='ip-address' drop"
```

Also run the command without the permanent command to make it immediately active without having to reload the firewall rules.

It using the script template shown above for iptables you should still write what you are blocking to a log file but there is no point in saving the commands to a command file as the –permanent option will ensure the rule survives across reboots.

Things you must take into consideration

Why the sample iptables script logic will work for either iptables (as is) or firewalld (if you change the sudo command to the appropriate firewall-cmd command) , the additional things I do that you **must** also **take into consideration** and most likely want to implement yourself are

- before the section that writes the page back to the hacker and adds the ip-address to the blacklist check the requesting ip-address
  - if from local addresses display a custom page asking the user to contact the admin to correct whatever link they followed and do not blacklist the ip-address, an internal user is most likely following a bad link you need to fix (depending on the number of users you have; you may actually want to blacklist and show a page asking them to contact the site admin to be unblacklisted)
  - have a list of addresses not to blacklist, these will be such things such as the gogglebot web crawling address range, they will try to re-index pages they know about even if the pages have been removed from the server and if they get a 404 response both google and bing ignore the 404 error and they will just try and re-index again later. For these addresses return a 410 (permanently deleted) rather than blacklist the address… and if they ignore that and still try and re-index the page then let them be blocked. Ideally you would not want to blacklist search engines but their insistence on ignoring 404 errors makes it inevitable unless you have special handling for them
- have a batch job that runs daily to sort|uniq the blacklist command file, sometimes is a hacker is hammering the site a few requests will trigger blacklist additions before the first trigger loads the drop rule (a cleanup job)
- review the blacklist log often to identify requests that do not appear to be hacking attempts but honest mistakes and unblock them
- have a check at the top of the script for any ip-address you use on the internal network for website scanning, and ‘fast exit’ the script if from that address to avoid blocking it
- in relation to the above point, scan your website whenever you make changes to identify any pages that may have links referring to non-existent pages and correct the links. You do not want to blacklist someone for innocently following a link on your site.

The main differences between iptables and netfilter

A note on this section

These observations do not in any way affect the methodology of blacklisting ip-addresses to your server. This section exists to point out the differences between the two, that may in some cases prevent your rules from being triggered.

Viewing the firewall rules

As noted earlier up until CentOS7/RHEL7/Fedora31 firewalld used iptables as its back-end. From CentOS8/RHEL8/Fedora32 onward firewalld uses netfilter as its back-end.

Iptables uses the iptables command and can dump out its rules in a simple text format that is reasonably easy to read. Netfilter uses the nft command and the rules are not so easy to read when dumped out.

Examples

```
iptables syntax to show rules
iptables -S
iptables -n -v -L

nft syntax to show rules
nft list ruleset
```

An important observation on the use of iptables and netfilter

On a CentOS8 machine where I use native iptables and do not run firewalld any rules added via iptables commands are automatically created in such a way that they are displayable as both iptables and netfilter rules. I do not have a F32 machine running only iptables so no observation there.

On Both Fedora32 and CentOS8 rules added via firewalld are displayable as netfilter rules but there are zero entries returned from iptables.

So at this point in time it may seem safer to use iptables as the rules are implemented into both solutions. However there are warnings about using both iptables and netfilter on the same system as there may be unexpected results.

However if using only firewalld as rules set for firewalld do not create iptables rules you must endure your server is configured to use only netfilter for all traffic.

An important observation for docker users

Both docker and docker-ce will add both netfilter and iptables rules regardless of which firewall solution you have chosen for you server when the docker engine is started. This is just something to be aware of.