

École polytechnique de Louvain

Advancing continuous integration for WAF engines by developing the ModSecurity Regression Test Set

Author: **Sébastien AMELINCKX**

Supervisor: **Ramin SADRE**

Readers: **Charles-Henry BERTRAND VAN OUYTSEL**, **Ervin HEGEDÜS**,
Steve MIHY

Academic year 2024–2025

Master [120] in Computer Science

Abstract

Web application security is an increasingly critical concern for organizations relying on web technologies to support their operations. Web Application Firewalls (WAFs) serve as the first line of defense against attacks targeting web applications. ModSecurity, a popular open-source WAF engine, recently underwent a development transition from Trustwave to the OWASP community. During this transition, no automated testing infrastructure was transferred to the new developers. Motivated by the absence of testing within ModSecurity's continuous integration and continuous delivery (CI/CD) pipelines, this work proposes a WAF engine testing methodology aimed at assisting developers in implementing a rigorous CI process. The work begins by evaluating the testing practices and tools of the OWASP Core Rule Set, followed by a description of the attempt of ModSecurity developers to adapt these tools into a utility known as the ModSecurity Regression Test Set (MRTS). To help advance MRTS to sufficient maturity for integration into ModSecurity's CI/CD pipelines, this thesis contributes new features and test cases to the MRTS testing framework. Finally, the test set is evaluated on ModSecurity2 and libmodsecurity3 infrastructures, and the work concludes by outlining future directions for MRTS's development.

Acknowledgements

I would like to express my profound gratitude to Charles-Henry Bertrand Van Ouytsel for his continuous support and for proposing this very interesting subject as my master thesis. I would also like to acknowledge Prof. Ramin Sadre for accepting to supervise this thesis and for his feedback throughout the year.

I would also like to thank the people at Approach Cyber for helping to make this work possible. In particular, I would like to thank Marc Stern for his guidance in the first part of this thesis, and Steve Mihy for accepting to review this work in his stead.

Furthermore, I want to express my sincere appreciation to Ervin Hegedüs for his involvement in this work, for reviewing my code contributions to MRTS, for accepting to further review this document, and ultimately for keeping the OWASP ModSecurity project well and alive. Additionally, I'd like to thank the developers of CRS for developing the tools used in this work. In particular, I'd like to thank Max Leske for answering my many questions and for implementing changes I suggested.

On a personal note, I would like to acknowledge my family, my friends, and my girlfriend Lisa for their support and care. Your encouragements helped me get through my education.

I would also like to thank my comrades. The moments we shared made this journey so much more invaluable. May we continue our struggle for equality until victory.

On a more serious note, I wish to express my strongest condemnation of all acts of harassment and sexual violence. Such behavior and any attempts to conceal it have no place at university or anywhere else in society. I share my solidarity with the victims and with those who have come forward to denounce these injustices.

Finally, I would like to remind us that, in this time, no subject is as important as the current events unfolding in Gaza. I urge our universities and governments to cut all academic, economic, and political ties complicit of the genocide of Palestinians.

Grammarly¹ and ChatGPT² were used to assist in the writing of this work. These tools were used to help me better express my ideas and ensure the clarity of my sentences.

¹<https://www.grammarly.com/>

²<https://www.openai.com/chatgpt>

Contents

1	Introduction	1
2	Background	3
2.1	Web Application Firewall	3
2.1.1	Concept	3
2.1.2	Deployment	4
2.1.3	Functionalities	6
2.1.4	Internals	7
2.2	ModSecurity	7
2.2.1	Project	7
2.2.2	Engine Functionalities	8
2.2.3	Transaction Lifecycle	8
2.3	Core Rule Set	9
2.3.1	Project	9
2.3.2	SecLang	9
2.4	Continuous Integration	10
2.4.1	Release Engineering	11
2.4.2	CI/CD Pipelines	12
2.5	YAML	12
3	Methodology	14
3.1	Objectives	14
3.1.1	Defining WAF Engine Functional Testing	14
3.1.2	Testing Framework Requirements	15
3.1.3	Initial Tool Evaluation	15
3.2	How CRS Does WAF Rule Testing	16
3.2.1	The CRS Testing Practice	16
3.2.2	A Framework for Testing WAFs: go-ftw	16
3.2.3	An HTTP Reflector: albedo	18
3.3	The ModSecurity Regression Test Set	18
3.3.1	Concept	18
3.3.2	Features of Version 0.1	19
3.3.3	Example: the ARGS_GET Target Test	19
4	Framework Features Design	23
4.1	Development Process	23
4.2	Categories of Features We Added	23
4.3	Rule Generation Improvements	24

4.3.1	ACTIONS and DIRECTIVES macros	24
4.3.2	Directives Before(-Each) and After(-Each) Generation	26
4.4	Testing Improvements	27
4.4.1	Extend Check Methods on Logs and Responses	27
4.4.2	Sending Malformed Requests	28
4.4.3	Manually Defining URI for Using Albedo's Reflection Capabilities	29
4.5	Constants in Configuration Files	30
4.6	WAF Infrastructures	31
5	Test Design and Evaluation	34
5.1	Test Case Selection and Categories	34
5.1.1	Objectives	34
5.1.2	Building on Previous Work	35
5.1.3	Selected Cases	36
5.2	Tests for Targets Popular in CRS	37
5.2.1	002 - ARGS	37
5.2.2	059 - REQUEST_COOKIES	39
5.2.3	061 - REQUEST_FILENAME and CVE-2024-1019	40
5.2.4	069 - RESPONSE_BODY	42
5.3	Tests for Other Targets	43
5.3.1	010 - DURATION	43
5.3.2	026 - MATCHED_VARS_NAMES	44
5.4	Evaluation	47
5.4.1	Experimental Setup	47
5.4.2	Evaluation on ModSecurity2	48
5.4.3	Evaluation on libmodsecurity3	48
5.5	Completeness of the Test Set	50
5.5.1	Coverage of Targets	50
5.5.2	Coverage of SecLang	50
5.5.3	Coverage of ModSecurity Functionalities	51
6	Conclusion and Future Work	52
Bibliography		
Appendices		
A	Auxiliary Framework Features	i
A.1	Improved Directory Structure	i
A.2	Complex Separator for Macros in Templates	ii
A.3	MRTS Utility	ii
B	Additional Feature Demonstrations	v
B.1	DIRECTIVES Macro	v
B.2	Encoded Requests	vii
C	Target Frequency in CRS v4.8.0	viii
D	Auxiliary Tests	ix
D.1	006 - ARGS_NAMES	ix

D.2	007 - ARGS_POST	x
D.3	008 - ARGS_POST_NAMES	xii
D.4	060 - REQUEST_COOKIES_NAMES	xiii

Chapter 1

Introduction

Since the wide deployment of computer networks, most organizations, governments, and businesses have been undergoing a digital transformation. One of the most visible transformations is the growing reliance on web technologies to carry out processes. While effective, deploying web applications poses risks to the information users provide and can potentially expose entry points to the organization's broader information system. The security of web applications is frequently compromised, as the numerous components that enable their functionality often contain vulnerabilities. When exploited, these weaknesses can result in data breaches, system damage, and significant financial losses.

Fortunately, the study of web application security is well-established. Experts and organizations have developed defense strategies to protect against exploitation and enhance threat detection. A key component of most strategies is the use of Web Application Firewalls (WAFs), which serve as the first line of defense against attacks. WAFs also provide crucial visibility, enabling analysts to monitor web traffic and identify potential threats.

However, developing a WAF is no simple task, and maintaining one presents a complex challenge that requires a rigorous methodology and the right tools. Furthermore, there are very few WAF engines available under an open-source license, which allows individuals and organizations to freely build security solutions upon them.

In August 2021, Trustwave announced the commercial end-of-life of its open-source WAF engine, ModSecurity, and searched for a new maintainer. In January 2024, the project entered into custodianship of the OWASP foundation, a non-profit organization fostering a large online community of developers working on open-source security software. The project, now fully reliant on voluntary contributions, has to grow a community of willing developers to survive. In response to the situation, Approach Cyber—a cybersecurity company offering ModSecurity based solutions—has committed to actively supporting the project's development by dedicating part of its resources to it. For Approach Cyber and many other companies, ensuring ModSecurity is actively maintained is critical to sustaining and evolving their security offerings for customers, guaranteeing the WAF engine remains robust and up-to-date.

When transferring the codebase, Trustwave did not provide all the tools that would be helpful to support its new developers. Among the lacking utilities, no end-to-end tests and framework to write them were given. Yet, a testing practice is crucial for developers

to check software behaves as expected, as unexpected behaviors can lead to functionality or even security failures.

CVE-2024-1019, published on the 30th of January 2024, is a high severity vulnerability affecting ModSecurity 3.0.0 up to 3.0.11. The vulnerability enables attackers to bypass WAF protections placed on the request URI parameters. The absence of well-designed testing practices and tools contributed heavily on leaving it unnoticed for up to 6 years. According to the lead developer of ModSecurity V3, a testing tool like the one we intend to contribute to could have helped prevent the vulnerability [1].

In an effort to contribute to the open development of WAFs, this thesis will develop a testing practice for WAF engines and demonstrate its application on ModSecurity V2 and ModSecurity V3.

The implementation of this testing practice takes the form of the ModSecurity Regression Test Set (MRTS), a utility build to write and execute test cases on ModSecurity compliant WAFs. The code for MRTS is located in a public repository belonging to the OWASP ModSecurity project and can be found at:

<https://github.com/owasp-modsecurity/MRTS>

This thesis is organised under six chapters:

Chapter 2 discusses the background information needed to contextualize this thesis. The chapter starts by describing what are WAFs by discussing their use and internals. It then proceeds to present ModSecurity and CRS, two open-source projects the thesis revolves around. Finally, the chapter discusses continuous integration, and its link to regression testing.

Chapter 3 presents the methodology used to select the testing practice we chose to pursue. The chapter starts by an initial assessment of tools that could have been considered, then proceeds by discussing the testing approach and tools developed by CRS for their purposes, and finally presents the original idea for MRTS.

Chapter 4 presents the features that were contributed to the framework. The chapter starts by discussing the process employed to come up with additional features. Each feature is given precise specifications and the reasoning behind them is duly justified.

Chapter 5 presents the test cases that were contributed to the regression test suite. The chapter starts by discussing the methodology used to select the functionalities of the WAF engine to be tested in this work. The tested functionalities are explained, and each test case is detailed. It then proceeds to evaluate the utility and existing test suite. The evaluation conducted of MRTS is on the two ModSecurity release lines.

Chapter 6 concludes by summarizing the work done and proposes further directions for future work.

Chapter 2

Background

2.1 Web Application Firewall

A Web Application Firewall (WAF) is a security solution aimed at protecting web applications. In this section, we will introduce WAFs as a concept, explore common deployment strategies to understand their use case, and present the internals that make them.

2.1.1 Concept

There is no currently standardized definition of a web application firewall, and different proposals-as well as the necessity of establishing a universal answer-remain subjects of ongoing debate [2, 3, 4]. According to the OWASP foundation, a *web application firewall* (WAF) is an application firewall for HTTP applications. It applies a set of rules to an HTTP conversation [5]. A WAF, as illustrated in figure 2.1, will examine the inbound and outgoing traffic of a web application, classify the traffic as safe or not for the security of the application, and perform actions. While network firewalls aim to protect systems by analyzing transport or network layer protocols such as TCP/IP, application firewalls check for malicious intent at the *application layer*.

WAFs are no ultimate security solution, but rather one layer among others of a *defense-in-depth* strategy [6]. In particular, WAFs are the first line of defense against attacks on web applications.



Figure 2.1: Web Application Firewall concept

2.1.2 Deployment

The deployment of a WAF is what determines what traffic a WAF can view and act upon. WAFs can be deployed on a network in many ways, with different advantages and disadvantages to each deployment setup. We will discuss elementary production WAF deployment models to understand simple use cases for WAFs.

In-line Reverse Proxy

A straight forward approach to WAF deployment is to configure the WAF as an *in-line reverse proxy* [7]. As illustrated in figure 2.2, the WAF is placed as the entry point to all traffic between the outside network and the web server(s).

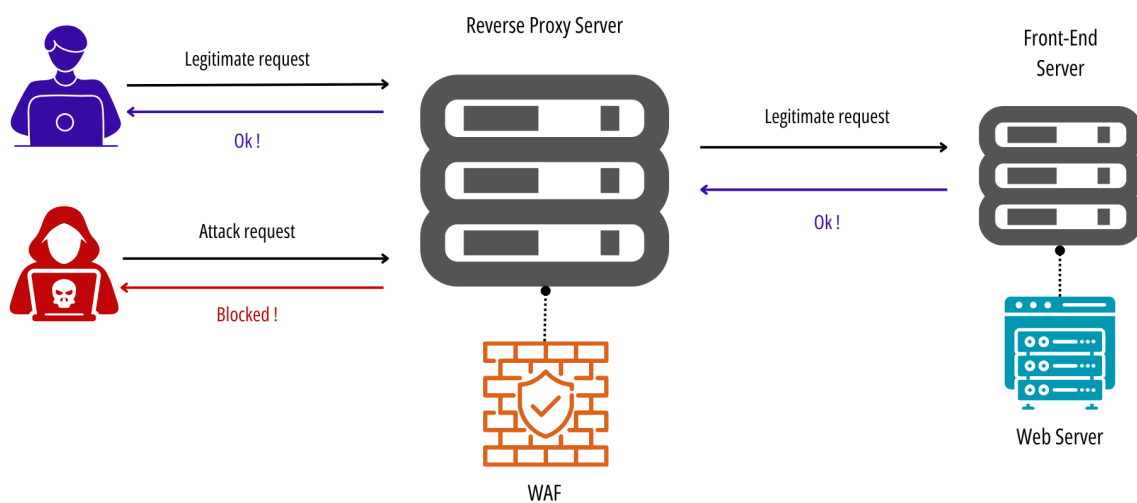


Figure 2.2: Web Application Firewall deployed as an in-line reverse proxy

The in-line reverse proxy WAF has *full view of the traffic* towards all applications behind it. Having a full view of the traffic has the advantage of allowing the WAF to log and to block any traffic exchanged with the web application. Being in-line has however the disadvantage of having to process all packets in real time which bears performance costs. Additionally, the blocking of traffic carries the risk of potentially classifying legitimate traffic as malicious. If managed in a separate software environment to the web servers, the in-line reverse proxy WAF allows for an on-premise deployment or an outsourced deployment as a cloud service.

Out-Of-Band

An alternative deployment model is to configure the WAF as an *out-of-bound* application [8]. As illustrated in figure 2.3, a copy of the traffic exchanged with the web application is sent to the WAF through techniques like port mirroring.

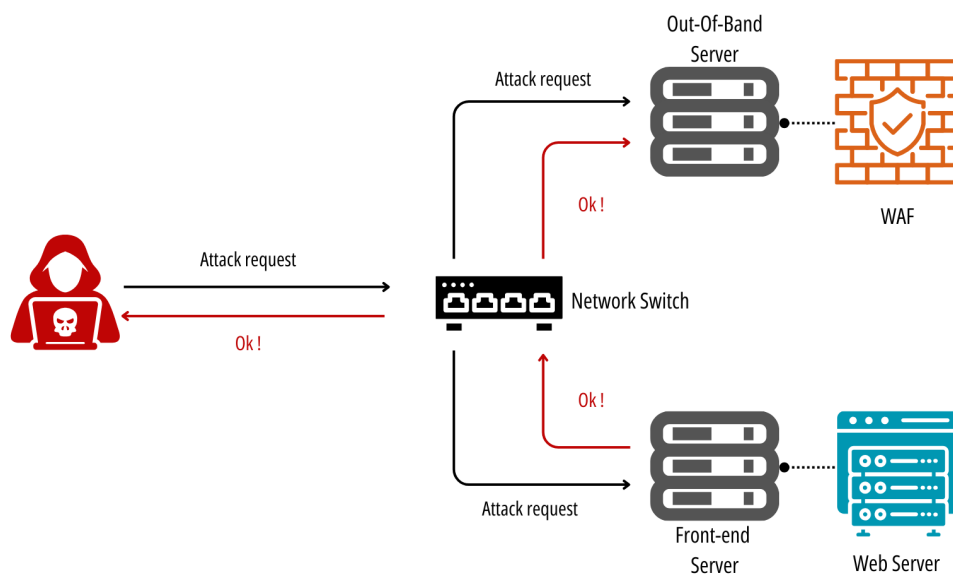


Figure 2.3: Web Application Firewall deployed as an out-of-bound server with port mirroring

The out-of-bound WAF is not meant to block traffic exchanged with the web application. Rather, it focuses on *logging the traffic for further analysis*. The out-of-bound WAF does not drop legitimate traffic classified as false positives and bears no performance cost on the traffic stream. However, it does not actively prevent attacks.

Software

In the above examples, we presented WAF deployments as if each component was installed on its own hardware. As illustrated in figure 2.4, a WAF and web applications can also be deployed as *software components* on shared hardware either as linked virtual machines, different processes part of the same software environment, or even with the WAF embedded in the web server process [9, 10].

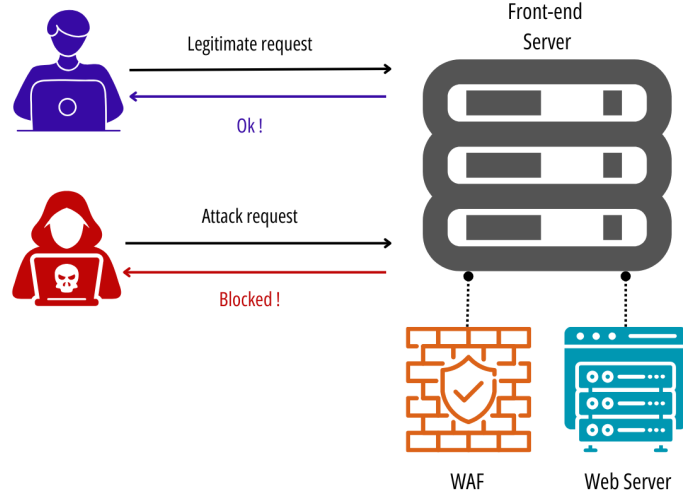


Figure 2.4: Web Application Firewall deployed on shared hardware

Software deployments can be set up in a variety of different architectures. They have the advantage of being flexible while requiring no modification of the network. Under a shared software environment, they have the disadvantage of having to share computing resources. WAFs embedded in web servers may save overall resource consumption by benefiting from traffic handling tasks the web server already performs, but managing both configurations as a single program can be more difficult.

In this work, our testing infrastructure will be deployed on a single machine under a shared environment.

2.1.3 Functionalities

Once deployed, WAFs can take actions based on the received traffic, either directly affecting the traffic or logging information related to it.

WAF actions affecting traffic conventionally include *blocking traffic* and can either be applied to malicious incoming traffic or outgoing sensitive traffic. A more recent use of WAF capabilities is *virtual patching* [11, 12], which involves quickly deploying a security fix to address a vulnerable application by applying a security policy that prevents exploitation of the newly discovered vulnerability. Virtual patches act as an interim solution to protect applications without requiring direct changes to the application’s source code, allowing organizations to mitigate attacks while awaiting the implementation of a full patch in production. Virtual patching is the only option to protect against vulnerabilities affecting code outside an organization’s control.

Logging traffic information is the other main function of a WAF. It provides visibility for manual traffic analysis, but also enables integration with other security solutions. Many security tools, such as Security Information and Event Management systems (SIEMs) or security testing software, can import WAF logs to power their functionality [13, 14, 15, 16]. Broader *threat intelligence* can also be extrapolated from the logged data, enabling the identification of emerging threats and patterns [17].

Some WAFs provide more advanced protection capabilities, such as bot and DDoS prevention [18]. Emergent machine learning techniques are also being developed [19]. In the context of this work we will not discuss those functionalities.

2.1.4 Internals

WAFs consists of two main components: a rule set and an engine. Each of these components serves distinct responsibilities in providing a WAF's functionality.

Rules

WAF rules are specifications instructing engines how to inspect HTTP traffic and what actions to perform. Rules are responsible for implementing the *security policies* provided by WAFs [20, 21, 22].

Rules will direct engines to check conditions and, when met, act during the HTTP transaction. More complex rule chains may require multiple conditions before acting, or instead of directly triggering actions, each trigger will increase a score, and actions are performed when a certain score threshold is reached [23].

Rules are implemented using a Domain Specific Language (DSL) which varies depending on the specific WAF software. Rules are often referred as the configuration of a WAF.

Engine

A WAF engine is responsible for parsing and buffering the HTTP requests and responses, managing the logging, handling any traffic modifications, and implementing a rule engine [24, 25, 26].

The implementation of WAF engines vary greatly depending on the targeted use, supported deployment models, intended platforms, implementation language, and rule language.

2.2 ModSecurity

The WAF engine we will discuss in this work is ModSecurity [27]. In this section, we will present the overall project and give some insights into the functionalities the engine and associated components provide.

2.2.1 Project

ModSecurity is, as of January 2024, developed under the custodianship of the OWASP foundation [28]. The ModSecurity codebases are licensed under the Apache 2.0 license. The implementation language of the ModSecurity engines is C/C++. ModSecurity is currently released under two release lines: ModSecurity2 and ModSecurity3, also known as libmodsecurity3. For historical reasons, ModSecurity2 was developed as an Apache 2 module making it available only for apache web server software. To circumvent this, a full rewrite of ModSecurity as a standalone library started in 2017. Associated with the release of ModSecurity 3.0.0 is the nginx connector for libmodsecurity3, integrating the

WAF engine with the nginx web server software. To this day, ModSecurity2 is intended for deployment with apache and ModSecurity3 for deployment with nginx. Efforts to stabilize the Apache connector for ModSecurity3 are undergoing, and if performance of the Apache connector is sufficient the project should move on to focus solely on the ModSecurity3 release. At the time of this writing, the latest releases are v2.9.9 and v3.0.14.

2.2.2 Engine Functionalities

ModSecurity's functionalities can be organized under different categories [24]:

- **Parsing:** ModSecurity includes several security-aware parsers aimed at extracting meaningful information from transaction data. Those parsers are responsible for turning raw bytes of data from HTTP transactions into comprehensive formats allowing WAF configurators to write rules capable of analyzing data in the context of a web application's functionality.
- **Buffering:** ModSecurity holds complete requests and responses in memory before forwarding them. This ensures accurate traffic blocking but consumes additional memory resources.
- **Logging:** Full transaction logging, a feature not commonly available in web server software, is offered by the ModSecurity engine. ModSecurity refers to this feature as audit logging. ModSecurity also provides different kinds of log files for multiple usages.
- **Rule Engine:** The rule engine in ModSecurity utilizes a language known as SecLang. Since versioning applies to the engines rather than the language itself, the specific implementation of SecLang can vary depending on the ModSecurity version. As a result, the current releases of ModSecurity2 and ModSecurity3 do not support identical versions of the language. However, ongoing efforts aim to reduce the differences between these implementations.

We will discuss some specific functionalities provided when describing the test cases implemented to evaluate them.

2.2.3 Transaction Lifecycle

ModSecurity will separate analysis of an HTTP transaction in 5 different phases [29]. The phases are:

- 1) **Arrival of the request headers:** The entry point to ModSecurity. The first phase is useful to prepare the analysis of the more computationally costly second phase.
- 2) **Arrival of the request body:** The main request analysis phase. Here, everything about the user's request can be analyzed to determine if the request should be blocked before reaching the web application.
- 3) **Arrival of the response headers:** If the transaction has not been intercepted, this phase happens when the web application has sent its response headers. The phase is useful to prepare the analysis of the more computationally costly fourth phase.

- 4) **Arrival of the response body:** The main response analysis phase. Here, everything about the web application's response can be analyzed to determine if the response should be blocked before reaching the user.
- 5) **Logging:** Happens when the HTTP transaction is over, implying it can't block any traffic. The final phase is useful to define how logging should be handled.

Each rule ModSecurity evaluates is associated to a corresponding phase. ModSecurity will apply the rules associated to a phase before passing on the next phase to be processed unless the HTTP transaction is interrupted.

2.3 Core Rule Set

The Core Rule Set (CRS) is an open-source project aimed at providing a generic set of rules covering a wide range of attacks [30]. In this section, we will present the overall project and the language used to implement the rules.

2.3.1 Project

The core rule set, publicly released in 2013, is currently developed under the custodianship of the OWASP foundation. The OWASP CRS codebases are licensed under the Apache 2.0 license. The core rule set is implemented with ModSecurity2 as its reference engine, but is compatible with multiple WAF engines [31]. The core rule set is also deployed in many configurations by default. CRS is designed to protect web applications from common security threats, including the OWASP top 10 which describe the most critical security risks to web applications. OWASP CRS is also responsible for developing other software aimed at helping rule developers.

2.3.2 SecLang

SecLang is a Domain Specific Language (DSL) used to write configurations for WAFs. SecLang was released as the DSL for configuring ModSecurity2 and was originally designed as an extension of the Apache HTTP server's configuration language, since ModSecurity2 is built as an Apache module. As the language used to implement the core rule set, SecLang has also become the language for other "ModSecurity compatible" WAF engines seeking to integrate these rules. Despite being used across different engines, the language was never standardized, and its description is mainly found in each engine's documentation [32, 33, 34]. The absence of a standard implies that the exact implementation and semantics of SecLang is dependent on the engine used, and even engine version.

Despite this, SecLang has influenced WAF rule languages across the industry. SecLang's adoption and influence beyond open-source WAFs can be partly attributed to the popularity of the core rule set itself. To integrate the CRS into their WAF deployments, many other proprietary WAF engines implement at least a subset of ModSecurity's SecLang with their own extensions. If they don't, they may translate the CRS into their own engine's DSL [31, 35].

Writing Rules with SecLang

To get an initial understanding of SecLang, let's discuss an example of how SecLang is used to instruct a WAF engine. Directing what the WAF engine should do is done by providing a *directive* in the configuration. There are many directives available in SecLang, but because we will focus on testing WAF engine rules in this work, we will describe the directive for writing rules: the *SecRule* directive [36].

```
SecRule VARIABLES OPERATOR [TRANSFORMATION_FUNCTIONS] ACTIONS
```

Let's dissect each part of the SecRule directive:

- **VARIABLES:** is the part of the HTTP transaction that will be examined by this rule. Variables are by default presented as raw bytes of data. Variables can encompass any extractable data from the HTTP transaction, including request and response contents, as well as computed data generated by the engine.
- **OPERATOR:** is the examination method used by the rule. The operator describes the condition to trigger the rule. Operators can involve pattern matching or various other evaluations.
- **TRANSFORMATION_FUNCTIONS:** are used to change variables before the rule operator is run. Transformation functions help extract meaningful information from the variable data or to counter evasion techniques. Providing transformation functions is optional.
- **ACTIONS:** define how the engine should act when a rule triggers. Actions can direct the engine to log, modify, or interrupt the HTTP transaction. Additionally, actions provide metadata, including the unique rule ID and the phase in which the rule should execute.

A Simple Example

A basic use case for a SecRule is illustrated with the following example:

```
SecRule REQUEST_BODY "@contains token" " \
id:1002, \
phase:2, \
log, \
deny, \
msg:'Token detected in request body'"
```

This rule, with id 1002, directs the engine to look if the request body variable, as found in phase 2, contains the string "token". If it does, the HTTP transaction is interrupted and logged with a message.

We will discuss exact rules and the functionalities they provide when describing test cases implemented in this work.

2.4 Continuous Integration

Continuous Integration (CI) is a software development practice where developers integrate code into a shared repository frequently [37]. In this section, we will contextualize the

practice of CI in software development, discuss its link to the practice of regression testing, and briefly discuss the systems that implement CI.

2.4.1 Release Engineering

Release Engineering is the discipline of building, packaging, testing, and distributing software releases in a reliable, reproducible, and automated manner. It involves processes, tools, and best practices ensuring that software is delivered efficiently and with high quality. As illustrated in figure 2.5, CI is one of the earlier steps of the release engineering pipeline. The primary objective of CI is to detect new issues early, enabling fast feedback and smoother collaboration among developers [38].

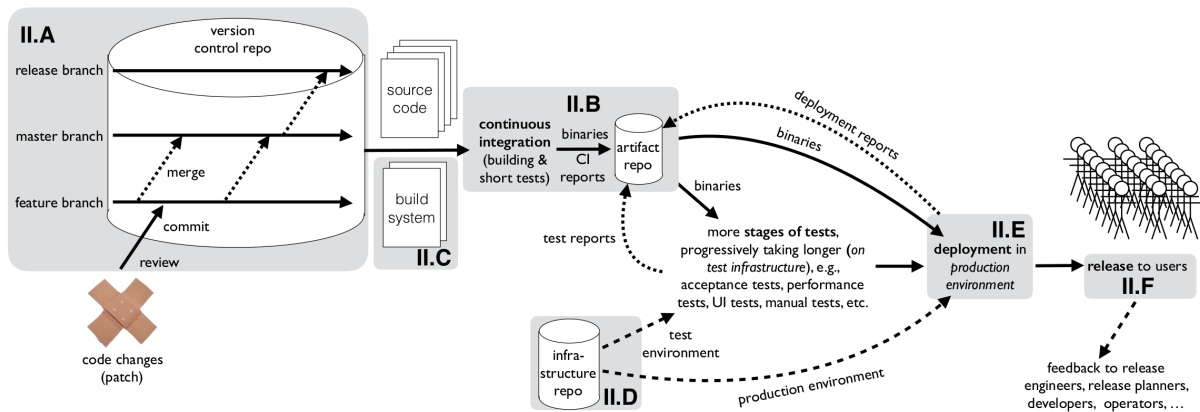


Figure 2.5: Overview of the modern release engineering pipeline from [38], © 2016 IEEE

Branching and Merging

For software developers to contribute changes to a shared project, they first need to create a copy of the main project's code on their local system. This process is known as *branching*. Typically, developers branch from the project's primary branch, often called the main or master branch, to create an isolated workspace where they can safely implement modifications. Once changes are made, developers save their work through a process called *committing*. When they are ready to integrate their updates into the shared codebase, they *merge* their changes into the target branch. This entire workflow is known as *integration*.

To accomplish this, developers use Version Control Systems (VCSs), software solutions that track project versions and maintain a history of changes. In this work, we will use Git [39], a widely adopted VCS, alongside the GitHub web application for hosting and collaboration [40].

Building and Regression Testing

In CI, an automated build and test process is triggered before merging changes to ensure they do not introduce issues. The *build* phase involves converting the code into its binary form, verifying that it compiles correctly without errors. The *testing* phase runs a suite of tests to confirm that the new changes do not break or regress existing functionality.

Regression testing is a quality assurance practice that involves running a set of tests to ensure that recent code changes have not introduced new defects or broken existing functionality. This type of testing can be applied at multiple stages of the release engineering pipeline. In this work, we will focus on performing regression testing before merging code into the master branch of the shared repository.

In academia, regression testing has been studied extensively to define methodologies to select and prioritize tests from the full set of tests of a project in order to balance long execution times and avoiding regressions [41]. However, industry practitioners have often criticized these approaches for being too time-consuming and impractical [42].

In this work, our objective is to develop a regression test suite from scratch. Rather than focusing on selecting or prioritizing test cases, we will implement a set of tests to be executed at each integration. To guide our approach, we will draw inspiration from the regression testing practice of the CRS, which currently runs 3955 unit tests in approximately 30 seconds on GitHub [43]. We aim to provide a few hundred unit tests, which will complete in less time due to their smaller number, ensuring the execution time of our test set remains acceptable.

2.4.2 CI/CD Pipelines

Continuous Integration (CI) and Continuous Delivery (CD) pipelines serve as the automation frameworks of release engineering, streamlining the process of building, testing, and deploying software. These pipelines enhance development workflows by ensuring that changes are continuously integrated, tested, and delivered efficiently. They are widely used to help developers accelerate the development cycle.

The regression test suite we develop will ultimately leverage GitHub’s CI/CD pipeline tools, specifically GitHub Actions, for automated execution [44]. GitHub Actions are configured directly within the repository using configuration files and can be extended with pre-built actions available in the GitHub Marketplace [45].

2.5 YAML

YAML Ain’t Markup Language (YAML) is a human-readable data serialization language widely used for configuration files and data exchange [46]. It is designed to be a simpler alternative to other data serialization languages by relying on indentation for structure, colons for key-value pairs, and dashes for lists. YAML supports three core data types: mappings, sequences, and scalars. The example below demonstrates a simple key-value pair, a dictionary, and a list containing mixed data types:

```
1 simple_key: simple_value
2
3 dictionary_example:
4   sub_key1: sub_value1
5   sub_key2: sub_value2
6
7 list_example:
8   - 1
```

```
9  - two
10 - "3"
```

YAML can be used to configure applications, with the structured format of a YAML file for configuration referred to as its YAML *schema*. Many programming languages offer libraries that can parse YAML files, allowing applications to dynamically adapt their behavior based on the configuration data, as will be demonstrated in this work.

Chapter 3

Methodology

3.1 Objectives

The aim of this thesis is to propose a regression testing process for WAF engines, with the goal of supporting engine developers by adopting a continuous integration practice. This work is motivated by the current limitations in the CI/CD pipelines of the two mainline ModSecurity releases. At present, the pipelines perform only compilation checks to ensure that submitted changes do not break the build process. While such checks are essential for catching syntax errors, they fall short of providing meaningful assurance about the correctness of code changes. In the absence of automated testing at the integration stage, identifying regressions relies solely on manual code review by contributors and maintainers, which is both inefficient and prone to errors.

In this section, we define what constitutes WAF engine functional testing and outline the requirements a testing framework must meet to implement it effectively. We also briefly review the initial set of tools considered for this purpose.

3.1.1 Defining WAF Engine Functional Testing

Functional testing verifies whether a software system behaves according to its specifications. In this work, we will focus on testing WAF functionalities. To test WAF features, understanding the division of responsibilities between the engine and the rules is essential for designing meaningful tests. Since our goal is to support WAF engine developers, it is important to distinguish between different WAF testing postures.

One naïve approach is to treat the WAF as a *black-box appliance*, testing it as a whole without separating its components. This involves sending attack and non-attack requests to a WAF configured with a given engine and rule set, then checking whether the observed behavior aligns with the specified security policy. While this helps assess the overall effectiveness of the WAF, it gives no indication into whether failures stem from the engine or the rules.

A second approach focuses on testing the *rules*. Here, a known reference engine is used to evaluate whether a rule set correctly enforces its intended security policy. This helps rule developers identify whether adjustments are needed, assuming the engine behaves correctly.

Our interest lies in a third approach: testing the WAF *engine* itself. This means evaluating whether the engine, when given any valid configuration, correctly executes its enabled functionalities as specified in its documentation. This posture provides feedback to engine developers, helping them detect implementation issues.

3.1.2 Testing Framework Requirements

A framework aiming to implement WAF engine functional testing should fulfill the following requirements.

First, it should have the capability to provide the configuration that instruct the engine to activate its functionalities. Ideally, the framework should allow writing tests that verify the correct behavior of a functionality for *any configuration* that enables it. In essence, it must be able to instruct the WAF engine to use a functionality across all its possible variations.

Next, the framework should be able to run a *live instance* of the WAF engine with the provided configuration and exchange HTTP *traffic of any form* through it. This live traffic enables testing the engine along with its surrounding components in a complete, end-to-end setup.

Finally, the framework must be capable of checking the WAF engine's *output*. This involves verifying that expected modifications to the HTTP traffic occur, and that the expected logging is generated. This ensures that the WAF engine is capable of defending an application, but also provides visibility into traffic exchanged through it.

3.1.3 Initial Tool Evaluation

We started by exploring existing tools used for general networking or testing purposes to assess if WAF engine functional testing was achievable with them. We found a few issues with the existing tools.

First, we found that many general-purpose networking tools are *not designed to send malformed HTTP requests*. A relevant tool for the task should be capable of deliberate deviations from protocol specifications to ensure the WAF engine is robust when handling any kind of data. However, tools like `curl` adhere strictly to protocol standards by design, prioritizing compliance over flexibility [47, 48]. As a result, unless a tool is explicitly built to handle malformed requests, it is unsuitable for this kind of testing and should not be considered.

Secondly, we observed that many existing network testing tools are *overly complex* for our needs. While they are technically capable of sending malformed HTTP requests, they come bundled with a wide range of features that are irrelevant to our specific use case. For instance, Apache JMeter includes capabilities like load and stress testing, which are unnecessary for validating functional requirements. Similarly, dynamic application security testing tools such as ZAP and Burp Suite offer extensive features for vulnerability assessment, but their focus on security testing makes them impractical for functional testing.

Finally, the reviewed tools were *not designed to synchronously correlate requests with WAF logs entries*. As a result, we would have needed to develop a custom solution capable of

verifying changes in the logs while sending requests through an existing utility or library. While building such a tool from scratch would have given us full control and met all our requirements, it would also have been too time-consuming.

3.2 How CRS Does WAF Rule Testing

As an open source project, CRS has developed one of the most advanced and publicly accessible WAF testing practices. In this section, we examine their approach and the tools they have built to support it

3.2.1 The CRS Testing Practice

The CRS repository include tests for rules in the set. Developers are expected to provide both positive and negative tests for inclusion in the regression test set when contributing a new rule [49]. Positive tests demonstrate the types of attacks the rule is intended to detect, while negative tests ensure the rule does not trigger on benign traffic.

When a developer opens a pull request on the CRS GitHub repository, the project's CI/CD pipeline is triggered to validate the proposed changes. This pipeline runs the full *regression test suite* against the version of CRS in the pull request using a predefined infrastructure with ModSecurity [43]. In addition to verifying that no existing rules break, the pipeline performs *quantitative tests* to ensure that the new or modified rule does not increase the false positive rate at paranoia level 1, the rule set's default and most conservative configuration [50].

If no *regressions* are found, the proposed changes move on to manual review by a CRS maintainer. In cases where the tests identify a *regression*, and the issue is attributed to the rule under development, contributors are expected to revise their rule accordingly. CRS developers may open discussions with engine maintainers if testing points to broader issues within the WAF engines. For open source engines, these conversations are facilitated by the fact that both ModSecurity and Coraza, like CRS, are OWASP projects.

3.2.2 A Framework for Testing WAFs: go-ftw

Fortunately, many of the challenges involved in supporting a robust WAF testing practice have already been addressed by the CRS developers.

The tool

To write and execute test cases on a live WAF instance, CRS developers created their own framework: the *Framework for Testing WAFs* (FTW) [51].

FTW was introduced in 2017 shortly after the release of CRS 3.0. It was developed in response to the challenges faced by CRS developers in testing the rule set. They needed a tool that could:

- Send both *compliant* and *non-compliant* HTTP traffic through a WAF instance
- Inspect the WAF's *output* to verify whether the appropriate rules were triggered, by analyzing both *responses* and *error logs*

- Run *quickly enough* to be integrated into the continuous integration process for CRS
- Be *user-friendly* for test writers, by using an existing popular scripting language

FTW is used under a few key assumptions. First, it is not meant to manage the WAF instance itself. The tool assumes that the WAF is already live and configured with the rules under test. Second, its default operation relies on access to the WAF's error log, which should contain entries for every rule that matches the incoming traffic. Third, FTW is designed to execute all test cases in a single run without restarting or reconfiguring the WAF between tests.

To improve performance and maintainability, FTW was entirely rewritten in the Go programming language in 2021. The resulting utility is known as `go-ftw` [52].

An example test case

Test cases in FTW are written using the YAML scripting language. To better understand how the tool works, let's examine an example test case that illustrates parts of the schema used to write test cases [53]. The following test taken from the CRS regression test set targets rule 942100, which is designed to detect SQL injection attacks [54]:

```

1 meta:
2   author: "Christian Folini, azurit"
3   description: Various SQL injection tests
4 rule_id: 942100
5 tests:
6   - test_id: 1
7     desc: "Simple SQL Injection"
8     stages:
9       - input:
10         dest_addr: 127.0.0.1
11         headers:
12           Host: localhost
13           User-Agent: "OWASP CRS test agent"
14           Accept: text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
15         method: POST
16         port: 80
17         uri: "/post"
18         data: "var=1234 OR 1=1"
19         version: HTTP/1.1
20       output:
21         log:
22           expect_ids: [942100]

```

The example gives us an understanding of what is required by the tool to execute a test:

- The destination *address* and *port* are needed to generate transport layer packets.
- A complete description of the HTTP request defines the test *input*.
- Explicit checks define the expected *output*. These may target either the HTTP response or logs, as shown in the example. The tester must know the expected

content precisely, such as the rule IDs present in the error log.

- Additional *metadata* provides context during test execution, for readability in the standard output.

Evaluation for testing a WAF engine

It is clear that `go-ftw` provides a versatile set of features for a range of WAF testing uses, from validating local configurations to evaluating the security of remote deployments.

However, when it comes to WAF engine testing, several limitations become apparent:

- A single test typically requires writing around 30 lines, much of which consists of *repeated information* across a test set. This verbosity slows down development.
- `go-ftw` assumes that the WAF is already *up and running*, pre-configured with the *existing rule set*. Because CRS engages in WAF rule testing, they already have the rules they want to test. Engine testing however, requires providing a rule set that enables the functionalities to be tested.

These limitations highlight the gap between what `go-ftw` currently offers and what is needed to effectively support WAF engine testing.

3.2.3 An HTTP Reflector: albedo

Albedo is an HTTP application designed to respond to requests sent by `go-ftw` [55]. By default, it returns a response with an empty body. However, it can be configured to generate specific responses by including data in the request, which dictates the response for that particular transaction, in effect *reflecting* the response it was given. CRS developed Albedo to facilitate testing of rules that act on the HTTP response.

The reflection feature is particularly valuable for defining responses *at runtime*, removing the need for external files that the application must reference. This approach not only speeds up execution but also enhances maintainability by keeping response definitions within the unit test configuration files.

3.3 The ModSecurity Regression Test Set

In June 2024, Ervin Hegedüs created a new repository for the ModSecurity project, dedicated to hosting the ModSecurity Regression Test Set (MRTS) [56]. The goal of this repository is to provide a collection of utilities for creating regression tests for the two mainline releases of ModSecurity or SecLang compliant WAF engines like Coraza. MRTS aims to adapt the testing tools used by CRS for WAF engine testing.

3.3.1 Concept

The purpose of MRTS is to facilitate the testing of ModSecurity's behavior by generating *combinations of rules* and configurations enabling the functionality we want to test. From a YAML file, a utility will generate a rule set and associated `go-ftw` test files. The rules

are then expected to be loaded into a *live WAF instance*, before being tested by executing `go-ftw` with the generated test files.

MRTS is designed to generate a wide variety of combinations to thoroughly test the functionality of the engine. The goal is to ensure that *each feature behaves correctly across all relevant variations*. ModSecurity employs a range of rule targets, operators, transformations, and phases all of which can interact in complex ways. Therefore, it is essential to test these behaviors systematically to ensure the engine performs as expected in different conditions.

MRTS can ensure thorough coverage of the engine’s functionalities by generating rules through *templating*. A global rule template is defined, and each individual test case specifies a set of parameters to be dynamically substituted into the template. The framework then generates all corresponding rules, with the various parameter combinations producing the variations necessary to test the functionality across all scenarios deemed relevant by the test case author.

Additionally, MRTS leverages the strengths of `go-ftw` to test the WAF engine along with its surrounding components as an *end-to-end* setup. Furthermore, MRTS mitigates the task of manually writing repeated information in `go-ftw`’s YAML schema by replacing common data automatically.

3.3.2 Features of Version 0.1

We found the project in its early development stage, released as version 0.1, but it already included several valuable features.

The utilities available at this stage were `generate-rules.py` and `collect-rules.py`. The first utility is responsible for interpreting the YAML schema and generating rule sets and tests accordingly. The second utility helps determine the coverage of SecLang by the generated rules, indicating which tests may be missing.

To generate rules, the utility loads templates from a global configuration file. A template contains a SecLang configuration that uses *macros* to replace parameters with values defined in specific test files. The utility then processes each provided file, generating combinations of the template with the various values defined in the test case. It also assigns a unique rule ID to each rule as it is a required parameter in SecRules.

For generating tests, the utility automatically writes files in the FTW schema. Repetitive information, such as the destination address and port, is replaced in each test file. The test itself automatically checks for the presence in the error log of the rule ID associated with the specific rule being generated.

3.3.3 Example: the ARGS_GET Target Test

Let us illustrate how MRTS tests a specific functionality of ModSecurity to understand its use. The process begins with a configuration file that includes a `global` section, which defines the *templates* and *default values* to be used during rule generation:

```
1 global:
2   version: MRTS/0.1
3   baseid: 100000
```

```

4  default_operator: "@rx"
5  templates:
6  - name: "SecRule for TARGETS"
7    template: |
8      SecRule ${TARGET}$ "${OPERATOR}$ ${OPARG}$" \
9        "id:${CURRID}$,\
10       phase:${PHASE}$,\
11       deny,\
12       t:none,\
13       log,\
14       msg:'%{MATCHED_VAR_NAME} was caught in phase:${PHASE}$',\
15       ver:'${VERSION}$'"
16  default_tests_phase_methods:
17  - 1: get
18  - 2: post
19  - 3: post
20  - 4: post
21  - 5: post

```

This template is designed to test the *target* functionalities of ModSecurity SecRules. One example is the ARGES_GET target, which inspects the values of HTTP GET request arguments to evaluate rule matches. Below is the test file for this functionality, which was already included in the repository:

```

1  target: ARGES_GET
2  rulefile: MRTS_004_ARGS_GET.conf
3  testfile: MRTS_004_ARGS_GET.yaml
4  templates:
5  - SecRule for TARGETS
6  colkey:
7  - - ''
8  - - arg1
9  - - arg1
10 - - arg2
11 - - /^arg_.*$/
12 operator:
13 - '@contains'
14 oparg:
15 - attack
16 phase:
17 - 1
18 - 2
19 - 3
20 - 4
21 testdata:
22   phase_methods:
23     1: get
24     2: get
25     3: get
26     4: get
27   targets:

```

```

28 - target: ''
29 test:
30     data:
31         foo: attack
32 - target: arg1
33 test:
34     data:
35         arg1: attack
36 - target: arg2
37 test:
38     data:
39         arg2: attack
40 - target: /^arg_.*$/
41 test:
42     data:
43         arg_foo: attack

```

We can see that the test uses the `SecRule` for TARGETS template and provides multiple parameters to generate various rule and test combinations.

First, it specifies the *request argument names* the rule should inspect under `colkey`. The first entry omits the argument name, meaning all arguments are considered. The second entry specifies `arg1`, indicating the rule should target that specific argument. The third entry includes both `arg1` and `arg2`, meaning the rule should inspect either. The final entry uses a regular expression, instructing the rule to match any argument name that fits the given pattern.

Next, the test defines the *operator* `@contains` and the *operator argument* `attack`, instructing the rule to check whether the targeted request arguments contain the substring "attack" in their values.

It then specifies the *phases* for which rules should be generated. In this example, the rules are created for phases 1 to 4.

Finally, it defines four *HTTP requests* that are expected to trigger the corresponding generated rules, serving as the input for validating the functionality's behavior under the specified conditions.

The rules will be generated and saved in a file named `MRTS_004_ARGS_GET.conf`. For example, the following two rules are among the rules created:

```

1  SecRule ARGS_GET "@contains attack" \
2      "id:100035,\
3      phase:4,\
4      deny,\
5      t:none,\
6      log,\
7      msg:'%{MATCHED_VAR_NAME} was caught in phase:4',\
8      ver:'MRTS/0.1'"
9
10 SecRule ARGS_GET:arg1 "@contains attack" \
11     "id:100036,\
12     phase:1,\

```

```
13 deny,\
14 t:none,\
15 log,\
16 msg:'%{MATCHED_VAR_NAME} was caught in phase:1',\
17 ver:'MRTS/0.1''"
```

We can observe that the *macros* in the template have been replaced with the corresponding values defined in the test file. The first rule looks for any argument inside an HTTP GET request containing the substring "attack" and executes in phase 4. The second rule looks in the `arg1` argument, if present in the GET request, to look if it contains the substring "attack" and executes in phase 1. Each rule has been assigned a unique ID incrementally. The `ARGS_GET` test results in the generation of **16 distinct rules** because of the combination of four different argument names across four separate phases.

The FTW tests will be saved in files named `MRTS_004_ARGS_GET.yaml`, with the rule ID appended to each file name. Each file corresponds to a specific rule generated earlier. A total of **32 unit tests** will be created for `go-ftw`, with the distribution as follows:

- **Tests 1 to 16** target the version of the rule that checks any argument by sending requests containing all four possible argument variations.
- **Tests 17 to 20** target the version of the rule inspecting `arg1`, and they trigger by sending requests where only `arg1` is present.
- **Tests 21 to 28** target the rule that checks both `arg1` and `arg2`. These tests are triggered by sending requests that include either `arg1` or `arg2` as the parameter.
- **Tests 29 to 32** are designed to target the rule that inspects arguments matching the regular expression `^arg_.*`. These tests are triggered by requests containing parameters such as `arg_foo`.

These unit tests comprehensively verify multiple rule variations that use `ARGS_GET` as their target, covering the relevant phases and parameters associated with this functionality.

Chapter 4

Framework Features Design

Now that we have defined the approach we aim to develop and selected MRTS as the project in which to implement it, we can begin contributing. We initially encountered MRTS in a very early development state, with only its general concepts implemented and very few features or test cases available. Our starting point will be to enhance the framework by adding features we consider necessary to make it more complete. In this chapter, we will begin by discussing the development process used to define the features we intended to add. We will then outline the categories of features we implemented, and finally, we will present the specifications of the features that were proposed to the project.

4.1 Development Process

The contributions to MRTS began with a phase dedicated to defining and specifying the features deemed necessary prior to expanding the suite of test cases. This initial work was conducted through a series of meetings with Marc Stern, the lead developer of ModSecurity2, during which we discussed and refined the proposed features. Following these discussions, I drafted detailed specifications for each feature. After several iterations of feedback and revision with Marc, we finalized our specification document. This document was then used to present the proposed features to Ervin Hegedüs, the maintainer of MRTS and lead developer of libmodsecurity3, as a basis for propositions to the project.

In addition to the planned features, a few minor enhancements were introduced opportunistically when their necessity became apparent during development.

4.2 Categories of Features We Added

The features we added can be classified into five categories:

- **Rule Generation Improvements:** A set of features aimed at extending the framework's capabilities for generating SecLang configuration for the WAF engine.
- **Testing Improvements:** Features designed to enhance the framework's ability to generate test files compatible with `go-ftw`.

- **Constants in Configuration Files:** a feature intended to assist developers in writing MRTS configuration files.
- **WAF Infrastructures:** A group of features focused on enabling the easy execution of the test suite on dedicated infrastructures.
- **Auxiliary Features:** Smaller contributions whose purpose were to help the project in additional ways. They are detailed in appendix A.

The remainder of this chapter presents the specification and design justification of each feature, organized by category.

4.3 Rule Generation Improvements

This section describes the features that improve rule generation in the testing framework. There are three features: the first two introduce new macros-*ACTIONS* and *DIRECTIVES*-for generating combinations, and the third adds the ability to insert content between around configurations.

4.3.1 ACTIONS and DIRECTIVES macros

Objective

The objective of those features is to allow for additional combinations to be formed on other aspects of the WAF configuration by introducing two new macros usable in templates: *ACTIONS* and *DIRECTIVES*. The first is for actions to be added to SecRules. It can also be used for *SecAction* directives, a SecLang directive resembling a SecRule but whose condition would be always true as it triggers the set of actions it is assigned for every transaction. The second is for adding entire directives to the configuration. The actions and directives are added in the test configuration file. It is possible to add multiple actions or directives and even multiple sets of them to be used in test case combinations.

This feature provides greater flexibility to test writers by enabling them to specify a larger part of SecLang in test configuration files, rather than being limited to writing more globally defined templates. A single template with those macros can be used to support generation for multiple test cases. It also expands the range of possible combinations when defining a test case. For example, they allow test cases checking if the functionality of a rule is compromised by certain actions inside the rule or adjacent directives. With this feature, the framework becomes more powerful.

Specification for the ACTIONS macro

We will illustrate those features with the ACTIONS macro. The use of the DIRECTIVES macro is similar and is fully illustrated at appendix B.1.

As with other macros, the idea is to introduce a new macro, `${ACTIONS}$`, which can be used in any template:

```

1 global:
2   - name: "ACTIONS Demonstration"
3     template: |
4       SecRule ${TARGET}$ "${OPERATOR}$ ${OPARG}$" \

```

```

5      "id:${CURRID}$/, \
6      phase:${PHASE}$/, \
7      log, \
8      ${ACTIONS} $"

```

In the YAML schema, a new **actions** section is used to define the actions to be replaced, represented as a list of lists:

```

1  target: ARGS
2  rulefile: MRTS_999_EXAMPLE.conf
3  testfile: MRTS_999_EXAMPLE.yaml
4  templates:
5    - ACTIONS macro example
6  operator:
7    - '@contains'
8  oparg:
9    - attack
10 actions:
11   - action:
12     - deny
13     - status:403
14   - action:
15     - deny
16     - status:404
17     - setenv:'variable=abc'
18 phase:
19   - 2

```

This example generates the following configuration:

```

1  SecRule ARGS "@contains attack" \
2    "id:100001,\
3    phase:2,\
4    log,\
5    deny,\
6    status:403"
7
8  SecRule ARGS "@contains attack" \
9    "id:100002,\
10   phase:2,\
11   log,\
12   deny,\
13   status:404,\
14   setenv:'variable=abc'"

```

Implementation

Implementations were done by modifying the `generate-rules.py` script and were merged on the 13/01/2025 for the ACTIONS macro [57, 58] and on the 14/01/2025 [59, 60] for the DIRECTIVES macro.

4.3.2 Directives Before(-Each) and After(-Each) Generation

Objective

The objective of this feature is to allow for the addition of directive blocks placed either before or after each generated rule, or surrounding the entire set of generated rules for a given test case. These directives are defined in the test configuration file.

This feature permits the setup of specific settings that are required for particular test scenarios. For example, certain variables necessary to implement a test case's logic can be declared or updated separately from the template employed. Overall, this feature enhances the flexibility of the testing framework, enabling it to better accommodate complex testing requirements.

Specification

In the configuration files, a **generation** section is added, containing special key-value pairs. The following keys can be defined:

- **before**: Adds the specified directives before all generated rules.
- **after**: Adds the specified directives after all generated rules.
- **before_each**: Adds the specified directives before each generated rule.
- **after_each**: Adds the specified directives after each generated rule.

For example:

```
1 generation:  
2   before: SecAction "id:100,phase:1,pass,ctl:requestBodyProcessor=XML"  
3   after_each: SecAction "id:${CURRID}$,phase:${PHASE}$,pass,log,msg: 'Matched  
   ↳  %{MATCHED_VAR_NAME}'"
```

Would generate the following configuration :

```
1 SecAction "id:100,phase:1,pass,ctl:requestBodyProcessor=XML" # Before  
2  
3 # Generated rule with template  
4 SecRule ARGS "@contains attack" "id:100001,phase:2,log"  
5  
6 SecAction "id:100002,phase:2,pass,log,msg: 'Matched %{MATCHED_VAR}'" # After-Each  
7  
8 # Generated rule with template  
9 SecRule ARGS "@contains attack" "id:100003,phase:3,log"  
10  
11 SecAction "id:100003,phase:3,pass,log,msg: 'Matched %{MATCHED_VAR}'" # After-Each
```

Implementation

Implementation was done by modifying the `generate-rules.py` script and was merged on the 28/02/2025 [61, 62].

4.4 Testing Improvements

This section describes the features that enhance the testing capabilities of the framework. Overall, these features enable the use of additional functionalities from `go-ftw` when generating intermediary test files. The first feature allows the use of the full set of checks in `go-ftw`, while the second and third features provide greater flexibility in defining what input request should be sent.

4.4.1 Extend Check Methods on Logs and Responses

Objective

The objective of this feature is to extend configuration files to enable the generation of tests using a variety of check methods. The checking features provided by `go-ftw` allow for detailed examination of both the HTTP response and error log. With this feature, we can for example write test cases that verify the transaction was blocked or modified as expected. We could also write negative tests, that is tests that verify a rule did not match, by checking for the absence of the associated log entry. By leveraging all of `go-ftw`'s check capabilities, we enhance our framework's ability to design more complex and diverse test cases.

Specification

The feature introduces the `output` key under the `test` section.

```
1 testdata:
2   phase_methods:
3     2: get
4   targets:
5     - target: ''
6       test:
7         output:
8           status: 400
9           response_contains: HTTP/1\..1
10          log:
11            expect_ids:
12              - 123456
13            no_expect_ids:
14              - 123456
15            match_regex: id[:\s"]*123456
16            no_match_regex: id[:\s"]*123456
17            expect_error: true
```

If `output` is not specified, the default behavior of the framework remains to check for the presence of the current rule ID in the logs. However, if the `output` section is specified in the configuration file, it will be included in every generated test file. Overriding the `output` section of tests allows test case writers to utilize the full set of checking features provided by `go-ftw`, which, at this time, are [63]:

- `status`: check if the HTTP response status matches the specified code
- `response_contains`: check if the HTTP response matches the given regex

- `log.expect_ids`: check if all IDs in the array of IDs are present in the log entry
- `log.no_expect_ids`: check if all IDs in the array of IDs are absent from the log entry
- `log.match_regex`: check if a regex matches the log entry
- `log.no_match_regex`: check if a regex does not match the log entry
- `expect_error`: check if the response status code is an error code (4XX)

Multiple different checks can be defined for each test case. It is important to note that a test is considered failed if at least one check does not pass, and it is considered successful only if all checks pass.

To conveniently write negative test cases in our framework, the `no_expect_ids` check is automatically updated with the current rule ID being tested, similar to how `expect_ids` is used as the framework’s default testing check.

Implementation

Implementation was done by modifying the `generate-rules.py` script and was merged on the 15/01/2025 [64, 65]. For negative test cases, the changes were proposed alongside the negative test case we contributed in this work on the 25/05/2025 [66].

4.4.2 Sending Malformed Requests

Objective

The objective of this feature is to extend configuration files to enable the generation of tests that send requests using the `encoded_request` feature of `go-ftw`.

Previous designers of `go-ftw` faced a limitation: testing WAF request parsing requires the ability to send malformed requests. While the tool already supported this to some extent, it was constrained by YAML’s limitations. Notably, YAML cannot parse invalid Unicode characters, making it impossible to include them in test requests. Another issue arises when specifying multiple HTTP headers with the same name: YAML parsers typically overwrite duplicate keys, preventing accurate representation of such headers in the `headers` section of the FTW YAML schema.

To address these issues, `go-ftw` introduced the `encoded_request` feature, which allows test writers to specify an entire HTTP request as a base64-encoded string. This string is decoded to raw bytes and sent as-is, overcoming the YAML parsing limitations. Allowing the use of `encoded_request` in MRTS files enables test cases with input requests containing invalid Unicode characters or duplicate headers, which could reveal issues with the engine’s HTTP parsing.

Contributing this feature reassured us that the framework was capable of sending HTTP traffic of any form, as prescribed by our requirements from section 3.1.2. It will be particularly useful when contributing test cases for the parsing functionalities of ModSecurity, which is beyond the scope of this work. A detailed illustration of its usage can be found in appendix B.2.

Implementation

Implementation was done by modifying the `generate-rules.py` script and was merged on the 15/01/2025 [67, 68].

4.4.3 Manually Defining URI for Using Albedo's Reflection Capabilities

Objective

The objective of this feature is to allow overwriting the default `uri` key in the generated `go-ftw` configuration. This feature is particularly useful for leveraging `albedo`'s reflection capability.

By default, the HTTP response from the `albedo` backend is a response with status code 200 and an empty body. For certain test cases, we may want to receive a specific response from the backend to verify correct transmission through the WAF or to test rules involving response-related variables. To support this, `albedo` provides two endpoints: `/reflect`, which generates a response based on data in the current transaction's request, and `/configure_reflection`, which allows defining a new endpoint that, when accessed, returns a specified response.

For the framework to use `albedo`'s reflection features, it must modify the `uri` field in the `go-ftw` configuration. This feature also brings the possibility to include input data for test cases in the request's URI. By enabling use of these capabilities, this feature supports the creation of more complex and diverse test cases.

Specification

The `uri` field can be defined under the `input` section:

```
1 testdata:
2   targets:
3     - target: ''
4     test:
5       data: '{"status": 201, "body": "<html>reflected-token</html>"}'
6       input:
7         headers:
8           - name: Content-Type
9             value: application/json
10        uri: '/reflect'
11      output:
12        status: 201
13        response_contains: "reflected-token"
```

The above example is a passing test that sends a request with URI `/reflect` instructing `albedo` to generate a response with status code 201 and an HTML document containing "reflected-token" in its body.

Implementation

Implementation was done by modifying the `generate-rules.py` script and was merged on the 03/03/2025 [69].

4.5 Constants in Configuration Files

This section describes a feature that improves the experience of writing configuration files. The feature introduces the use of constants to reduce repetitive writing.

Objective

The objective of this feature is to introduce constant variables that can be substituted into parameter values when parsing MRTS YAML files. Two types of constants are supported: global cross-file constants and local per-file constants.

While YAML already supports a similar feature through anchors and aliases, those are limited to the file in which they are defined [70]. This is useful for large YAML files, but in this project we typically work with small files, usually under 50 lines, to maintain readability. We favor this approach by limiting each configuration file to testing a single functionality. Nevertheless, using cross-file constants helps eliminate repeated patterns in the schema and provides a quality-of-life improvement for test writers. This feature will be particularly useful once the test set contains a larger number of cases.

Specification

For global constants, a `default_constants` section is added as a subsection of `global` sections:

```
1 global:
2   version: MRTS/0.1
3   default_constants:
4     One: 1
5     Two: "two"
6     Three: 'three'
```

For per-file constants, a `constants` section is added and is located as a root section:

```
1 rulefile: MRTS_999_EXAMPLE.conf
2 testfile: MRTS_999_EXAMPLE.yaml
3 constants:
4   One: 'one'
```

Constants can have any value admissible in YAML, including list and dictionaries:

```
1 constants:
2   Dict:
3     foo: bar
4   List:
5     - 1
6     - '2'
```

Constants are accessible within other sections by enclosing their names with the `~{` prefix and `~}` suffix, signaling to the parser that the enclosed content is a constant to be substituted. The `~` symbol was chosen because it avoids conflicts with reserved characters in YAML, and does not interfere with syntax used in SecLang.

```
1 testdata:
2   targets:
3     - target: ''
4     test:
5       input:
6       headers:
7         - name: Content-Type
8           value: ~{XML}~
9         - name: Content-Length
10          value: ~{ONE}~
```

Implementation

Implementation was done by modifying the `generate-rules.py` script and was merged on the 15/01/2025 [71, 72].

4.6 WAF Infrastructures

This section describes the infrastructures that enable executing the test set on a live WAF along with its associated components. The features introduce the necessary configurations to prepare the two mainline releases of ModSecurity for testing.

Objective

The objective of the infrastructure features is to provide the necessary configuration files and automation scripts to run a WAF to be tested with the test set. In a CI/CD pipeline, the software environment is determined by the operating system and the specific software being tested. For the WAF to run correctly, its binaries must be properly installed by the automation software used in the pipeline. On the other hand, the configuration files are responsible not only for specifying which WAF rules to apply, but also for defining where information is logged, where traffic is routed after passing through the WAF, what runtime parameters are set, etc. In this project, we provide default configurations that connect to the framework's generated rules, setting a standard configuration suitable for CI/CD pipelines and offering a reference for developers.

Alongside these configuration files, we contributed the `mrts.py` utility to manage the entire toolchain required to execute the test set. The utility manages the generation of files, the web servers and their WAF configuration, `albedo` for the backend application, and `go-ftw` to execute tests. It is fully described in appendix A.3.

The provided infrastructures support testing of the two mainline releases of ModSecurity, bringing the project closer to integration into their respective CI/CD pipelines.

Specification

The provided configuration files are designed to closely resemble the default installations of the respective web servers, while also integrating ModSecurity as a reverse proxy. They support running the web servers as non-privileged users by providing writable file locations for logs. Figure 4.1 illustrates the test bed architecture. In this setup, the web servers function as inline reverse proxies, listening at address 0.0.0.0 on port 80 for test requests sent by `go-ftw`. They forward incoming traffic to 0.0.0.0 on port 8000, where `albedo` handles the requests and generates the corresponding responses.

Error logs are written to `MRTS/config_infra/[WAF infrastructure directory]/infra/log/error.log`, where they can be read by `go-ftw` for validation.

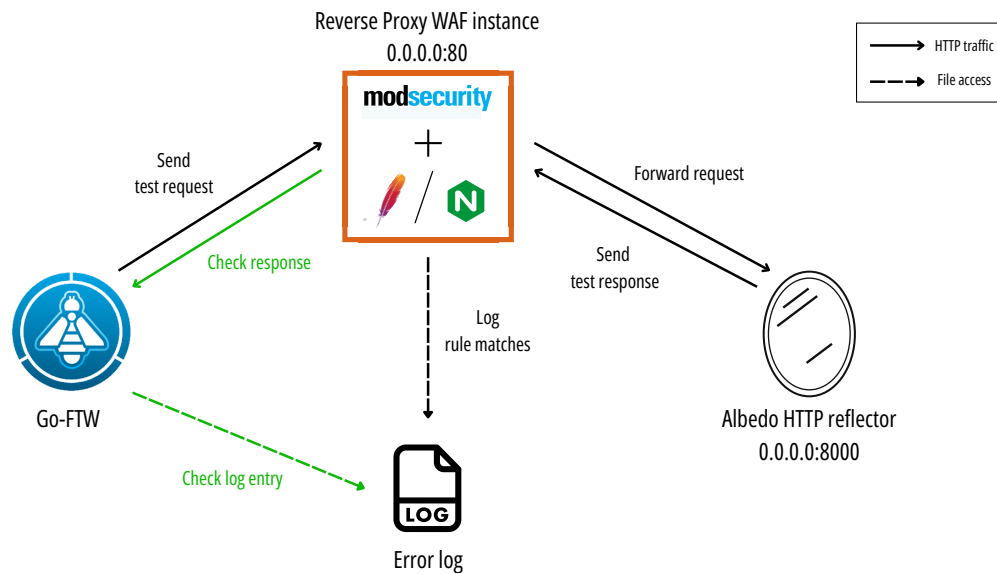


Figure 4.1: Test Bed Architecture

Apache + ModSecurity2 on Ubuntu

The infrastructure provided to test ModSecurity2 is enabled through the `apache2` utility, intended for Debian-based Linux distributions. Since ModSecurity2 is designed as a module for the Apache web server, the configuration assumes that `apache2` and the `security2_module` have been correctly installed.

While ModSecurity2 is available on multiple platforms, Debian-based distributions are an appropriate first target due to their widespread use among developers and compatibility with common automation systems. In particular, GitHub Actions provides containers based on the `ubuntu-latest` image.

Nginx + libmodsecurity3 on Linux

The infrastructure provided to test libmodsecurity3 is enabled through the `nginx` utility, designed for Linux distributions. Since libmodsecurity3 functions as a standalone library that interacts with web servers via connectors, the configuration assumes that `nginx` has been correctly installed along with the Nginx connector for libmodsecurity3 as a dynamic module.

While both libmodsecurity3 and its Nginx connector are available on multiple platforms, providing the initial configuration for Linux is appropriate given its popularity among developers and widespread use in automation systems. In particular, GitHub Actions provides multiple Linux-based containers.

Implementation

Implementation was done by providing the configuration files and automation scripts located under `apache2_ubuntu` and `nginx_linux`. The Apache infrastructure was merged on the 21/04/2025 [73, 74] and the Nginx infrastructure was proposed on the 23/04/2025 [75].

Chapter 5

Test Design and Evaluation

After extending the capabilities of MRTS, we can move forward by demonstrating the testing framework through the contribution of new test cases. However, before regression testing can be fully integrated into ModSecurity’s continuous integration pipelines, the test set must reach a sufficient level of maturity. We must demonstrate that the test set is capable of uncovering defects in major features to support the claim that it can enhance confidence in proposed changes by reducing the risk of introducing functional or security regressions. To this end, we will spend the remainder of this work contributing and evaluating test cases.

In this chapter, we begin by outlining the methodology used for selecting test cases, along with the test categories we defined to structure our contributions. We then provide a detailed discussion of each test’s purpose and implementation, organized according to these categories. After that, we run the test set on both release lines of ModSecurity and analyze the results. Finally, we assess the completeness of the test set to provide insight into what remains untested.

5.1 Test Case Selection and Categories

Contributing a test set capable of evaluating all the functionalities of the ModSecurity engines will require many hours of development, even with a framework tailored for WAF engine testing. We found the MRTS project with only 5 tested functionalities, generating 54 SecRules and 110 `go-ftw` unit tests. Thoroughly testing all requirements will demand hundreds of additional test cases to cover every usage and combination of a WAF engine configuration. As contributing a complete test set is beyond the scope of this work, our goal is to propose a general direction for its development through the contribution of test cases at the project’s currently early development stage.

In this section, we will discuss the selection methodology used to choose which test cases to provide. Following that, we will present each contributed test case by outlining the engine functionality being tested, explaining the rationale behind the test case, and providing insight into the generated rules and unit tests.

5.1.1 Objectives

To determine a suitable selection of test cases, we first outline its objectives:

- **Advance the maturity of the test set** to bring it closer to a first release, enabling its integration into the CI/CD pipelines of ModSecurity, which will help developers with continuous integration.
- **Provide initial test cases** for functionalities that are among the most *critical* for the engine, ensuring that essential features behave as expected.
- **Demonstrate the capabilities of our framework** through the test cases, helping future developers understand the relevance of MRTS and the features introduced in chapter 4.

The test cases we contribute constitute our attempt to address these objectives within the scope of this work.

5.1.2 Building on Previous Work

Before we involved ourselves with this project, MRTS contributors had provided 5 test cases along with the initial scripts to generate the corresponding rules and tests. These 5 test cases all focus on ModSecurity’s ability to store information about the current HTTP transaction in variables. The information held in these variables can pertain to various aspects of the transaction, such as data present in the request and response, or data computed by ModSecurity itself such as extrapolated details based on the transaction or internal performance metrics. These variables play a crucial role in rule writing, as they can be referenced in the SecRule directive to identify which part of the HTTP transaction should be inspected to determine whether a rule should perform a set of actions. They are often referred to as rule *targets*.

Since work has already begun on SecRule targets, and given that targets are a major feature of ModSecurity, it is logical for us to continue this effort to help achieve the objectives outlined above. As we build upon earlier work, we will examine the previous design decisions that will most significantly influence our own.

A first template: SecRules for TARGETS

The first target tests use a simple template intended to create generic combinations of a SecRule matching targets to conditions, composed of operators and their arguments:

```

1 - name: "SecRule for TARGETS"
2   template: |
3     SecRule ${TARGET}$ "${OPERATOR}$ ${OPARG}$" \
4       "id:${CURRID}$,\
5       phase:${PHASE}$,\
6       deny,\
7       t:none,\
8       log,\
9       msg:'%{MATCHED_VAR_NAME} was caught in phase:${PHASE}$',\
10      ver:'${VERSION}$'"

```

When triggered, this SecRule interrupts the transaction and logs a message. Since the template is effective for testing targets, we will reuse it whenever applicable to generate our SecLang configurations. In cases where it is not suitable, we will derive from it a new template and specify the modifications we introduced.

Testing the relevant phases

When writing test cases, we must carefully select the phases relevant to the functionalities we aim to test. Due to historical reasons, the current implementation of the ModSecurity engines may change the value of targets across phases. For example, consider the `REQUEST_URI` target, which contains the request's uri: in phase 1, it may appear URL-encoded (e.g., `%2F%3Fthis%2Bis%2Ba%2Buri`), while in phase 2 and onward, it may appear URL-decoded (e.g., `/?this+is+a+uri`). Covering as many relevant phases as possible helps identify issues caused by such transformations. However, including irrelevant phases increases the number of generated unit tests and, consequently, the test suite's execution cost unnecessarily.

The five existing test cases of the set test their functionalities from phases 1 to 4, and don't test them in phase 5. This design decision comes from the fact that the targets tested, like for most targets, obtain their value from the data exchanged in the HTTP transaction. Testing the phases for which the targets are usually used, such as phase 1 and 2 for targets related to the request and phase 3 and 4 for targets related to the response, would be sufficient to find the most severe regressions they might be subjected to. For good measure and because it can be afforded for the moment, it was decided by previous test contributors for test cases of targets related to the HTTP request to cover phases 1 to 4. We will continue to use this principle for the test cases we will contribute, in particular those related to the HTTP requests.

As we will see, some targets still get updated in the fifth logging phase, or at least we suspect they could. In those instances, we will justify why we include the fifth phase as a parameter to the test case.

The state of the MRTS framework

In its current form, MRTS is primarily designed to test one category of ModSecurity's functionalities: the *SecLang rule engine*. Testing the rule engine is useful to ensure the engines meet the requirements described in their reference guides, which mostly document SecLang. The rule engine also relies on other features provided by ModSecurity, namely parsing, buffering, and logging. However, test cases specifically targeting these features can be designed and should be included for the regression test set to be thorough. In this work, because we also wanted to demonstrate extensively the current capabilities of MRTS, we chose to limit our contributions to tests for the rule engine.

Another important point is that MRTS is designed to generate numerous distinct unit tests without optimizing for efficiency. Since the goal is to build a regression test set from the ground up, the tool focuses on exhaustively generating all relevant combinations. As long as the overall execution time remains acceptable for integration into CI/CD pipelines, this is not an issue. Consequently, we define test combinations liberally throughout this work, with the assumption that any performance concerns can be addressed later through optimization strategies if necessary.

5.1.3 Selected Cases

Our test case selection is justified in several ways.

First, we selected targets based on their *usefulness to rule developers*. Not all targets are

equally valuable for constructing effective security policies. Some are far more frequently used, as can be observed in the core rule set [76]. By verifying the correct behavior of the most commonly used targets in CRS, our tests help detect regressions in features critical to the widely deployed core rule set, thereby hopefully avoiding the introduction of defects with the greatest potential security impact on real-world deployments of the engines. To know the frequency of targets used by CRS, we produced a table containing each occurring target in CRS 4.8.0, one of the recent releases of CRS. It is available in appendix C.

Secondly, we included targets that, while less commonly used by rule developers, serve to *demonstrate the framework's capabilities*. They exercise framework features not covered by other test cases, thereby showcasing the relevance of our previous contributions.

Furthermore, we contributed targets that required testing logic similar to existing cases, demonstrating how easily additional tests can be created for related functionalities. To avoid repeating similar descriptions, these tests are fully documented in appendix D, and we will reference them when discussing the test cases they resemble.

Finally, when appropriate, we included tests related to past or ongoing security and functional issues to demonstrate the framework's ability to identify incorrect or ambiguous behavior.

5.2 Tests for Targets Popular in CRS

We will now discuss the contributed test cases targeting variables frequently used in the core rule set. The selection is based on their frequency of use in CRS version 4.8.0.

5.2.1 002 - ARGS

The ARGS target contains the HTTP request arguments. ARGS is employed 175 times in CRS 4.8.0. It closely corresponds to the ARGS_NAMES target, which contains the names of HTTP request arguments and appears 171 times in CRS 4.8.0. The similar test case for ARGS_NAMES is detailed in appendix D.1.

Functionality

The ARGS variable refers to the collection of all arguments included in an HTTP request. Requests can contain arguments in two locations, depending on the HTTP verb used. For HTTP GET requests, arguments are found in the query string within the request header. For other HTTP requests, most commonly POST requests, arguments are located in the request body. The ARGS variable retrieves all arguments from the request, regardless of their specific location. SecLang also provides targets ARGS_GET and ARGS_POST, referring to the collection of arguments in an HTTP request located in the query string or in the body respectively.

The ARGS variable is a collection, meaning it holds all request arguments indexed by their name. Collection variables can be used directly in a rule to indicate that the rule should match against any argument in the request. Alternatively, a collection key, either a string or a regular expression, can be supplied to target only the arguments whose names match the given key.

Here are some example SecRules, similar to the ones generated by our test:

```
1 SecRule ARGS "@contains attack" "id:0, phase:1, log"
```

Will inspect all request arguments and trigger if one of them contains "attack".

```
1 SecRule ARGS:arg1|ARGS:arg2 "@contains attack" "id:0, phase:1, log"
```

Will inspect request arguments named "arg1" or "arg2" and trigger if one of them contains "attack".

```
1 SecRule ARGS:/^arg./ "@contains attack" "id:0, phase:1, log"
```

Will inspect request arguments starting with "arg_" and trigger if one of them contains "attack".

Test

To test the ARGS target, we generate SecRules with the following parameters:

- Template: SecRule for TARGETS
- Target: ARGS
- Phases: [1],2,3,4
- Operator: @contains
- Argument: attack
- Collection Keys: None, arg1, arg1 & arg2, /^arg_.* /

This test generates **16 SecRules** that inspects ARGS and triggers if it contains "attack". The variations span four phases and use four different collection keys. We begin testing at phase 1, where the ARGS target first becomes available for GET requests. Following previous decisions, we test the target up to phase 4. The test covers rules using no collection key, a single static key, two static keys, and a regular expression key to cover the various ways collection keys may be used.

We generate four requests with the following arguments: `foo=attack`, `arg1=attack`, `arg2=attack`, and `arg_foo=attack`. Each request is designed to trigger the rule with no collection key. Additionally, the last three requests are intended to trigger the rules with the `arg1`, `arg1 & arg2`, and `/^arg_.* /` collection keys, respectively.

To test the entire feature, we test rules with both GET and POST requests to cover the two possible locations for arguments. This is implemented through two variations of the test. Variation A runs from phase 1 to 4 using GET requests, where arguments are found in the query string. Variation B runs from phase 2 to 4 using POST requests, where arguments are placed in the request body, which is only accessible from phase 2. These variations have identical logic to the `ARGS_GET` and `ARGS_POST` test cases. The `ARGS_GET` test case has the same logic as variation A, and was already present in the test set. The `ARGS_POST` test case has the same logic as variation B, and was contributed to the test set as part of this work. A detailed description of this test can be found at appendix D.2.

With both variations, the test case generates **56 unit tests**.

Implementation

The ARGS test contributed was a rewrite of the originally present ARGS test. The original test used a GET request in phase 1 and POST requests in phase 2 to 4, but this logic forgot to test the target for GET requests in phase 2 to 4. The test was implemented by providing files `CONF_002_TARGET_ARGS_A-GET.yaml` and `CONF_002_TARGET_ARGS_B-POST.yaml`, and was merged on the 20/04/2025 [77].

5.2.2 059 - REQUEST_COOKIES

The REQUEST_COOKIES target contains the HTTP request cookies. REQUEST_COOKIES is employed 151 times in CRS 4.8.0. It closely corresponds to the REQUEST_COOKIE_NAMES target, which contains the names of HTTP request cookies and appears also 151 times in CRS 4.8.0. The similar test case for REQUEST_COOKIE_NAMES is detailed in appendix D.4.

Functionality

The REQUEST_COOKIES variable refers to the collection of all cookies included in an HTTP request. HTTP cookies are pieces of data used to maintain state across multiple requests and are transmitted via the `Cookie` header in HTTP requests [78].

Like the ARGS variable, REQUEST_COOKIES is a collection, meaning it stores all cookies indexed by their names. Individual cookies can be accessed using collection keys, allowing rules to inspect specific cookie values or search for patterns across multiple cookie names.

Here are some example SecRules, similar to the ones generated by our test:

```
1 SecRule REQUEST_COOKIES "@contains attack" "id:0, phase:1, log"
```

Will inspect all cookies and trigger if one of them contains "attack".

```
1 SecRule REQUEST_COOKIES:cookie1|REQUEST_COOKIES:cookie2 "@contains attack" "id:0,  
↪ phase:1, log"
```

Will inspect cookies named "cookie1" or "cookie2" and trigger if one of them contains "attack".

```
1 SecRule REQUEST_COOKIES:/^cookie_.*/ "@contains attack" "id:0, phase:1, log"
```

Will inspect cookies starting with "cookie_" and trigger if one of them contains "attack".

Test

To test the REQUEST_COOKIES target, we generate SecRules with the following parameters:

- Template: SecRule for TARGETS

- Target: REQUEST_COOKIES
- Phases: 1,2,3,4
- Operator: @contains
- Argument: attack
- Collection Keys: None, cookie1, cookie1 & cookie2, /^cookie_.*\/

This test generates **16 SecRules** that inspects REQUEST_COOKIES and triggers if it contains "attack". The variations span four phases and use four different collection key configurations. We begin testing at phase 1, where the REQUEST_COOKIES target first becomes available. Following previous decisions, we test the target up to phase 4. The test covers rules using no collection key, a single static key, two static keys, and a regular expression key to cover the various ways collection keys may be used.

We generate four requests with the following headers: `Cookie:cookie=attack`, `Cookie:cookie1=attack`, `Cookie:cookie2=attack`, and `Cookie:cookie_foo=attack`. Each request is designed to trigger the rule with no collection key. Additionally, the last three requests are intended to trigger the rules with the `cookie1`, `cookie1 & cookie2`, and `/^cookie_.*\/` collection keys, respectively. Furthermore, 3 requests are defined containing headers: `Cookie:cookie1=attack;cookie2=hello`, `Cookie:cookie1=hello;cookie2=attack` and `Cookie:cookie_bar=hello;cookie _ foo=attack; cookie_foobar=world` intended to trigger the SecRule with collection key `cookie1 & cookie2` and `/^cookie_.*\/` respectively and in order to assert the correct parsing of multiple cookies.

The test case generates **60 unit tests**.

Implementation

The test was implemented by providing file `CONF_059_TARGET_REQUEST_COOKIES.yaml`, and was merged on the 01/06/2025 [79].

5.2.3 061 - REQUEST_FILENAME and CVE-2024-1019

The REQUEST_FILENAME target contains the file name from the request's URI. REQUEST_FILENAME is employed 56 times in CRS 4.8.0 and was affected by the CVE-2024-1019 vulnerability. This test case also utilizes our *manual URI definition* feature.

Functionality

The REQUEST_FILENAME variable contains the data from the HTTP request's file name, that is the URI up to the start of the query string indicated by the presence of the `?` delimiter:

```

1 GET /file/name/index?query=string
2
3 REQUEST_FILENAME: /file/name/index

```

REQUEST_FILENAME is not a collection and therefore does not support the use of collection keys.

Here is an example of a SecRule, similar to the ones generated by our test:

```
1 SecRule REQUEST_FILENAME "@contains attack" "id:0, phase:1, log"
```

Will inspect the request's file name in phase 1 and trigger if it contains "attack".

CVE-2024-1019

CVE-2024-1019 is a vulnerability affecting ModSecurity version 3.0.0 to version 3.0.11. It has a base score of 8.6 on the Common Vulnerability Scoring System version 3 (CVSS3), categorizing it as being of high severity [80].

The vulnerability arose because of the order of URL decoding of the request's URI. Contrary to indications in RFC 3986 [81], the engine decoded the URI before separating it into multiple components. This created an exploit where using the URL encoded version of the ? delimiter (%3F) hid content after it, bypassing rules inspecting the filename:

```
1 GET /file/name/index%3Fattack?query=string
2
3 REQUEST_FILENAME: /file/name/index
```

Unfortunately, the content also didn't appear in the QUERY_STRING target.

To help prevent the vulnerability from resurfacing in the future, we take the opportunity to incorporate its exploit into our test case.

Test

To test the REQUEST_FILENAME target, we generate SecRules with the following parameters:

- Template: SecRule for TARGETS
- Target: REQUEST_FILENAME
- Phases: 1,2,3,4
- Operator: @contains
- Argument: attack
- Collection Keys: None

This generates **4 SecRules** inspecting REQUEST_FILENAME and triggering if it contains "attack". We begin testing at phase 1, where the REQUEST_FILENAME target first becomes available for requests. Following previous decisions, we test the target up to phase 4.

Utilizing our *manual URI definition* feature, we define 4 requests with the following URIs: /attack/in/uri?arg=value to check the matching at the start of the filename, /in/uri/attack?arg=value to check the matching at the end of the filename, /in/uri/is%3Fattack?arg=value to check for CVE-2024-1019, and a final /in/uri/is?attack coupled with the no_expect_ids check to confirm the non-matching of the rule when the token is located after the ? delimiter.

The test case generates **16 unit tests**.

Implementation

The test was implemented by providing file `CONF_061_TARGET_REQUEST_FILENAME.yaml`, and was merged on the 01/06/2025 [82].

5.2.4 069 - RESPONSE_BODY

The RESPONSE_BODY target contains the HTTP response body. RESPONSE_BODY is employed 55 times in CRS 4.8.0 and testing this target also demonstrates how the framework can utilize the reflection capabilities of `albedo` through our *manual URI definition* feature.

Functionality

The RESPONSE_BODY variable contains the data from the HTTP response body and functions only when ModSecurity is configured to buffer the response, as this part is typically the largest component of an HTTP transaction and configurators might not want to bear the performance costs of large bufferings. In our case, our test infrastructures are configured to buffer response bodies in order to test functionalities related to them. RESPONSE_BODY is not a collection and therefore does not support the use of collection keys.

Here is an example of a SecRule, similar to the ones generated by our test:

```
1 SecRule RESPONSE_BODY "@contains attack" "id:0, phase:4, log"
```

Will inspect the response body, in phase 4 as it is not available in earlier phases, and trigger if it contains "attack".

Test

To test the RESPONSE_BODY target, we generate SecRules with the following parameters:

- Template: Non-disruptive SecRule for TARGETS
- Target: RESPONSE_BODY
- Phases: 4,5
- Operator: @contains
- Argument: attack
- Collection Keys: None

This generates **2 SecRules** inspecting RESPONSE_BODY and triggering if it contains "attack". We begin testing from phase 4, as it is the phase where the RESPONSE_BODY target becomes available. Since RESPONSE_BODY cannot be tested in phases 1 to 3, we take the opportunity to explore an additional combinations by including phase 5. For testing in phase 5, we avoid using the `deny` action, as disruptive actions are not supported in that phase.

For the requests, we generate four requests containing data that prompts **albedo** to produce our input responses, allowing us to control the content passing through the WAF. We test the rules using responses with four different content types: plain text, HTML, XML, and JSON. The purpose of varying content types is to verify that the engine correctly buffers response bodies for the MIME types it is configured to handle. In our infrastructure, this behavior is controlled by the `SecResponseBodyMimeType` directive, which specifies which document types should be buffered by the WAF engine. These four content types are included in CRS's default regression testing configuration, and we have adopted the same set by default. This approach allows us to detect whether the response body of a given content type is not being buffered correctly.

The test case generates **8 unit tests**.

Implementation

The test was implemented by providing file `CONF_069_TARGET_RESPONSE_BODY_.yaml`, and was merged on the 01/06/2025 [83].

5.3 Tests for Other Targets

We will now discuss the other target tests contributed in this work. Although these targets are less frequently used in CRS, they still serve specific purposes and are necessary for building a complete test set. Additionally, they help illustrate various capabilities of our framework.

5.3.1 010 - DURATION

The DURATION target holds the processing time of the current transaction. Its test case demonstrates the use of the *ACTIONS* macro and the *before generation* feature.

Functionality

The DURATION target is computed by ModSecurity and contains the time elapsed during the processing of a transaction. In practice, it is most useful in logging, helping configurators assess the engine's processing performance. Unlike the other targets tested in this work, it holds a numeric value representing the time elapsed since the start of the processing phases. The further along the transaction is being processed, the higher the value.

Here are some examples of SecActions logging the duration of the transaction:

```
1 SecAction "id:1, phase:1, log, msg:'Time since processing start: %{DURATION}'"
```

Will log the DURATION of since the start of processing in phase 1.

```
1 SecAction "id:5, phase:5, log, msg:'Time since processing start: %{DURATION}'"
```

Will log the DURATION value at phase 5. Since the engine must complete all earlier phases before reaching phase 5, we expect the logged duration to be higher than the duration recorded in phase 1.

Test

Since we expect the DURATION target to increase between subsequent processing phases, we use the `setvar` action to define a variable `duration_marker` for comparison. Initially, we set the marker to 0. In each following rule, we update `duration_marker` to the previously matched value of DURATION using target `MATCHED_VAR`, so that the rule in the next phase can compare its current DURATION value against the one from the previous phase, expecting it to be greater.

To generate those SecRules, we use the following parameters:

- Template: SecRule for TARGETS with ACTIONS
- Target: DURATION
- Phases: 1,2,3,4,5
- Operator: @gt
- Argument: tx.duration_marker
- Collection Keys: none
- Action: setvar:tx.duration_marker=%{MATCHED_VAR}

Using our *before generation* feature, we place the following directive:

```
1 # Set starting duration marker
2 SecAction "id:${CURRID}$,phase:1, setvar:tx.duration_marker=0"
```

This directive initializes the duration marker to zero.

We use a variation of the SecRule for TARGETS template that incorporates the *ACTIONS* macro to place the `setvar` action and is non-disruptive to allow us to test the target in phase 5. **5 SecRules** are generated, one for each phase, each with two responsibilities: the first is to log their ID if the current DURATION exceeds the previously set duration marker and the second is to update the marker with the previously matched DURATION value.

The test case generates **5 unit tests**.

Implementation

The test was implemented by providing file `CONF_010_TARGET_DURATION.yaml`, and was proposed on the 06/05/2025 [84].

5.3.2 026 - MATCHED_VARS_NAMES

The `MATCHED_VARS_NAMES` target contains the names of all targets matched by the previous SecRule. The test demonstrates the use of the *extended check methods* and the *before-each generation* features. It also shows the use of negative tests in the set. It was designed as an illustration of issue #3382 of the ModSecurity repository, opened on the 15/05/2025 [85].

Functionality

The `MATCHED_VARS_NAMES` target contains the collection of the names of targets matched in the previous `SecRule`. Usually, this target is employed in a `SecRule` chain. A rule chain is constituted of multiple `SecRules`, with their conditions linked through logical AND operators. They are defined by the use of the `chain` action:

```
1 SecRule ARGS token "chain,deny,id:1"
2 SecRule MATCHED_VARS_NAMES "@contains ARGS:token"
```

However, a user opened issue #3382 because of unclear semantics of the target when used outside a rule chain. The user provided this example:

```
1 SecRule ARGS "@rx 1" "id:1, phase:1, pass"
2 SecRule ARGS "@rx 2" "id:2, phase:1, pass"
3 SecRule MATCHED_VARS_NAMES "@contains ARGS:foo" "id:3, phase:1, deny, status:403"
```

When sending request `GET /?foo=1&bar=2`, matching both `SecRules` targeting `ARGS`, his intuition was that the third `SecRule` should not trigger because its condition looks for the name of the first targeted match which is not from the previous `SecRule`. However, it was confirmed by libmodsecurity3's lead developer that the request did in fact trigger the last rule. Because this behavior appears to be a bug, we contributed a negative test to demonstrate it is not correct behavior. We also took the opportunity to provide a positive test for the `MATCHED_VARS_NAMES` target.

Negative Test

The negative test is meant to recreate the bug described above. It uses the following parameters:

- Template: Non-disruptive `SecRule` for `TARGETS`
- Target: `MATCHED_VARS_NAMES`
- Phases: 1,2,3,4,5
- Operator: `@contains`
- Argument: `ARGS:matched_vars_names_negative_test`
- Collection Keys: none

We include testing of phase 5 because the target can be useful in all phases. To allow testing in phase 5, we use a non-disruptive variation of the `SecRule` for `TARGETS` that does not use the `deny` action which is forbidden in the logging phase.

Using our *before-each generation* feature, we place the following directives:

```
1 SecRule ARGS "@rx matched_vars_names_negative_test" "id:${CURRID}$,"
  ↳ phase:${PHASE}$, pass"
2 SecRule ARGS "@rx matched_vars_names_test" "id:${CURRID}$, phase:${PHASE}$, pass"
```

Their intended purpose is for both rules to match, but we will expect only the second target to appear in `MATCHED_VARS_NAMES`.

The input requests contain two token as arguments to trigger the SecRules. To confirm that the tested rule's ID should be absent from the logs, we use our *extended checks* feature with the `no_expect_ids`.

In this way we recreate the conditions of the potential bug described in issue #3382, by generating **5 SecRules** and **5 unit tests** across all 5 phases.

Positive Test

The positive test is meant to confirm that the target functions as expected when used outside a rule chain. The rationale of the test case is to trigger a rule with multiple targets, each of which is matched. According to the requirements, their names should all appear in the `MATCHED_VARS_NAMES` target when processing the next SecRule. The test case verifies that each name is present individually.

The test case uses the following parameters:

- Template: Non-disruptive SecRule for TARGETS
- Target: `MATCHED_VARS_NAMES`
- Phases: 1,2,3,4,5
- Operator: `@contains`
- Arguments: `ARGS:matched_vars_names_test`, `ARGS_NAMES:matched_vars_names_test`, `REQUEST_COOKIES:matched_vars_names_test`
- Collection Keys: none

We again include testing of phase 5 because the target can be useful in all phases. This is the first time we use variations of the operator argument to implement our testing logic.

Using our *before-each generated feature*, we place the following directive:

```
1 SecRule ARGS|ARGS_NAMES|REQUEST_COOKIES "@rx matched_vars_names_test"
   ↳ "id:${CURRID}$/, phase:${PHASE}$/, pass"
```

This directive matches a token located in either the arguments, the argument names, or the request cookies. For the input requests, we place the token in each of these three locations, ensuring that the SecRule directive matches every target. The generated SecRules verify that `MATCHED_VARS_NAMES` includes one of these targets across all phases. Through the combination of 5 phases and 3 operator arguments, the test case generates **15 SecRules** and **15 unit tests**, confirming that all relevant targets are correctly included in the `MATCHED_VARS_NAMES` collection at each phase.

Implementation

The test was implemented by providing files `CONF_026_TARGET_MATCHED_VARS_NAMES.yaml` and `CONF_026_TARGET_MATCHED_VARS_NAMES-NEG.yaml`, and was proposed on the 25/05/2025 [84].

5.4 Evaluation

Now that we have contributed test infrastructures and test cases, we proceed to execute the test set and discuss the results.

In this section, we describe an experiment in which the test set is executed on both release lines of ModSecurity. We begin by outlining the execution environment, followed by a discussion of the results.

5.4.1 Experimental Setup

To execute the test set, we first use the latest version of MRTS, with the additional proposed contributions that are not yet merged in the main project. The primary goals of this experiment are to verify the correctness of the test results and to measure the average execution time required to complete a full test run. While the pass or fail status of test cases depends solely on the versions of the WAF engine and MRTS, the execution time can vary significantly based on the underlying hardware, system configuration, and software versions of all involved components.

Hardware & Operating System

The experiment was executed on a laptop with the following hardware:

- **CPU:** AMD Ryzen 5 4500U, 6 cores @ 2.375GHz
- **RAM:** 16 GB DDR4
- **Storage:** Western Digital PC SN730 NVMe SSD, 1858MB/s read speed, 1172MB/s write speed
- **OS:** Linux Mint 21.3 x86_64

This hardware was sufficient to execute the entire test suite without hitting resource limits during typical runs. Since the infrastructure is tested on the same machine, the primary bottleneck is the storage I/O, as MRTS relies on reading and writing files to generate inputs and verify test results. Therefore, the timing metrics we discuss will be based on real time, including both CPU processing and I/O latency.

Software versions

The experiment was executed with the following software and their versions:

- **Python3:** version 3.10.12
- **Apache2:** version 2.4.52 (for Ubuntu)
- **Nginx:** version 1.18.0 (for Ubuntu)
- **Albedo:** version 0.2.0
- **Go-FTW:** version 1.3.0
- **ModSecurity2:** version 2.9.9
- **libmodsecurity3:** version 3.0.14

- **Nginx connector for libmodsecurity3:** version 1.0.3

Experimental Process

The experiment consists of running the entire toolchain with the `mrts.py` utility. The utility starts by generating the rules and `go-ftw` test files, launches `albedo` as the backend, launches the WAF loaded with the generated rules, executes the tests with `go-ftw` and reports the results before shutting down the infrastructure. To evaluate the execution time, we will give the average result of ten runs.

5.4.2 Evaluation on ModSecurity2

The execution of the test set on ModSecurity2 indicates 15 failed tests out of 408:

```
1 Generate rules and tests
2 Launch backend
3 Launch infrastructure
4 Executing test set...
5 Error: failed 15 tests
6 Failure: test set failed
7 Backend shutdown
8 Infrastructure shutdown
9 MRTS completed
```

The failing tests relate to the positive `MATCHED_VARS_NAMES` test case. In ModSecurity2, this target only contains the previously matched variable inside the same rule chain. This corresponds to the usual use of the target, and is therefore not worrisome. Ideally, both versions of the engines should have the same behavior. The required alignment will be determined once the developers complete their investigation of issue #3382.

As we expected, all the other test cases pass without issues.

Over ten runs, the entire toolchain completed in an average of 1.682 seconds ($\sigma = 0.1010$). Of this, `go-ftw` accounted for an average of 0.9488 seconds ($\sigma = 0.0974$).

5.4.3 Evaluation on libmodsecurity3

The execution of the test set on libmodsecurity3 indicates 10 failed tests out of 408.

```
1 Generate rules and tests
2 Launch backend
3 Launch infrastructure
4 Executing test set...
5 Error: failed 10 tests
6 Failure: test set failed
7 Backend shutdown
8 Infrastructure shutdown
9 MRTS completed
```

The 5 first failing tests relate to the negative `MATCHED_VARS_NAMES` test case, which is expected as it was designed to illustrate the ongoing issue #3382.

The 5 other failing tests are due to the DURATION tests. Because of differences in the implementation of the DURATION target across engines, the test cases we provided fail. In ModSecurity2, the DURATION target is implemented as an integer counting microseconds. In libmodsecurity3, it is implemented as a floating-point number counting microseconds. Consequently, when using the DURATION target in a SecRule with the @gt operator, the value of the DURATION target is resolved to an empty value in libmodsecurity3 as demonstrated by the debug log:

```

1  [174653147029.073614] [/] [4] (Rule: 100094) Executing operator "Gt" with param
   ↪  "0" Was: "" against DURATION.
2  [174653147029.073614] [/] [9] Target value: "0.001536" (Variable: DURATION)
3  [174653147029.073614] [/] [4] Rule returned 0.

```

While the correct behavior should ultimately be determined by the engine developers, it is less than ideal that the engines exhibit differing behaviors. ModSecurity3's implementation of the DURATION target should either be aligned with that of ModSecurity2, or ModSecurity2 should be adjusted accordingly. Since ModSecurity3's DURATION target is effectively unusable as a SecRule condition, we are of the opinion that it should be the one to change.

In some runs on ModSecurity3, test cases in later phases may intermittently fail. This is due to the WAFs logging threads that are sometimes so delayed that the thread processing the next request sent by go-ftw writes its log entries first, causing the framework to be unable to associate the input request to the right log entry. Although the provided configuration attempts to mitigate this problem, a definitive solution will only be achieved with additional fixes in the testing tools or once the engine itself is patched. A quick solution is to set go-ftw's rate limiter high enough, around 5ms in our setup, to avoid threads outpacing each other.

Over ten runs, the entire toolchain completed in an average of 1.5855 seconds ($\sigma = 0.0499$). Of this, go-ftw accounted for an average of 0.9420 seconds ($\sigma = 0.0361$).

Comparison with vulnerable v3.0.11

By slightly modifying our experiment to use ModSecurity v3.0.11, we can verify our tests to detect CVE-2024-1019 work as intended:

```

1  executing tests in file MRTS_061_REQUEST_FILENAME.yaml
2  running 100199-1: passed in 14.921342ms (RTT 54.70747ms)
3  running 100199-2: passed in 14.873932ms (RTT 54.586733ms)
4  running 100199-3: 100199-3 failed in 15.540877ms (RTT 54.460335ms)
5  running 100199-4: passed in 15.065743ms (RTT 54.833647ms)
6  executing tests in file MRTS_061_REQUEST_FILENAME.yaml
7  running 100200-1: passed in 14.578708ms (RTT 53.41528ms)
8  running 100200-2: passed in 14.639623ms (RTT 54.688294ms)
9  running 100200-3: 100200-3 failed in 14.866068ms (RTT 54.283192ms)
10 running 100200-4: passed in 14.772142ms (RTT 55.953844ms)
11 executing tests in file MRTS_061_REQUEST_FILENAME.yaml
12 running 100201-1: passed in 15.603404ms (RTT 55.516451ms)
13 running 100201-2: passed in 14.139452ms (RTT 54.663387ms)
14 running 100201-3: 100201-3 failed in 15.402797ms (RTT 54.31911ms)
15 running 100201-4: passed in 15.381407ms (RTT 54.786318ms)

```

```

16 | executing tests in file MRTS_061_REQUEST_FILENAME.yaml
17 |   running 100202-1:  passed in 14.864065ms (RTT 55.79244ms)
18 |   running 100202-2:  passed in 14.914549ms (RTT 54.713721ms)
19 |   running 100202-3:  100202-3 failed in 14.400022ms (RTT 54.496323ms)
20 |   running 100202-4:  passed in 15.456809ms (RTT 55.752976ms)

```

The additional 4 failed tests each correspond to the exploit we included, confirming that our test set is capable of detecting security regressions.

5.5 Completeness of the Test Set

To provide an evaluation of the work we contributed, it is important to assess the completeness of the regression test set and understand how our added test cases advance this goal. In this section, we first analyze the test set’s coverage of SecRule targets, then broaden the discussion to its overall coverage of SecLang, and finally consider the prospects for achieving full coverage of ModSecurity’s functionalities.

5.5.1 Coverage of Targets

ModSecurity’s reference documentation (for both V2 and V3) defines a total of 109 variables. Although ModSecurity3 implements only 85 of these, a complete regression test set should ideally include tests for all of them. With our contributions, MRTS includes tests for just 14 targets, leaving 95 targets still untested to achieve full coverage of this set of features.

Some targets may require multiple test cases to fully cover their functionality. A notable example is the TX target, which represents the collection of all variables set by configurators during transaction processing. Not only is TX widely used by rule developers, but it also has unique behavior not shared by other targets, as it stores and manages access to variables. Additionally, some TX variables have reserved purposes, serving as internal flags essential to ModSecurity’s internal operations. All of these aspects must be carefully considered when designing comprehensive tests for this target. Special cases like these will undoubtedly arise for other targets as well.

In this work, the test cases we provided generated 153 additional SecRules and 298 additional `go-ftw` unit tests to the test set.

5.5.2 Coverage of SecLang

Of course, targets represent only a portion of SecLang’s overall functionality. To fully test the SecRule directive alone, it is necessary to cover its additional components, namely:

- Operators, of which there are 39
- Transformation functions, of which there are 38
- Actions, of which there are 47

While our test cases make use of various operators and actions, dedicated tests that specifically verify their behavior would provide greater clarity for developers. Such targeted

tests are essential for pinpointing which component fails when changes are introduced to the engine. Special language operators should also be tested, like the not (!) and size (&) operators.

Beyond SecRule, all other SecLang directives must also be tested. The reference documentation lists 81 directives in total. Excluding SecRule and SecAction leaves 79 directives, suggesting that at least 79 additional test cases are needed to achieve complete coverage of the language.

Covering the SecLang language also serves multiple WAF engines and rule developers. As SecLang is also adopted by engines beyond ModSecurity, primarily seeking integration with the core rule set, providing a common testing reference becomes essential. Such a reference would help engine developers resolve inconsistencies in SecLang implementations, while also enabling rule developers to easily write portable rule sets. Additionally, many security issues reported to open-source engine developers stem from failures to support functionalities required by CRS rules, sometimes because of unnoticed regressions. With CRS serving as the primary driver of open-source WAF development through its wide base of support and adoption, ensuring the quality of SecLang implementations is more relevant than ever.

5.5.3 Coverage of ModSecurity Functionalities

Most of the testing currently performed in MRTS is designed to validate one core functionality of ModSecurity: its rule engine. This focus explains why the framework is built to generate combinations of configuration parameters to exercise the rule evaluation process thoroughly. However, other critical functionalities of the engine must also be tested explicitly. These include verifying that data parsing operates correctly, ensuring that HTTP transaction buffering behaves as expected, and confirming that the logging output conforms to the defined format.

Testing these aspects will require a different set of capabilities within the framework. Expanding MRTS to support such tests will be essential for achieving a complete evaluation of ModSecurity's overall behavior.

Chapter 6

Conclusion and Future Work

In this work, we have discovered the purpose and challenges of WAF engine regression testing. We have proposed a testing approach tailored for WAF engines, explored the testing practice of WAF rule developers, and helped adapt their tools to support WAF engine testing through the development of the ModSecurity Regression Test Set.

We contributed features enhancing the framework’s capabilities to write complex test cases and manage the different tools to execute the tests. To enhance rule generation, we developed two new macros and created the possibility to surround generated rules with additional configurations. For better tests, we extended the framework to leverage three key features provided by `go-ftw`: the ability to inspect all WAF outputs, support for sending malformed requests, and integration with `albedo`’s reflection capabilities. To help managing the future test set, we proposed a feature to define constants across configuration files. We also proposed infrastructures for running the test set on ModSecurity2 and libmodsecurity3.

Furthermore, we contributed multiple test cases, proposing a test case selection method for the short-term goals of the project and demonstrating the effectiveness of MRTS’s features. In total, we provided test cases for 10 SecRule targets, 6 of which are used extensively in the OWASP Core Rule Set. We included a test for CVE-2024-1019 to demonstrate how MRTS can be used to prevent the reintroduction of previously addressed vulnerabilities. We also provided a negative test case illustrating an ongoing issue of the ModSecurity engines.

Altogether, the contributions in this work enriched the MRTS repository with 9 new issues, 22 new pull requests containing 34 commits and 15020 proposed lines. At the time of writing, 19 pull requests containing 30 commits and 12747 lines have already been merged while the others are still under review.

Finally, we presented the current results of the test set on both ModSecurity2 and libmodsecurity3, showing that its execution time is acceptable for integration into CI pipelines, and highlighting its secondary benefit of revealing behavioral differences across SecLang implementations. Overall, we have advanced the MRTS project’s development, bringing it closer to serving as ModSecurity’s regression testing framework to support continuous integration for its developers.

Despite these contributions, much work remains. Many functionalities are still untested,

and achieving a comprehensive test set for all of ModSecurity's features will require sustained development efforts.

In the short term, the project can continue expanding coverage of target tests and broaden its scope to include more aspects of the SecLang rule engine. Specifically, the test set should aim to cover all SecLang actions, transformation functions, and operators to cover the other parameters of the SecRule directive. It should also aim to test the wide range of additional directives provided by SecLang. A high priority should be to test features heavily used by CRS, as regressions in these functionalities would have the greatest security impact due to CRS's widespread adoption.

Additional features could also be developed. The project could expand the generation of diverse test cases with additional macros and generation options. The utility could simplify test execution management further, notably by benefiting from the many features go-ftw provides in the matter. It could also welcome configurations for testing on other WAF engines and infrastructures, such as ModSecurity2 on Red Hat-based linux distributions or Coraza integrated with the Caddy web server. By revealing the differences in behavior of different engines with the same SecLang configuration, MRTS could also take a major role in helping developers close discrepancies between their respective implementations of the SecLang language.

In the long term, the project should aim to test all the features provided by the ModSecurity engines. Notably, MRTS should be expanded with payload generation features covering a wide range of data to test the parsing and buffering features of the engine. The project should also include tests highlighting potential issues with the interactions of the engines with their associated components, especially when handling malformed HTTP traffic. Test cases checking the logs, ensuring entries respect the right formats and hold the expected contents, must also be considered.

Given MRTS's reliance on tools from the CRS project, closer collaboration with CRS developers to help guide the development of their testing tools would also prove useful for the project's future, and is ultimately necessary as ModSecurity's future is closely tied to CRS amidst their common custodianship under the OWASP foundation.

Finally, when MRTS reaches sufficient maturity, it should be integrated into ModSecurity's CI/CD pipelines to support continuous integration across both mainline releases.

In conclusion, we have made a meaningful contribution to the OWASP ModSecurity project. With its custodianship now under OWASP, ModSecurity relies on building a new community of contributors to sustain its development. A robust and well-implemented testing practice is critical to enabling both new contributors and core maintainers to continue improving the most widely deployed open source WAF engine. While significant work remains to secure the project's future, this work shows that there are ample opportunities for new WAF developers to learn, contribute, and bring innovative ideas to the software that enforces the first line of defense for web applications.

Bibliography

- [1] OWASP Foundation. *OWASP ModSecurity – Ervin Hegedüs*. Discussion by Ervin Hegedüs on the new organization of the ModSecurity project (12:30). YouTube, 2024. URL: <https://www.youtube.com/watch?v=9RAjby44iAM>.
- [2] Ivan Ristić and Christian Folini. *ModSecurity Handbook, 2nd edition*. Feisty Duck, 2017, p. 6. ISBN: 1907117024.
- [3] The Web Application Security Consortium. *Web Application Firewall Evaluation Criteria*. WAFEC was a standardization effort to define WAFs and evaluation criterias to select different solutions. WAFEC 1.0 was released in 2006, and in 2013 an attempt at modernising WAFEC began. However, WAFEC 2.0 remains unreleased due to difficulties maintaining the project. URL: <http://projects.webappsec.org/w/page/13246985/Web%20Application%20Firewall%20Evaluation%20Criteria>.
- [4] OWASP Foundation. *OWASP Best Practices: Use of Web Application Firewalls/Version 1.0.5*. Archived OWASP wiki page on WAFs, with a section defining the term WAF. It also mentions it's differences with the WAFEC definition of a WAF. URL: https://wiki.owasp.org/index.php/Category:OWASP_Best_Practices:_Use_of_Web_Application_Firewalls/Version_1.0.5#A1.2_Definition_of_the_term_WAF_-_Web_Application_Firewall.
- [5] OWASP Foundation. *Web Application Firewall*. OWASP wiki page defining Web Application Firewall. URL: https://owasp.org/www-community/Web_Application_Firewall.
- [6] OWASP Foundation. *Secure Product Design Cheat Sheet - The principle of Defense-in-Depth*. Cheatsheet of security design concepts, including a description of defense-in-depth. URL: https://cheatsheetseries.owasp.org/cheatsheets/Secure_Product_Design_Cheat_Sheet.html#2-the-principle-of-defense-in-depth.
- [7] Chad Russel. *Web Application Firewalls*. O'Reilly Media, 2018, pp. 62–63. ISBN: 9781492032281. URL: <https://www.oreilly.com/library/view/web-application-firewalls/9781098135378/>.
- [8] Chad Russel. *Web Application Firewalls*. O'Reilly Media, 2018, p. 63. ISBN: 9781492032281. URL: <https://www.oreilly.com/library/view/web-application-firewalls/9781098135378/>.
- [9] Chad Russel. *Web Application Firewalls*. O'Reilly Media, 2018, p. 64. ISBN: 9781492032281. URL: <https://www.oreilly.com/library/view/web-application-firewalls/9781098135378/>.
- [10] Ivan Ristić and Christian Folini. *ModSecurity Handbook, 2nd edition*. Feisty Duck, 2017, pp. 7–8. ISBN: 1907117024.

- [11] Chad Russel. *Web Application Firewalls*. O'Reilly Media, 2018, pp. 29–30. ISBN: 9781492032281. URL: <https://www.oreilly.com/library/view/web-application-firewalls/9781098135378/>.
- [12] OWASP Foundation. *Virtual Patching Cheat Sheet*. Concise virtual patching definition and framework. URL: https://cheatsheetseries.owasp.org/cheatsheets/Virtual_Patching_Cheat_Sheet.html.
- [13] Microsoft. *Using Microsoft Sentinel with Azure Web Application Firewall*. Documentation page on how to use the Azure WAF with the Microsoft Sentinel SIEM. URL: <https://learn.microsoft.com/en-us/azure/web-application-firewall/waf-sentinel>.
- [14] Waf Charm. *Check AWS WAF log using SIEM on Amazon Elasticsearch Service*. Tutorial to use open-source tooling to integrate AWS WAF and Amazon Elasticsearch. URL: <https://www.wafcharm.com/en/blog/check-aws-waf-log-using-siem-on-amazon-elasticsearch-service/>.
- [15] Logz.io. Integrations for the Logz.io Cloud SIEM. AWS WAF and ModSecurity are listed as having integration with Logz.io. URL: <https://docs.logz.io/docs/user-guide/cloud-siem/integrations/>.
- [16] Zed Attack Proxy. *Log File Importer*. Logs importer add-on help page, allowing to import ModSecurity logs to populate the ZAP Site Tree. URL: <https://www.zaproxy.org/docs/desktop/addons/log-file-importer/>.
- [17] Chad Russel. *Web Application Firewalls*. O'Reilly Media, 2018, pp. 46–47. ISBN: 9781492032281. URL: <https://www.oreilly.com/library/view/web-application-firewalls/9781098135378/>.
- [18] Chad Russel. *Web Application Firewalls*. O'Reilly Media, 2018, pp. 54–56. ISBN: 9781492032281. URL: <https://www.oreilly.com/library/view/web-application-firewalls/9781098135378/>.
- [19] Simon Applebaum, Tarek Gaber, and Ali Ahmed. “Signature-based and Machine-Learning-based Web Application Firewalls: A Short Survey”. In: *Procedia Computer Science* 189 (2021). AI in Computational Linguistics, pp. 359–367. ISSN: 1877-0509. DOI: 10.1016/j.procs.2021.05.105.
- [20] AWS WAF. *AWS WAF rules*. Documentation explaining AWS WAF rules. URL: <https://docs.aws.amazon.com/waf/latest/developerguide/waf-rules.html>.
- [21] Microsoft. *Custom rules for Web Application Firewall v2 on Azure Application Gateway*. Documentation on what are custom rules for Azure WAF. URL: <https://docs.aws.amazon.com/waf/latest/developerguide/waf-rules.html>.
- [22] Ivan Ristić and Christian Folini. *ModSecurity Handbook, 2nd edition*. Feisty Duck, 2017, p. 10. ISBN: 1907117024.
- [23] OWASP CRS. *ANOMALY SCORING*. CRS 3 is designed as an anomaly scoring rule set. This documentation page explains what anomaly scoring is and how to use it. URL: https://coreruleset.org/docs/2-how-crs-works/2-1-anomaly_scoring/.
- [24] Ivan Ristić and Christian Folini. *ModSecurity Handbook, 2nd edition*. Feisty Duck, 2017, pp. 9–10. ISBN: 1907117024.

- [25] Microsoft. *WAF engine on Azure Application Gateway*. Description of the Azure WAF engine functionalities. URL: <https://learn.microsoft.com/en-us/azure/web-application-firewall/ag/waf-engine>.
- [26] OWASP Coraza. *Introduction*. Documentation of the Coraza WAF engine. URL: <https://coraza.io/docs/tutorials/introduction/>.
- [27] OWASP ModSecurity. *ModSecurity*. Public Github repository of the ModSecurity project's code. It holds the code of both the V2 and V3 versions of the project. URL: <https://github.com/owasp-modsecurity/ModSecurity>.
- [28] Trustwave. *Trustwave Transfers ModSecurity Custodianship to the Open Worldwide Application Security Project (OWASP)*. Blog post announcing the transfer of ModSecurity to the OWASP Foundation. URL: <https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/trustwave-transfers-modsecurity-custodianship-to-the-open-worldwide-application-security-project/>.
- [29] Ivan Ristić and Christian Folini. *ModSecurity Handbook, 2nd edition*. Feisty Duck, 2017, pp. 11–12. ISBN: 1907117024.
- [30] OWASP CRS. *coreruleset*. Public Github repository of the Core Rule Set. URL: <https://github.com/coreruleset/coreruleset>.
- [31] OWASP CRS. *Engine and Integration Options*. Lists the WAF engines with integration of the CRS. At the bottom of the article, there is a discussion on WAF engine's rule languages and their implementation of at least a subset of SecLang. URL: https://coreruleset.org/docs/deployment/engine_integration_options/.
- [32] OWASP ModSecurity. *Reference Manual (v2.x)*. Reference Manual for ModSecurity V2.X versions in the ModSecurity Github wiki. It documents the usage of ModSecurity2 and its rule language. URL: <https://github.com/owasp-modsecurity/ModSecurity/wiki/Reference-Manual-%28v2.x%29>.
- [33] OWASP ModSecurity. *Reference Manual (v3.x)*. Reference Manual for ModSecurity V3.X versions in the ModSecurity Github wiki. It documents the usage of ModSecurity3 and its rule language. URL: [https://github.com/owasp-modsecurity/ModSecurity/wiki/Reference-Manual-\(v3.x\)](https://github.com/owasp-modsecurity/ModSecurity/wiki/Reference-Manual-(v3.x)).
- [34] OWASP Coraza. *SecLang*. SecLang documentation for Coraza. URL: <https://coraza.io/docs/seclang/>.
- [35] Cloudflare. *Cloudflare OWASP Core Ruleset*. Cloudflare's implementation of the OWASP CRS. URL: <https://developers.cloudflare.com/waf/managed-rules/reference/owasp-core-ruleset/>.
- [36] Ivan Ristić and Christian Folini. *ModSecurity Handbook, 2nd edition*. Feisty Duck, 2017, p. 70. ISBN: 1907117024.
- [37] Martin Fowler. *Continuous Integration*. Blog Article from author Martin Fowler, who wrote several publications on software engineering and extreme programming. 2024. URL: <https://martinfowler.com/articles/continuousIntegration.html>.
- [38] Bram Adams and Shane McIntosh. "Modern Release Engineering in a Nutshell – Why Researchers Should Care". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 5. 2016, pp. 78–90. ISBN: 978-1-5090-1855-0. DOI: 10.1109/SANER.2016.108.
- [39] *Git*. The Git documentation website. URL: <https://git-scm.com/>.

- [40] *Github*. The Github website. URL: <https://github.com/>.
- [41] Shin Yoo and Mark Harman. “Regression testing minimization, selection and prioritization: a survey”. In: *Software testing, verification and reliability* 22.2 (2012), pp. 67–120. DOI: 10.1002/stvr.430.
- [42] Emelie Engström and Per Runeson. “A Qualitative Survey of Regression Testing Practices”. In: *Product-Focused Software Process Improvement*. Ed. by M. Ali Babar, Matias Vierimaa, and Markku Oivo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 3–16. ISBN: 978-3-642-13792-1. DOI: 10.1007/978-3-642-13792-1_3.
- [43] OWASP Core Rule Set. *Regression Tests - Workflow runs*. Log of the regression testing jobs runned in the CRS CI pipeline. URL: <https://github.com/coreruleset/coreruleset/actions/workflows/test.yml>.
- [44] Github. *GitHub Actions documentation*. URL: <https://docs.github.com/en/actions>.
- [45] Github. *Marketplace*. The GitHub Actions Market Place. URL: <https://github.com/marketplace?type=actions>.
- [46] YAML Language Development Team. *YAML Ain’t Markup Language (YAML™) version 1.2, Revision 1.2.2*. YAML specification, updated on the 01/10/2021. URL: <https://yaml.org/spec/1.2.2/>.
- [47] Curl. *HTTP Request smuggling with curl #6144*. Issue discussing HTTP smuggling with curl. Developpers mention that curl tries to adhere to protocol specifications and that sending malformed requests may be out of the scope of the project. URL: <https://github.com/curl/curl/issues/6144>.
- [48] Ariary. *HTTPCustomHouse*. Tool developped by a security engineer to generate HTTP smuggling requests. The project description includes an evaluation on some general purpose networking utilities and deems them unsatisfying for dealing with malformed http requests. URL: <https://github.com/ariary/HTTPCustomHouse>.
- [49] OWASP Core Rule Set. *Testing the Rule Set*. The CRS documentation on testing the rule set on a custom environment, indicating to developers how to develop tests and encouraging them to add positive and negative tests. URL: <https://coreruleset.org/docs/6-development/6-5-testing-the-rule-set/>.
- [50] OWASP Core Rule Set. *Quantitative tests - Workflow runs*. Log of the quantitative testing jobs runned in the CRS CI pipeline. URL: <https://github.com/coreruleset/coreruleset/actions/workflows/quantitative.yaml>.
- [51] OWASP Core Rule Set. *Testing WAFs, FTW Version 1.0 released*. Blog post announcing the release of FTW 1.0. URL: <https://coreruleset.org/20170810/testing-wafs-ftw-version-1-0-released/>.
- [52] OWASP Core Rule Set. *go-ftw*. go-ftw’s Github repository. URL: <https://github.com/coreruleset/go-ftw>.
- [53] OWASP Core Rule Set. *ftw-test-schema - v2.3.0*. go-ftw’s YAML format documentation, version 2.3.0 generated on the 15/02/2025. URL: <https://github.com/coreruleset/ftw-tests-schema/blob/main/spec/v2.3.0/ftw.md>.

- [54] OWASP Core Rule Set. *Various SQL injection tests*. Test case YAML file for rule 942100 covering SQL injection attacks. URL: <https://github.com/coreruleset/coreruleset/blob/main/tests/regression/tests/REQUEST-942-APPLICATION-ATTACK-SQLI/942100.yaml>.
- [55] OWASP Core Rule Set. *albedo*. albedo's Github repository. URL: <https://github.com/coreruleset/albedo>.
- [56] OWASP ModSecurity. *MRTS*. MRTS's Github repository. URL: <https://github.com/owasp-modsecurity/MRTS>.
- [57] OWASP ModSecurity Regression Test Set. *Custom Actions for templates #4*. Issue for the ACTIONS macro feature. URL: <https://github.com/owasp-modsecurity/MRTS/issues/4>.
- [58] OWASP ModSecurity Regression Test Set. *Utility change: custom actions for templates #14*. Pull request for the ACTIONS macro feature. URL: <https://github.com/owasp-modsecurity/MRTS/pull/14>.
- [59] OWASP ModSecurity Regression Test Set. *Custom Directives for templates #5*. Issue for the DIRECTIVES macro feature. URL: <https://github.com/owasp-modsecurity/MRTS/issues/5>.
- [60] OWASP ModSecurity Regression Test Set. *Utility change: custom directives for templates #15*. Pull request for the DIRECTIVES macro feature. URL: <https://github.com/owasp-modsecurity/MRTS/pull/15>.
- [61] OWASP ModSecurity Regression Test Set. *Placing Directives before(-each) and after(-each) rule generation #9*. Issue for the before/after/each feature. URL: <https://github.com/owasp-modsecurity/MRTS/issues/9>.
- [62] OWASP ModSecurity Regression Test Set. *Utility change: placing directives before(-each) and after(-each) rule generation #18*. Pull request for the before/after/each feature. URL: <https://github.com/owasp-modsecurity/MRTS/pull/18>.
- [63] OWASP ModSecurity Regression Test Set. *Ftw Test Schema - Version 2.3.0*. Go-FTW YAML schema specification used for configuration files, version 2.3.0, published on the 15/02/2025. URL: <https://github.com/coreruleset/ftw-tests-schema/blob/main/spec/v2.3.0/ftw.md>.
- [64] OWASP ModSecurity Regression Test Set. *Extend check methods on error log and responses #6*. Issue for the the extended checks feature. URL: <https://github.com/owasp-modsecurity/MRTS/issues/6>.
- [65] OWASP ModSecurity Regression Test Set. *Utility change: extend check methods on error log and responses #16*. Pull request for the extended checks feature. URL: <https://github.com/owasp-modsecurity/MRTS/pull/16>.
- [66] OWASP ModSecurity Regression Test Set. *Test: 026 - MATCHED_VARS_NAMES #33*. Pull request for the MATCHED_VARS_NAMES test case and introducing negative tests. URL: <https://github.com/owasp-modsecurity/MRTS/pull/33>.
- [67] OWASP ModSecurity Regression Test Set. *Send encoded requests as input #12*. Issue for the the extended checks feature. URL: <https://github.com/owasp-modsecurity/MRTS/issues/12>.

- [68] OWASP ModSecurity Regression Test Set. *Utility change: send encoded requests as input #13*. Pull request for the encoded request feature. URL: <https://github.com/owasp-modsecurity/MRTS/pull/13>.
- [69] OWASP ModSecurity Regression Test Set. *Utility change: allow manually defining URI for using reflection capability of albedo #20*. Pull request for the manually defined URI feature. URL: <https://github.com/owasp-modsecurity/MRTS/pull/20>.
- [70] YAML Language Development Team. *YAML Ain't Markup Language (YAML™) version 1.2 - Section 3.2.2.2: anchors and aliases*. YAML specification describing YAML's alias feature. It links to section 9.1 that states: "A YAML character stream may contain several documents. Each document is completely independent from the rest.", indicating that aliases work only in the same file. URL: <https://yaml.org/spec/1.2.2/#3222-anchors-and-aliases>.
- [71] OWASP ModSecurity Regression Test Set. *Reusable constants in yaml schema #7*. Issue for the the reusable constants feature. URL: <https://github.com/owasp-modsecurity/MRTS/issues/7>.
- [72] OWASP ModSecurity Regression Test Set. *Utility change: reusable constants in yaml schema #17*. Pull request for the reusable constants feature. URL: <https://github.com/owasp-modsecurity/MRTS/pull/17>.
- [73] OWASP ModSecurity Regression Test Set. *Infrastructure and test environment #11*. Issue for the the test infrastructure feature. URL: <https://github.com/owasp-modsecurity/MRTS/issues/11>.
- [74] OWASP ModSecurity Regression Test Set. *Utility change: infrastructure for apache2 with modsec2 on ubuntu #21*. Pull request for the mrts utility and apache2 infrastructure. URL: <https://github.com/owasp-modsecurity/MRTS/pull/21>.
- [75] OWASP ModSecurity Regression Test Set. *Utility change: nginx + libmodsecurity3 infrastructure, with fail-fast as new CL argument #24*. Pull request for the libmodsecurity3 infrastructure feature. URL: <https://github.com/owasp-modsecurity/MRTS/pull/24>.
- [76] *modsec-vue*. Web application documenting multiple versions of the core rule set. It lists many informations about the rule set including used targets, and was used to produce our table in appendix C. URL: <https://crsdoc.digitalwave.hu/>.
- [77] OWASP ModSecurity Regression Test Set. *Test; ARGS test rewrite - test GET and POST on phases 1-4 and 2-4 #22*. Pull request for the ARGS test case. URL: <https://github.com/owasp-modsecurity/MRTS/pull/22>.
- [78] Internet Engineering Task Force (IETF). *RFC6265: HTTP State Management Mechanism*. Request For Comment defining the HTTP Cookie and Set-Cookie header fields. URL: <https://datatracker.ietf.org/doc/html/rfc6265>.
- [79] OWASP ModSecurity Regression Test Set. *Test: target REQUEST_COOKIES - 059 #29*. Pull request for the REQUEST_COOKIES test case. URL: <https://github.com/owasp-modsecurity/MRTS/pull/29>.
- [80] OWASP Foundation. *CVE 2024-1019*. Description of CVE 2024-1019 affection the REQUEST_FILENAME target. URL: https://owasp.org/www-project-modsecurity/tab_cves#cve-2024-1019-2024-01-30.

- [81] Network Working Group. *RFC3986: Uniform Resource Identifier (URI): Generic Syntax*. Request For Comment defining the HTTP URI syntax. Section 2.4 discusses. URL: <https://datatracker.ietf.org/doc/html/rfc3986>.
- [82] OWASP ModSecurity Regression Test Set. *Test: 061 - target REQUEST_FILENAME - recreation of CVE-2024-1019 #34*. Pull request for the REQUEST_FILENAME test case and recreating CVE-2024-1019. URL: <https://github.com/owasp-modsecurity/MRTS/pull/34>.
- [83] OWASP ModSecurity Regression Test Set. *Test: target RESPONSE_BODY - 069 #31*. Pull request for the RESPONSE_BODY test case. URL: <https://github.com/owasp-modsecurity/MRTS/pull/31>.
- [84] OWASP ModSecurity Regression Test Set. *Test: target DURATION - 010 #26*. Pull request for the DURATION test case. URL: <https://github.com/owasp-modsecurity/MRTS/pull/26>.
- [85] OWASP ModSecurity. *Semantic of MATCHED_VARS / MATCHED_VARS_NAMES #3382*. Issue discussing clarity of the semantic of MATCHED_VARS_NAMES. URL: <https://github.com/owasp-modsecurity/ModSecurity/issues/3382>.
- [86] OWASP ModSecurity Regression Test Set. *Directory Structure Change #3*. Pull request for the new directory structure. URL: <https://github.com/owasp-modsecurity/MRTS/pull/3>.
- [87] OWASP ModSecurity Regression Test Set. *Complex separator for macros in templates #8*. Issue for the the complex separator feature. URL: <https://github.com/owasp-modsecurity/MRTS/issues/8>.
- [88] OWASP ModSecurity Regression Test Set. *Utility change: complex separator for macros in templates #10*. Pull request for the complex separator feature. URL: <https://github.com/owasp-modsecurity/MRTS/pull/10>.
- [89] OWASP ModSecurity Regression Test Set. *Test: ARGS_NAMES #23*. Pull request for the ARGS_NAMES test case. URL: <https://github.com/owasp-modsecurity/MRTS/pull/23>.
- [90] OWASP ModSecurity Regression Test Set. *Test: target tests 007 & 008 - ARGS_POST & ARGS_POST_NAMES #25*. Pull request for the ARGS_POST and ARGS_POST_NAMES test cases. URL: <https://github.com/owasp-modsecurity/MRTS/pull/25>.
- [91] OWASP ModSecurity Regression Test Set. *Test: target REQUEST_COOKIES_NAMES - 060 #30*. Pull request for the REQUEST_COOKIES_NAMES test case. URL: <https://github.com/owasp-modsecurity/MRTS/pull/30>.

Appendices

Appendix A

Auxiliary Framework Features

In this appendix, we present auxiliary features not fundamental to this work, but that were nonetheless contributed to the framework.

A.1 Improved Directory Structure

Objective

Our first contribution to the project was an adapted project directory structure. The original structure was partially inherited from the testing procedure of the Core Rule Set. As the project evolved, it required a directory layout better suited to its various components, allowing for more convenient writing and execution of test sets.

Specification

The main directories organize the project as follows:

- **config_infra**: organizes the different configurations for web servers and their connection to ModSecurity. These configurations vary depending on the WAF engine being tested, the web server software used, and the operating system of the testing infrastructure. Since the final utility is intended to support dynamic setup selection at runtime, the project needs to provide commonly used configurations, especially those relevant for CI/CD pipeline integration.
- **config_tests**: contains the configuration files for the main regression test set. This directory is where test developers contribute test cases, and it is placed at the root of the project for easy accessibility. Currently, it contains a single test set targeting the ModSecurity engines.
- **feature_demo**: a directory for demonstrating features of the testing framework. It is particularly useful when developing or showcasing features related to the YAML schema. It also serves as a reference to help other developers understand the available features and how to use the schema.
- **generated**: the target directory for rule files and test files generated for inclusion in the WAF configuration and execution by `go-ftw`. It provides a standardized

location for intermediary files, which is useful for debugging the testing framework at different stages of the toolchain.

- **mrts**: contains the Python scripts of the project. The various utilities are organized so that every operation can be executed easily from the root of the project.

Implementation

Implementation was done by creating the different directories and moving the existing files in the correct locations. Changes were merged on the 29/11/2024 [86].

A.2 Complex Separator for Macros in Templates

Objective

Macros were initially written using a simple prefix, like \$PHASE, but this was unclear in complex templates. To improve readability, we adopted the more structured format \${...}\$, which makes macros easier to identify and distinguish from surrounding text.

We chose the \${...}\$ symbols because they are clear enough to indicate that a substitution is happening, while not conflicting with macros inside SecLang rules using %{...} as separator and also not conflicting with constant invocation using ~{...}~.

Specification

As seen throughout this document, we have used the \${...}\$ separator for macros in template:

```
1 global:
2   template: |
3     SecRule ${TARGET}$ "${OPERATOR}$ ${OPARG}$" \
4       "id:${CURRID}$,\
5       phase:${PHASE}$,\
6       log,\
7       msg: '%{MATCHED_VAR_NAME} was caught in phase:${PHASE}$',\
8       ver: '${VERSION}$'"
```

Implementation

Implementation was done by modifying the `generate-rules.py` script and was merged on the 12/01/2025 [87, 88].

A.3 MRTS Utility

Objective

The objective of this feature is to run the entire toolchain through a single utility. To execute the test set, the rules and test files must be generated, the backend and WAF reverse

proxy loaded with the generated rules must be started, then **go-ftw** must be executed with the generated test files, and finally, the infrastructure must be shut down.

The utility simplifies the use of the test set by automating tedious tasks such as restarting a fresh WAF instance with the latest generated rules, restarting the backend, and regenerating files. It also facilitates switching between test beds or test sets through its parameterized arguments.

Specification

The **mrts.py** utility performs the following actions:

- 1) Generates rules and **go-ftw** test files based on MRTS configuration files using the **generate_rules.py** script.
- 2) Launches **albedo** as a backend HTTP application.
- 3) Launches an inline reverse proxy WAF through a provided directory containing start and stop python scripts along with the necessary configuration files to run the WAF, including the generated rules.
- 4) Run **go-ftw** with the generated test files on the WAF test infrastructure.
- 5) Checks the output of **go-ftw** and indicates if the test set succeeded or if at least one of the tests failed.
- 6) Shuts down the WAF and the backend server.

The utility has the following usage:

```
1 mrts.py [-h] -i /path/to/infra/ -r /path/to/config/ -e /path/to/gen/rules/ -t  
  ↪ /path/to/gen/tests/ [-f /path/to/mrts/ftw.config.yaml] [-c] [-v] [-F]
```

The parameters are:

- **-i** Directory containing the WAF infrastructure
- **-r** MRTS configuration files
- **-e** Directory where SecLang configurations will be written
- **-t** Directory where **go-ftw** test files will be written
- **-f** Optional, provides a configuration file for **go-ftw**. This file provides information for the execution of **go-ftw**, such as the location of the log file to check for log content and the log format. If not provided, uses the configuration file located in the WAF infrastructure directory.
- **-c** Optional, cleans the previously generated files before proceeding. The option is useful when adding new files to the tests set in order to avoid conflicting configurations. Newly generated rules might have rule IDs that were previously attributed to other rules, and identical IDs between rules will crash the WAF at startup.
- **-v** Optional, provides a verbose output. The verbose output shows the detail of each step in the toolchain, and shows the output of **generate_rules.py** and **go-ftw**. The option is useful for debugging.

- `-F` Optional, interrupts the execution of the test set at the first failed test by setting the `-fail-fast` flag of `go-ftw`. The option is useful to save on computing resources by preventing the execution of the entire test set whenever we observe a failed test.

Implementation

Implementation was done by creating the `mrts.py` script and was merged on the 21/04/2025 [73, 74].

Appendix B

Additional Feature Demonstrations

In this appendix, we provide demonstrations of features whose detailed illustrations were not fundamental to this work.

B.1 DIRECTIVES Macro

As with other macros, the idea is to introduce a new macro, `${DIRECTIVES}$`, which can be used in any template:

```
1 global:
2 - name: "DIRECTIVES Demonstration"
3   template: |
4     SecRule ${TARGET}$ "${OPERATOR}$ ${OPARG}$" \
5       "id:${CURRID}$,\
6       phase:${PHASE}$,\
7       log"
8
9     ${DIRECTIVES}$
```

In the YAML schema, a new `directives` section is used to define the directives to be replaced, represented as a list of lists. Similar to templates, other macros can be used inside the added directives, including the `${CURRID}$` macro, which sets unique IDs incrementally, as well as all other macros that are replaced with their values for the current combination:

```
1 target: ARGS
2 rulefile: MRTS_999_EXAMPLE.conf
3 testfile: MRTS_999_EXAMPLE.yaml
4 templates:
5 - DIRECTIVES macro example
6 operator:
7 - '@contains'
8 oparg:
9 - attack
10 directives:
11 - directive:
12   - SecAction "id:${CURRID}$,phase:${PHASE}$, pass, setenv:'var=abc'"
```

```

13 - directive:
14   - SecAction "id:${CURRID}$,phase:${PHASE}$, pass, setenv:'var=def'"
15 phase:
16   - 2

```

This example generates the following configuration:

```

1  SecRule ARGS "@contains attack" \
2    "id:100001,\
3    phase:2,\
4    log"
5
6  SecAction "id:100002,phase:2, pass,setenv:'var=abc'"
7
8  SecRule ARGS "@contains attack" \
9    "id:100003,\
10   phase:2,\
11   log"
12
13 SecAction "id:100004,phase:2, pass,setenv:'var=def'"

```

To add multiple directives in the same combination, the **directive** section can take a list of directives.

```

1  target: ARGS
2  rulefile: MRTS_999_EXAMPLE.conf
3  testfile: MRTS_999_EXAMPLE.yaml
4  templates:
5    - DIRECTIVES macro example
6  operator:
7    - '@contains'
8  oparg:
9    - attack
10 directives:
11   - directive:
12     - SecAction "id:${CURRID}$,phase:${PHASE}$, pass, setenv:'123=abc'"
13     - SecAction "id:${CURRID}$,phase:${PHASE}$, pass, setenv:'456=def'"
14 phase:
15   - 2

```

This example generates the following configuration:

```

1  SecRule ARGS "@contains attack" \
2    "id:100002,\
3    phase:2,\
4    log"
5
6  SecAction "id:100001,phase:2, pass,setenv:'123=abc'"
7  SecAction "id:100003,phase:2, pass,setenv:'456=def'"

```

B.2 Encoded Requests

The feature allows defining an `encoded_request` field in MRTS configuration files to overwrite the default `input` field in tests:

```
1 testdata:
2   phase_methods:
3     2: post
4   targets:
5     - target: ''
6     test:
7       data: null
8       input:
9         encoded_request: aGVsbG9fd29ybGQK
```

The above example generates the following test configuration for `go-ftw`:

```
1 tests:
2 - test_title: 100000-1
3   ruleid: 100000
4   test_id: 1
5   desc: 'Test case for rule 100000, #1'
6   stages:
7 - description: Send request
8   input:
9     dest_addr: 127.0.0.1
10    port: 80
11    protocol: http
12    method: POST
13    headers:
14      User-Agent: OWASP MRTS test agent
15      Host: localhost
16      Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,
17        ↪ text/plain;q=0.8,image/png,*/*;q=0.5
18    uri: /post
19    version: HTTP/1.1
20    encoded_request: aGVsbG9fd29ybGQK
21  output:
22    log:
23      expect_ids:
24        - 100000
```

Note that in the above example, the fields `method`, `headers`, `uri`, and `version` are ignored, as `go-ftw` will decode the content of `encoded_request` and send it as-is, instead of constructing the request from those fields.

Appendix C

Target Frequency in CRS v4.8.0

Target	Freq.	Target	Freq.
TX (total usage)	282	REQUEST_COOKIES [†]	151
TX:DETECTION_PARANOIA_LEVEL	192	REQUEST_COOKIES_NAMES [†]	151
TX:BLOCKING_PARANOIA_LEVEL	16	REQUEST_HEADERS (total usage)	135
TX:1	10	REQUEST_HEADERS:User-Agent	26
TX:REPORTING_LEVEL	5	REQUEST_HEADERS	25
TX:BLOCKING_INBOUND_ANOMALY_SCORE	3	REQUEST_HEADERS:Referer	21
TX:BLOCKING_OUTBOUND_ANOMALY_SCORE	3	REQUEST_HEADERS:Content-Type	8
TX:0	2	REQUEST_HEADERS:Content-Length	5
TX:2	2	REQUEST_HEADERS:Range	4
TX:922100_CHARSET	2	REQUEST_HEADERS:Request-Range	4
TX:/rfi_parameter_*/	2	REQUEST_HEADERS:X-Filename	4
TX:BLOCKING_ANOMALY_SCORE	1	REQUEST_HEADERS:X_Filename	4
TX:content_type	1	REQUEST_HEADERS:X-File-Name	4
TX:content_type_charset	1	REQUEST_HEADERS:Accept	3
TX:CRS_VALIDATE_UTF8_ENCODING	1	REQUEST_HEADERS:X.Filename	3
TX:DETECTION_ANOMALY_SCORE	1	REQUEST_HEADERS:Accept-Encoding	2
TX:DETECTION_INBOUND_ANOMALY_SCORE	1	REQUEST_HEADERS:Host	2
TX:DETECTION_OUTBOUND_ANOMALY_SCORE	1	REQUEST_HEADERS:Cache-Control	1
TX:detection_paranoia_level	1	REQUEST_HEADERS:Connection	1
TX:ENABLE_DEFAULT_COLLECTIONS	1	REQUEST_HEADERS:Sec-CH-UA-Mobile	1
TX:enforce_bodyproc_urlencoded	1	REQUEST_HEADERS:Sec-Fetch-User	1
TX:EXTENSION	1	&REQUEST_HEADERS:Content-Type	3
TX:/header_name_920450_/	1	&REQUEST_HEADERS:Transfer-Encoding	3
TX:/header_name_920451_/	1	&REQUEST_HEADERS:Content-Length	2
TX:sampling_percentage	1	&REQUEST_HEADERS:User-Agent	2
TX:sampling_rnd100	1	&REQUEST_HEADERS:Accept	1
TX:ua_hash	1	&REQUEST_HEADERS:Cache-Control	1
&TX:allowed_http_versions	1	&REQUEST_HEADERS:Host	1
&TX:allowed_methods	1	&REQUEST_HEADERS:Range	1
&TX:allowed_request_content_type	1	&REQUEST_HEADERS:Referer	1
&TX:allowed_request_content_type_charset	1	&REQUEST_HEADERS:x-up-devcap-post-charset	1
&TX:ARG_LENGTH	1	REQUEST_FILENAME [†]	56
&TX:ARG_NAME_LENGTH	1	RESPONSE_BODY [†]	55
&TX:blocking_paranoia_level	1	REQUEST_BODY	17
&TX:COMBINED_FILE_SIZES	1	FILES	8
&TX:critical_anomaly_score	1	REQUEST_URI	8
&TX:crs_setup_version	1	REQUEST_LINE	7
&TX:crs_validate_utf8_encoding	1	REQUEST_METHOD	7
&TX:detection_paranoia_level	1	MATCHED_VARS	6
&TX:early_blocking	1	REQUEST_BASENAME	6
&TX:enforce_bodyproc_urlencoded	1	RESPONSE_HEADERS:Content-Encoding	6
&TX:error_anomaly_score	1	MATCHED_VAR	5
&TX:inbound_anomaly_score_threshold	1	ARGS_GET (total usage) [‡]	4
&TX:MAX_FILE_SIZE	1	ARGS_GET	2
&TX:MAX_NUM_ARGS	1	ARGS_GET:fbclid	1
&TX:notice_anomaly_score	1	ARGS_GET:gclid	1
&TX:outbound_anomaly_score_threshold	1	MULTIPART_PART_HEADERS (total usage)	4
&TX:reporting_level	1	MULTIPART_PART_HEADERS	3
&TX:restricted_extensions	1	&MULTIPART_PART_HEADERS:_charset_	1
&TX:restricted_headers_basic	1	REQUEST_URI_RAW	4
&TX:restricted_headers_extended	1	FILES_NAMES	3
&TX:sampling_percentage	1	REQBODY_PROCESSOR	3
&TX:TOTAL_ARG_LENGTH	1	REQUEST_HEADERS_NAMES	3
&TX:warning_anomaly_score	1	REMOTE_ADDR	2
ARGS (total usage) [†]	176	REQUEST_PROTOCOL	2
ARGS	175	RESPONSE_STATUS	2
&ARGS	1	ARGS_COMBINED_SIZE [‡]	1
XML (total usage) [‡]	175	ARGS_GET_NAMES [†]	1
XML:/*	161	FILES_COMBINED_SIZE	1
XML://@*	14	QUERY_STRING	1
ARGS_NAMES [†]	171	UNIQUE_ID	1

Table C.1: Frequency of occurrences of SecLang targets in CRS v4.8.0, ordered by frequency, grouped by collection, according to <https://crsdoc.digitalwave.hu/?v=v4.8.0>.

[†] These target tests were contributed as part of this work.

[‡] These target tests were already included in the test set prior to this work

Appendix D

Auxiliary Tests

In this appendix, we present auxiliary test cases whose detailed description were not fundamental to this work, but that were nonetheless contributed to the test set.

D.1 006 - ARGS_NAMES

The ARGS_NAMES target contains the HTTP request argument names. ARGS_NAMES is employed 171 times in CRS 4.8.0.

Functionality

The ARGS_NAMES variable refers to the collection of all argument names included in an HTTP request. It is equivalent to the ARGS target, but inspects argument names rather than their values. SecLang also provides targets ARGS_GET_NAMES and ARGS_POST_NAMES, referring to the collection of argument names in an HTTP request located in the query string or in the body respectively.

Here are some example SecRules, similar to the ones generated by our test:

```
1 SecRule ARGS_NAMES "@contains attack" "id:0, phase:1, log"
```

Will inspect all request argument's names and trigger if one of them contains "attack".

```
1 SecRule ARGS_NAMES:attack1|ARGS_NAMES:attack2 "@contains attack" "id:0, phase:1,  
  ↳ log"
```

Will inspect request arguments named "attack1" or "attack2" and trigger if either name contains "attack".

```
1 SecRule ARGS_NAMES:/^attack_.* / "@contains attack" "id:0, phase:1, log"
```

Will inspect request arguments starting with "attack_" and trigger their name contains "attack".

Test

To test the `ARGS_NAMES` target, we generate `SecRules` with the following parameters:

- Template: `SecRule` for `TARGETS`
- Target: `ARGS_NAMES`
- Phases: `[1],2,3,4`
- Operator: `@contains`
- Argument: `attack`
- Collection Keys: `None, attack1, attack1 & attack2, /^attack_.*\/`

This test generates **16 `SecRules`** that inspect `ARGS_NAMES` and trigger if it contains "attack". The variations span four phases and use four different collection keys. We begin testing at phase 1, where the `ARGS_NAMES` target first becomes available for GET requests. Following previous decisions, we test the target up to phase 4. The test covers rules using no collection key, a single static key, two static keys, and a regular expression key to cover the various ways collection keys may be used.

We generate four requests with the following data: `attack=test`, `attack1=test`, `attack2=test`, and `attack_foo=test`. Each request is designed to trigger the rule with no collection key. Additionally, the last three requests are intended to trigger the rules with the `attack1`, `attack1 & attack2`, and `/^attack_.*\/` collection keys, respectively.

To test the entire feature, we test rules with both GET and POST requests to cover the two possible locations for arguments. This is implemented through two variations of the test. Variation A runs from phase 1 to 4 using GET requests, where arguments are found in the query string. Variation B runs from phase 2 to 4 using POST requests, where arguments are placed in the request body, which is only accessible from phase 2. These variations have identical logic to the `ARGS_GET_NAMES` and `ARGS_POST_NAMES` test cases. The `ARGS_GET_NAMES` test case has the same logic as variation A, and was already present in the test set. The `ARGS_POST_NAMES` test case has the same logic as variation B, and was contributed to the test set as part of this work. A detailed description of this test can be found at appendix D.3.

With both variations, the test case generates **56 unit tests**.

Implementation

The test was implemented by providing files `CONF_006_TARGET_ARGS_NAMES_A-GET.yaml` and `CONF_006_TARGET_ARGS_NAMES_B-POST.yaml`, and was merged on the 21/04/2025 [89].

D.2 007 - `ARGS_POST`

The `ARGS_POST` target contains the request arguments located in the request body. The test case is very similar to the `ARGS` target.

Functionality

The ARGES_POST variable refers to the collection of all arguments located in the body of an HTTP request. Essentially, it is a subset of the ARGES target, limited to requests that contain arguments in their body. Like other collection variables, ARGES_POST organizes request body arguments by name and allows access through collection keys.

Here are some example SecRules, similar to the ones generated by our test:

```
1 SecRule ARGES_POST "@contains attack" "id:0, phase:1, log"
```

Will inspect all request arguments located in the body and trigger if one of them contains "attack".

```
1 SecRule ARGES_POST:args1|ARGES_POST:args2 "@contains attack" "id:0, phase:1, log"
```

Will inspect request arguments located in the body named "args1" or "args2" and trigger if one of them contains "attack".

```
1 SecRule ARGES_POST:/^args_.*/ "@contains attack" "id:0, phase:1, log"
```

Will inspect request arguments in the request body starting with characters "args_" and trigger if one of them contains "attack".

Test

The test is identical to version B of the ARGES test. We generate SecRules with the following parameters:

- Target: ARGES_POST
- Phases: 2,3,4
- Operator: @contains
- Argument: attack
- Collection Keys: None, arg1, arg1 & arg2, /^arg_.*/

This test generates **12 SecRules** that inspects the ARGES_POST variable and triggers if it contains "attack". The variations span 3 phases and use 4 different collection key configurations. We begin testing at phase 2, where the ARGES_POST target first becomes available for POST requests. Following previous decisions, we test the target up to phase 4. The test covers rules using no collection key, a single static key, two static keys, and a regular expression key to cover the various ways collection keys may be used.

We generate 4 test requests with the following data: `foo=attack`, `arg1=attack`, `arg2=attack`, and `arg_foo=attack`. Each request is designed to trigger the rule with np collection key. Additionally, the last 3 requests are intended to trigger the rules with the `arg1`, `arg1 & arg2`, and `/^arg_.*/` collection keys, respectively.

The test case generates **12 unit tests**.

Implementation

The test was implemented by providing file `CONF_007_TARGET_ARGS_POST.yaml`, and was merged on the 01/06/2025 [90].

D.3 008 - ARGS_POST_NAMES

The `ARGS_POST_NAMES` target contains the request argument names located in the request body. The test case is very similar to the `ARGS_NAMES` target.

Functionality

The `ARGS_POST_NAMES` variable refers to the collection of all argument names located in the body of an HTTP request. Essentially, it is a subset of the `ARGS_NAMES` target, limited to requests that contain arguments in their body. Like other collection variables, `ARGS_POST_NAMES` organizes request body argument names by name and allows access through collection keys.

Here are some example `SecRules`, similar to the ones generated by our test:

```
1 SecRule ARGS_POST_NAMES "@contains attack" "id:0, phase:1, log"
```

Will inspect all request argument names in the body and trigger if one of them contains "attack".

```
1 SecRule ARGS_POST_NAMES:attack1|ARGS_POST_NAMES:attack2 "@contains attack" "id:0,  
↪ phase:1, log"
```

Will inspect request arguments in the body named "attack1" or "attack2" and trigger if either name contains "attack".

```
1 SecRule ARGS_NAMES:/^attack_.* / "@contains attack" "id:0, phase:1, log"
```

Will inspect request arguments in the body starting with characters "attack_" and trigger if their name contains "attack".

Test

The test case is identical to version B of the `ARGS_NAMES` test. We generate `SecRules` with the following parameters:

- Target: `ARGS_POST_NAMES`
- Phases: 2,3,4
- Operator: `@contains`
- Argument: `attack`
- Collection Keys: `None`, `attack1`, `attack1 & attack2`, `/^attack_.* /`

This test generates **12 `SecRules`** that inspects the `ARGS` variable and triggers if it contains "attack". The variations span 3 phases and use 4 different collection key configurations. We begin testing at phase 2, where the `ARGS_POST` target first becomes

available for POST requests. Following previous decisions, we test the target up to phase 4. The test covers rules using no collection key, a single static key, two static keys, and a regular expression key to cover the various ways collection keys may be used.

We generate four test requests with the following data: `attack=test`, `attack1=test`, `attack2=test`, and `attack_foo=test`. Each request is designed to trigger the rule without a collection key. Additionally, the last three requests are intended to trigger the rules with the `attack1`, `attack1 & attack2` and `/^attack_.*` collection keys, respectively.

The test case generates **12 unit tests**.

Implementation

The test was implemented by providing file `CONF_008_TARGET_ARGS_POST_NAMES.yaml`, and was merged on the 01/6/2025 [90].

D.4 060 - REQUEST_COOKIES_NAMES

The `REQUEST_COOKIES_NAMES` target contains the HTTP request cookie's names. `REQUEST_COOKIES_NAMES` is employed 151 times in CRS 4.8.0.

Functionality

The `REQUEST_COOKIES_NAMES` variable refers to the collection of all cookie names included in an HTTP request. It is the equivalent of the `REQUEST_COOKIES` target, but for cookie names rather than their values.

Here are some example `SecRules`, similar to the ones generated by our test:

```
1 SecRule REQUEST_COOKIES_NAMES "@contains attack" "id:0, phase:1, log"
```

Will inspect all request cookie's names and trigger if one of them contains "attack".

```
1 SecRule REQUEST_COOKIES_NAMES:attack1|REQUEST_COOKIES_NAMES:attack2 "@contains  
  ↳ attack" "id:0, phase:1, log"
```

Will inspect request cookies named "attack1" or "attack2" and trigger if either name contains "attack".

```
1 SecRule REQUEST_COOKIES_NAMES:/^attack_.* / "@contains attack" "id:0, phase:1, log"
```

Will inspect request cookies starting with "attack_" and trigger if their name contains "attack".

Test

To test the `REQUEST_COOKIES_NAMES` target, we generate `SecRules` with the following parameters:

- Template: `SecRule` for `TARGETS`
- Target: `REQUEST_COOKIES_NAMES`

- Phases: 1,2,3,4
- Operator: @contains
- Argument: attack
- Collection Keys: None, attack1, attack1 & attack2, /^attack_.*/

This test generates **16 SecRules** that inspects REQUEST_COOKIES_NAMES and triggers if it contains "attack". The variations span four phases and use four different collection key configurations. We begin testing at phase 1, where the REQUEST_COOKIES_NAMES target first becomes available. Following previous decisions, we test the target up to phase 4. The test covers rules using no collection key, a single static key, two static keys, and a regular expression key to cover the various ways collection keys may be used.

We generate four requests with the following headers: `Cookies:attack=test`, `Cookies:attack1=test`, `Cookies:attack2=test`, and `Cookies:attack_foo=test`. Each request is designed to trigger the rule with no collection key. Additionally, the last three requests are intended to trigger the rules with the `attack1`, `attack1 & attack2` and `/^attack_.*/` collection keys, respectively. Furthermore, 3 requests are defined containing headers: `Cookies:attack1=test;hello=test`, `Cookies:hello=test;attack2=test`, `Cookies:hello=test;attack_foo=test;world=test` intended to trigger the SecRule with collection key `attack1`, `attack1 & attack2` and `/^attack_.*/` respectively and in order to assert the correct parsing of multiple cookie names.

The test case generates **60 unit tests**.

Implementation

The test was implemented by providing file `CONF_060_TARGET_REQUEST_COOKIES_NAMES.yaml`, and was merged on the 01/06/2025 [91].

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl