Red Hat JBoss Core Services 2.4.37

Red Hat JBoss Core Services ModSecurity Guide

For use with Red Hat JBoss middleware products.

Last Updated: 2020-06-24

Red Hat JBoss Core Services 2.4.37 Red Hat JBoss Core Services ModSecurity Guide

For use with Red Hat JBoss middleware products.

Legal Notice

Abstract

This book is a guide for using the Red Hat JBoss Core Services Mod Security module.

Table of Contents

CHAPTER 1. MODSECURITY INTRODUCTION

1.1. OVERVIEW

This guide includes information and examples for the ModSecurity v2.9 module for use with Red Hat JBoss Core Services 2.4.37. The effectiveness of ModSecurity is based on user generated rules. As such, this guide will explain how to create and implement rules, but will not give you a set of rules to use.

1.2. MODSECURITY

ModSecurity is a web application firewall(WAF). Web application firewalls filter, monitor, and block HTTP traffic to a web application. As opposed to a regular firewall, a WAF uses filters to determine what and who interacts with your HTTPD server application. ModSecurity accomplishes this task by making use of user generated security rules. These rules allow for configurable, realtime monitoring of HTTP traffic in order to detect attacks instantly.

CHAPTER 2. MODSECURITY CONFIGURATION

2.1. RHEL CONFIGURATION

2.1.1. Dependencies

ModSecurity has several dependencies to function. Some of these are already included as a part of Red Hat JBoss Core Services. A full listing is provided below:

Dependency	Part of JBCS
Apache Portable Runtimes	Y
APR-Util	Y
mod_unique_id	Y
libcurl	Y
PCRE	Y
libxml2	N
Lua5.x	N

NOTE

Red Hat JBoss Core Services for windows contains libxml2 and Lua5.x

2.1.2. Installation with Red Hat JBoss Core Services

ModSecurity comes as a part of the Red Hat JBoss Core Services package. For complete instructions on installing Red Hat JBoss Core Services, please refer to the installation guide for JBCS

2.1.3. Initial Configuration

ModSecurity is contained as a part of the Red Hat JBoss Core Services. This means that ModSecurity comes preconfigured and you will **NOT** have to use the **apxs** tool to build and install the module. In order to further configure Mod_sec, we must load the module into HTTPD.

2.1.3.1. Starting ModSecurity

You must load the libxml2 and lua5.1 libraries before loading the ModSecurity module. Use the following commands to load the libraries:

LoadFile /usr/lib/libxml2.so

LoadFile /usr/lib/liblua5.1.so

Then, load the ModSecurity module using the following:

LoadModule security2_module modules/mod_security2.so

2.1.3.2. Configuring the rules folder

The backbone of ModSecurities' functionality is the creation of rules that the system uses. However, we must know where to store rules in order to use them. JBCS comes with a preconfigured folder located at ***HTTPD_HOME*/modsecurity.d**. You can store rules in this folder or the **activated_rules** folder located underneath it.

Additionally, to use these rules we must setup the **mod_security.conf.sample** file to our specifications. Most importantly you **MUST** change the file name to **mod_security.conf**. This can be done with the following command:

mv mod_security.conf.sample ./mod_security.conf

In the file itself, you can set parameters for all of the configuration directives you will use with ModSecurity rules.

2.1.3.3. PCRE Warning

You want to avoid Apache using the bundled PCRE library and ModSecurity linking against the one provided by the operating system. The easiest way to do this is to compile Apache against the PCRE library provided by the operating system (or you can compile it against the latest PCRE version you downloaded from the main PCRE distribution site). You can do this at configure time using the --with-pcre switch. If you are not in a position to recompile Apache, then, to compile ModSecurity successfully, you'd still need to have access to the bundled PCRE headers (they are available only in the Apache source code) and change the include path for ModSecurity (as you did in step 7 above) to point to them (via the --with-pcre ModSecurity configure option). Do note that if your Apache is using an external PCRE library you can compile ModSecurity with WITH_PCRE_STUDY defined,which would possibly give you a slight performance edge in regular expression processing. Non-gcc compilers may have problems running out-of-the-box as the current build system was designed around the gcc compiler and some compiler/linker flags may differ. To use a non-gcc compiler you may need some manual Makefile tweaks if issues cannot be solved by exporting custom CFLAGS and CPPFLAGS environment variables.

2.1.3.4. Key Configuration Options

enable-pcre-jit: Enables JIT support from pcre >= 8.20 that can improve regex performance.

enable-lua-cache: Enables lua vm caching that can improve lua script performance. Difference just appears if ModSecurity must run more than one script per transaction.

enable-request-early: On ModSecurity 2.6 phase one has been moved to phase 2 hook, if you want to play around it use this option.

enable-htaccess-config: It will allow the follow directives to be used into .htaccess files when AllowOverride Options is set

2.2. WINDOWS CONFIGURATION

2.2.1. Windows Dependencies

ModSecurity has several dependencies to function on Windows. Some of these are already included as a part of Red Hat JBoss Core Services. A full listing is provided below:

Dependency	Part of JBCS on Windows
Apache Portable Runtimes	Y
APR-Util	Y
mod_unique_id	Y
libcurl	Y
PCRE	Y
libxml2	Y
Lua5.x	Y

2.2.2. Windows Installation

Many of the items required to run ModSecurity on windows are already contained in the JBCS package. However, for ModSecurity to function properly certain criteria must be met. Ensure the directory where you build software from source contains the Apache source you used to build the Apache web server and the ModSecurity source. For example:

- Apache source is in **C:\ *sourceDirectory*\httpd-2.4.37**
- Apache has been installed to **C:\Apache2437**
- ModSecurity source is in **C:\ *sourceDirectory*\mod_security**

In this case ***sourceDirectory*** is a generic directory used in conjunction with the project.

Additonally, ensure your build environment has the correct setup. The **PATH** environment variable must include the Visual Studio variables as set by **vsvars32.bat AND** also the CMAKE **bin**\ directory. Lastly, set and environment variable to the Apache source code directory: **C:\ *sourceDirectory*\httpd-2.4.37**

Once you are certain you have met the above criteria, you can install JBCS to the appropriate location on your C: drive.

ModSecurity comes as a part of the Red Hat JBoss Core Services package. For complete instructions on installing Red Hat JBoss Core Services, please refer to the installation guide for JBCS

2.2.3. Windows Configuration

2.2.3.1. Starting ModSecurity

You must load the libxml2 and lua5.1 libraries before loading the ModSecurity module. Use the following commands to load the libraries:

LoadFile /usr/lib/libxml2.so

LoadFile /usr/lib/liblua5.1.so

Then, load the ModSecurity module using the following:

LoadModule security2_module modules/mod_security2.so

2.2.3.2. Configuring Rules Directory

The backbone of ModSecurities' functionality is the creation of rules that the system uses. However, we must know where to store rules in order to use them. JBCS comes with a preconfigured directory located at ***HTTPD_HOME*/modsecurity.d**. You can store rules in this directory or the **activated_rules** directory located underneath it.

Additionally, to use these rules we must setup the **mod_security.conf.sample** file to our specifications. Most importantly you **MUST** change the file name to **mod_security.conf**.

2.2.3.3. Key Configuration Options

enable-pcre-jit: Enables JIT support from pcre >= 8.20 that can improve regex performance.

enable-lua-cache: Enables lua vm caching that can improve lua script performance. Difference just appears if ModSecurity must run more than one script per transaction.

enable-request-early: On ModSecurity 2.6 phase one has been moved to phase 2 hook, if you want to play around it use this option.

enable-htaccess-config: It will allow the follow directives to be used into .htaccess files when AllowOverride Options is set

CHAPTER 3. MODSECURITY RULES MAKING

3.1. CONFIGURATION EXAMPLES

3.1.1. Background Information

Configuration Directives: The configuration directives for ModSecurity are similar to the Apache Server directives. Most of the ModSecurity directives can be used inside the various Apache Scope Directives. There are others, however, that can only be used once in the main configuration file. These rules, along with the Core rules files, should be contained in files outside of the httpd.conf file and called up with Apache "Include" directives. This allows for easier updating/migration of the rules. A full listing of configuration directives can be found in the reference section of this document:

3.2. RULE EXAMPLES

3.2.1. Background Information

ModSecurity primarily functions when the user creates custom rules. These custom rules dictate what Mod_sec checks for when running. The rules are implemented at any of the 5 phases for the Apache request cycle. The five phases and their syntax names are given below:

Request headers → REQUEST_HEADERS

Request body → REQUEST_BODY

Response headers → RESPONSE_HEADERS

Response body → RESPONSE_BODY

Logging → LOGGING

3.2.2. Example Rule 1:

This section explains an example custom rule. The custom rule given checks whether or not history was changed by a request. Generally rules have 4 parts: a configuration directive, variable(s), operator(s), and action(s). For full listings of configuration directives, variables, operators, and transformations/actions, please refer to the reference section of this guide. The rule is given below:

SecRule REQUEST_URI "@streq /index.php" "id:1,phase:1,t:lowercase,deny"

The rule begins with **SecRule**. This is a configuration directive which creates a rule that will analyze the selected variables using the selected operator. Most of your rules will use this configuration directive.

The rule then continues with the variables: **REQUEST_URI** This variable holds the full request URL including the query string data.

Next, operators the operator is defined as **"@streq /index.php"**. the **@streq** checks for string values equal to the **/index.php** file. However, before this occurs, our transformations/actions will take place. They are defined as **"id:1,phase:1,t:lowercase,deny"**. From this command, the rule, during phase 1, will obtain the URI portion of the HTTP request, and transform the value to lowercase. it will then check the transformed value to see if it is equal to **/index.php**. If it is equal, Modsecurity will deny the request, and no further rules will be processed.

3.2.3. Example Rule 2:

This section explains a more complex example rule. The custom rule given checks whether or not history was changed by a request. Generally rules have 4 parts: a configuration directive, variable(s), operator(s), and action(s). For full listings of configuration directives, variables, operators, and transformations/actions, please refer to the reference section of this guide. The rule is given below:

SecRule REQUEST_URI|REQUEST_BODY|REQUEST_HEADERS_NAMES|REQUEST_HEADERS "history.pushstate|history.replacestate" "phase:4,deny,log,msg:'history-based attacks detected'"

The rule begins with **SecRule**. This is a configuration directive which creates a rule that will analyze the selected variables using the selected operator. Most of your rules will use this configuration directive .

The rule then continues with the variables:
REQUEST_URI|REQUEST_BODY|REQUEST_HEADERS_NAMES|REQUEST_HEADERS There are 4 variables here and each defines a different part of the request that the rule checks.

Next, operators are defined for the request. In this case, the rule checks for the JS methods **history.pushstate()** and **history.replacestate()**. Assuming these values are found, the last section will be executed.

The last section defines transformations and actions that will take place. In this case it first defines the processing phase in which the rule occurs. Then it runs the **deny**, **log**, and **msg** actions. **Deny** will stop the rule processing and intercept the transaction. **Log** will indicate that a log will need to be created. **Msg** will output a message with the log. the message is defined as 'history-based attacks detected'.

CHAPTER 4. MODSECURITY REFERENCES

4.1. APPENDIX

This section contains important references to the ModSecurity documentation provided by SpiderLabs. It has links to full listings of the following items used for creating rules.

4.1.1. Actions

Actions

4.1.2. Configuration Directives

Configuration Directives

4.1.3. Operators

Operators

4.1.4. Transformation Functions

Transformation Functions

4.1.5. Variables

Variables