Bypassing a Web Application Firewall

Wissam El Labban

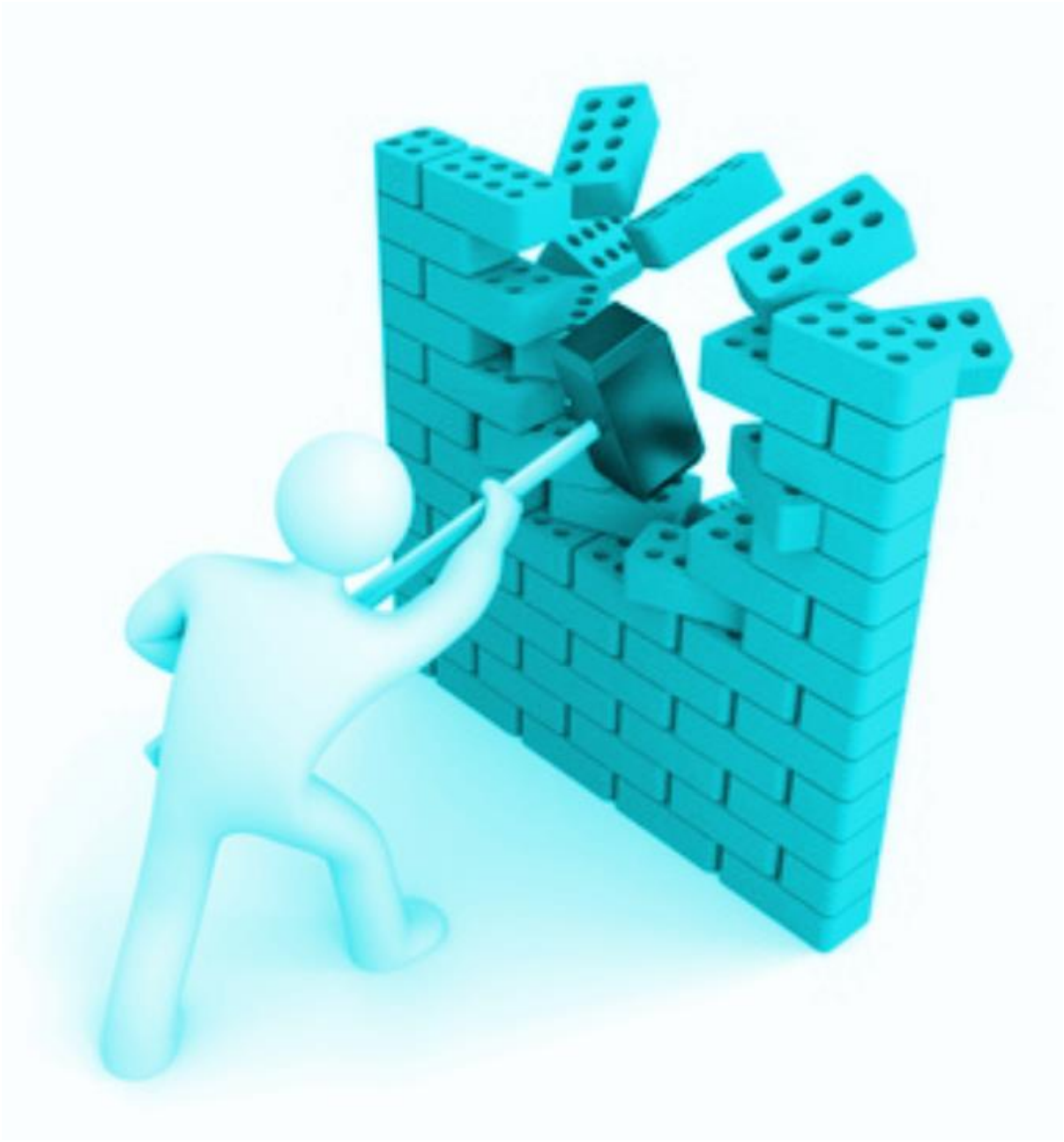


Table of contents

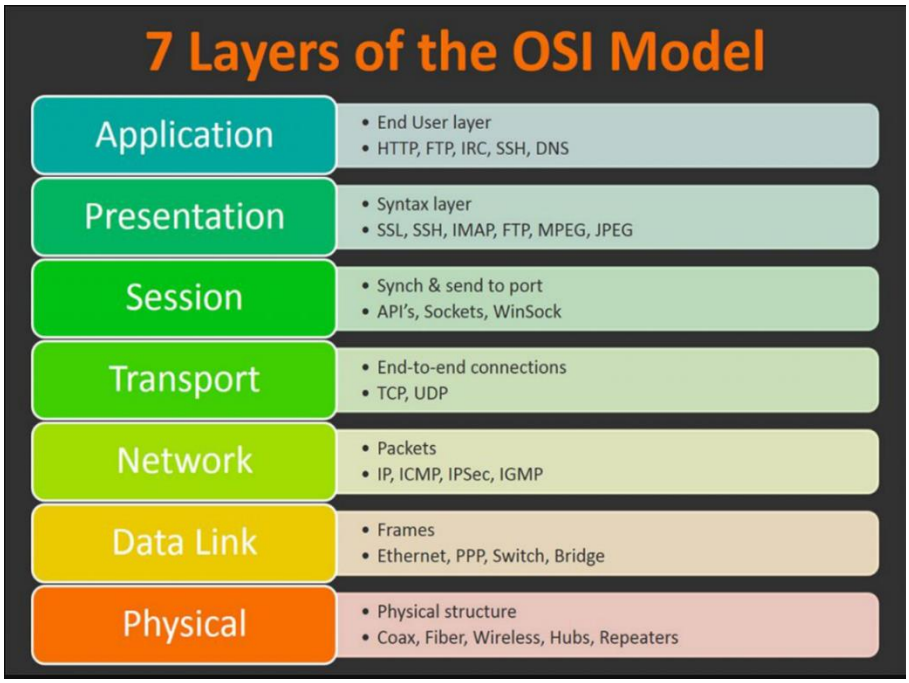
Contents

Required from reader of report

To get the full benefit of this report you will need: beginner to intermediate knowledge of networks, basic understating of ELK stack setup, and internet access.

What is a WAF

We are aware of the concept of firewalls in networks. However, those aren't enough to protect our web applications. Those are applications on web servers that host our websites that are accessible through a client's web browser. Those connections, however, are on the higher layer of the OSI model.



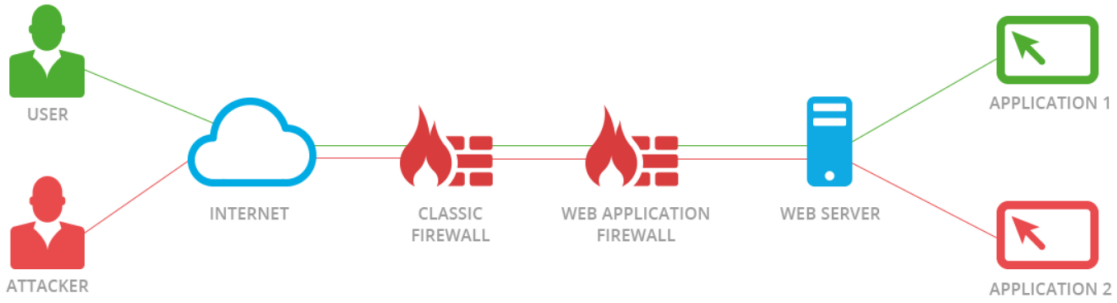
The OSI Model shown above has 7 layers, and the traditional firewall only protects from layers 3-4. This means that a regular firewall will do nothing to stop suspicious HTTP traffic. A web application firewall on the other hand works on layers 4-7 of the OSI stack. This combination of firewall and WAF (short for web application firewall) covers the entire stack and is a necessity for any self-respecting organization to mitigate risk of their networks, and web applications hosted on them.

Now that we know about WAFs, we will go into the concept of implementing one and understanding how it works on a deeper level.

How a WAF works

A WAF inspects http traffic and passes clients input parameters through its core rule set. A core rule set (CRS for short) is exactly what it sounds like. A set of rules that if violated, would block a user from processing a request that the WAF would consider dangerous.

Where should a WAF be on the network of an organization? From a network architecture perspective, a WAF should sit between a traditional firewall and a web server. A general understanding is in the image below.



A firewall will be the first line of defence to filter out known attacks on the lower level of the OSI model. If that firewall does not detect a dangerous request, then the connection will go on to the WAF. If the WAF does not detect a dangerous request, then that request will go through to the web servers.

Keep in mind that this setup is meant to mitigate risk and is not a 100% effective solution to stop any external attacker as I will demonstrate soon how a WAF can be bypassed.

Building a penetration testing tool for a WAF

This is where the fun begins. I will be building an Ideal setup for pen testing a WAF. For my WAF, I will be using mod security. This is an open-source web application firewall that can be downloaded from GitHub alongside its CRS made by the OWASP organisation. The server that mod security will sit on is an ubuntu server running the Nginx web service. Mod security which is an Apache built service will be using a Nginx connector to connect and function in tandem with Nginx.

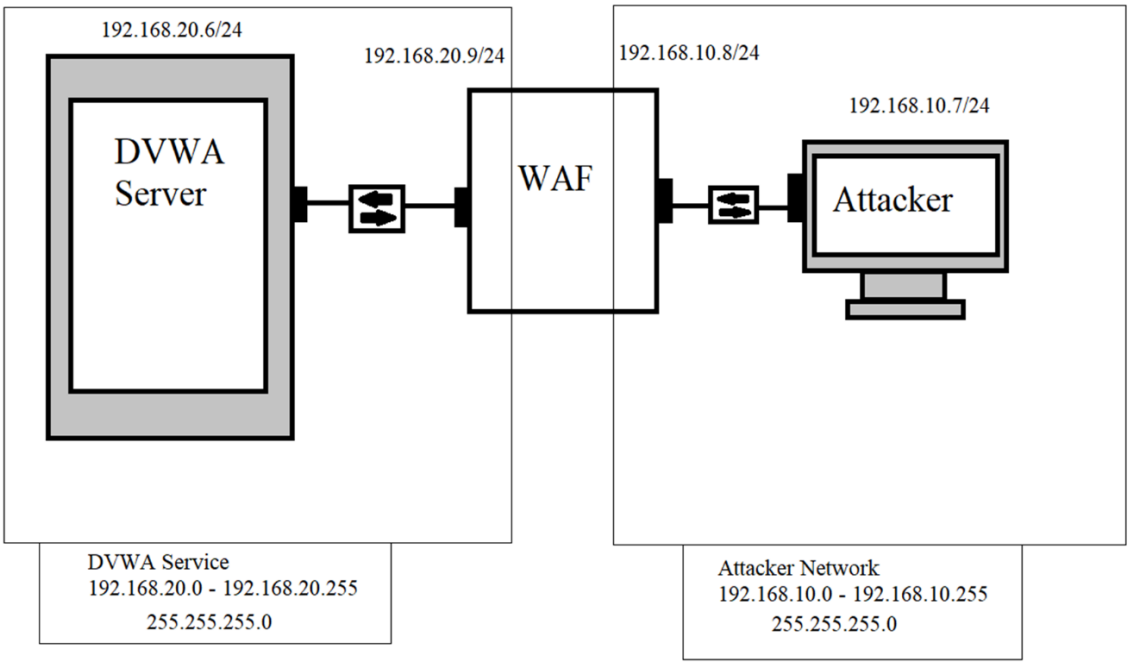
For the Web application, I will be using the DVWA web application. This again is an open-source web application that can be downloaded from GitHub. DVWA will be a website with web applications that will take parameters. There are different levels of code that protect the web application from malicious requests. For this test, I will be keeping the security of those applications on low. At that level, there is no security whatsoever on the coding layer. The only line of defence will be the mod security-nginx WAF. The DVWA server will run on kali Linux, a Debian based Linux distribution like ubuntu.

```
server {
        listen 80;
        listen [::]:80;
        server_name dvwa.com;
        modsecurity on;
        location / {
                proxy_pass http://192.168.20.6;
                modsecurity_rules_file /etc/nginx/modsec/main.conf;
        }
}
```

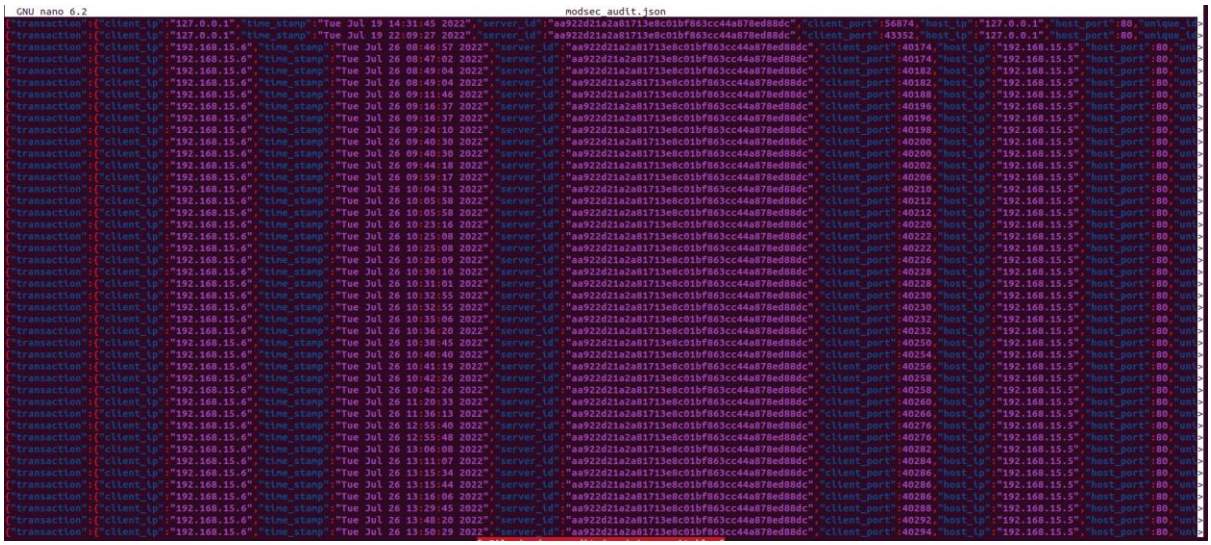
The image above is from the Modsecurity-Nginx WAF. It will be connecting to our DVWA server via a reverse proxy connection. This is a connection Nginx provides to connect to backend servers on the http level. We have enforced mod security rules on that connection to our dvwa server.

We will of course have an attacker machine. This machine will be connected to our Modsecurity-Nginx WAF. This machine will be a kali Linux machine.

Now our network looks like this:

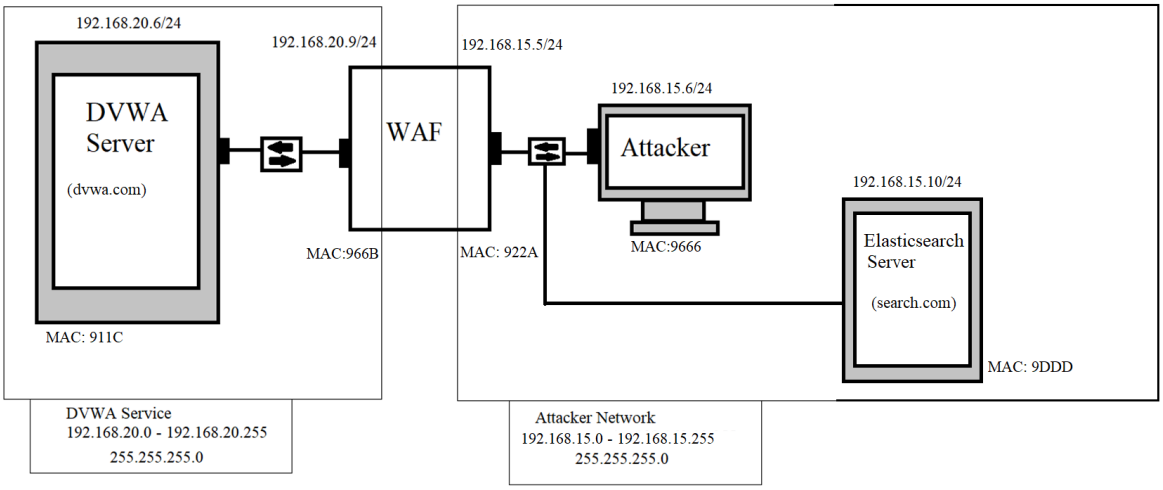


This setup may seem like enough, but in order to pen-test a WAF properly, we need to understand which rules are being violated and how to bypass them. Every time a rule is violated, and a block happens, mod security logs it in the audit log as seen below.



However, there is an overload of information that anyone would find daunting. To circumvent this issue, I will be using the Elasticsearch service to read and display logs.

Our modified network will now look like this:



The Elasticsearch server is a service running on ubuntu using the ELK stack (Elasticsearch, Logstash, Kibana). My logs will go from the mod security audit log in the WAF to the Elasticsearch server using the file beats service.

```
###################### Filebeat Configuration Example #########################

# This file is an example configuration file highlighting only the most common
# options. The filebeat.reference.yml file from the same directory contains all the
# supported options with more comments. You can use it as a reference.
#
# You can find the full configuration reference here:
# https://www.elastic.co/guide/en/beats/filebeat/index.html

# For more available modules and options, please see the filebeat.reference.yml sample
# configuration file.

# ============================== Filebeat inputs ===============================

filebeat.inputs:

# Each - is an input. Most options can be set at the input level, so
# you can use different inputs for various configurations.
# Below are the input specific configurations.

# filestream is an input for collecting log messages from files.
- type: log

  # Unique ID among all inputs, an ID is required.
  id: my-filestream-id

  # Change to true to enable this input configuration.
  enabled: true

  # Paths that should be crawled and fetched. Glob based paths.
  paths:
    - /var/log/modsec_audit.json
    #- c:\programdata\elasticsearch\logs\*
```

```
# ------------------------------ Logstash Output -------------------------------
output.logstash:
  # The Logstash hosts
  hosts: ["192.168.15.10:5044"]

  # Optional SSL. By default is off.
  # List of root certificates for HTTPS server verifications
  #ssl.certificate_authorities: ["/etc/pki/root/ca.pem"]

  # Certificate for SSL client authentication
  #ssl.certificate: "/etc/pki/client/cert.pem"

  # Client Certificate Key
  #ssl.key: "/etc/pki/client/cert.key"
```

The two images above are of the file beats config sending logs to the logstash component of the Elasticsearch server.

The image below is of the configuration in our /etc/logstash/conf.d/logstash.conf file receiving the modsecurity auditlog info from the logstash port, parsing json, and then sending it to elasticsearch where the information will be presented by kibana in the “modsec” indice.

```
input {
        beats {
                port => "5044"
        }
}

filter {
        json {
                source => "message"
        }
}

output {
        elasticsearch {
                hosts => "localhost:9200"
                index => "modsec-%{+YYYY.MM}"
        }
}
```

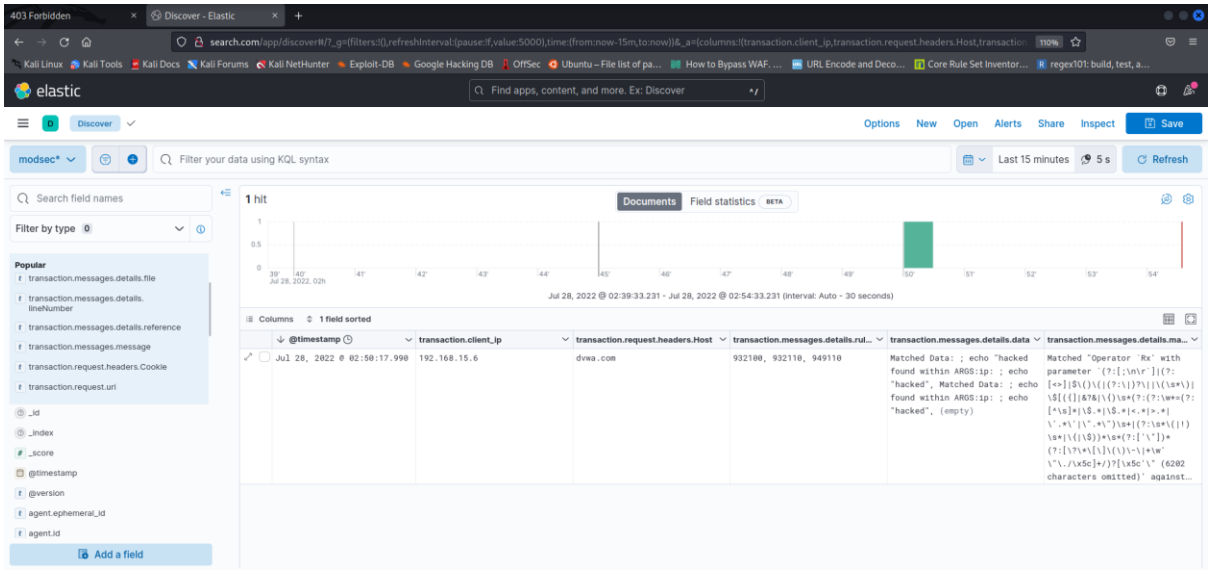
We will now have a modsec index that shows information from our modsecurity_audit log.

We do not have a DNS server on our network. So to let our attacker resolve the host names of the websites, we will go into the etc/hosts file of the attacker machine and make the following changes:

```
  GNU nano 6.2                          /etc/hosts
127.0.0.1       localhost
127.0.1.1       kali

192.168.15.5  dvwa.com
192.168.15.10 search.com

# The following lines are desirable for IPv6 capable hosts
::1     localhost ip6-localhost ip6-loopback
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Now let’s go to dvwa.com and do a violation.

Now let’s go to our mod security index.



All the information we need is presented in a beautiful format that gives us the data in real time. We can see our rule ID, and what illegal parameter got patched and to which rule as well as the order of rules that were triggered.

Our setup is now complete. Its time for some penetration testing.

Pentesting using WAF Bypass Techniques

I will be using a set of techniques from the two following sources:

https://hacken.io/researches-and-investigations/how-to-bypass-waf-hackenproof-cheat-sheet/

https://owasp.org/www-pdf-archive/OWASP_Stammtisch_Frankfurt_-_Web_Application_Firewall_Bypassing_-_how_to_defeat_the_blue_team_-_2015.10.29.pdf

The technique grading scheme will be as follows:

Grade	Meaning
fail	Complete failure
null	Bypassed but gave no output
partial pass	Bypassed but gave useless output
Pass	Bypassed and gave desired output

The pass grade is the only desirable outcome on this list.

- Command and injection:

127.0.0.1 (| |) echo "hack"

The above command was a bypass that was found at random

- SQL Injection

‘ select * from password

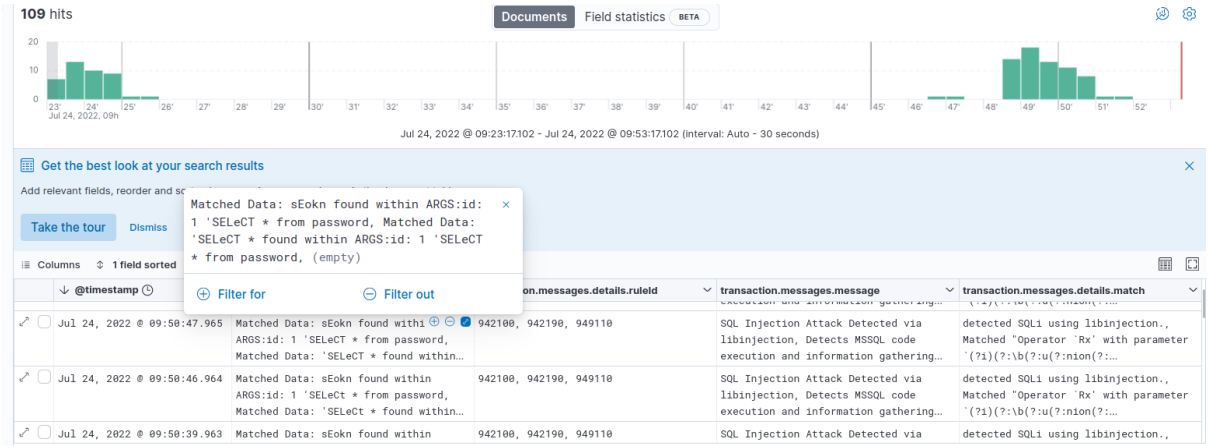
Case toggling technique: fail

```
GET /vulnerabilities/sqli/?id=1+%27§select§+*+from+password&Submit=Submit HTTP/1.1
```

Then we used every case combination of select and performed a simple list attack only to get 403 on every response (403 forbidden page by nginx WAF).

51	SEleCt	403			723	2
52	SEleCT	403			723	2
53	SElEct	403			723	2
54	SElEcT	403			723	2
55	SElECt	403			723	2
56	SElECT	403			723	2
57	SELect	403			723	2
58	SELecT	403			723	2
59	SELeCt	403			723	2
60	SELeCT	403			723	2
61	SELEct	403			723	2
62	SELEcT	403			723	2
63	SELECt	403			723	2
64	SELECT	403			723	2

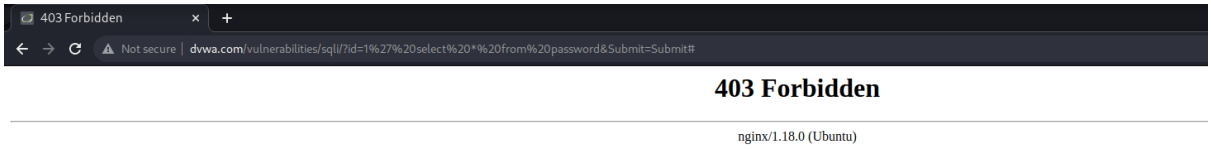
Our logs had the same rules triggered with every case of select *. This meant that case changing technigues would not help in this command



URL encoding technique: fail

We endoded our paylod into the url and found that that did not work

1%27%20select%20*%20from%20password



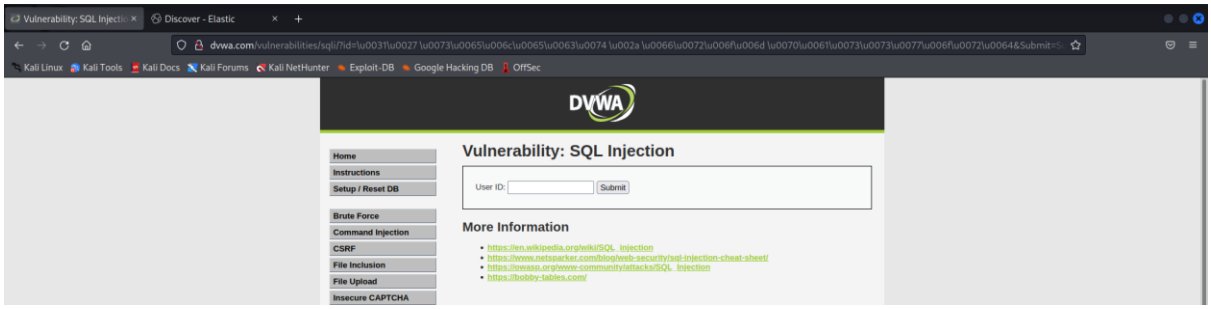
@timestamp	transaction.messages.details.da...	transaction.messages.details.rul...	transaction.messages.message	transaction.messages.details.m...	transaction.request.uri
Jul 24, 2022 @ 09:59:21.019	Matched Data: sEokn found within ARGS:id: 1' select * from password, Matched Data: ...	942100, 942190, 949110	SQL Injection Attack Detected via libinjection, Detects MSSQL code execution and...	detected SQLi using libinjection., Matched "Operator `Rx' with parameter...	/vulnerabilities/sqli/?id=1%27%20select%20*%20from%20password&Submit=Submit

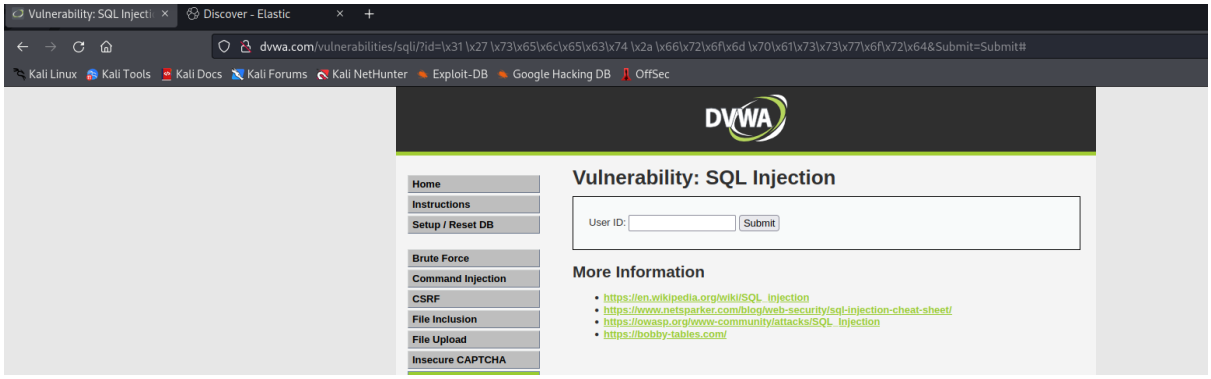
It is also worth noting that case toggling in addition to the URL encoding technique did not work since the WAF can detect all variations of select * case toggle just like we found before

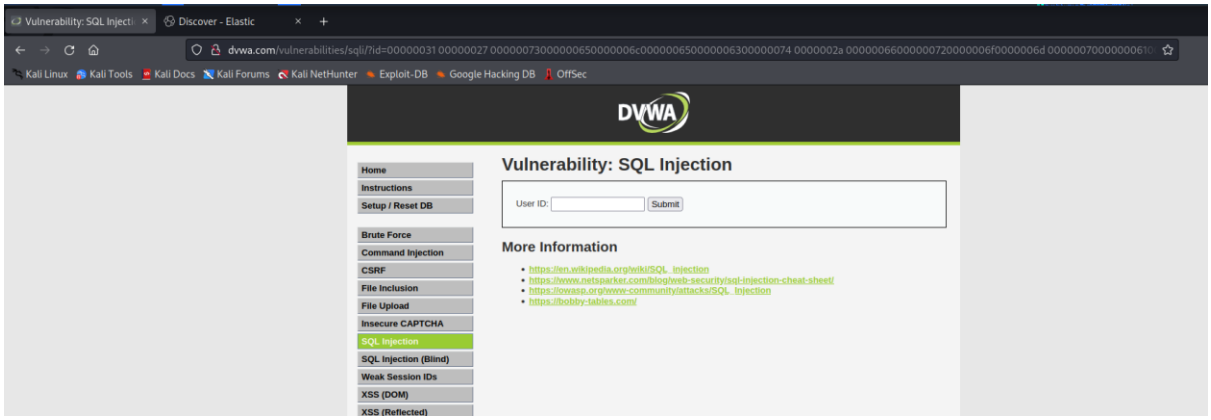
Unicode technique: null

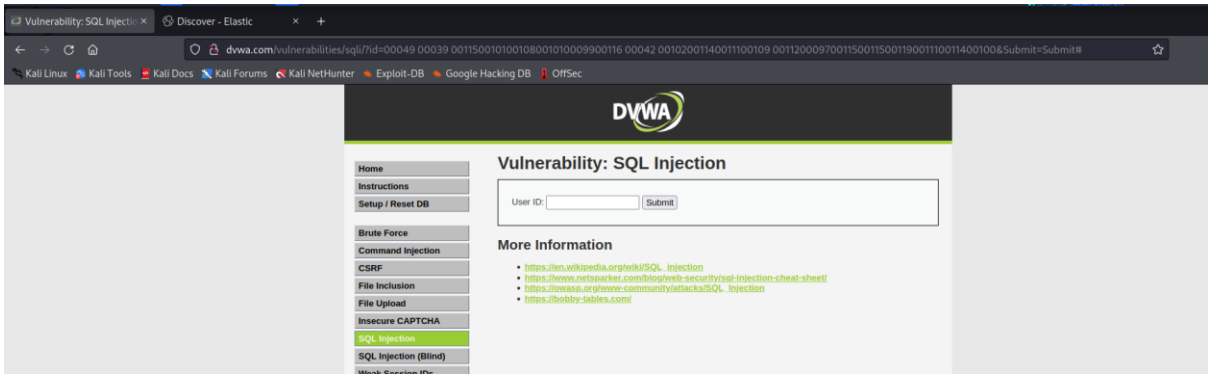
We were able to bypass the waf using the Unicode technique

We converted 1' select * from password into utf-16, utf-32, utf-8, and decimal encoding and the technique was not caught by the WAF.



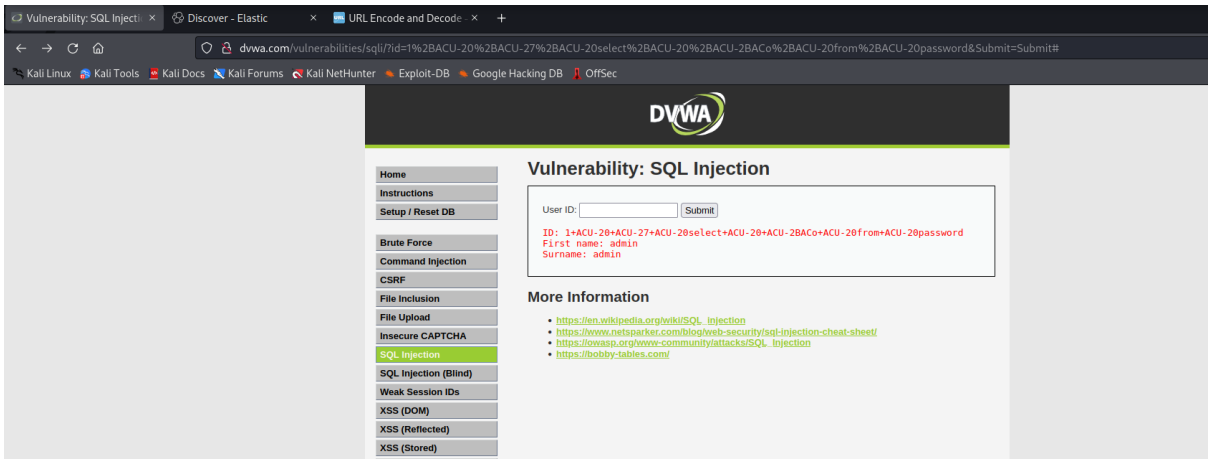






While the technique did not trigger any WAF rules, we still did not get any viable output.

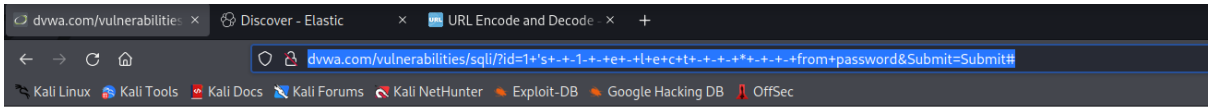
Double url encoding: partial pass



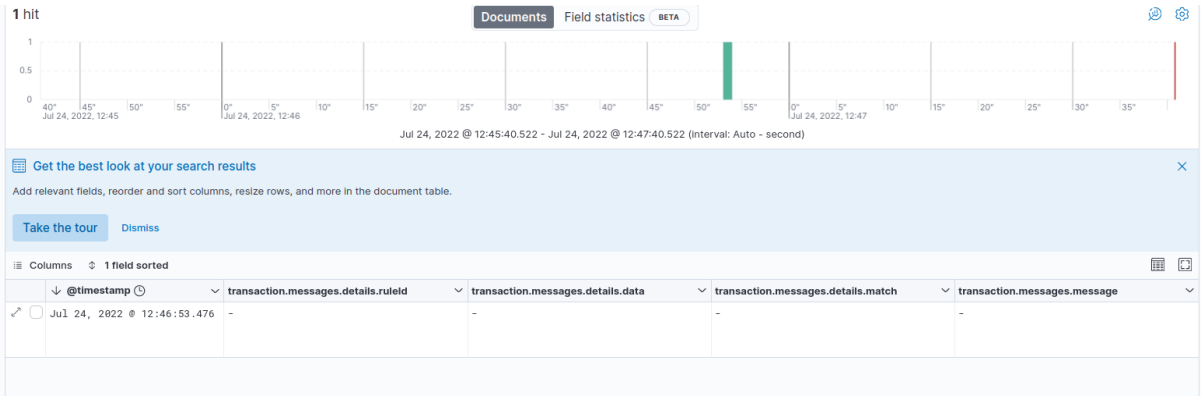
Double encoded 1’ select * from password but still did not get any valid input

Junk characters technique: Pass

http://dvwa.com/vulnerabilities/sqli/?id=1+%27s+-+-1-+-+e+-+l+e+c+t+-+-+-+*+-+-+-+from+password&Submit=Submit#



The technique was a complete success as I obfuscated the payload in the most confusing way possible. The server does not handle the request 1' select * from password and displays a blank page as its version of an sql error. It is worth noting however that we received a waf log with no input



While using those techniques if great for understanding how a WAF behaves, it is inefficient and can take time to find a desired bypass. If I were an outside attacker brute forcing techniques in this manner from a single IP, then an enterprise CSOC team (Cybersecurity Operations centre/ Blue team) would pick up my activity from their SIEM and block my IP address and/or range.

However, in this situation, I am a pen tester with access to the core rule set of the WAF and a real time input of rules triggered. This gives us a beautiful opportunity to find vulnerabilities in a more efficient way. Reverse engineering attack from the ruleset.

The campion of vulnerability finding. Reverse engineering attacks from the ruleset

Using information from the logs, we will identify and examine the rules triggered, and view the matched expressions in the rules and find alternatives to them. We will also use the regex expressions to derive obfuscation and bypasses.

Command injection:

Let's try the command, echo

This command obviously got blocked by our WAF.

But looking at Elasticsearch we can see the rule which blocked it:

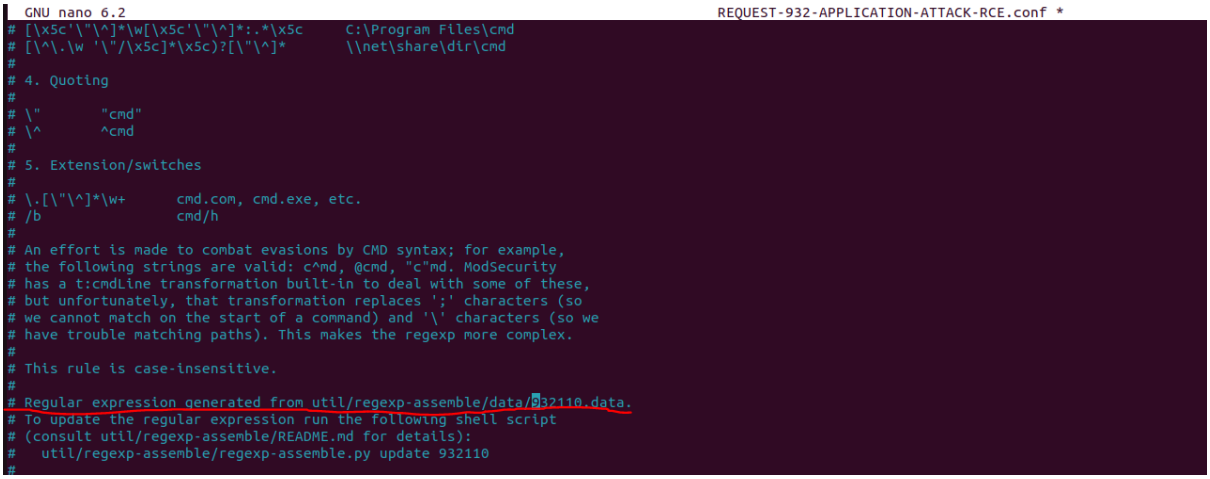
@timestamp	transaction.messages.details.ruleId	transaction.messages.details.data	transaction.messages.details.match
Jul 27, 2022 @ 02:01:41.418	932110, 949110	Matched Data: ; echo found within ARGS:ip: ; echo, (empty)	Matched "Operator `Rx' with parameter `(?i)(?:[;\n\r`]|(?:|)?||&?&|\(\)\s*(?:['(,@\"\s])*(?:(?:(?:[\x5c'\"\^]*\w[\x5c'\"\^]*:.*|[\^\.\w '\"/\x5c]*)\x5c|[\w'\"\./]+\/))?[\"\^]*(?:m[\"\^]*(?:y[\"\^]*s[\"\^]*q[\"\^]*l(?:[\"\^]*(?:d[\"\^]*u[\ (5003 characters omitted)' against variable `ARGS:ip' (Value: `; echo' ), Matched "Operator `Ge' with parameter `5' against variable `TX:BLOCKING_INBOUND_ANOMALY_SCORE' (Value: `5')

We can also see that our command was matched to a regular expression

@timestamp	transaction.messages.details.ruleId	transaction.messages.details.data	transaction.messages.details.match
Jul 27, 2022 @ 02:01:41.418	932110, 949110	Matched Data: ; echo found within ARGS:ip: ; echo, (empty)	Matched "Operator `Rx' with parameter `(?i)(?:[;\n\r`]|(?:|)?||&?&|\(\)\s*(?:['(,@\"\s])*(?:(?:(?:[\x5c'\"\^]*\w[\x5c'\"\^]*:.*|[\^\.\w '\"/\x5c]*)\x5c|[\w'\"\./]+\/))?[\"\^]*(?:m[\"\^]*(?:y[\"\^]*s[\"\^]*q[\"\^]*l(?:[\"\^]*(?:d[\"\^]*u[\ (5003 characters omitted)' against variable `ARGS:ip' (Value: `; echo' ), Matched "Operator `Ge' with parameter `5' against variable `TX:BLOCKING_INBOUND_ANOMALY_SCORE' (Value: `5')

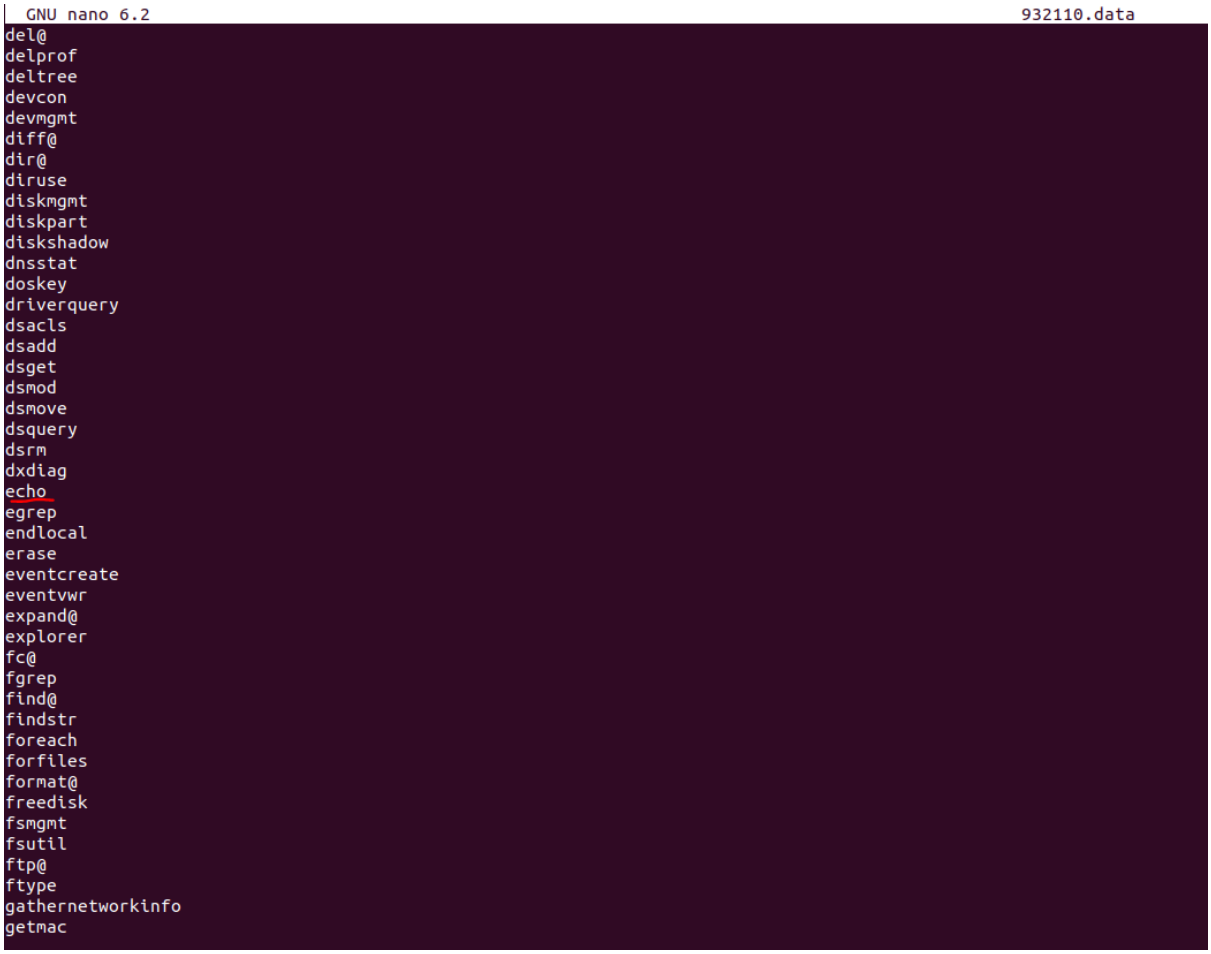
The data for those regular expressions are found in the util/regexp-assemble/data/ folder.

This can be seen in the comments of the 932-rule file itself (located in the rules directory).



```
  GNU nano 6.2                                                REQUEST-932-APPLICATION-ATTACK-RCE.conf *
# [\x5c'\"\^]*\w[\x5c'\"\^]*:.*\x5c     C:\Program Files\cmd
# [\^\.\w '\"/\x5c]*\x5c)?[\"\^]*       \\net\share\dir\cmd
#
# 4. Quoting
#
# \"        "cmd"
# \^        ^cmd
#
# 5. Extension/switches
#
# \.[\"\^]*\w+      cmd.com, cmd.exe, etc.
# /b                cmd/h
#
# An effort is made to combat evasions by CMD syntax; for example,
# the following strings are valid: c^md, @cmd, "c"md. ModSecurity
# has a t:cmdLine transformation built-in to deal with some of these,
# but unfortunately, that transformation replaces ';' characters (so
# we cannot match on the start of a command) and '\' characters (so we
# have trouble matching paths). This makes the regexp more complex.
#
# This rule is case-insensitive.
#
# Regular expression generated from util/regexp-assemble/data/932110.data.
# To update the regular expression run the following shell script
# (consult util/regexp-assemble/README.md for details):
#   util/regexp-assemble/regexp-assemble.py update 932110
#
```

In the data file itself. We found echo sure enough.

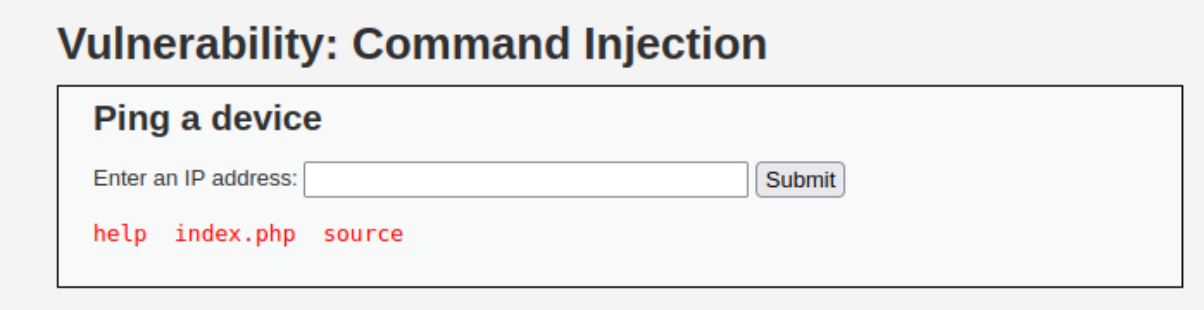


We can also see that there are starting operators (since command injections are added to a parameter of an already existing command in a website text bar)

```
##!$ \b

##! starting tokens prefix
##!> assemble
##! ;cmd
;
##! {cmd
\{
##! |cmd
\|
##! ||cmd
\|\|
##! &cmd
&
##! &&cmd
&&
##! \ncmd
\n
##! \rcmd
\r
##! `cmd
`
##!=>

##! match possible white space between prefix expressions
\s*
##!=>

##! commands prefix
(?:
##!=>

##! (cmd)
(
##! ,cmd
,
##! @cmd
@
##! 'cmd'
'
##! "cmd"
\"
##! spacing+cmd
\s
```

However. There are some commands that are not listed within the data file. 3 examples of commands that were not found are dir., who, and w. after hitting submit, the passed through the waf with no issue as demonstrated below.

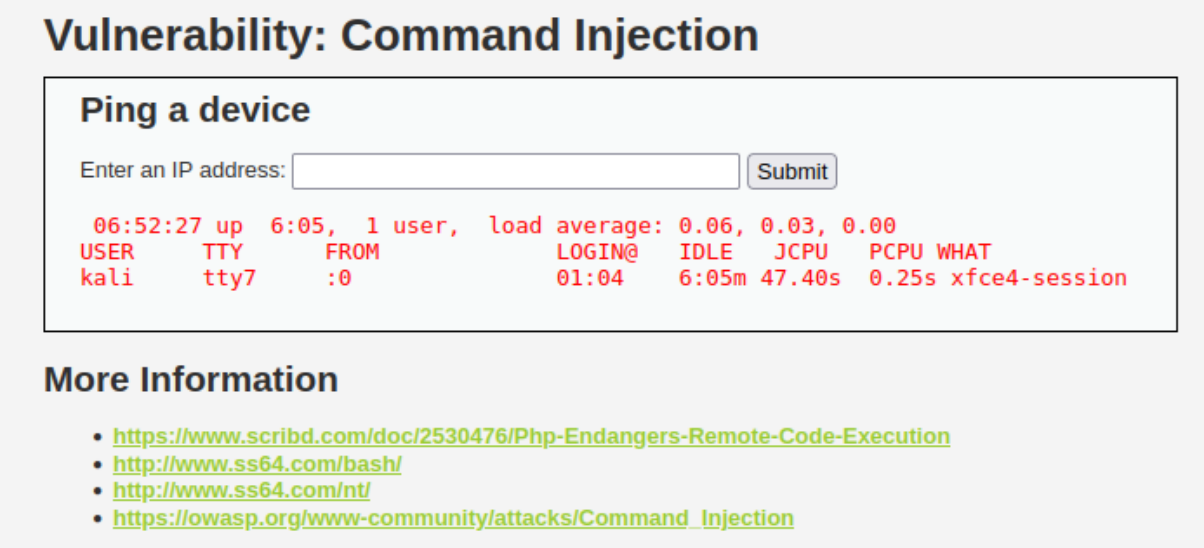
Dir



Dir is another command that substituted the blocked ls command

19

W



Who

While those exploits are there does that mean that there are ironclad rules against listed commands like echo?

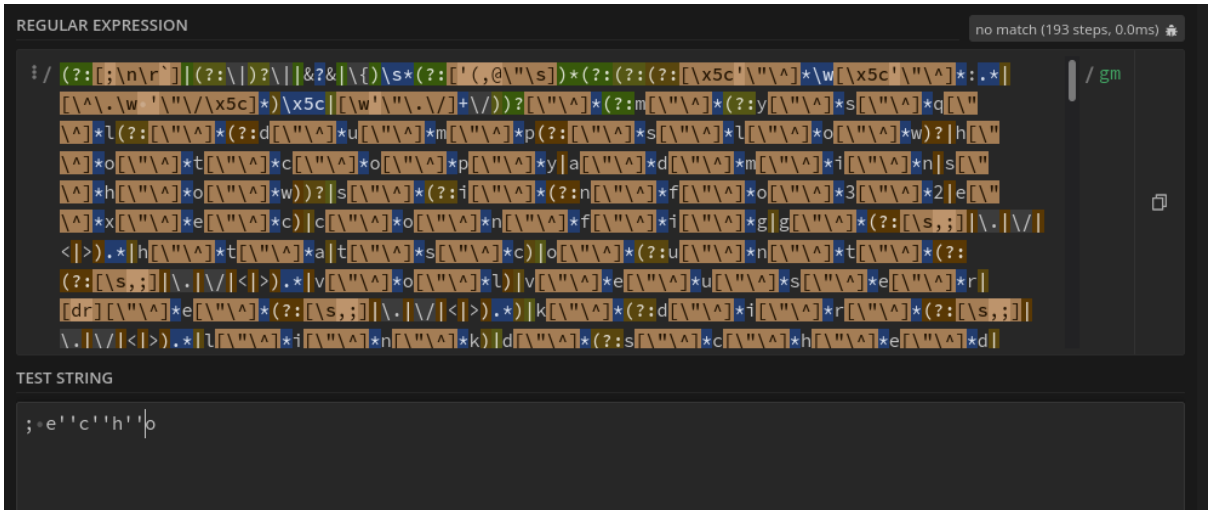
No. not by a longshot.

In Linux we can obfuscate our code with simple string separators like ‘’.

Let’s go deeper into regex and input the regex code from our rule into regex101. This Is a website that can test regex with matches.

In the image below, we can see that the command echo got caught by our ruleset as expected.

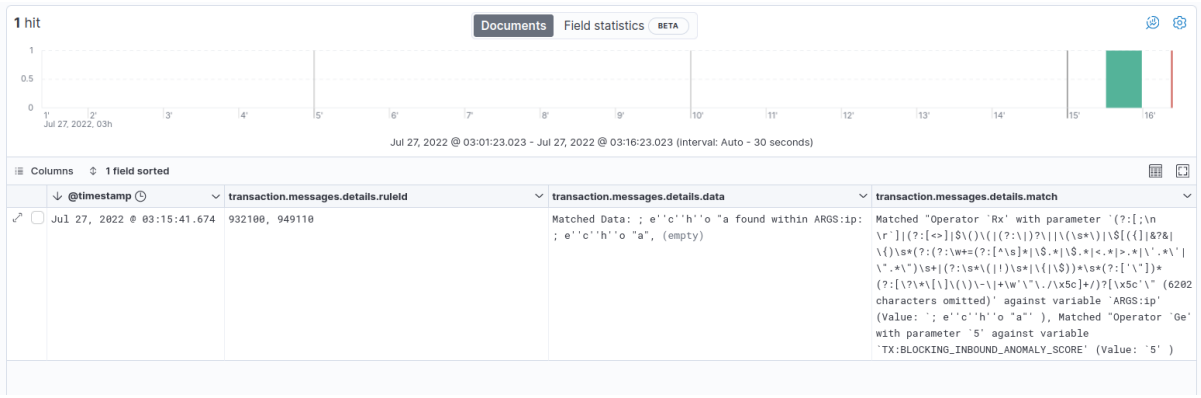
However, using obfuscation, this command bypasses the ruleset.



Now that we bypassed that rule, we're all good right? Well… Not quite.

Let's try the command; e''c''h''o "a"

Our WAF caught it with the flowing rules triggered:



Keep in mind that rule 932100 is a different rule than the rule that our regex expression bypassed (rule 932110).

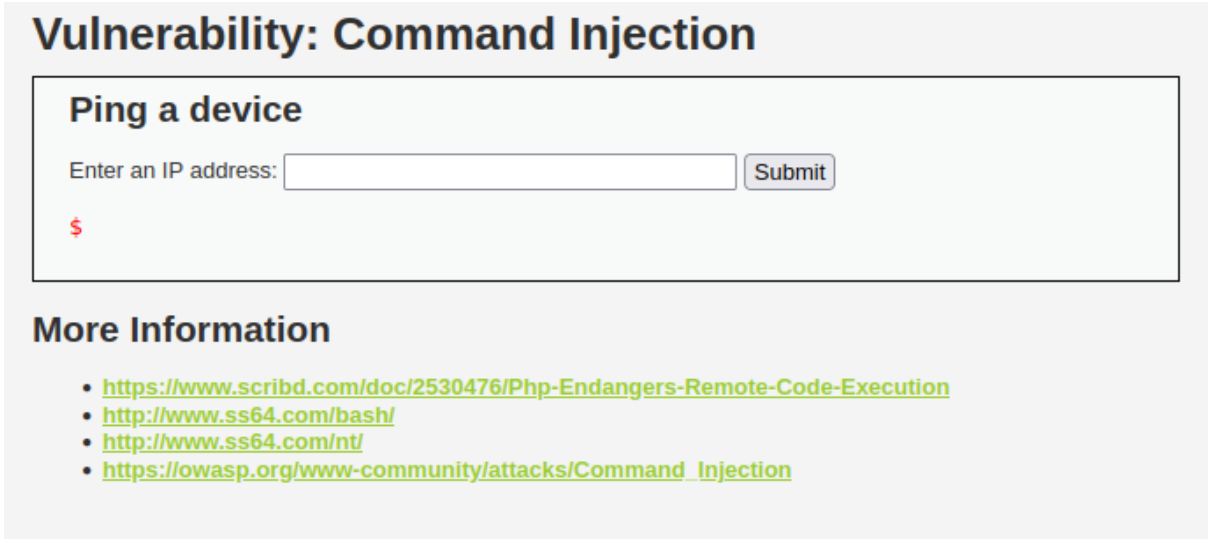
The next step is to repeat our process until we bypass that rule.

Lets look at our regex expression for rule 932110

Sure enough, it got caught.

Let’s try the a different expression that won’t get caught.

And it was able to go through the WAF.



We bypassed all rulesets. However, this output isn't exactly meaningful of life destroying.

Encoding isn't a situation that can help us here since we only get post requests

- Conclusion

A WAF is a must have tool for organization willing to protect their web applications. However, it should not be the only line of defence. It is also a coders job to secure web applications by adding more secure code (ex: only enter an IP address or reject the input).

As for the mod security WAF, it is a decent WAF as far as mitigating risk from general attacks. However, it has vulnerabilities that can easily exploited by a seasoned hacker. My pen testing level at the time of this is novice and even I was able to find some vulnerabilities with a bit of effort. On top of that this WAF has no machine learning capabilities, so it has the capacity to generate many false positives. (ex: let's say someone's name was ; sudo (very unlikely scenario)). Some WAFs can learn how their web applications work in order to understand how to protect them better unlike our open-source mod security WAF. Moreover, our modsecuity core rule set Is open-source and available to the public. An attacker who knows the WAF model and CRS can engineer an attack that can be bypassed and harm a vulnerable web application. Overall, I strongly discourage any organization to use this WAF to protect their web applications. As a learning tool however, it is perfect for a novice pen tester like me to understand how to bypass such a device and understand how WAFs Work.