

# **ModSecurity**

Web Application Firewalling

# ModSecurity Introduction

Our next step is to protect the web applications running on our web server using an Apache module called ModSecurity.

ModSecurity is a Web Application Firewall (WAF).

As with other WAFs, ModSecurity examines both client requests and server responses, blocking or modifying them when they're very likely an attack.

You can run this in logging-only mode, to check if it would interfere with normal operation.

# Rules and Two Models

ModSecurity works by checking every request and response against a set of rules.

If a rule matches a request or response, ModSecurity can take an action, generally to block or log the request/response.

There are two models for these rules:

- Default deny – rules match requests/responses that don't fit expected behavior
- Default allow – rules match requests/responses that look hostile


# Default Deny Example: SQL Injection

Let's protect a web application form by blocking input that doesn't match expectation.

Suppose we are protecting the DVWA SQL injection vulnerability. If you want to try this live, download the OWASP Broken Web Applications virtual machine after class.

We'll be doing this same kind of thing in our DonkeyDocker bonus exercise, where you'll write your own rule to protect DonkeyDocker.

# SQL Injection in DVWA



## Vulnerability: SQL Injection

**User ID:**

ID: ' OR 1=1; #  
First name: admin  
Surname: admin

ID: ' OR 1=1; #  
First name: Gordon  
Surname: Brown

ID: ' OR 1=1; #  
First name: Hack  
Surname: Me

ID: ' OR 1=1; #  
First name: Pablo  
Surname: Picasso

# Protecting the Form

To use ModSecurity to block this attack, consider the URL:

<http://owaspbwa/dvwa/vulnerabilities/sqli/?id=INPUT&Submit=Submit#>

The user/attacker submits the parameters to:

`/dvwa/vulnerabilities/sqli/`

`id` : a search string, intended to be numeric

`Submit` : the submit button

## Determine Reasonable Constraints

If we were dealing with a complete web application, we'd likely choose our constraint by exercising the user creation part of the application.

If the user gets to choose their own id, it will be obvious to us what the constraints are on that choice.

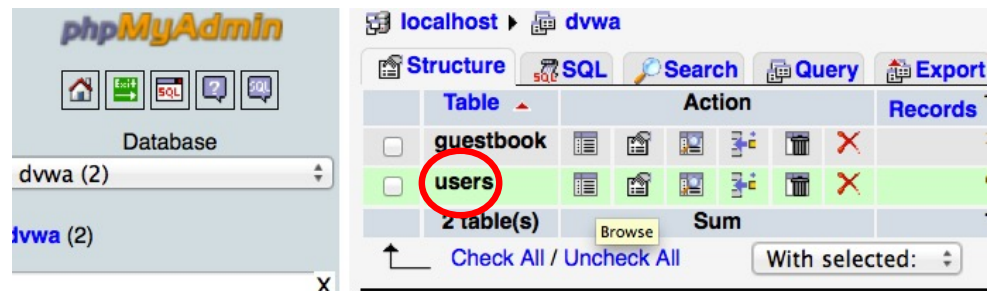
If the id is generated by the application and never revealed to the user, we could look at the database to see what values are being set. We could also read the user creation source code, if it was available. It often isn't. Further, the bang-for-your-buck ratio isn't as high.

# Investigate Constraints

We could define constraints for the id field.

Look at the users already in the database.

1. Browse to <http://owaspbwa/phpmyadmin>.
2. Login as root with password owaspbwa.
3. Browse the users table.





## Decide on Constraints

It looks like it'd be completely safe to restrict this field to numbers.

To be a bit more lenient, we could allow letters and numbers.

## Enforce the Constraints

```
<Location /dvwa/vulnerabilities/sqli/>  
SecRule ARGS_GET: id "!^[0-9]{1,15}$"  
"phase:2,t:none,deny,log,auditlog, \  
msg:'Input Validation Alert on id - Data not in the correct format.',  
logdata:'%{MATCHED_VAR} '"  
</Location>
```

We can create a rule that restricts the GET request's id variable to match the regular expression:

`^[a-zA-Z0-9]{1,15}$` or even `^[0-9]{1,15}$`

Regular expressions are powerful. If you don't speak regexp yet, we strongly recommend learning.

# RegEx Exploration Tools

**Regular Expressions 101:**

<https://regex101.com/>

**RegExr:**

<https://regexr.com/>

# Beating Evasions: POST Requests

Attackers can evade WAF and IDS rules by finding alternate ways to get their attacks through. One is to send the attack via POST instead of GET. The application server might still accept the attack, even though our first WAF rule wasn't looking for it.

Let's make sure that we don't allow POST requests to this URL:

```
<Location /dvwa/vulnerabilities/sqli/>  
SecRule &ARGS_POST_NAMES "!@eq 0" "phase:2,t:none,deny,log,auditlog, \\  
msg:'Input Validation Alert - Arguments in Post  
Payload',logdata:'%{MATCHED_VAR}' "
```

# Beating Evasions: Duplicate Argument Submissions

The URL on this evasion would look like this:

```
/dvwa/vulnerabilities/sqli/?id=1&id=%27%20OR%201%3D1%3B%20%23
```

Let's also make sure that the id parameter appears only once:

```
SecRule &ARGS_GET_NAMES:id "@gt 1" "phase:2,t:none,deny,log,auditlog,\  
msg:'Input Alert - Multiple id parameters.',logdata:'%{MATCHED_VAR}'"
```

# Test the Defense

Always test the defense – find mistakes or errors before an attacker does.

We use the same attack string:

```
' OR 1=1; #
```

This time, ModSecurity rejects the form submission.

## Bonus Exercise: DonkeyDocker

Open the Firefox browser on the class machine to:

<http://localhost:10000/exercises/donkeydocker-modsecurity>

## ModSecurity: CRS Mode

What we just did is actually an unusual use of ModSecurity. Most people use ModSecurity in deny list-only mode, where it just runs the OWASP ModSecurity Core Rule Set (CRS).

This rule set blocks a tremendous number of attacks, including the SQL injection attack we just killed through allow listing.

While these use a deny list methodology, they are incredibly effective. They're able to block attacks against unknown vulnerabilities in generic web apps.



# Core Rule Set (CRS): Techniques

- SQL Injection (SQLi)
- Cross Site Scripting (XSS)
- Local File Inclusion (LFI)
- Remote File Inclusion (RFI)
- Code injection, including PHP, Java, Shell
- Session Fixation
- Real-time Blocklist Lookups
- Google Safe Browsing API Lookups
- Known Web Shells
- Robots / Scanners via User Agent Strings
- Anti-virus/malware Scanning
- Sensitive Data Leakage
- Trojan Protection
- Application Misconfigurations
- Applications Returning Error Messages

Reference: <https://github.com/coreruleset/coreruleset>

## CRS – Collaborative Method

The Core Rule Set blocks attacks best using ModSecurity's Anomaly Scoring Detection mode.

Rather than an individual rule blocking an interaction outright, each rule instead contributes to an interaction's overall reputation. If the scores end up high enough, ModSecurity blocks the interaction.

The rules use ModSecurity's `setvar` action to increment both a general anomaly score and one specific to the attack type (e.g.: SQLi, XSS, LFI, RFI, ...).

# CRS: Activate Anomaly Scoring Detection

Activate ModSecurity's Anomaly Scoring Detection mode by changing the `SecDefaultAction` to `pass`.

In `modsecurity_crs_10_setup.conf` change this line:

```
SecDefaultAction "phase:1,deny,log"
```

to:

```
SecDefaultAction "phase:2,pass,log"
```

## CRS: Anomaly

When we use the anomaly scoring detection model, we need to choose thresholds for blocking. Anomaly scores get totaled by rules at these levels:

- 5 pts - Critical (web attack rules, 40-level files)
- 4 pts - Error (outbound leakage rules, 50-level files)
- 3 pts - Warning (malicious client rules, 35-level files)
- 2 pts - Notice (protocol policy and anomaly files)

# CRS: Severities

Severities range this way, from least to most severe:

|  |   |
|--|---|
| Notice (protocol policy and anomaly files)       | 5 |
| Warning (malicious client rules, 35-level files) | 4 |
| Error (outbound leakage rules, 50-level files)   | 3 |
| Critical (web attack rules, 40-level files)      | 2 |

There are two even more severe levels, only for correlation:

|   |   |
|---|---|
| Alert (inbound attack causing application error)    | 1 |
| Emergency (inbound attack causing outbound leakage) | 0 |

## CRS: Setting Thresholds

Based on this, we can set thresholds in the same file.

```
setvar:tx.inbound_anomaly_score_level=5, \    (one web attack rule fired)
setvar:tx.outbound_anomaly_score_level=4, \    (one data leakage rule fired)
```

These thresholds will be used in the following two files. For fewer false positives, change those numbers to 20.

```
modsecurity_crs_49_inbound_blocking.conf
modsecurity_crs_59_outbound_blocking.conf
```

# CRS: Activate Blocking

Activate blocking by editing this file: `modsecurity_crs_10_config_log.conf`

Change:

```
setvar:tx.anomaly_score_blocking=off,
```

to:

```
setvar:tx.anomaly_score_blocking=on,
```

## CRS Demo Page

ModSecurity used to have a demo page where you could input an attack string.

The results that site gave for our ' `OR 1=1 ; #` string are on the next slide.

You could also try attacks on your own ModSecurity-protected site and check out your logs.



# Demo Page Results

Results (txn: U9WnlsCo8AoAAH@oEjQAAAAF)

---

CRS Anomaly Score Exceeded (score 43): 981242-Detects classic SQL injection probings 1/2

## All Matched Rules Shown Below

---

**1000** Hash Validation Violation.  
Matched **Connection** at ARGS

---

**981261**SQL Injection Attack Detected via LibInjection  
Matched **s&1;c** at ARGS:test

---

**981261**SQL Injection Attack Detected via LibInjection  
Matched **s&1;** at ARGS:test

---

**981261**SQL Injection Attack Detected via LibInjection  
Matched **s&1;c** at QUERY\_STRING

---

**981261**SQL Injection Attack Detected via LibInjection  
Matched **s&1;** at QUERY\_STRING

---

**981318**SQL Injection Attack: Common Injection Testing Detected  
Matched **'** at ARGS:test

# More about ModSecurity

## Books on ModSecurity:

*Web Application Defender's Cookbook: Battling Hackers and Protecting Users* by Ryan C. Barnett and Jeremiah Grossman (Dec 10, 2012)

*ModSecurity Handbook: The Complete Guide to the Popular Open Source Web Application Firewall* by Ivan Ristic (2017)

*ModSecurity 2.5* by Magnus Mischel (Nov 23, 2009)