Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

WAF NextGen

AUTORES

IGNACIO CARLOS MONZALVO MILAN
JUAN PABLO MARTÍNEZ DELBUGIO

TUTORES

GUSTAVO BETARTE
JUAN DIEGO CAMPO
FELIPE ZIPITRÍA

ABRIL, 2021

PRESENTADO AL TRIBUNAL EVALUADOR COMO REQUISITO DE GRADUACIÓN DE LA CARRERA INGENIERÍA EN COMPUTACIÓN

Agradecimientos

A nuestros padres, hermanos, familias, quienes desde su lugar nos brindaron apoyo y afecto durante todo el tiempo que duró este proyecto, en un contexto muy particular.

A Toti, por tantas horas de compañía.

A nuestros amigos, siempre presentes en el transcurso de este trabajo.

A nuestros colegas, siempre dispuestos a darnos una mano.

A nuestros jefes, por la flexibilidad y por el interés.

A la UDELAR, por brindarnos la posibilidad de estudiar aquello que nos apasiona.

A Felipe, Gustavo y Juan Diego, por proponernos este proyecto
e impulsarnos a dar lo mejor de nosotros.

Gracias a todos.

Abstract

Dentro de las distintas alternativas de "Web Application Firewalls" (WAFs) de código abierto, ModSecurity es conocida por ser la más popular. ModSecurity es ampliamente utilizado tanto por gobiernos como comercialmente, siendo una primera defensa para múltiples aplicaciones web. Usualmente, es configurado para utilizar un conjunto de reglas de seguridad mantenidas por la organización sin fines de lucro OWASP, las cuales son conocidas como el "Core Rule Set" (CRS). A pesar de su alto nivel de popularidad, ModSecurity presenta distintas limitaciones que no han podido ser solucionadas en los últimos años. En este trabajo, se analizan dichas limitaciones y se construye un WAF moderno con el fin de resolverlas. Particularmente, el nuevo WAF provee un lenguaje de programación imperativa para definir reglas de seguridad, el cual permite recrear gran parte de las reglas del CRS. Además, el WAF fue implementado en el lenguaje Rust, un lenguaje que se ha vuelto popular en el desarrollo de sistemas de seguridad.

Índice

1 | Introducción

En la última década, a partir de la masificación de las aplicaciones web, se ha visto un aumento en la variedad de ataques hacia las mismas. Además de los esfuerzos de los desarrolladores de dichas aplicaciones para no introducir vulnerabilidades, se ha popularizado el uso de "Web Application Firewalls" (WAFs) como medida para la prevención de distintos ataques.

Un WAF es un servidor web, el cual actúa como intermediario entre un cliente y otro servidor web al cual el cliente desea acceder. De esta manera, un WAF permite filtrar y/o monitorear el tráfico HTTP que se dirige hacia el servidor final. Al inspeccionar dicho tráfico, el WAF puede prevenir ataques conocidos como son inyecciones SQL, y ataques XSS.

De dicha definición, se desprende que un WAF es, en particular, un servidor proxy HTTP. Justamente, un servidor proxy HTTP es un servidor web que actúa de intermediario entre un cliente y otro servidor web. Más aún, a aquellos servidores proxy que actúan en nombre del servidor final, se les denomina proxys inversos. Observamos que un WAF, más que un servidor proxy, es un servidor proxy inverso.

Entre las distintas alternativas a "Web Application Firewalls", ModSecurity se volvió la más popular entre aquellas que son de código abierto. ModSecurity ofrece un lenguaje específico que permite definir reglas de seguridad, las cuales son creadas por los administradores de aplicaciones web para filtrar posibles ataques.

A pesar de que cada administrador puede definir sus propias reglas de seguridad, usualmente ModSecurity es configurado para utilizar el "Core Rule Set" (CRS). El CRS es un conjunto de reglas de ModSecurity, mantenido por la organización OWASP, ampliamente utilizado para prevenir distintos tipos de ataques.

En los últimos años, el futuro de ModSecurity se ha tornado incierto ya que no ha recibido grandes actualizaciones. Por otro lado, los requerimientos del CRS se han vuelto cada vez más complejos, y no parecen ser satisfechos apropiadamente por la tecnología de ModSecurity. Esto ha llevado a que la comunidad considere que podría

ser el momento de encontrar una alternativa a dicho WAF.

1.1 Objetivos del trabajo

El objetivo de este trabajo es crear un "Web Application Firewall" moderno, seguro y que ofrezca un lenguaje sofisticado para la definición de reglas de seguridad. Este lenguaje debe permitir recrear el CRS (o, al menos, gran parte de él), con el fin de que pueda ser considerado una alternativa a la utilización de ModSecurity.

Por otra parte, el WAF debe ser implementado en el lenguaje Rust, el cual cuenta con mecanismos específicos que garantizan un mayor nivel seguridad en las aplicaciones desarrolladas en el mismo.

1.2 Estructura del trabajo

En la sección 2 se introduce ModSecurity y su lenguaje de especificación de reglas, y también se describe en qué consisten OWASP y el "Core Rule Set". La sección 3 presenta un análisis de distintos conceptos relacionados al "Core Rule Set" y su organización, al igual que las principales limitaciones de ModSecurity. Al final de dicha sección, se definen los requerimientos del "Web Application Firewall" a implementar.

En la sección 4 se presenta el diseño del nuevo WAF, empezando por su arquitectura desde el punto de vista de un proxy inverso, y continuando con el diseño de las funcionalidades de seguridad que debe proveer. En esta sección, particularmente, se introduce un lenguaje de programación que permite definir reglas de seguridad, y se justifica por qué el WAF diseñado permite implementar gran parte de las reglas de seguridad del CRS.

El WAF implementado es descrito en la sección 5. Además de dicho WAF, se describe la librería Rhodium, la cual es una librería para el desarrollo de servidores web en Rust, y el CRS-Parser, un analizador sintáctico el cual es capaz de descomponer y examinar las reglas del CRS. Todo esto fue implementado en el marco de este proyecto. Principalmente, se describen las funcionalidades ofrecidas por el WAF, el CRS-Parser y Rhodium, e instrucciones para su configuración. Finalmente, en la sección 6 se presentan las conclusiones y el trabajo a futuro.

En el apéndice A1 se encuentra el relevamiento de las arquitecturas de distintos proxys inversos utilizados a gran escala. Dicho relevamiento fue de utilidad para el diseño de la arquitectura del WAF desarrollado. Por otro lado, en el apéndice A2 se

describe formalmente el lenguaje introducido en la sección 4, tanto su sintaxis, como su semántica y las funciones predefinidas del mismo.

Con el fin de definir las librerías a utilizar en la implementación del WAF, se realizó un análisis de algunos proxys inversos desarrollados en Rust, y las librerías para el manejo de mensajes HTTP que utilizaban. Este análisis se puede encontrar en el apéndice A3.

Por último, en el apéndice A4 se encuentra un instructivo para ejecutar las aplicaciones descritas en la sección 5.

1.3 Código fuente

Todo el código fuente generado como parte de este trabajo se encuentra disponible en [1].

2 | Estado del Arte

En esta sección comenzamos realizando una introducción del "Web Application Firewall" ModSecurity y describiendo el lenguaje que provee para implementar reglas de seguridad. Luego, presentamos brevemente la fundación sin fines de lucro OWASP y, en particular, uno de sus principales proyectos: el "Core Rule Set".

2.1 ModSecurity

ModSecurity es el WAF de código abierto más utilizado en la industria. La primera versión fue lanzada en 2002 como un módulo del servidor web Apache, aunque actualmente puede ser utilizado en otros servidores web.

ModSecurity cuenta con un lenguaje propio que permite al administrador del WAF definir sus propias reglas para el filtrado y monitoreo de su aplicación. Esto hace que sea muy flexible a la hora de definir políticas de seguridad.

En un flujo normal, ModSecurity recibe un pedido HTTP, lo retransmite al servidor final, recibe la respuesta HTTP, y luego retransmite dicha respuesta al cliente. A dicho procesamiento se le denomina **transacción**.

Las reglas de ModSecurity se pueden aplicar en cinco fases distintas del ciclo de vida de una transacción, dichas fases son identificadas por el contenido procesado en cada una de ellas [2]:

1. **Request Headers:** En la primera etapa se procesan las cabeceras del pedido HTTP recibido por el WAF. Es el paso previo a procesar el cuerpo del mismo.
2. **Request Body:** Una vez procesadas las cabeceras, el cuerpo del pedido es procesado en la segunda etapa.
3. **Response Headers:** De la misma manera que con los pedidos, las cabeceras de una respuesta HTTP son examinadas antes de procesar el cuerpo de la misma.
4. **Response Body:** Una vez procesadas las cabeceras de la respuesta, se procede a procesar el cuerpo de la misma.

5. **Logging:** Una vez finalizada la transacción, se guarda la información sobre las fases anteriores. En esta fase, las reglas sólo pueden influir para controlar qué información debe registrarse.

2.1.1 Lenguaje de especificación de reglas

El comportamiento de ModSecurity es definido en archivos de configuración, los cuales contienen una serie de directivas que establecen las reglas a ser ejecutadas. En general, las reglas se definen utilizando la directiva "SecRule" de la siguiente manera [3]:

```
SecRule VARIABLES OPERADOR
 FUNCIONES DE TRANSFORMACIÓN Y ACCIONES
```

Variables.

Las variables se corresponden con partes del pedido o de la respuesta con las cuales la regla trabajará. Por ejemplo, una regla puede utilizar la variable REQUEST_METHOD para saber qué método HTTP utilizó un pedido.

Las variables a las cuales puede acceder una regla dependen de la fase para la cual se defina dicha regla. Por ejemplo, en la primera fase ("Request Headers"), las reglas no pueden acceder a la variable que contiene el cuerpo del pedido, o variables que contengan información de la respuesta.

ModSecurity provee una gran cantidad de variables, las cuales son divididas en dos grandes grupos:

- **Variables o Variables Simples:** Contienen información específica, normalmente una cadena de bytes o un valor numérico. Por ejemplo, la variable ARGS_COMBINED_SIZE contiene el tamaño en bytes de los argumentos de un pedido. A su vez, las variables simples pueden ser divididas en varios subgrupos:
  - Variables del pedido
  - Variables de la respuesta
  - Variables de tiempo
  - Variables del servidor
  - Otras variables

- **Colecciones:** Son colecciones de variables simples. Un ejemplo es la colección ARGS, la cual es un mapa con los argumentos recibidos en un pedido y sus correspondientes valores. Pueden ser divididas en distintos subgrupos:
  - Colecciones especiales: Algunas veces se necesita tratar como colecciones a objetos que no lo son. Por ejemplo, para los archivos XML se utiliza la colección XML. A través de dicha colección se puede extraer información de un archivo XML recibido en un pedido HTTP.
  - Colecciones de lectura: Son colecciones que no pueden ser modificadas por las reglas.
  - Colecciones de lectura y escritura: Son colecciones que pueden ser utilizadas por las reglas para almacenar información durante una transacción.
  - Colecciones persistentes: Son colecciones de lectura y escritura, cuyo contenido es transversal a todas las transacciones.

Operadores.

El propósito de las variables es permitir evaluar operadores. Los operadores definen condiciones y son utilizados para decidir cuándo aplicar las acciones definidas en una regla. El operador más utilizado es el de expresiones regulares, el cual permite validar que el valor de una variable verifique una expresión regular fija. Aún así, ModSecurity ofrece una gran variedad de operadores, los cuales pueden actuar en distintos escenarios y sobre distintos tipos de datos. Por ejemplo, se puede validar que una variable con un valor numérico sea menor que un valor determinado, o que una variable con una cadena de bytes empiece con ciertos valores.

Los operadores se dividen en tres categorías:

- **Operadores de cadenas:** Utilizados para evaluar condiciones sobre cadenas de bytes.
- **Operadores numéricos:** Utilizados para evaluar condiciones sobre variables numéricas.
- **Otros operadores:** En esta categoría están los operadores que no pertenecen a ninguna de las categorías anteriores.

Acciones.

Cuando una variable seleccionada cumple con la condición de un operador, se procede a realizar acciones sobre la transacción. Por lo tanto, luego del operador, se

define una lista de acciones. Las mismas se dividen en distintos tipos:

- **Acciones disruptivas:** Todas las reglas deben tener asociada una, y sólo una, de estas acciones, las cuales le indican a ModSecurity qué hacer. Pueden bloquear la transacción, permitir que continúe, redirigirla, pausarla por un tiempo dado u otras opciones.
- **Acciones de flujo:** Pueden alterar el flujo en el que se procesa un pedido o una respuesta en una fase determinada. Por ejemplo, permiten saltear otras reglas definidas.
- **Acciones de metadata:** Proporcionan información extra sobre una regla. Estas acciones no son acciones en sí, sino que almacenan información de la regla como puede ser un identificador único, la fase de la regla, entre otras cosas.
- **Acciones sobre variables:** Estas acciones permiten definir, modificar o eliminar variables.
- **Acciones de "Logging":** Estas acciones determinan en qué medida se registrará información sobre la regla en cuestión.
- **Otras acciones:** Acciones que no pertenecen a ninguno de los otros grupos.

Es importante observar que es estrictamente necesario que toda regla incluya al menos una variable, un único operador y, como ya se mencionó, exactamente una acción disruptiva.

A continuación, presentamos un ejemplo de una regla con una única variable (REQUEST_FILENAME), un operador de cadenas ("contains") y distintas acciones: una disruptiva ("drop") y dos de metadata ("id" y "phase").

```
SecRule REQUEST_FILENAME "@contains admin"
    "phase:1,id:1,drop"
```

Esta regla es una regla de fase uno, cuyo fin es filtrar aquellos pedidos cuyo archivo de destino contenga la palabra "admin" en el nombre. Por ejemplo, esta regla filtra pedidos HTTP a direcciones como "www.example.com/admin.php". Aún así, no filtra pedidos a direcciones como "www.example.com/adMIN.html". Si quisiéramos que fuese el caso, deberíamos trabajar con una expresión regular o con una función de transformación.

Funciones de transformación.

Previo a la evaluación de un operador, se pueden aplicar funciones para modificar las variables a evaluar. Para esto, ModSecurity ofrece distintas funciones conocidas

como funciones de transformación.

Las funciones de transformación de una regla son definidas junto con las acciones y se pueden distinguir debido a que cuentan con un prefijo característico: "t:". Siguiendo el ejemplo anterior, se podría modificar la regla de la siguiente manera:

```
SecRule REQUEST_FILENAME "@contains admin"
    "phase:1,id:1,t:lowercase,drop"
```

Esto tendría como efecto que, previo a evaluar el operador, se convierta el valor de la variable a minúsculas. De esta manera, se filtrarán también pedidos a direcciones como "www.example.com/adMIN.html", o a cualquier archivo que contenga la palabra "admin", sin distinguir entre mayúsculas y minúsculas.

Reglas con múltiples variables.

Como se comentó al principio, una misma regla puede contener múltiples variables, las cuales deben ir separadas por una barra vertical ("|"). Al ejemplo anterior se le podría agregar la búsqueda de la palabra "admin" en la cadena de consulta:

```
SecRule REQUEST_FILENAME|QUERY_STRING "@contains admin"
    "phase:1,id:1,t:lowercase,drop"
```

Para evaluar una regla con múltiples variables, se evalúa el operador para cada una de las variables. Luego, en caso de que alguna de las variables haya cumplido con la condición del operador, se ejecutan las acciones de la regla.

Evaluación de reglas con colecciones.

Análogamente, para evaluar una regla con una variable que es una colección, se evalúa utilizando los valores de cada una de las entradas de la colección.

Un dato importante es que, a pesar de que no se especifica en la documentación, las colecciones pueden ser de dos tipos: colecciones propiamente dichas, y mapas. Una colección de tipo mapa es, por ejemplo, ARGS. ARGS contiene claves y valores: las claves son los nombres de los argumentos recibidos en un pedido HTTP, y los valores son el valor correspondiente al argumento recibido.

Diferenciar entre los dos tipos de colecciones es fundamental debido a que su evaluación es diferente. Veamos dos ejemplos de colecciones: ARGS y ARGS_NAMES. El primero es un mapa como se indicó previamente, y el segundo es una colección con el nombre de los argumentos recibidos (lo que en el mapa ARGS correspondería a las claves).

Consideremos las siguientes reglas:

```
SecRule ARGS_NAMES "@eq prohibido"
    "phase:1,id:1,drop"
SecRule ARGS "@eq prohibido"
    "phase:1,id:2,drop"
```

La primera regla, como es de esperar, filtra un pedido HTTP si contiene un argumento de **nombre** "prohibido". Por otro lado, la segunda regla filtra un pedido HTTP si contiene un argumento cuyo **valor** es "prohibido".

En otras palabras, para evaluar una regla cuya variable es un mapa, si pensamos el mapa como un conjunto de pares *(clave, valor)*, se evalúa la regla para cada *valor* del mapa.

Las colecciones que son mapas, permiten también utilizar una entrada particular como variable. En este caso, si dicha entrada no se encuentra entonces directamente no se evalúa el operador. Un ejemplo podría ser el siguiente:

```
SecRule ARGS:user "@eq prohibido"
    "phase:1,id:1,drop"
```

En este caso, se filtra el pedido si contiene un argumento de nombre "user" y de valor "prohibido".

Además de la ya vista SecRule, el lenguaje de reglas utilizado por ModSecurity ofrece otras directivas, de las cuales se destacan las siguientes:

- **SecAction:** Ejecuta una acción de manera incondicional.
- **SecDefaultAction:** Define una lista con acciones que serán utilizadas por las siguientes reglas.
- **SecMarker:** Define una marca que será utilizada por otras acciones. Se utiliza, por ejemplo, en conjunto con la acción "skipAfter" para crear flujos del tipo "if-else".
- **SecRuleRemoveById:** Elimina la regla con un ID dado.
- **SecRuleScript:** Crea una regla implementada en Lua.
- **SecRuleUpdateActionById:** Reemplaza la lista de acciones de una regla.

Si bien el lenguaje de ModSecurity permite expresar una gran cantidad de reglas, se vuelve una tarea difícil mantenerlas e ir creando reglas para prevenir ataques nuevos.

Afortunadamente, OWASP creó un proyecto llamado "Core Rule Set" el cual brinda un conjunto de reglas para ModSecurity que es actualizado constantemente y permite proteger una aplicación web de distintos ataques ya conocidos.

2.2 "Core Rule Set"

OWASP es el acrónimo de Open Web Application Security Project. Es una fundación sin fines de lucro que se dedica a la seguridad en aplicaciones web [4]. En particular, OWASP apoya y difunde proyectos que aporten en el área de seguridad de aplicaciones web, y también se encarga de publicar artículos y documentación.

Quizás su proyecto más conocido sea el "OWASP Top 10". Este proyecto, que es actualizado regularmente, incluye los 10 riesgos más importantes para cualquier aplicación web y es utilizado como guía por empresas de todo el mundo para prevenir posibles ataques.

Además del "OWASP Top 10", esta organización es responsable de diversos proyectos tanto de desarrollo como de documentación. Uno de los proyectos de desarrollo más activos es OWASP ZAP, el cual es un escáner de seguridad web, y su objetivo es ser utilizado como una herramienta para realizar pruebas de penetración. Otro de los proyectos más populares de OWASP es el "Core Rule Set".

Si bien las aplicaciones web funcionan de maneras muy variadas, todas ellas pueden ser vulnerables a un conjunto de ataques relativamente comunes. Por lo tanto, aunque ModSecurity le ofrece flexibilidad al usuario para definir reglas que se ajusten específicamente a su aplicación, gran parte de estas reglas son muy similares para la mayoría de las aplicaciones.

Es por esta razón que OWASP publica y mantiene un conjunto de reglas denominado "Core Rule Set" (CRS) [5]. Este conjunto de reglas genéricas, pretende proteger a las aplicaciones de un gran número de ataques, incluyendo aquellos que pertenecen a la lista de los diez ataques más comunes, publicada por la propia organización. Al ser público, permite que los usuarios lo utilicen libremente y lo modifiquen según sus propias necesidades.

El CRS es utilizado, por ejemplo, por los principales proveedores de servicios en la nube como Microsoft Azure (particularmente, por el "Azure Application Gateway" [6]) y Google Cloud (a través de "Google Cloud Armor" [7]). Aún así, no se especifica si los mismos utilizan ModSecurity, una variación de ModSecurity, o un WAF propio para el cual recrearon las reglas del CRS.

Las reglas del "Core Rule Set" permiten detener distintos tipos de ataques como

inyecciones SQL, ataques XSS, LFI y RFI. Su organización es descrita en la sección de análisis, en particular en la sección 3.1.

Una característica importante del CRS es que si bien es un conjunto de reglas estándar, puede ser configurado por el usuario según sus necesidades en distintos aspectos. Un ejemplo de esto son los distintos "niveles de paranoia" configurables en el mismo. Existen cuatro niveles distintos que pueden ser configurados mediante variables de ambiente, siendo 1 el nivel menos estricto y 4 el nivel más estricto. Mientras más alto el nivel de paranoia, más reglas serán ejecutadas.

Si bien el lenguaje de ModSecurity cuenta con una gran cantidad de acciones, variables, transformaciones y operadores, las reglas del CRS sólo utilizan un subconjunto reducido de estas. Aquellas utilizadas por el CRS son presentadas en la sección 4.4.

El "Core Rule Set", al momento de escribir este informe, se encuentra en su versión 3.3.0, conteniendo más de 30 archivos, que suman más de 700 reglas.

3 | Análisis

En los últimos años, la poca evolución de ModSecurity ha causado que surja la necesidad de encontrar una alternativa, entre los WAFs de código abierto, que pueda reemplazarlo. En ese sentido, es fundamental que dicho WAF pueda recrear las reglas del CRS. En el marco de este proyecto, se nos fue solicitado el diseño y la implementación de un WAF moderno que no tenga las limitaciones de ModSecurity y que brinde las funcionalidades necesarias para soportar (al menos, una buena parte) el CRS.

En esta sección, analizamos el CRS y dos de las principales limitaciones de ModSecurity, su lenguaje de definición de reglas y la evaluación de expresiones regulares. Finalizamos presentando los requerimientos del WAF a implementar, con el fin de poder diseñar una solución que sea adecuada al problema.

Para poder crear un WAF que soporte las funcionalidades requeridas por el CRS, es necesario comprender primero cómo se organizan y se pueden diferenciar las distintas reglas del mismo.

3.1 Organización del CRS

Al crecer la cantidad de reglas en el CRS, las mismas fueron separadas en distintos archivos con el fin de mantener una buena organización. Rápidamente, cada uno de esos archivos comenzó a tomar un sentido propio: actualmente, hay un archivo con todas las reglas para prevenir ataques XSS, otro para las reglas que previenen ataques de inyección SQL, etc.

Además de agrupar reglas relacionadas, las reglas de un mismo archivo tienen un flujo de ejecución que no depende de las reglas de los demás archivos. En otras palabras, las reglas para prevenir ataques XSS, por dar un ejemplo, tienen un flujo de ejecución con condiciones (que una regla actúe, puede depender del resultado de otra), pero no dependen de las reglas de los demás archivos.

Recordando las cinco fases de ModSecurity, cada uno de estos archivos contiene sólo

reglas para los pedidos (sólo reglas de fase uno y dos), o sólo reglas para las respuestas (reglas de fase tres y cuatro), o sólo reglas de logging (reglas de fase cinco).

En la figura 3.1 se observa el flujo de ejecución de las reglas del CRS. Primero se ejecutan las reglas de fase uno (que se encuentran sólo en los archivos con reglas para los pedidos) y luego las de fase dos (que se encuentran en los mismos archivos). Después de ejecutar estas reglas, hay un conjunto de reglas de fase dos que son reglas de decisión: son las que determinan si hay que bloquear o no el pedido.

Es importante recordar que en realidad cualquier regla puede bloquear o no una transacción (a través de su acción disruptiva) pero, en el caso del CRS, se utilizan reglas específicas para esto. Esto es porque el CRS está basado en puntajes: cada regla que detecte un posible ataque agrega cierto valor a un puntaje acumulado de la transacción. Cuando se terminan de evaluar las reglas, se compara el puntaje del pedido con un umbral constante y, dependiendo de su valor, se filtra o no dicho pedido.

Cada regla agrega uno de los siguientes puntajes al acumulado: *critical_anomaly_score*, *error_anomaly_score*, *warning_anomaly_score* o *notice_anomaly_score*. Estos puntajes son variables de ambiente del CRS (hablaremos de las variables de ambiente más adelante), y sus valores por defecto son 5, 4, 3 y 2 respectivamente.

Además, los umbrales son también variables de ambiente. Se mencionan umbrales en plural, porque existen dos: uno para la primera toma de decisiones (luego de las reglas de fase uno y fase dos), y otro para la segunda (luego de las reglas de fase tres y fase cuatro). Dichos umbrales son conocidos como *inbound_anomaly_score_threshold* y *outbound_anomaly_score_threshold*, y sus valores por defecto son 5 y 4 respectivamente.

Si, por ejemplo, una regla de fase uno detecta una anomalía y agrega un puntaje de valor *critical_anomaly_score* al acumulado, entonces dicha transacción será intervenida al ejecutar la primera toma de decisiones (en el caso de utilizar los valores por defecto).

Continuando con el flujo de ejecución del CRS, luego de recibir la respuesta (en caso de que el pedido no sea filtrado) se ejecutan las reglas de fase tres, las cuales se encuentran sólo en los archivos con reglas para las respuestas, y luego las de fase cuatro, que se encuentran en los mismos archivos. Similar a la ejecución de las primeras dos fases, la fase cuatro finaliza con la ejecución de reglas de decisión para filtrar o no la respuesta.

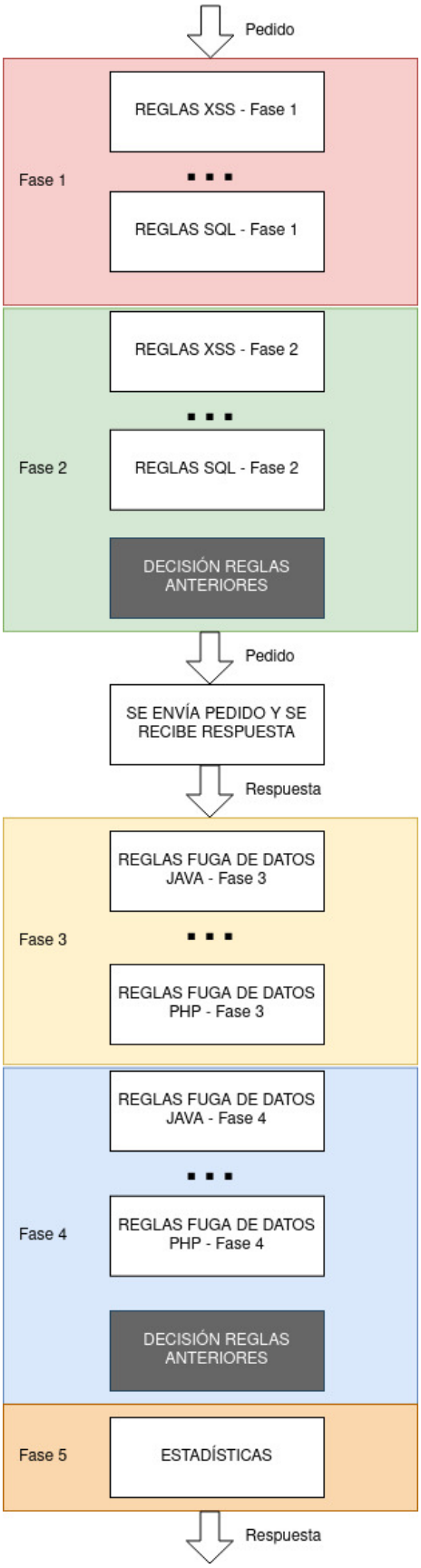

Figura 3.1: Ejecución del CRS sin incluir excepciones

Por último, en caso de no haberse filtrado ni el pedido ni la respuesta, se ejecutan reglas con el fin de mantener información de, por ejemplo, transacciones no bloqueadas, entre otras cosas.

Hasta este punto describimos cinco tipos de archivos:

1. Aquellos que cuentan con reglas para pedidos, como por ejemplo el archivo con reglas para prevenir ataques XSS o el archivo con reglas para prevenir ataques de inyección SQL.
2. El archivo que contiene las reglas para la primera toma de decisión.
3. Aquellos que cuentan con reglas para respuestas, como por ejemplo el archivo con reglas para prevenir ataques de fuga de datos de código en PHP.
4. El archivo que contiene las reglas para la segunda toma de decisión.
5. El archivo que contiene las reglas de fase cinco, cuyo fin es registrar información.

Además de estos archivos, hay otro grupo de archivos del CRS con reglas con otra finalidad: crear excepciones. Estos archivos, que no fueron incluidos en la figura 3.1 para simplificarla, contienen reglas cuyas acciones deshabilitan reglas de los archivos descritos previamente, o no le permiten a algunas reglas el acceso a variables específicas.

Una diferencia particular es que cada regla en estos archivos es independiente, es decir, su ejecución no depende de la ejecución de las demás reglas como en el caso de las reglas de los otros archivos. Por lo tanto, cada regla en un archivo de estos la podemos ver como una excepción independiente, con una condición y un resultado (deshabilitar ciertas reglas y/o negarle el acceso a variables específicas a otras reglas).

Observando esta organización del CRS podemos abstraer algunos conceptos subyacentes, como son los agrupamientos de reglas en distintos archivos y las reglas que denominamos reglas de decisión. A continuación, definimos qué será para nosotros una regla (en el sentido abstracto), una política, una excepción y un evaluador.

3.1.1 Reglas y Políticas

Comenzamos mostrando en la figura 3.2 los distintos conceptos extraídos y cómo se relacionan. En este contexto, un **objeto** es lo que se va a evaluar, puede ser un pedido o una respuesta. Con **configuración**, nos referimos a variables de ambiente que influyen en una evaluación. Por ejemplo, podría haber una política cuyo objetivo sea

bloquear ciertas direcciones IP, en ese caso las direcciones a bloquear serían parte de la configuración.

El **estado**, por otro lado, es utilizado para compartir información a través de las políticas y las reglas. El mismo puede ser separado en dos: el estado actual de la transacción y el estado global. Profundizamos en este concepto más adelante.

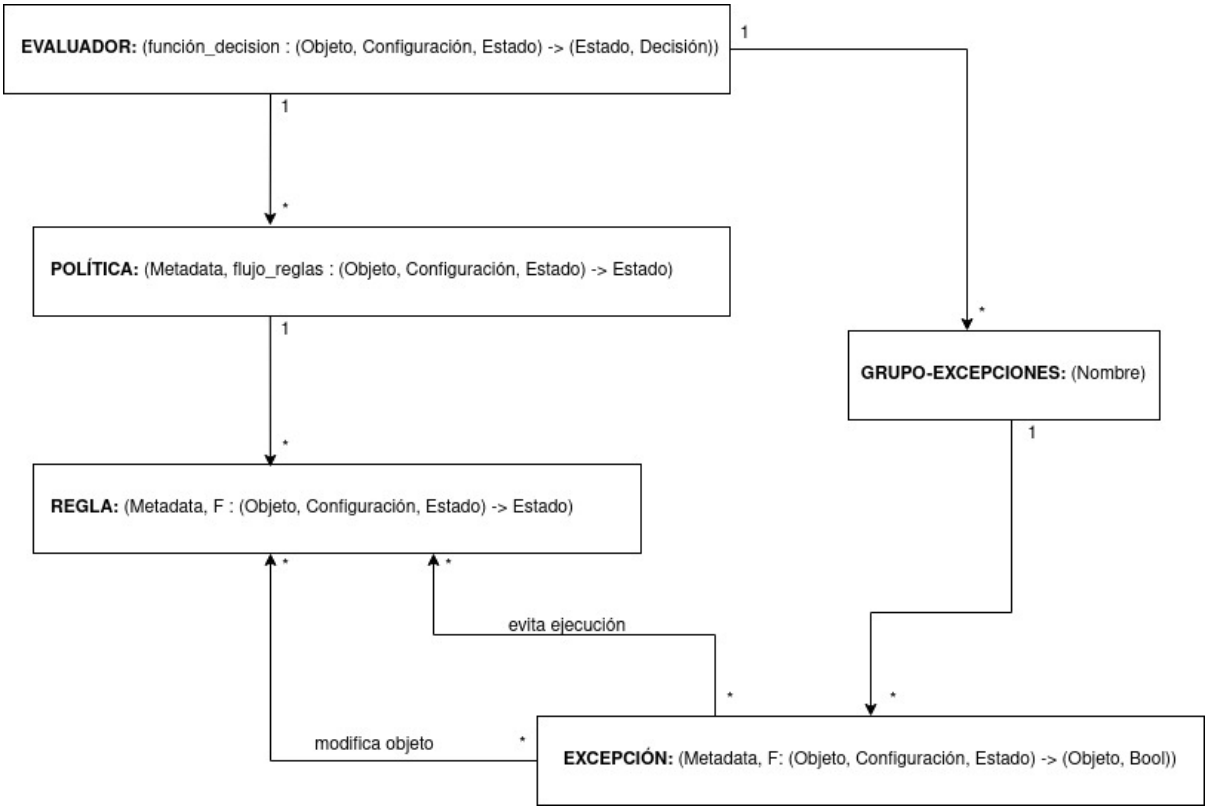

Figura 3.2: Evaluador, políticas, reglas y excepciones

Luego de describir a qué nos referimos con objeto, configuración y estado, comenzamos con los conceptos más importantes. Una **regla** (en el sentido abstracto, no en el sentido de ModSecurity) está compuesta por información (como puede ser un identificador, una versión, etc) a la que le denominamos *metadata*, y por una función que recibe un objeto, un estado y una configuración, y retorna un nuevo estado.

Por otro lado, una **política** también está compuesta por su *metadata* y por una función del mismo tipo que la de una regla, pero se le agrega un conjunto de reglas. La función de la política puede invocar a las funciones de sus respectivas reglas y, por lo tanto, representa el flujo de ejecución de dichas reglas. Los archivos con reglas para pedidos del CRS, por ejemplo, los podemos ver como políticas, que están compuestas por distintas reglas y un flujo de ejecución para las mismas.

Por último, un **evaluador** tiene un conjunto ordenado de políticas, un conjunto de

excepciones y una función, a la cual le denominamos función de decisión (aunque, como comentamos luego, no es su único propósito). Pensando en el caso de un pedido HTTP, el objeto contendría información de dicho pedido y, para evaluar qué decisión tomar (bloquearlo, redirigirlo, etc) sobre dicho pedido, se deben invocar las funciones de las políticas del evaluador en orden y luego, utilizando el estado final, ejecutar la función de decisión.

De esta manera, podríamos ver al CRS como dos evaluadores (véase la figura 3.3): uno con las reglas que actúan sobre los pedidos, y otro con las reglas que actúan sobre las respuestas. Además, las reglas de ModSecurity de fase cinco no son reglas en sí, sino que se enfocan en guardar información; este trabajo podría ser realizado en la función de decisión del segundo evaluador. En otras palabras, la función de un evaluador no tiene por qué ser estrictamente sólo de decisión, sino que es una función que se ejecuta al final de las políticas y, por lo tanto, puede también utilizarse para recopilar información de la ejecución de las políticas y, entonces, almacenar información que parezca conveniente.

Recordemos que en ModSecurity las reglas de fase uno se ejecutan antes que las reglas de fase dos, y así para todas las fases. Aún así, las reglas dentro de distintos archivos son independientes como comentamos previamente. Por lo tanto, no es necesario que las reglas de fase dos del archivo de reglas de XSS, por dar un ejemplo, se ejecuten antes que las reglas de fase uno del archivo de reglas de inyecciones SQL. De este modo, desde nuestra abstracción, podemos pensar todas las reglas de XSS (tanto de fase uno, como de fase dos) como una única política, y lo mismo con los demás archivos.

En otras palabras, si bien el flujo de la figura 3.3 no es estrictamente el que vemos en la figura 3.1 (en este último, todas las reglas de fase uno se ejecutan antes que las de fase dos), ambos son equivalentes.

3.1.2 Excepciones

Otra vez, para simplificar el flujo de ejecución, evitamos en un primer momento mencionar el papel de las excepciones en esta ejecución.

En la figura 3.2 podemos ver como cada excepción está dada por su información (identificador, versión, etc), otra vez denominada *metadata*, y una función que recibe un objeto, una configuración y un estado, y retorna un objeto y un *booleano*.

Cada regla puede tener asociadas múltiples excepciones, la relación entre una regla y una excepción puede ser una de dos: la excepción puede **evitar la ejecución de la regla**, o la excepción puede **modificar el objeto a recibir por la regla**.

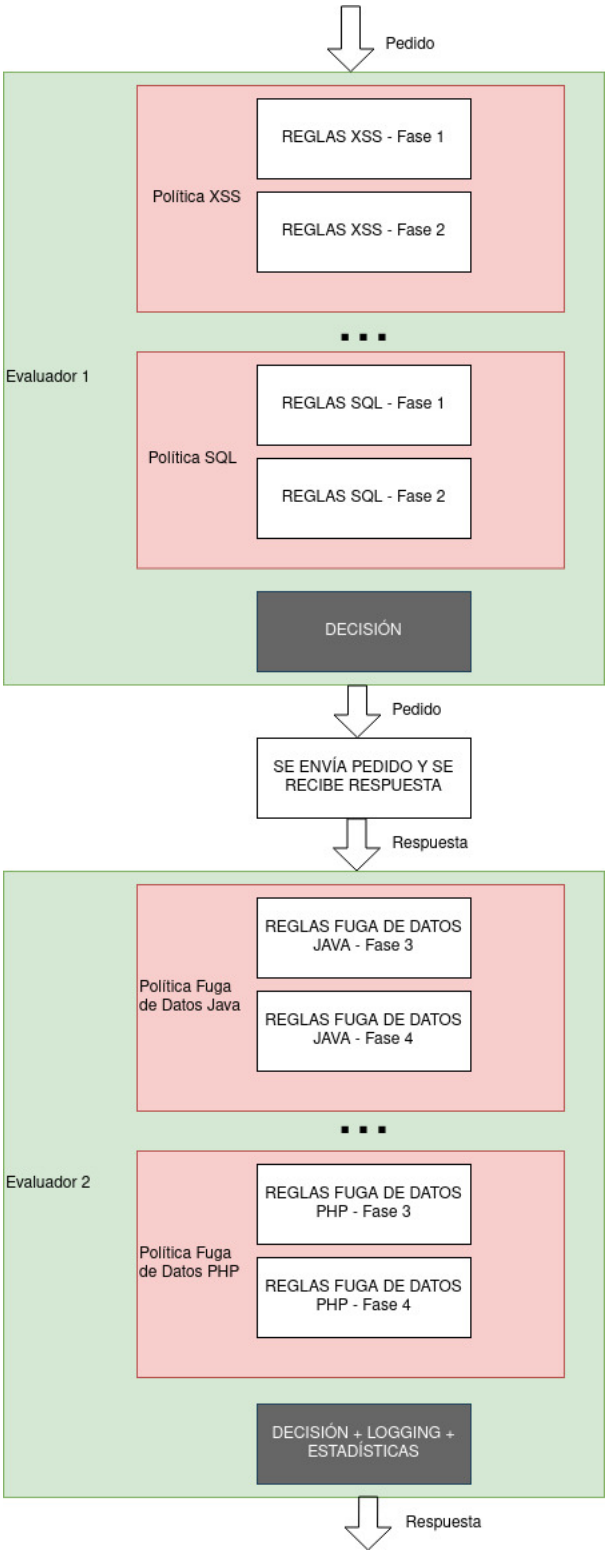

Figura 3.3: Viendo el CRS como dos evaluadores, con sus políticas y sus funciones de decisión

El valor *booleano* retornado por la función de la excepción indica si la misma se debe aplicar o no. Recordemos que las excepciones pueden evitar la ejecución de una regla, pero también pueden evitar que dicha regla acceda a información específica del

objeto. En ese sentido, el objeto retornado por la función es el resultado de quitarle dicha información al objeto que, en un principio, iba a recibir la regla.

Al ejecutar una regla con una única excepción, se ejecuta primero la función de la excepción. Si el resultado de la excepción indica que la misma no se debe aplicar, entonces se ejecuta la función de la regla sin ningún cambio. Si, por el contrario, la excepción se debe aplicar, hay dos casos dependiendo de la relación entre la regla y la excepción:

1. La regla no se debe ejecutar.
2. La regla se debe ejecutar, pero utilizando el objeto retornado por la función de la excepción (un objeto con menos información). Esto no modifica el objeto para el resto de las reglas.

En caso de que una regla cuente con más de una excepción, las mismas deben tener un orden de aplicación. En caso de que una de las excepciones indique que la regla no debe ser ejecutada, entonces la misma no será ejecutada. En caso contrario, el objeto utilizado por la regla será el resultado de las distintas funciones de aquellas excepciones que se deben aplicar.

Las excepciones se organizan en distintos grupos. En el CRS hay distintos archivos con excepciones y cada uno de ellos corresponde a un grupo. Por ejemplo, se cuenta con un archivo con excepciones para algunas reglas en caso de estar ejecutando una aplicación de WordPress.

3.1.3 Variables globales

En el análisis previo, describimos lo que llamamos un estado, y comentamos que dicho estado es el medio por el cual las distintas reglas y políticas pueden comunicarse entre sí: cada política puede leer o modificar dicho estado en su flujo de ejecución, y en cada una de sus reglas. Además, dicho estado luego es utilizado por la función de decisión para tomar la decisión final.

Además, el estado se puede separar en dos: el estado de la transacción y el estado global. En el caso del CRS, el estado de la transacción es almacenado en la colección "TX", mientras que el estado global se almacena en la colección "IP" (es una colección persistente de ModSecurity).

Información de un cliente.

La justificación de la utilización de la colección "IP" es simple: las reglas del CRS necesitan acceder a información global de un cliente (identificado por su dirección

IP), esencialmente para detectar ataques de denegación de servicio.

ModSecurity ofrece una colección "IP" para compartir información a través de distintas transacciones: dada una dirección IP, todas las transacciones donde el cliente tenga dicha dirección acceden a la misma colección global a través de "IP".

Esta colección es la única utilizada por el CRS para mantener información sobre el estado global. En otras palabras, es la única variable que permite a distintas transacciones (en caso de ser iniciadas por el mismo cliente) compartir información entre ellas.

Concurrencia.

Como "IP" es una colección persistente, que puede ser accedida por distintas transacciones iniciadas por el mismo cliente, un cliente podría iniciar dos transacciones distintas, en paralelo, y ambas podrían intentar modificar la colección "IP" al mismo tiempo. Naturalmente, surge la necesidad de entender cómo ModSecurity maneja dichos accesos concurrentes a la misma información.

En ModSecurity las colecciones persistentes son mapas (pares *(clave, valor)*). El manejo de variables de ModSecurity permite tres operaciones para modificar dichos mapas:

- Eliminar un par *(clave, valor)*.
- Ingresar un par *(clave, valor)*, eventualmente sustituyendo a otro par con la misma clave.
- Modificar el valor de un par *(clave, valor)*: dada una clave del mapa y un *número*, se actualiza el par *(clave, valor)* por el par *(clave, valor + número)*.

Podemos pensar que, para cada dirección IP, ModSecurity almacena una colección con información de dicha dirección. Al iniciar una transacción, se copia en la colección "IP" la información almacenada por ModSecurity para el respectivo cliente. De esta manera, si dos transacciones del mismo cliente inician al mismo tiempo, ambas tendrán la misma información en la colección "IP", pero si una modifica su colección la otra no notará dicho cambio ya que acceden a copias distintas.

Para cada transacción, ModSecurity registra las operaciones realizadas sobre la colección "IP". Al finalizar la transacción, se aplican (en orden) todos los cambios realizados a la copia, sobre la colección verdadera.

Vemos que si quisiéramos almacenar la información de cuántas veces un cliente ha iniciado una transacción, por ejemplo, bastaría crear una regla que aumente en uno la entrada con nombre "contador_total" de la colección "IP". Sin embargo, esto **no** es lo

mismo a obtener el par *(contador_total, valor)* de la colección "IP", y sustituirlo ingresando el par *(contador_total, valor + 1)*.

El ejemplo anterior es el porqué de la existencia de una tercera operación que modifique el valor de un par en la colección. Si registramos que una transacción **ingresó** un par *(clave, valor)*, entonces cuando actualicemos la información estaremos sustituyendo el valor anterior. En cambio, si registramos que una transacción **aumentó** el valor de una entrada, no sustituiremos el valor anterior, sino que lo aumentaremos.

Configuración.

Además de la colección "IP", otra colección interesante es la colección "TX". La misma es una colección de lectura/escritura, y fue creada para almacenar información de la transacción que está siendo evaluada. Se utiliza, por ejemplo, para llevar el puntaje con el cual se tomará la decisión final. Si bien está claro que la función de dicha colección es la de mantener gran parte del estado de la transacción, en el CRS se utiliza también dicha variable para almacenar las variables de ambiente.

Hay distintas variables de ambiente en el CRS, una en particular es el "nivel de paranoia". Según el nivel de paranoia que se le configure al CRS, algunas reglas se ejecutarán o no. No sólo el nivel de paranoia, sino otros valores como cada cuantos pedidos por segundo se diagnostica que una dirección IP está realizando un ataque de denegación de servicio, se pueden configurar.

Para esto, el CRS cuenta con una regla que se ejecuta al comienzo de cada transacción y le asigna los valores a las variables de ambiente: por ejemplo, agrega una entrada con el nivel de paranoia en la colección "TX". Luego, todas las reglas siguientes pueden acceder al nivel de paranoia a través de la colección "TX".

Aquí observamos un problema de separación de responsabilidades. La colección "TX" se debería utilizar para almacenar información de una transacción, no información global a todas las transacciones, como es el caso de la configuración.

Observamos también que la configuración no puede ser modificada en tiempo de ejecución. Si una regla modifica el valor del nivel de paranoia de la colección "TX", lo está modificando para la transacción que se está ejecutando: en otras palabras, modifica el nivel de paranoia para las reglas siguientes que se ejecuten, pero dentro de la misma transacción.

3.2 Lenguaje de ModSecurity

Al empezar a abstraer los conceptos de política, regla, excepción y función de decisión, se empieza a vislumbrar que el lenguaje ofrecido por ModSecurity no permite representar adecuadamente estos conceptos. El problema es simple: en ModSecurity "no hay más" que reglas. El concepto de política, por ejemplo, no existe más allá de separar reglas en distintos archivos.

Lo que denominamos función de decisión se implementa como un conjunto de reglas de ModSecurity. Esto no debería ser así, ya que hay una distinción clara entre las reglas (como concepto abstracto) y la función de decisión. El mismo problema se repite constantemente, y en distintos niveles. Veamos la definición de las siguientes reglas consecutivas:

```
SecRule REQUEST_FILENAME "@endsWith login.php"
    "id:1,pass,skip:2,phase:1"
SecRule ARGS_GET:arg "@ge 3" "id:2,drop,phase:1"
SecAction "pass,skip:1,phase:1"
SecRule ARGS_GET:arg "@ge 2" "id:4,drop,phase:1"
```

La directiva "SecAction" ejecuta acciones incondicionalmente, la podemos ver como una regla sin operador (y, por lo tanto, sin variables). Mientras que la acción "skip:*k*" evita la ejecución de las siguientes *k* reglas.

Conceptualmente hay sólo dos reglas, la de identificador igual a 2, y la de identificador igual a 4. Lo que hace dicha secuencia es ejecutar la regla 4 si el archivo de destino del pedido termina con "login.php", y sino ejecutar la regla 2. En resumen, necesitamos cuatro reglas para simular un "if-else" entre dos reglas.

Lo anterior es un claro ejemplo de un flujo que, a priori, sería simple (un flujo "if-else"), pero que se vuelve complicado. No sólo se vuelve tedioso al trabajar con múltiples reglas, sino que es clara la falta de separación de responsabilidades: el flujo de ejecución de las reglas, se define con más reglas.

Además, ModSecurity ofrece un conjunto de acciones que permiten realizar flujos más interesantes que un "if-else", como son "ruleRemoveById", "ruleRemoveByTag", "ruleRemoveTargetById" y otras. Estas acciones sirven para eliminar reglas en tiempo de ejecución, o no permitir que algunas reglas accedan a variables específicas. Si bien permiten implementar flujos más complejos, estas acciones hacen que un conjunto de reglas se vuelva mucho más complicado de analizar y de mantener.

En el ejemplo anterior también se visualiza otra característica de ModSecurity: el

identificador de la regla está dentro de las acciones a tomar. No sólo el identificador, sino los "tags" (las categorías) de una regla, y cualquier información de la misma, se encuentra dentro de las acciones a tomar. Esto no parece intuitivo: las acciones asociadas a una regla se ejecutan sólo si una de las variables dadas cumple la condición del operador, pero la información de una regla no depende de su ejecución.

Además de la clara falta de separación de responsabilidades, otro de los principales problemas del lenguaje de ModSecurity es que no es un lenguaje familiar. Para un administrador de servidores que no haya trabajado previamente con ModSecurity, escribir reglas de seguridad no se le hará sencillo en un principio.

De todos modos, las falencias de ModSecurity no están necesariamente ligadas a su lenguaje. En la próxima sección, describimos otra de las principales limitaciones del mismo.

3.3 Evaluación de expresiones regulares en ModSecurity

Uno de los operadores más populares de ModSecurity es el operador **rx**. Dicho operador permite utilizar expresiones regulares con el fin de verificar si se cumple alguna condición en datos de un pedido o una respuesta. Para evaluar dichas expresiones regulares, ModSecurity utiliza la librería PCRE del lenguaje C, la cual no siempre es eficiente (puede llegar a alcanzar tiempos exponenciales).

El problema es que PCRE utiliza *backtracking* recursivo para la evaluación de las expresiones regulares y, por lo tanto, puede sufrir ataques de ReDoS (denegación de servicio a expresiones regulares) [8].

Los ataques de ReDoS se aprovechan del hecho de que un evaluador de expresiones regulares tenga una complejidad exponencial en el tiempo, para el peor caso (el tiempo puede crecer exponencialmente en relación con el tamaño de la entrada). De esta manera, un ataque puede utilizar un valor específico que haga que al evaluarse en una expresión regular tome un tiempo muy alto de procesamiento, logrando consumir una gran cantidad de recursos.

Particularmente, PCRE cuenta con una implementación basada en *backtracking* recursivo para poder ofrecer funcionalidades de *lookaround*. Dichas funcionalidades permiten definir expresiones regulares donde la coincidencia, o no, de una cadena con una expresión regular depende de su entorno. Más específicamente, sean q y u expresiones regulares, PCRE permite definir las siguientes expresiones [9]:

1. $q(?=u)$ - Dada una cadena, una subcadena coincide con dicha expresión regular

si coincide con la expresión regular q, y **está seguida** por una subcadena que coincide con la expresión u (no hace a dicha subcadena parte de la coincidencia).

2. $q(?!u)$ - Dada una cadena, una subcadena coincide con dicha expresión regular si coincide con la expresión regular q, y **no está seguida** por una subcadena que coincida con la expresión u.
3. $(?<=u)q$ - Dada una cadena, una subcadena coincide con dicha expresión regular si coincide con la expresión regular q, y **está precedida** por una subcadena que coincide con la expresión u (no hace a dicha subcadena parte de la coincidencia).
4. $(?<!u)q$ - Dada una cadena, una subcadena coincide con dicha expresión regular si coincide con la expresión regular q, y **no está precedida** por una subcadena que coincida con la expresión u.

A diferencia de PCRE, otros evaluadores de expresiones regulares -cuya implementación está basada en autómatas- garantizan tiempos lineales de ejecución. Dichos evaluadores no permiten la utilización de las expresiones *lookaround*.

El CRS, por ejemplo, cuenta con tres expresiones regulares (de un total de 272) que utilizan dichas funcionalidades.

3.4 Requerimientos

El WAF a implementar no sólo debe proveer las funcionalidades básicas de cualquier proxy inverso, sino que debe dar solución a las principales limitaciones de ModSecurity. Principalmente, debe ofrecer un lenguaje que permita expresar las políticas del CRS, diferenciando de manera acorde los distintos conceptos relevados en este análisis.

A continuación, listamos los requerimientos separados entre requerimientos no funcionales, y requerimientos funcionales.

3.4.1 Requerimientos funcionales

En el caso de los requerimientos funcionales, empezamos listando los requerimientos generales de un proxy inverso. El nuevo WAF debe actuar como intermediario entre clientes y un grupo de servidores, comunicándose, a través del protocolo HTTP o HTTPS, con cualquiera de estos. Dichos requerimientos se listan a continuación.

RF_01	El WAF debe poder configurarse para escuchar en distintos puertos. Pudiendo aceptar conexiones utilizando el protocolo HTTP o el protocolo HTTPS.
RF_02	El WAF debe poder transferir un pedido recibido hacia una aplicación final a configurar, y retornar su respuesta.
RF_03	El WAF debe poder acceder a múltiples servidores finales, los cuales podrán utilizar tanto el protocolo HTTP como el protocolo HTTPS.
RF_04	Al WAF se le debe poder configurar una estrategia de balanceo de carga para elegir con cuál servidor final comunicarse.

Además de las funcionalidades generales de un proxy inverso, es necesario que el WAF permita representar los conceptos relevados en esta sección y utilizarlos para evaluar y filtrar, en caso de ser necesario, las transacciones correspondientes. Para esto, debe poder definir reglas, políticas, excepciones y funciones de decisión para **pedidos**:

RF_05	El WAF debe permitir definir reglas para los **pedidos**.
RF_06	El WAF debe permitir agrupar reglas para los **pedidos** en políticas, permitiendo definir el flujo de ejecución de las mismas.
RF_07	El WAF debe permitir definir una función de decisión sobre los **pedidos**.
RF_08	El WAF debe permitir definir excepciones para las reglas que actúen sobre **pedidos**. Pudiendo agrupar dichas excepciones en grupos.
RF_09	El WAF debe poder evaluar **pedidos** a partir de un conjunto ordenado de políticas para los mismos, una función de decisión y un conjunto de excepciones. Ejecutando secuencialmente las distintas políticas, previo a la ejecución de la función de decisión.

Análogamente, el WAF debe poder definir reglas, políticas, excepciones y funciones de decisión para **respuestas**:

RF_10	El WAF debe permitir definir reglas para las **respuestas**.
RF_11	El WAF debe permitir agrupar reglas para las **respuestas** en políticas, permitiendo definir el flujo de ejecución de las mismas.
RF_12	El WAF debe permitir definir una función de decisión sobre las **respuestas**.
RF_13	El WAF debe permitir definir excepciones para las reglas que actúen sobre **respuestas**. Pudiendo agrupar dichas excepciones en grupos.
RF_14	El WAF debe poder evaluar **respuestas** a partir de un conjunto ordenado de políticas para las mismas, una función de decisión y un conjunto de excepciones. Ejecutando secuencialmente las distintas políticas, previo a la ejecución de la función de decisión.

Requerimientos del lenguaje.

Para poder evaluar pedidos y respuestas a partir de políticas y funciones de decisión, es necesario que las funciones asociadas a dichos conceptos puedan ser definidas en un lenguaje apropiado:

RF_15	El WAF debe contar con un lenguaje con la finalidad de expresar las funciones de decisión y las funciones asociadas a las reglas, políticas y excepciones. En otras palabras, dicho lenguaje será utilizado para implementar las funciones que se encuentran en la figura 3.2.

Dicho lenguaje debe ser más simple que el lenguaje de ModSecurity. En particular, que sea un lenguaje de programación imperativa facilitará el aprendizaje del mismo, debido a la popularidad de estos. Además, el lenguaje **no** debe ser utilizado para definir la información de las reglas (su *metadata*), como sí ocurre en ModSecurity.

Con el fin de definir el flujo de ejecución de las distintas reglas (recordemos que el flujo de ejecución de las reglas, es dado por la función de la política correspondiente), el lenguaje debe contar con estructuras de control básicas. En base a estos dos puntos, se listan los siguientes dos requerimientos:

RF_16	El WAF debe separar claramente la definición de la *metadata* de una regla, política o excepción, de su función asociada (la cual será implementada, como se describió recientemente, en un lenguaje particular).
RF_17	El lenguaje debe tener una sintaxis simple, similar a un lenguaje imperativo, contando con estructuras de control básicas que permitan definir flujos de ejecución.

Utilizando dicho lenguaje se implementarán las funciones de las reglas, políticas, etc. Las mismas deberán, utilizando el lenguaje, acceder a información del pedido o de la respuesta. Además, el lenguaje deberá contar con tipos básicos que permitan representar dicha información. Particularmente, deberá contar con un tipo especial para el manejo de direcciones IP, las cuales en ModSecurity no se distinguen de las cadenas de bytes.

RF_18	El lenguaje debe permitir a las reglas, políticas y funciones de decisión de **pedidos**, acceder a información del **pedido**.
RF_19	El lenguaje debe permitir a las reglas, políticas y funciones de decisión de **respuestas**, acceder a información de la **respuesta**.
RF_20	El lenguaje debe contar con tipos de datos básicos como son: números, cadenas de bytes, booleanos, colecciones y mapas. Además, deberá contar con un tipo especial para el manejo de direcciones IP.

Uno de los problemas que observamos de ModSecurity, particularmente de cómo el CRS utiliza ModSecurity, es que no se distingue aquellas variables que son de ambiente (configuración del CRS), o aquellas que son información propia de la transacción (como puntajes acumulados, etc). Además, en el caso de las variables globales (la información de un cliente, por ejemplo), el manejo de concurrencia de ModSecurity parece ser anticuado o, al menos, poco intuitivo. Al trabajar con un lenguaje de programación imperativa, se pueden utilizar mecanismos primitivos de concurrencia para administrar los accesos a las variables globales. Es importante observar que cuando nos referimos a variables globales, no estamos considerando como tales a las variables de ambiente. En esta dirección, se establecen los siguientes dos requerimientos.

RF_21	El lenguaje debe distinguir, claramente, aquellas variables que sean de ambiente, aquella información que sea propia de la transacción (contenida en variables locales), y aquella información que sea global (como información de cada cliente).
RF_22	El lenguaje debe proveer mecanismos de exclusión mutua en caso de contar con variables globales.

Por último, con el fin de que el WAF pueda ser considerado una alternativa a ModSecurity, el mismo debe poder recrear el CRS, a menos de aquellas reglas que utilicen expresiones regulares que no puedan ser expresadas sin las funcionalidades de *lookaround*. Esto se debe a que, como se explica en los requerimientos no funcionales, el WAF deberá utilizar otro evaluador de expresiones regulares distinto a PCRE, y no todas las expresiones regulares utilizadas en el CRS podrán ser admitidas por dicho evaluador.

RF_23	El lenguaje debe permitir la evaluación de expresiones regulares.
RF_24	El WAF debe poder expresar las políticas, excepciones y funciones de decisión del CRS, a menos de aquellas reglas que evalúen expresiones regulares que no puedan ser expresadas sin las funcionalidades de *lookaround*

Para finalizar, describimos los requerimientos no funcionales.

3.4.2 Requerimientos no funcionales

En el marco de este proyecto, se nos solicitó que el WAF a implementar fuese desarrollado en Rust. El principal motivo para desarrollar sistemas de seguridad en Rust es que garantiza seguridad de memoria (evitando ataques como desbordamiento de buffers o punteros colgantes), además de evitar condiciones de carrera (acceso no sincronizado a una misma sección de la memoria desde dos o más hilos de ejecución) y accesos a punteros nulos. Más aún, la seguridad de memoria es alcanzada sin utilizar un "colector de basura", manteniéndose así a la par de C/C++ en eficiencia. Por lo tanto, el primer requerimiento no funcional es el siguiente.

RNF_1	El WAF debe ser desarrollado en Rust.

Como ya anticipamos, el WAF a desarrollar utilizará un evaluador de expresiones regulares distinto a PCRE. De hecho, debido a la posibilidad de recibir ataques de ReDoS, para la evaluación de expresiones regulares el WAF deberá utilizar un evaluador cuyo orden de tiempo de ejecución sea lineal con respecto al largo de la entrada.

RNF_2	El WAF debe poder evaluar expresiones regulares en un tiempo lineal con respecto al tamaño de la entrada.

4 | Diseño

Luego de haber definido los requerimientos del WAF a implementar, continuamos con el diseño de dicho WAF. Empezamos con el diseño de la arquitectura de un proxy inverso y luego con el diseño de sus funcionalidades de seguridad.

Recordemos que un WAF es un caso particular de un proxy inverso y, por lo tanto, empezamos diseñando la arquitectura de un proxy inverso que cumpliera con lo definido en los requerimientos y que además permitiera ser extendido fácilmente para incorporar otras funcionalidades.

Una vez diseñado este proxy inverso, continuamos con el diseño de una componente particular, a la que le denominamos "CRS-Engine", que permitirá brindar funcionalidades de seguridad, transformando el proxy inverso específicamente en un WAF. Para esto empezamos describiendo el lenguaje a ser utilizado por dicha componente para definir políticas de seguridad, continuando con su diseño y, por último, justificando por qué dicho "Engine" permite expresar las políticas del CRS.

4.1 Arquitectura

Para el diseño de esta arquitectura hicimos un relevamiento de la arquitectura de cuatro proxys inversos ampliamente utilizados: Traefik, Nginx, Envoy y HAProxy. Dicho relevamiento se encuentra en el apéndice A1.

Utilizando el relevamiento de estos proxys inversos diseñamos uno que pudiera brindar las funcionalidades básicas de otros proxys inversos, pero al mismo tiempo que se pudiera extender fácilmente para ofrecer más funcionalidades. Con ese objetivo en mente, el diseño final se puede ver en la figura 4.1.

Una de las primeras decisiones tomadas fue que, debido a que el objetivo final era implementar un "Web Application Firewall", no era necesario en nuestro caso ofrecer funcionalidades de TCP proxy como son los casos de Envoy y HAProxy, sino que debíamos trabajar al nivel del protocolo HTTP.

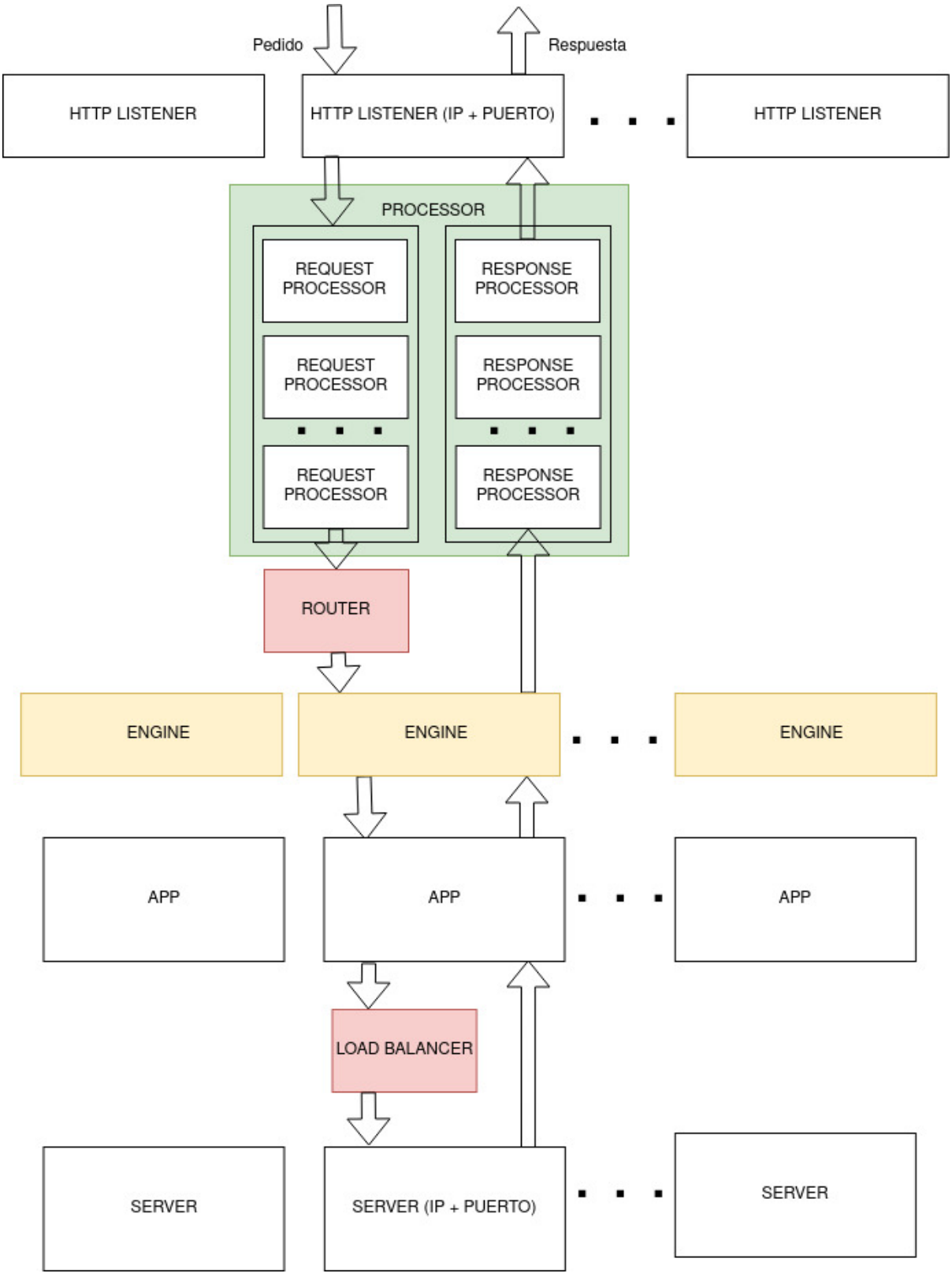

Figura 4.1: Arquitectura del WAF

En todos los proxys inversos relevados, observamos cómo se los podía utilizar para recibir pedidos en más de una interfaz de red o, incluso, en distintos puertos. Esta es una de las funcionalidades esperadas de cualquier proxy inverso y, como se indica en

los requerimientos, este proyecto no es la excepción. Para esto creamos el concepto de "HTTP Listener", que es el encargado de recibir pedidos HTTP (o HTTPS) en una dirección IP y puerto fijos.

Luego de recibir un pedido, el mismo será procesado por el "Processor", el cual no depende del "HTTP Listener" que haya recibido el pedido. Este componente realizará tareas estándar, como son verificar que el pedido esté bien formado y eliminar las **cabeceras "hop-by-hop"**.

Las cabeceras "hop-by-hop" son un grupo de cabeceras que sólo tienen sentido en una única conexión a nivel de transporte y, por lo tanto, no deben ser retransmitidas por servidores proxys. Algunas de las mismas son, por ejemplo, las cabeceras "Connection" y "Keep-Alive".

Una vez procesado por el "Processor", será el turno del "Router" (también único), de decidir qué "Engine" será aplicado al pedido y cuál será la aplicación final del mismo. Cada "Engine" puede modificar o incluso bloquear un pedido, o a su respuesta.

Por último, si bien el "Router" indica la aplicación final, puede haber más de un servidor ejecutando la misma aplicación. Por lo tanto, deberá utilizarse un balanceador de carga (al que denominamos "Load Balancer") para decidir a cuál servidor se le enviará el pedido. El "Load Balancer" también se encargará de asignarle los valores correspondientes a la **cabecera "Forwarded"** y las **cabeceras "X-Forwarded-*"**.

La cabecera "Forwarded" y las cabeceras "X-Forwarded-For", "X-Forwarded-Host" y "X-Forwarded-Proto" (denominadas cabeceras "X-Forwarded-*") permiten brindarle información al servidor final sobre el camino que realizó un pedido antes de llegar a dicho servidor. En otras palabras, un servidor proxy utiliza dichas cabeceras para indicarle al servidor final que el mensaje es un mensaje retransmitido, e incluso indicarle quién es el cliente real. La cabecera "Forwarded" reúne toda la información brindada por las cabeceras "X-Forwarded-*", siendo esta la cabecera estándar. Aún así, las cabeceras "X-Forwarded-*" son ampliamente utilizadas, siendo consideradas el estándar *de-facto*.

Luego de que el servidor final responda, dicha respuesta será también procesada por el "Engine" en un primer momento, y luego por el "Processor".

En la imagen se puede observar que dentro de un "Processor" hay múltiples "Request processors" y "Response processors". Cada uno de estos tendrá una responsabilidad única y la idea es que, en un futuro, otros desarrolladores puedan agregar más de estos a medida que sea necesario, y que incluso se pueda configurar cualás "Request/Response processors" incluir en el "Processor", y cuáles no.

Dentro de lo que denominamos "Engine" en el diseño anterior se podrán brindar funcionalidades de seguridad. En particular, diseñamos un "Engine" específico, al cual denominamos "CRS-Engine", que ofrecerá la posibilidad de ejecutar políticas de seguridad sobre pedidos y respuestas. Para esto, el "Engine" deberá recibir políticas de seguridad que serán implementadas en un lenguaje diseñado con dicho fin.

4.2 Lenguaje para políticas de seguridad

El lenguaje a ser utilizado para implementar políticas de seguridad por parte del "CRS-Engine" es un lenguaje simple de programación imperativa, con el fin de que se pueda obtener un código limpio y mantenible, así como ser familiar para cualquier administrador de "Web Application Firewalls".

En esta sección, describimos conceptos generales del lenguaje, mientras que en la sección 4.3 se describe cómo utiliza el "CRS-Engine" dicho lenguaje para implementar distintas políticas. La formalización de la sintaxis y la semántica del lenguaje, y la especificación de funciones predefinidas en el mismo, se encuentran en el apéndice A2.

4.2.1 Tipos

El lenguaje tiene tipado dinámico, por lo cual a una misma variable se le pueden asignar valores de distintos tipos. De todos modos, los valores tienen un único tipo, el cual puede ser: *bool*, *int*, *string*, *array*, *hash*, *IP* o *undefined*. El tipo *undefined* se puede ver tanto como un tipo, como un valor: básicamente, hay un valor especial que se denomina *undefined* y cuyo tipo es *undefined*.

A los tipos *bool*, *int* y *string* los denominamos tipos primitivos. Al igual que en la mayoría de los lenguajes de programación:

- Hay dos valores de tipo *bool*: *true* y *false*.
- Un valor de tipo *int* representa un número entero entre -2.147.483.648 (-2^{31}) y 2.147.483.647 ($2^{31} - 1$).

Los valores de tipo *string* representan una cadena de bytes cualquiera. A lo largo de este informe, muchas veces hablamos de una cadena de caracteres como un *string*, en dicho caso hacemos referencia a la codificación ASCII de dicha cadena de caracteres. Por ejemplo, si hablamos del *string* "hola", hacemos referencia a la cadena de bytes *(0x68, 0x6F, 0x6C, 0x61)*.

Un valor de tipo *IP* representa una dirección IPv4 o una dirección IPv6. Por otro lado, un *hash* es un conjunto (potencialmente vacío) de pares (*clave*, *valor*), donde *clave* es

un valor de tipo primitivo, y *valor* es un valor de cualquier tipo. En un *hash*, un valor de tipo primitivo puede aparecer una única vez como *clave* de un par. Le denominaremos un *valor de tipo hash* a una referencia a un *hash* en memoria: en otras palabras, las variables con un valor de tipo *hash* en realidad contendrán una referencia a un *hash* en memoria, por lo cual más de una variable podría tener una referencia al mismo *hash*.

Por último, un *array* es una lista ordenada (potencialmente vacía) de valores de cualquier tipo. A la cantidad de entradas le denominamos el *largo del array*. Podemos ver, de otra manera, a los *arrays* como una colección de pares (*índice*, *valor*), en donde:

- *índice* es un valor de tipo *int*, mayor o igual a cero, y menor al largo del *array*.
- *valor* es un valor de cualquier tipo.

De esta manera, el par (0, *valor*) corresponde al valor en la primera posición, (1, *valor*) al de la segunda, etc.

Le denominaremos un *valor de tipo array* a una referencia a un *array* en memoria: en otras palabras, las variables con un valor de tipo *array* en realidad contendrán una referencia a un *array* en memoria, por lo cual más de una variable podría tener una referencia al mismo *array*.

4.2.2 Funciones

La construcción más compleja del lenguaje es una función, la sintaxis de una función es:

```
fn nombre(par_1, par_2, ··· , par_n) {
    //conjunto de instrucciones
}
```

Para ejecutar una función, se ejecutan secuencialmente las instrucciones definidas en su cuerpo. Si se terminan de ejecutar todas las instrucciones de una función, y no se ejecutó la instrucción "return" para alguna expresión, entonces dicha función retorna el valor *undefined*.

Cada ejecución de una función tiene un ambiente, el cual puede ser modificado por las instrucciones del cuerpo de la función. El ambiente de una función es un concepto abstracto, que relaciona variables con valores y representa el valor contenido en una variable en un momento dado. El ambiente de una función, inicialmente, contiene sólo los valores de los argumentos recibidos y el de las variables globales del

"CRS-Engine", las cuales se describen más adelante.

4.2.3 Instrucciones

Cada instrucción dentro del cuerpo de una función puede ser una de las siguientes:

1. La asignación de un valor a una variable, o a una entrada de un *array* o *hash*. La sintaxis para estas asignaciones es:

```
variable = valor;
array[indice] = valor;
hash[clave] = valor;
```

2. La estructura de control "if-else" (el bloque "else" es opcional), cuya sintaxis es:

```
if (condicion) {
    //instrucciones
} else {
    //instrucciones
}
```

3. La estructura de control "for", que permite recorrer las entradas de un *array* o un *hash*. La sintaxis para los dos casos es:

```
for (clave, valor) in hash {
    //instrucciones
}
for (indice, valor) in array {
    //instrucciones
}
```

4. La instrucción "return", que finaliza la ejecución de una función. Su sintaxis es:

```
return expr;
```

5. La instrucción "break", que finaliza la ejecución de un bucle. Su sintaxis es:

```
break;
```

6. La invocación de una función, cuya sintaxis es:

```
funcion(arg_1, ··· , arg_n);
```

Recordemos que en el apéndice A2 se formaliza la sintaxis de estas instrucciones, además de la sintaxis de las distintas expresiones que se pueden formar (al igual que la semántica de instrucciones y expresiones). Aún así, conociendo las instrucciones previamente definidas y observando los distintos ejemplos que se dan a lo largo del informe, se puede obtener una idea general del lenguaje.

4.3 CRS-Engine

Como ya se mencionó, el "CRS-Engine" es un "Engine" del proxy inverso especificado en la sección 4.1, el cual permite definir políticas de seguridad como las del CRS, que filtran pedidos/respuestas, utilizando el lenguaje definido previamente.

El "CRS-Engine" se configura con dos listas de políticas, una de políticas para pedidos, y otra para respuestas. Además, necesita de dos funciones de decisión (una para pedidos y otra para respuestas), una función de configuración y grupos de excepciones.

En cada transacción, el "CRS-Engine" ejecuta primero las políticas para pedidos siguiendo el orden de la lista. Después, ejecuta la función de decisión para los pedidos y, en caso de no bloquear el pedido, una vez recibida la respuesta ejecuta las políticas para las respuestas, terminando con su función de decisión. Dicho flujo se puede ver resumido en la figura 4.2.

En las siguientes secciones describimos cómo se implementan políticas y funciones de decisión para el caso de pedidos, ya que para respuestas es análogo. Luego, comentamos la importancia de la función de configuración (la cual no se observa en la figura 4.2 debido a que no se ejecuta en cada transacción, sino una única vez). Por último, describimos cómo se implementan excepciones para reglas y políticas para pedidos.

4.3.1 Reglas para pedidos

Como se describió en el análisis, una política está compuesta por un conjunto de reglas. Para el "CRS-Engine", una regla para pedidos es una función del lenguaje previamente definido, más información de la regla a la que le denominamos *metadata*. La *metadata* está compuesta por:

- Un identificador único (dentro de una política) que consiste en una cadena de caracteres alfanuméricos, guiones y guiones bajos.
- Una lista de categorías, donde cada categoría consiste en una cadena de caracteres alfanuméricos, guiones, guiones bajos, barras ("/") y puntos.

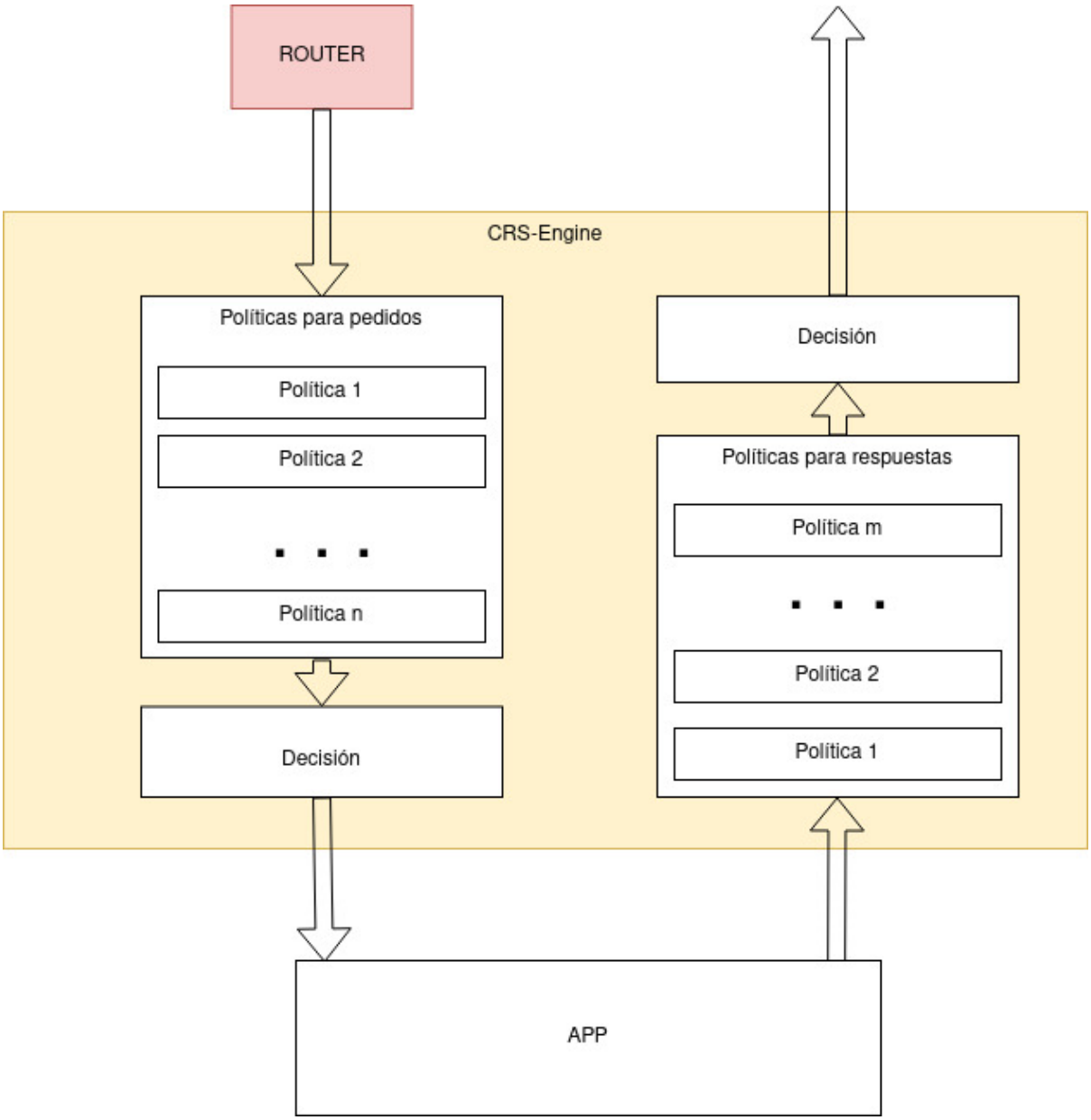

Figura 4.2: El "CRS-Engine" dentro de la arquitectura 4.1

- Una versión que consiste en una cadena de caracteres alfanuméricos, guiones, guiones bajos, barras ("/") y puntos.
- Un nivel de severidad, que puede ser: EMERGENCY, ALERT, CRITICAL, ERROR, WARNING, NOTICE, INFO o DEBUG.

La función de la regla debe ser una función del lenguaje con dos parámetros. Ambos parámetros recibidos serán objetos del lenguaje de tipo *hash*: uno tendrá información del pedido (url, parámetros POST, parámetros GET, el cuerpo, etc), y el otro será utilizado para compartir información de una transacción entre las distintas reglas y políticas.

A continuación, vemos la función de una regla que, en caso de que el método del pedido sea GET, aumenta en uno un puntaje que es almacenado en el segundo parámetro.

```
fn regla(req, tx) {
    if (req["method"] == "GET") {
        tx["puntaje"] = tx["puntaje"] + 1;
    }
}
```

4.3.2 Políticas para pedidos

Por otro lado, una política para pedidos está dada por un conjunto de reglas para pedidos, más información de la política (a la que también le denominamos *metadata*) y una función del lenguaje que permite definir el flujo de ejecución de las reglas. Podemos ver la relación entre una política y una regla en la figura 4.3.

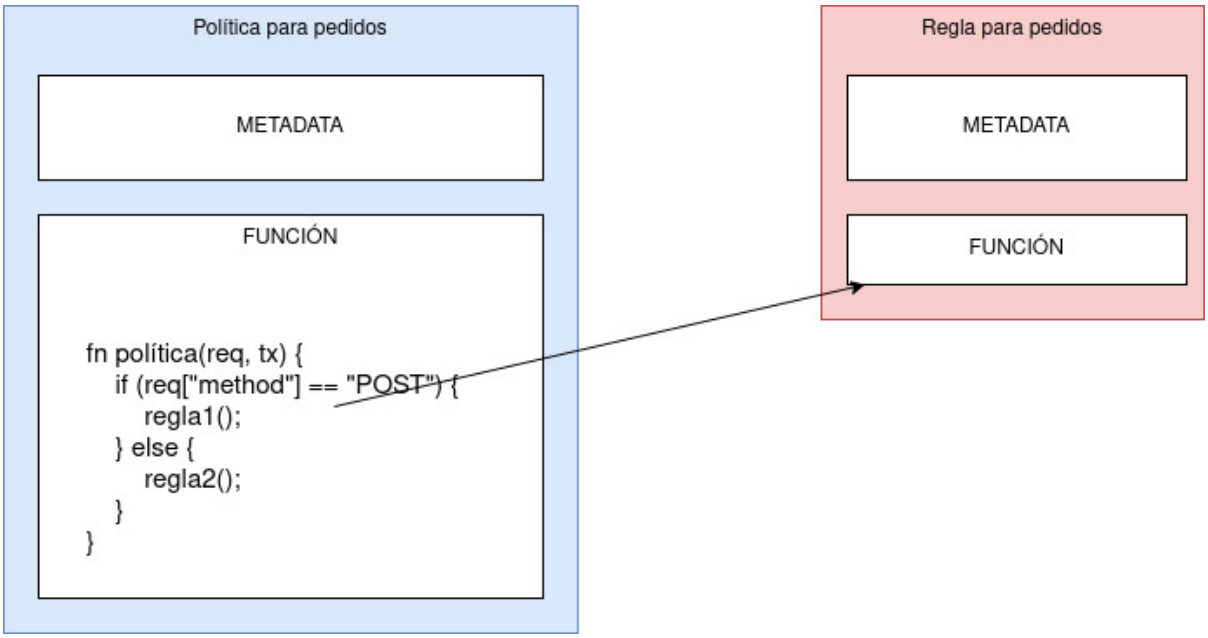

Figura 4.3: Relación entre política y regla

En el caso de las políticas, la *metadata* sólo está compuesta por un identificador único y una versión análogos a los de una regla.

La función de la política también será una función del lenguaje con dos parámetros como los de una regla. A diferencia de las reglas, una política podrá invocar a las funciones de sus reglas, y su función es definir el flujo de ejecución de las reglas de la política.

Al invocar la función de una regla **no** se deben utilizar argumentos. Las reglas siempre reciben los dos argumentos que describimos previamente. El tener que declarar los parámetros en la firma de la función tiene dos objetivos:

1. Permitir modificar los nombres de dichos parámetros. Por ejemplo, para acceder a la información del pedido (recibido en el primer parámetro), se pueden utilizar los identificadores *req*, *request*, *pedido*, o como desee el desarrollador, basta con modificar el nombre de dicho parámetro.
2. Distinguir aquellas variables que son propias de la transacción (recibidas por parámetro), de aquellas que son globales (serán descritas luego).

La función de una regla no tiene por qué retornar ningún valor. Aún así, puede retornar cualquier valor y dicho valor puede ser utilizado por la función de la política que la invocó.

4.3.3 Funciones de decisión

Además de reglas y políticas, tenemos funciones de decisión. En este caso, una función de decisión está compuesta por una versión y una función del lenguaje definido previamente que, al igual que una política o una regla, recibe dos parámetros: un valor de tipo *hash* que representa la información del pedido, y otro con información compartida por las distintas reglas y políticas. Dicha función debe retornar la acción a realizar.

Recordemos que, en este caso, estamos hablando de funciones de decisión para el caso de políticas para **pedidos**. Análoga a esta descripción, tenemos una definición para el caso de **respuestas**, en donde la función de decisión recibe información de la respuesta como parámetro, en lugar de información del pedido.

Además de los tipos básicos del lenguaje (*bool*, *int*, *string*, *array*, *hash*, *IP* y *undefined*), tenemos otro tipo: *action*. *Action* representa una acción a realizar por el "CRS-Engine". La función de decisión debe retornar un valor de este tipo.

Los valores de tipo *action* pueden ser:

- *pass*: Si se retorna dicho valor, no se filtra el pedido. Para obtener un objeto de dicho valor, se puede llamar a la función *pass()* (sin ningún argumento).
- *drop*: Si se retorna dicho valor, no se le envía una respuesta al cliente. Para obtener un objeto de dicho valor, se puede llamar a la función *drop()* (sin ningún argumento).
- *deny*: Un valor de estos es creado utilizando la función *deny(status)*. Si se retorna dicho valor, se le envía una respuesta al cliente con el estado utilizado para

invocar la función *deny*. Por ejemplo, si se retorna el valor retornado por *deny(404)*, el cliente recibirá una respuesta con un error 404.

A continuación, vemos una función de decisión que, en caso de que cierto puntaje acumulado supere el valor diez, retorna un error 404 al cliente. En caso contrario, no interrumpe el procesamiento de la transacción.

```
fn decision(req, tx) {
    if (tx["puntaje"] > 10) {
        return deny(404);
    } else {
        return pass();
    }
}
```

4.3.4 Variables globales

Como se describió anteriormente, las políticas y reglas podrán compartir información de una transacción a través de un objeto del lenguaje que será compartido entre sus funciones. Además de eso, las reglas del CRS necesitan acceder y/o modificar información global.

Esto se cumple en dos casos particulares: acceso a variables de ambiente y acceso a información de un cliente en particular (a través de la colección "IP").

Variables de ambiente.

El "CRS-Engine" cuenta con un objeto global que puede ser accedido por todas las reglas, políticas y funciones de decisión: "env". Por ejemplo, para acceder al nivel de paranoia, bastaría con almacenarlo en una entrada *env["paranoia_level"]*, y luego leerlo desde allí.

Las funciones que describimos previamente no pueden modificar el objeto "env", por lo tanto es necesario contar con una función que se encargue de crear el objeto "env" con los valores de ambiente. Dicha función es la **función de configuración**.

Además de las políticas y las funciones de decisión, mencionamos que el "CRS-Engine" necesitaba una función de configuración. Aún así, en el flujo de ejecución del "CRS-Engine" (que se vio en la figura 4.2) no se encontraba dicha función de configuración. La diferencia con la función de configuración es que dicha función se ejecuta una única vez, al iniciar el WAF, y no para cada transacción.

La función de configuración está compuesta por una versión y una función del

lenguaje que no recibe ningún parámetro, ni debe retornar ningún objeto. La importancia de la función de configuración recae en que es la única que permite modificar la variable global "env". El valor final de "env" al finalizar la ejecución de esta función, será el valor que contendrá dicha variable global durante la ejecución de todas las políticas. A continuación, vemos una función de configuración que almacena el nivel de paranoia en el objeto "env".

```
fn setup() {
    env["paranoia_level"] = 2;
}
```

Información de un cliente.

Por otro lado, el "CRS-Engine" cuenta con otro objeto global que puede ser accedido (y modificado) por las funciones de las reglas, políticas y de decisión: "client". "client" tendrá la misma información para aquellas transacciones con el mismo cliente (identificado por su dirección IP). Además, el lenguaje cuenta con una primitiva "lock_client" la cual es un cierre de exclusión mutua: ninguna de las otras reglas, ejecutando en una transacción con el mismo cliente, podrá acceder o modificar la variable "client" (se bloquearán) mientras se ejecutan las instrucciones dentro de "lock_client". Su sintaxis es muy simple:

```
lock_client {
    //conjunto de instrucciones
}
```

Las funciones descritas anteriormente (las de las reglas, pedidos y de decisión) pueden acceder a las variables globales "env" y "client" a pesar de no recibir dichas variables por parámetro. Por otro lado, claramente la función de configuración no podrá acceder a la variable global "client", debido a que se ejecuta una única vez, previo a comenzar a procesar transacciones.

4.3.5 Excepciones

Como en el análisis, dejamos las excepciones para el final. En el "CRS-Engine" tenemos grupos de excepciones, cada uno compuesto por un conjunto de excepciones e información a la que le denominamos *metadata*.

La *metadata* de un grupo de excepciones está conformada por:

1. Un nombre (que sirve como identificador) que consiste en una cadena de

caracteres alfanuméricos, guiones y guiones bajos.

2. Una versión que consiste en una cadena de caracteres alfanuméricos, guiones, guiones bajos, barras ("/") y puntos.

Por otro lado, cada excepción consiste de información a la que también denominamos *metadata* y de dos funciones del lenguaje, ambas con un único parámetro: la información del pedido. La primera función retorna un valor de tipo *bool*, e indica si se debe aplicar o no la excepción. La segunda retorna un objeto de tipo *hash* con el pedido modificado.

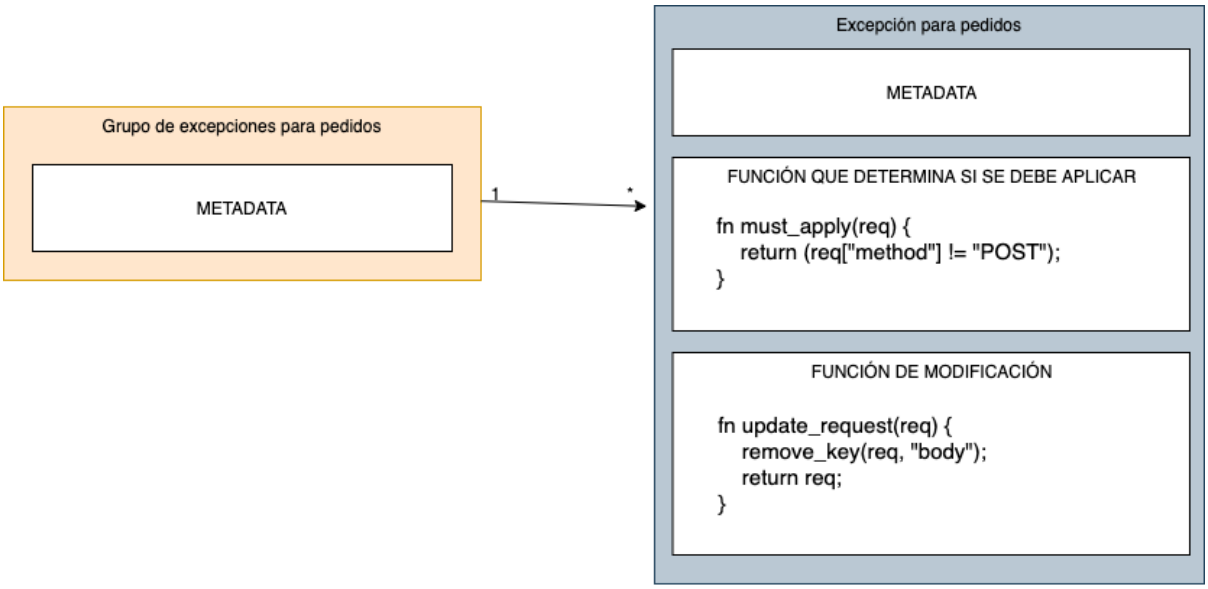

Figura 4.4: Relación entre grupo de excepciones y excepciones

La *metadata* de una excepción consiste en un identificador análogo al nombre de un grupo de excepciones, y único dentro del grupo, y también de una versión análoga a la de un grupo de excepciones. Podemos ver la relación entre grupos de excepciones y excepciones en la figura 4.4.

Recordemos que, en este caso, estamos hablando de excepciones para el caso de políticas y reglas para **pedidos**. Análoga a esta descripción, tenemos una definición para el caso de **respuestas**, en donde las funciones de las excepciones reciben información de la respuesta como parámetro, en lugar de información del pedido.

Las reglas se definen con la siguiente sintaxis, donde la función *must_apply* debe retornar un valor *booleano*, e indica si la excepción se debe aplicar o no; y la función *update_request* debe retornar un valor de tipo *hash*, usualmente siendo una modificación del parámetro recibido.

```
exception Excepcion {
     fn must_apply(req) {
         //instrucciones
     }

     fn update_request(req) {
         //instrucciones
     }
}
```

Dichas excepciones deben estar definidas dentro de un grupo de excepciones, los cuales se definen de la siguiente manera:

```
exceptions Grupo {
     // excepciones
}
```

Excepciones para reglas.

En la definición de cada excepción se indican las reglas a las cuales la misma será aplicada. Por un lado, podemos indicar que una excepción puede bloquear la ejecución de una regla utilizando la siguiente sintaxis:

```
#[exclude(POLICY)(RULE-ID)]
exception Excepcion {
     fn must_apply(req) { ... }
     fn update_request(req) { ... }
}
```

Similarmente, se puede indicar que una excepción puede modificar el pedido de una regla utilizando la siguiente sintaxis:

```
#[modify(POLICY)(RULE-ID)]
exception Excepcion {
     fn must_apply(req) { ... }
     fn update_request(req) { ... }
}
```

Como es de esperar, en caso de que a una regla se le asocien múltiples excepciones, si se cumple una de las excepciones de exclusión, entonces la regla no se ejecuta. De

otro modo, se modifica el pedido recibido por dicha regla, utilizando aquellas excepciones que se deban aplicar (aplicando las excepciones en el orden en el que fueron definidas).

Grupo de excepciones para reglas.

La notación anterior puede ser utilizada en la definición de un grupo de excepciones, haciendo que todas las excepciones del grupo se apliquen a las reglas indicadas. Un ejemplo, sería:

```
#[exclude(POLICY1)(RULE-ID-1)]
#[modify(POLICY2)(RULE-ID-2)]
exceptions Grupo {
     // excepciones
}
```

Excepciones para políticas.

Por último, se puede aplicar una excepción a una política entera. Para esto basta utilizar la sintaxis anterior pero sin especificar el identificador de una regla. En este caso la notación quedaría, por ejemplo, *#[exclude(POLICY)]*.

Incluso, se puede utilizar la sintaxis *#[exclude(ALL)]*, o *#[modify(ALL)]*, para aplicar una excepción a todas las políticas definidas.

4.3.6 Información de un pedido

En las distintas funciones descritas previamente, gran parte de las mismas recibían un parámetro con información del pedido. A continuación, se presenta una tabla con toda la información recibida en dicho parámetro (al que le denominamos **req**).

Notación: variable1 $\rightarrow$ variable2: Indica que "variable1" es de tipo *hash*, con una clave de tipo *string* e igual a "variable2", cuyo contenido es el especificado

Variable	**Contenido**
req → args → combined_size	*int* con la suma del tamaño de todos los argumentos del pedido (en bytes).
req → args → get	*hash* con claves de tipo *string* y valores de tipo *string*, conteniendo el nombre de los argumentos y su valor correspondiente en la cadena de consulta del pedido.
req → args → post	*hash* con claves de tipo *string* y valores de tipo *string*, conteniendo el nombre de los argumentos y su valor correspondiente en el cuerpo del pedido.
req → body	*string* con el cuerpo del pedido.
req → body_processor	Procesador utilizado para el pedido. Puede ser uno de los siguientes *strings*: URLENCODED, MULTIPART, XML o JSON.
req → cookies	*hash* con claves de tipo *string* y valores de tipo *string*, conteniendo el nombre de las cookies y su valor correspondiente.
req → files → combined_size	*int* con la suma del tamaño de todos los archivos que se encuentran en el cuerpo del pedido (en bytes).
req → files → form_names	*array* con objetos de tipo *string* que contiene los nombres de los campos de formulario utilizados para subir los archivos del pedido.
req → files → remote_names	*array* con objetos de tipo *string* que contiene los nombres originales de los archivos del pedido.
req → headers	*hash* con claves de tipo *string* y valores de tipo *string*, conteniendo el nombre de las cabeceras del pedido (en minúscula) y su valor correspondiente.
req → ip	*IP* que representa la dirección IP del cliente.
req → json → attributes	En caso de que el cuerpo tenga formato JSON, es un *array* de *strings* con todos las cadenas que aparezcan como atributos en dicho cuerpo.

req → json → values	En caso de que el cuerpo tenga formato JSON, es un *array* de *strings* con todas las cadenas que aparezcan como valores en dicho cuerpo.
req → method	*string* con el método del pedido en mayúscula.
req → protocol	*string* con el protocolo del pedido.
req → request_line	*string* con la línea del pedido.
req → url → filename	*string* con el nombre del archivo de la URL del pedido.
req → url → path	*string* con la dirección del archivo de la URL del pedido (directorios y el nombre del archivo).
req → url → query	*string* con la cadena de consulta del pedido.
req → url → raw_url	*string* que contiene la URL completa del pedido incluyendo el nombre de dominio y la cadena de consulta sin haber decodificado los caracteres URL-codificados.
req → xml → attributes	En caso de que el cuerpo tenga formato XML, es un *array* de *strings* con todos los atributos de los distintos nodos XML.
req → xml → nodes	En caso de que el cuerpo tenga formato XML, es un *array* de *strings* con todos los nodos del mismo.

4.3.7 Información de una respuesta

Análogamente, presentamos la información del argumento (al que denominaremos **res**) a ser recibido por las funciones de las reglas, políticas, excepciones y de decisión para el caso de las respuestas.

Variable	**Contenido**
res → body	*string* con el cuerpo de la respuesta.
res → status	*int* con el estado de la respuesta.

4.4 Justificación del diseño

Para finalizar, nos centramos en justificar por qué es posible utilizar el "CRS-Engine" para implementar las políticas del CRS. Para esto, describimos las variables de

ModSecurity utilizadas por el CRS, al igual que las acciones, transformaciones y operadores, y explicamos cómo se podría obtener dicha información, u obtener un comportamiento similar, en el nuevo lenguaje.

4.4.1 Variables utilizadas en el CRS

Primero, describimos las variables de ModSecurity utilizadas por las reglas del CRS, y luego cómo obtener dicha información en el nuevo lenguaje. Las variables utilizadas en el CRS, junto a sus respectivas descripciones, se listan a continuación.

Colecciones.

- **ARGS:** Es un mapa que asocia los argumentos recibidos en un pedido (tanto en el cuerpo como en la cadena de consulta del pedido), con el valor recibido. Si el cuerpo del pedido está en formato JSON, los valores de dicho JSON serán agregados a la colección ARGS. Por ejemplo, si el cuerpo es el siguiente:

```
{
   attr1: {
       attr2: value
   }
}
```

  Entonces se agregará el valor *value* en ARGS.

- **ARGS_NAMES:** Es una colección que contiene los nombres de los argumentos recibidos en un pedido. En caso de que el cuerpo tenga formato JSON, para cada valor se agregará una concatenación de sus atributos, separados con un punto. Siguiendo el ejemplo anterior, ARGS_NAMES contendría el valor *attr1.attr2* [10].

- **ARGS_GET:** Similar a ARGS, pero sólo contiene argumentos recibidos en la cadena de consulta de un pedido.

- **ARGS_GET_NAMES:** Es una colección que contiene los nombres de los argumentos recibidos en la cadena de consulta de un pedido.

- **FILES:** Es una colección con los nombres originales de los archivos de un pedido. Disponible para pedidos de tipo *multipart/form-data*.

- **FILES_NAMES:** Es una colección con los nombres de los campos utilizados para subir cada archivo de un pedido. Disponible para pedidos de tipo

multipart/form-data.

- **GEO:** Es un mapa con información de la ubicación de la dirección IP de origen, que es cargado al utilizar el operador "geoLookup". GEO contiene las siguientes entradas:
  - **COUNTRY_CODE:** El código del país en dos caracteres. Ejemplos: US, GB.
  - **COUNTRY_CODE3:** El código del país en hasta tres caracteres.
  - **COUNTRY_NAME:** El nombre completo del país.
  - **COUNTRY_CONTINENT:** El continente del país en dos caracteres. Ejemplo: EU.
  - **REGION:** El nombre de la ciudad en caso de poder obtenerse.
  - **POSTAL_CODE:** El código postal en caso de poder obtenerse.
  - **LATITTUDE:** La latitud en caso de poder obtenerse.
  - **LONGITUDE:** La longitud en caso de poder obtenerse.
  - **DMA_CODE:** El código de área metropolitana en caso de poder obtenerse.
- **IP:** Es un mapa persistente a través de distintas transacciones, se comparte entre aquellas transacciones con la misma dirección IP de origen. Puede ser modificado por las reglas de ModSecurity.
- **MATCHED_VARS:** Es un mapa que contiene el nombre de aquellas variables que cumplieron la condición del operador para una regla, junto con el valor de dicha variable al evaluarse el operador.
- **MATCHED_VARS_NAMES:** Es una colección que contiene el nombre de aquellas variables que cumplieron la condición del operador para una regla.
- **REQUEST_COOKIES:** Es un mapa que asocia los nombres de las cookies de un pedido, con su valor.
- **REQUEST_COOKIES_NAMES:** Es una colección que contiene los nombres de las cookies recibidas en un pedido.
- **REQUEST_HEADERS:** Es un mapa que contiene las cabeceras de un pedido, asociando su nombre con su valor.
- **REQUEST_HEADERS_NAMES:** Es una colección que contiene los nombres de las cabeceras de un pedido.

- **TX:** Es un mapa que es preservado a través de una transacción completa. Puede ser modificado por las reglas de ModSecurity.
- **XML:** Es una colección especial la cual debe ser utilizada junto a una expresión XPath. Se utiliza cuando el cuerpo recibido tiene formato XML. Si se utiliza una variable *XML:xpath* en una regla, la regla es evaluada con los valores del cuerpo extraídos por la expresión *xpath*.

Variables Simples.

- **ARGS_COMBINED_SIZE:** La suma del tamaño de todos los parámetros de un pedido (en bytes).
- **DURATION:** Milisegundos desde el inicio de una transacción.
- **FILES_COMBINED_SIZE:** La suma del tamaño de todos los archivos de un pedido (en bytes).
- **MATCHED_VAR:** Valor de la última variable que haya cumplido la condición del operador para una regla.
- **MATCHED_VAR_NAME:** Nombre de la última variable que haya cumplido la condición del operador para una regla.
- **QUERY_STRING:** Contiene la cadena de consulta extraída de la URL de un pedido.
- **REMOTE_ADDR:** Dirección IP del cliente.
- **REQBODY_PROCESSOR:** Nombre del procesador utilizado para un pedido. Puede ser uno de los siguientes: URLENCODED, MULTIPART, XML.
- **REQUEST_BASENAME:** Sólo contiene el nombre del archivo de REQUEST_FILENAME. Ejemplo: index.php
- **REQUEST_BODY:** Cuerpo de un pedido.
- **REQUEST_FILENAME:** URL relativa de un pedido sin la cadena de consulta. Ejemplo: /folder/index.php
- **REQUEST_LINE:** Línea completa de un pedido.
- **REQUEST_METHOD:** Método de un pedido.
- **REQUEST_PROTOCOL:** Protocolo utilizado por el cliente.
- **REQUEST_URI_RAW:** URI completa de un pedido, incluyendo el nombre de dominio y la cadena de consulta. A diferencia de las demás variables

relacionadas a la URI, los caracteres URL-codificados no son decodificados. Ejemplo: http://www.example.com/folder/index.php?p=X

- **REQUEST_URI:** URI completa de un pedido, incluyendo la cadena de consulta pero no el dominio. Ejemplo: folder/index.php?p=X
- **RESPONSE_BODY:** Cuerpo de una respuesta.
- **RESPONSE_STATUS:** Estado HTTP de una respuesta.
- **UNIQUE_ID:** Identificador único de un pedido.

A continuación, presentamos la lista de colecciones de ModSecurity utilizadas por el CRS y cómo pueden ser obtenidas en el nuevo lenguaje.

Colecciones de ModSecurity	**Implementación en el nuevo lenguaje**
ARGS	Accediendo a req → args → get, req → args → post y a req → json → values
ARGS_NAMES	Accediendo a get_keys(req → args → get), get_keys(req → args → post) y a req → json → attributes
ARGS_GET	req → args → get
ARGS_GET_NAMES	get_keys(req → args → get)
FILES	req → files → remote_names
FILES_NAMES	req → files → form_names
GEO	Simplemente se almacena el valor de la función predefinida "geo_lookup" en una variable.
IP	A través de la variable global "client".
MATCHED_VARS	Esta colección no tendría sentido en el nuevo lenguaje. Es una colección auxiliar para acceder a las variables que cumplieron con la condición del operador de una regla, dentro de la regla misma. Si se quiere evaluar una condición (simulando un operador de ModSecurity) en múltiples variables, se recorren dichas variables (con un "for" por ejemplo), se evalúa la condición con un "if", y dentro del "if" (donde se tomarían las acciones), se puede acceder a la variable que cumplió con la condición del operador.

MATCHED_VARS_NAMES	Análogo a MATCHED_VARS.
REQUEST_COOKIES	req → cookies
REQUEST_COOKIES_NAMES	get_keys(req → cookies)
REQUEST_HEADERS	req → headers
REQUEST_HEADERS_-NAMES	get_keys(req → headers)
TX	En el argumento recibido por las funciones de decisión, reglas y políticas, se puede almacenar información de la transacción.
XML	En el CRS, sólo se utilizan expresiones para obtener todos los atributos o todos los nodos XML del cuerpo. Estos se pueden obtener accediendo a req → xml → attributes y a req → xml → nodes.

Al igual que con las colecciones, presentamos la lista de variables simples de ModSecurity utilizadas por el CRS y cómo pueden ser obtenidas en el nuevo lenguaje.

Variables de ModSecurity	**Implementación en el nuevo lenguaje**
ARGS_COMBINED_SIZE	req → args → combined_size
DURATION	En el CRS se utiliza para crear valores aleatorios. Por lo tanto, no será necesaria debido a que se cuenta con la función predefinida "random".
FILES_COMBINED_SIZE	req → files → combined_size
MATCHED_VAR	Análogo a MATCHED_VARS.
MATCHED_VAR_NAME	Análogo a MATCHED_VARS.
QUERY_STRING	req → url → query
REMOTE_ADDR	req → ip
REQBODY_PROCESSOR	req → body_processor
REQUEST_BASENAME	req → url → filename
REQUEST_BODY	req → body
REQUEST_FILENAME	req → url → path
REQUEST_LINE	req → request_line
REQUEST_METHOD	req → method
REQUEST_PROTOCOL	req → protocol
REQUEST_URI_RAW	req → url → raw_url

REQUEST_URI	Concatenando req → url → path y req → url → query
RESPONSE_BODY	res → body
RESPONSE_STATUS	res → status
UNIQUE_ID	En el CRS se utiliza para crear valores aleatorios. Por lo tanto, no será necesaria debido a que se cuenta con la función predefinida "random".

4.4.2 Operadores utilizados en el CRS

Además de las variables, vimos que otro componente principal de las reglas de ModSecurity son los operadores. En la siguiente tabla, describimos los operadores utilizados por las reglas del CRS, y cómo se implementan dichos operadores en el nuevo lenguaje.

Operador de ModSecurity	**Funcionalidad**	**Implementación en el nuevo lenguaje**
Contains	Permite determinar si una cadena de bytes contiene a otra	Utilizando la función predefinida "contains"
DetectSQLi	Permite detectar ataques de SQLi en una cadena de bytes utilizando la librería **libinjection**	Utilizando la función predefinida "is_sqli"
DetectXSS	Permite detectar ataques de XSS en una cadena de bytes utilizando la librería **libinjection**	Utilizando la función predefinida "is_xss"
EndsWith	Permite determinar si una cadena de bytes termina con otra.	Utilizando la función predefinida "ends_with"
Eq	Permite determinar si un número es igual a otro	Utilizando el operador ==
Ge	Permite determinar si un número es mayor o igual a otro	Utilizando el operador >=

GeoLookup	Busca la dirección IP del cliente en una base de datos con la información de las ubicaciones de distintas direcciones IP. En caso de encontrarla, carga la colección "GEO" con la información descrita previamente.	Utilizando la función predefinida "geo_lookup"
IpMatch	Permite determinar si una dirección IP está en una lista de direcciones IP y subredes dadas	Utilizando la función predefinida "in_subnet" y el operador ==
Lt	Permite determinar si un número es menor a otro	Utilizando el operador <
PM	Permite determinar si una cadena de bytes es igual a alguna de las cadenas de una lista. No distingue entre mayúsculas y minúsculas. Utiliza el algoritmo Aho-Corasick.	Utilizando la función predefinida "aho_corasick"
PmFromFile	Análogo a PM, pero recibe el nombre de un archivo que contiene la lista de cadenas a comparar, separadas por un salto de línea	Utilizando la función predefinida "aho_corasick_from_file"
Rx	Permite determinar si una cadena de bytes contiene una subcadena que coincide con una expresión regular	Utilizando las funciones predefinidas "match_regex" o "capture_regex"
Streq	Permite determinar si dos cadenas de bytes son iguales	Utilizando el operador ==
ValidateByteRange	Permite determinar si los bytes utilizados de una cadena caen fuera de un rango de bytes dado	Utilizando la función predefinida "in_byte_range"

ValidateUrlEncoding	Permite determinar si son válidos los caracteres URL-codificados de una cadena de bytes	Utilizando la función predefinida "validate_-url_encoding"
ValidateUTF8Encoding	Permite determinar si una cadena de bytes es una cadena UTF-8 válida	Utilizando la función predefinida "validate_-utf8_encoding"
Within	Permite determinar si una cadena de bytes contiene alguna de las cadenas de una lista	Utilizando la función predefinida "contains" junto con un bucle que itere sobre la lista

4.4.3 Acciones utilizadas en el CRS.

Para analizar las distintas acciones utilizadas en el CRS, separamos las mismas en distintas categorías: acciones disruptivas, información de una regla, flujos de ejecución, registro de información, manejo de variables y manejo de excepciones.

Acciones disruptivas.

Las acciones disruptivas utilizadas por el CRS son cuatro: *block*, *deny*, *drop* y *pass*. Además, se utiliza la acción *status* para complementar a la acción *deny*. En la siguiente tabla se puede observar la descripción de dichas acciones y su implementación en el nuevo lenguaje.

Acción de ModSecurity	**Descripción**	**Implementación en el nuevo lenguaje**
Block	Ejecuta la acción disruptiva por defecto. La misma se puede modificar con la directiva "SecDefaultAction". En el caso del CRS, coincide con *pass*	No corresponde

Deny	Finaliza la ejecución de las reglas y retorna una respuesta HTTP con el estado dado por la acción *status*	Se utiliza la función predefinida "deny" para retornar un objeto de tipo *action* en la función de decisión
Drop	Finaliza la ejecución de las reglas sin retornar una respuesta al cliente	Se utiliza la función predefinida "drop" para retornar un objeto de tipo *action* en la función de decisión
Pass	No finaliza la ejecución de las reglas	Se utiliza la función predefinida "pass" para retornar un objeto de tipo *action* en la función de decisión
Status	Se utiliza junto con la acción *deny* para asignarle el estado a la respuesta	La función "deny" recibe el estado deseado por parámetro

Información de una regla.

Recordemos que la información de una regla de ModSecurity era indicada en las acciones de la misma. En la tabla que sigue, describimos las acciones utilizadas por el CRS cuya finalidad es brindar información de la regla. Dicha información se corresponde, directamente, con la información que denominamos *metadata* en las reglas del "CRS-Engine".

Acción de ModSecurity	**Descripción**
Id	Asigna un identificador único a una regla
Severity	Asigna un nivel de severidad a la regla. En el CRS se utilizan los siguientes niveles: EMERGENCY, ALERT, CRITICAL, ERROR, WARNING, NOTICE, INFO y DEBUG
Tag	Asigna un "tag" (una categoría) a una regla. Se puede utilizar múltiples veces para asignar múltiples categorías
Ver	Asigna una versión a una regla

Flujos de ejecución.

Más allá de finalizar o no la ejecución de las restantes reglas, hay acciones que, de una u otra forma, influyen en el orden o el flujo de ejecución de las distintas reglas. Presentamos dichas acciones, su descripción y cómo obtener el comportamiento deseado en el nuevo lenguaje, en la siguiente tabla.

Acción de ModSecurity	**Descripción**	**Implementación en el nuevo lenguaje**
Chain	Encadena una regla con la que le sigue. En otras palabras, la regla siguiente sólo se ejecuta si se ejecuta la actual.	Las reglas de ModSecurity encadenadas corresponden a una única regla del "CRS-Engine", con estructuras "if" anidadas en su función
Phase	Le asigna a una regla la fase en la cual será ejecutada.	No corresponde ya que no es necesario definir la fase de una regla en el CRS-Engine.
SkipAfter	En caso de que se ejecute la regla donde se encuentra dicha acción, esta acción permite saltear las siguientes reglas.	Se utilizan estructuras de control en la función de la política para crear flujos de ejecución de sus reglas.
MultiMatch	En caso de que una regla tenga definidas múltiples transformaciones, el operador se evaluará para cada variable luego de aplicar cada transformación (si hay *n* transformaciones, se evaluará *n* veces para cada variable).	Al evaluar una variable, se puede ir modificando dicha variable al aplicar transformaciones, y luego de aplicar cada transformación evaluar el operador.

Registro de información.

El registro de información de ModSecurity se divide en dos archivos: el archivo de registro y el archivo de auditoría. El registro de información se hace, mayoritariamente, en el archivo de registro, que es donde registran información las reglas que así lo deseen.

El registro de auditoría, por otro lado, se puede realizar una única vez por transacción, y registra información de distintas partes de la transacción. Las reglas

pueden decidir qué partes de la transacción registrar, y basta con que una de las reglas decida que se debe auditar dicha transacción, para que esto ocurra (al final de la transacción).

Para el registro de información en el archivo de registro, el CRS utiliza tres acciones: *log*, *logData* y *msg*. La acción *log* indica que la regla debe registrar información (en caso de que se ejecuten las acciones de la regla), la cual está dada por las acciones *logData* y *msg*. La acción *msg* indica el mensaje principal a ser registrado, el cual irá acompañado de la información de la regla (identificador, versión, categorías y severidad), mientras que la acción *logData* se puede utilizar más de una vez, y permite registrar el valor de distintas variables.

La acción *log* implica también que se marque a la transacción como una que debe ser auditada. Aún así, no siempre que se quiere que una regla registre información también se desea auditar la transacción. Por lo tanto, se utiliza la acción *noauditlog* para indicar que la regla no debe utilizarse como criterio para decidir si se debe auditar la transacción.

Para auditar una transacción, las reglas pueden utilizar dos acciones: *auditLog* y *ctl:auditLogParts*. La primera trae como resultado que si se ejecutan las acciones de la regla, entonces la transacción será auditada. La segunda, es utilizada por las reglas para indicar qué partes de una transacción registrar en caso de que se audite la misma (en el CRS, sólo se audita el cuerpo de las respuestas).

En el "CRS-Engine", también se separa el registro de información entre el archivo de registro y el archivo de auditoría. Para el registro de información en el archivo de registro se puede utilizar la función predefinida "log" desde dentro de las reglas, mientras que para auditar una transacción se puede utilizar la función predefinida "audit" desde las funciones de decisión.

La función "log" no sólo registra un mensaje recibido por parámetro, sino que es acompañado por la información de la regla/política/excepción/función de decisión que invoque dicha función. Por otro lado, la función "audit" recibe qué información de la transacción debe ser registrada en el archivo de auditoría.

Manejo de variables.

Para la modificación, creación y eliminación de variables, el CRS utiliza cuatro acciones. En la siguiente tabla se pueden observar dichas acciones, junto a su descripción y su implementación en el nuevo lenguaje.

Acción de ModSecurity	Descripción	Implementación en el nuevo lenguaje
Capture	Almacena el valor de las capturas de una expresión regular sobre una variable. Puede almacenar hasta diez capturas en la colección "TX" (utilizando las entradas "TX.0" hasta "TX.9")	Utilizando la función predefinida "capture_regex"
ExpireVar	Configura un temporizador para eliminar una entrada de la colección "IP" luego de cierto tiempo (en segundos)	Utilizando la función predefinida "expire_client_key"
SetVar	Se utiliza para asignarle un valor o eliminar entradas de las colecciones "IP" y "TX"	Utilizando el operador de asignación y las funciones predefinidas "remove_key" y "remove_entry".
InitCol	Se utiliza en ModSecurity para indicar que se va a utilizar la colección global "IP"	No es necesario

Manejo de excepciones.

Para poder implementar las distintas excepciones, el CRS utiliza las acciones *ctl:ruleRemoveById*, *ctl:ruleRemoveTargetById*, *ctl:ruleRemoveByTag* y *ctl:ruleRemoveTargetByTag*, las cuales sólo son invocadas desde las excepciones del CRS.

La acción *ctl:ruleRemoveById* es utilizada para evitar que otra regla se ejecute. Este comportamiento se puede lograr al agregar la notación *#[exclude(POLICY)(RULE-ID)]* en la excepción correspondiente.

Análogamente, la acción *ctl:ruleRemoveTargetById* es utilizada para eliminar variables momentáneamente al ejecutar una regla. Es invocada, justamente, desde una excepción para no permitirle, a una regla en particular, acceder a información específica del pedido. Esto se puede lograr al utilizar la notación

#[modify(POLICY)(RULE-ID)] en una excepción, y eliminar las entradas correspondientes en la función *update_request*.

Por otro lado, la acción *ctl:ruleRemoveByTag* permite eliminar todas las reglas con una cierta categoría en tiempo de ejecución. Esta acción es utilizada una única vez, para evitar que se ejecuten todas las reglas de una política específica. Para esto, todas las reglas de dicha política tienen una categoría en común y se utiliza la acción en cuestión para evitar su ejecución. Este comportamiento se puede obtener utilizando la notación *#[exclude(POLICY)]*.

Por último, la acción *ctl:ruleRemoveTargetByTag* es utilizada para no permitir que todas las reglas con una cierta categoría accedan a una variable en particular. Esta acción es utilizada únicamente junto a la categoría *OWASP_CRS*. La categoría *OWASP_CRS* es una categoría que tienen todas las reglas del CRS. Por lo tanto, la acción *ctl:ruleRemoveTargetByTag* es utilizada con el fin de que ninguna regla del CRS acceda a ciertas variables específicas del pedido. Este comportamiento puede ser obtenido utilizando la notación *#[modify(ALL)]*.

4.4.4 Transformaciones utilizadas en el CRS

Por último, describimos cómo obtener el comportamiento de las distintas transformaciones de ModSecurity utilizadas por el CRS. En este caso, a cada transformación le corresponde una función análoga en el nuevo lenguaje. Evitamos describir las transformaciones en esta sección debido a que su descripción se corresponde con la definición de la función predefinida asociada (las cuales se pueden observar en el apéndice A2.4). A continuación, se presenta una tabla indicando dicha correspondencia.

Transformación de ModSecurity	**Función en el nuevo lenguaje**
Base64Decode	base64_decode
CmdLine	cmd_line
CompressWhiteSpace	compress_white_space
CssDecode	css_decode
HexEncode	hex_encode
HtmlEntityDecode	html_entity_decode
JsDecode	js_decode
Length	str_length
Lowercase	to_lowercase
NormalizePath	normalize_path
NormalizePathWin	normalize_path_win
RemoveNulls	remove_nulls

ReplaceComments	replace_comments
Sha1	sha1
UrlDecode	url_decode
UrlDecodeUni	url_decode_uni
Utf8toUnicode	utf8_to_unicode

5 | Implementación del WAF

En esta sección, describimos las distintas componentes implementadas. En particular, implementamos dos aplicaciones y una librería, todas ellas utilizando el lenguaje de programación Rust:

- En primer lugar, implementamos un analizador sintáctico al que le denominamos **CRS-Parser**, el cual es capaz de leer las reglas del CRS y convertirlas a estructuras de Rust.
- En segundo lugar, implementamos una librería a la que le denominamos **Rhodium** cuyo fin es proveer facilidades para implementar servidores HTTP en Rust.
- Por último, implementamos una parte del WAF diseñado a lo largo del proyecto, utilizando la librería **Rhodium**.

En las próximas secciones, describimos qué fue implementado en cada uno de estos.

5.1 CRS-Parser

Al comienzo del proyecto, se nos solicitó la creación del CRS-Parser con el fin de obtener un acercamiento a las reglas del CRS. El CRS-Parser lee archivos de ModSecurity, y crea estructuras de Rust con las distintas reglas de dichos archivos, dividiendo entre acciones, variables y operadores.

La ejecución del programa es muy sencilla, el mismo recibe la dirección de un archivo con las reglas e imprime en consola las estructuras creadas para cada regla (por cada regla, una estructura). En el apéndice A4 se pueden observar ejemplos de ejecución.

Se crearon pruebas unitarias para las distintas acciones, operadores y variables, y también se utilizaron todas las reglas del CRS para probar el correcto análisis de las mismas. Como resultado, se obtuvo un cubrimiento de 98.01% del código en dichas

pruebas.

Además, el CRS-Parser nos permitió extraer las distintas expresiones regulares utilizadas en las reglas del CRS (con el operador de expresiones regulares). El CRS-Parser intenta utilizar dichas expresiones regulares para crear una expresión regular en Rust, utilizando la librería "regex" e indica, en la estructura creada, si es posible utilizar dicha cadena para crear una expresión regular o no.

Por ejemplo, veamos la siguiente regla de ModSecurity:

```
SecRule REQUEST_HEADERS:Content-Length "!@rx ^\d+$"
    "id:920160,
    phase:1,
    drop,
    msg:'Content-Length HTTP header is not numeric',
    logdata:'%{MATCHED_VAR}',
    tag:'application-multi',
    ver:'OWASP_CRS/3.3.0',
    severity:'CRITICAL'"
```

Al ser analizada por el CRS-Parser, este nos retorna:

```
SecRule(SecRuleStruct {
    variables: [
        Collection {
            negated: false,
            count: false,
            name: RequestHeaders,
            arg: Some(Simple("Content-Length"))
        }
    ],
    operator: Operator {
        negated: true,
        operator: Rx(Parsed(^\d+$))
    },
    actions: Some([
        Id(920160),
        Phase(Phase1),
        Disruptive(Drop),
        Msg("Content-Length HTTP header is not numeric"),
```

```
        LogData([Macro(Var(MatchedVar))]),
        Tag("application-multi"),
        Ver("OWASP_CRS/3.3.0"),
        Severity(Critical)
    ])
})
```

En este caso, la expresión regular de la regla de ModSecurity es admitida por la librería "regex". El CRS utiliza, en total, 272 expresiones regulares, de las cuales 16 no son aceptadas por la librería "regex" de Rust.

La utilización de la librería "regex", la cual fue utilizada en la implementación del WAF, responde a uno de los requerimientos del nuevo WAF: garantizar tiempos de ejecución lineales al evaluar expresiones regulares.

La librería "regex" de Rust, es una librería cuya sintaxis es del estilo de las expresiones regulares de Perl (al igual que PCRE, cuyo nombre viene de "Perl Compatible Regular Expressions"), pero que no provee funcionalidades de *lookaround* con el fin de asegurar un tiempo de ejecución lineal con respecto al tamaño de la entrada (la entrada consiste en la expresión regular y el texto a evaluar) [11].

Utilizando la librería "regex", no sólo utilizamos la librería estándar de Rust para el trabajo con expresiones regulares, sino que también damos solución a este problema y, como comentamos, al trabajar con las expresiones regulares del estilo de Perl (como ModSecurity), gran parte de las expresiones regulares del CRS son admitidas en esta librería (256 de 272).

De todos modos, las diferencias entre las librerías PCRE y "regex" llevan a que el CRS-Parser deba realizar una pequeña modificación a las expresiones regulares obtenidas de los archivos del CRS. En PCRE, hay múltiples caracteres que pueden ser utilizados escapados (con una barra inversa antes), o sin escapar. Dichos caracteres **no** pueden ser utilizados escapados en "regex" y, por lo tanto, es necesario quitarles la barra inversa. Dichos caracteres son: las comillas dobles ("), el símbolo de porcentaje (%), la barra (/), la coma(,), el símbolo de exclamación (!), los dos puntos (:), el arroba (@), las comillas simples (’), los símbolos de mayor y menor (< y >), y el símbolo de igual (=).

De las 16 expresiones regulares que no son admitidas por la librería "regex":

- Tres utilizan funcionalidades de *lookaround*.
- Una es demasiado grande.
- Dos dan error debido a una pequeña diferencia entre PCRE y "regex": En PCRE,

dentro de una clase de caracteres (ejemplo: *[abc123]*), se puede ingresar el carácter "[". Por otro lado, en "regex" es necesario ingresar dicho carácter con una barra inversa antes. En otras palabras, la expresión *[abc[123]* de PCRE, corresponde a la expresión *[abc\[123]* de "regex".

- Diez no pueden ser convertidas ya que, en la librería "regex", el carácter "{" debe ser utilizado con una barra inversa antes, siempre que no se utilice para ingresar cuantificadores. Los cuantificadores permiten definir expresiones donde se define que una subexpresión se debe repetir una cierta cantidad de veces (por ejemplo, la expresión regular *[abc123]{2}* admite la cadena *a3*, pero no la cadena *a*). En PCRE, el carácter "{" se puede utilizar sin una barra inversa antes, incluso aunque no se esté utilizando para definir un cuantificador.

5.2 Rhodium

Para la implementación del WAF, se utilizó la librería Hyper de Rust. Hyper es una librería que permite trabajar con pedidos y respuestas HTTP a bajo nivel, manipulando directamente dichos pedidos y respuestas. Dicha característica, además de su alta popularidad, fue el principal motivo por el cual utilizamos Hyper en lugar de otras alternativas como H2 o Actix.

En el apéndice A3 se puede encontrar una descripción de algunos proyectos que relevamos previo al comienzo de esta implementación, al igual que la descripción de algunas librerías para el manejo de conexiones HTTP.

Como parte del desarrollo del WAF, se creó una librería llamada Rhodium la cual permite construir servidores, utilizando la librería Hyper, de una manera más ordenada y sencilla. Para utilizar Rhodium, se necesita definir un conjunto ordenado de "Handlers" y un "Service".

A grandes rasgos, un "Handler" es un par de funciones, una que procesa pedidos (recibe un pedido y retorna un pedido), y otra que procesa respuestas (recibe una respuesta y retorna una respuesta). Mientras que un "Service" es una función que recibe un pedido y retorna una respuesta.

Rhodium recibe un conjunto ordenado de "Handlers", un "Service", y una configuración HTTP o HTTPS, y crea un servidor que, al recibir un pedido, retorna la respuesta obtenida al ejecutar el flujo de la figura 5.1.

En caso de querer crear un servidor que trabaje con el protocolo HTTP, basta indicarle la interfaz y el puerto en donde recibir los pedidos. En caso de querer crear un servidor HTTPS, hay que indicarle también las direcciones (locales) del certificado

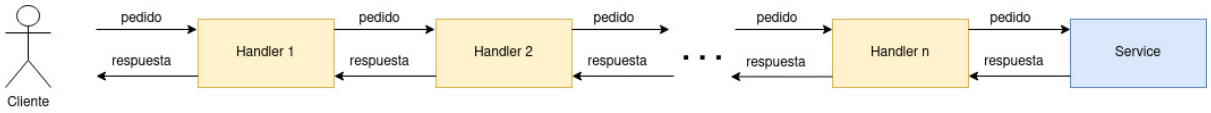

Figura 5.1: Flujo al utilizar Rhodium

y de la clave privada asociada.

Las funciones ejecutadas por los "Handlers" y el "Service" pueden ser funciones asíncronas. El implementar dicha librería, nos permitió separar la implementación del WAF, de los aspectos más técnicos de Hyper.

5.3 WAF

Al igual que en el diseño, dividimos la descripción del WAF implementado en dos partes: empezamos con la descripción de las funcionalidades básicas ofrecidas, y su configuración, y luego continuamos con las funcionalidades específicas de seguridad.

5.3.1 Funcionalidades de proxy inverso

Utilizando la librería Rhodium, implementamos un proxy inverso con la arquitectura descrita en la sección 4.1 (más específicamente, la de la figura 4.1), y luego implementamos el "CRS-Engine" para convertir el proxy inverso en un WAF. Las funcionalidades básicas del WAF como proxy inverso son:

- El WAF permite crear múltiples "HTTP Listeners", especificados en un archivo YAML, a los cuales se les configura: IP y puerto donde escucharán, y el protocolo utilizado para comunicarse con el cliente (puede ser HTTP o HTTPS). En caso de ser HTTPS, se debe especificar un certificado y su clave privada.
- Todos los pedidos recibidos en un "HTTP Listener", y sus correspondientes respuestas, son procesadas por el "Processor". Actualmente, el "Processor" elimina las cabeceras "hop-by-hop" de los pedidos y las respuestas.
- Los pedidos, una vez procesados, son ruteados por el "Router" (decide cuál "Engine" los procesará y cuál aplicación final les corresponde), el cual actualmente tiene la implementación más básica: le asigna a todos los pedidos el único "Engine" existente, y la primera aplicación ingresada.
- Actualmente, existe un único "Engine" (el "CRS-Engine") el cual brinda las funcionalidades de seguridad.

- El WAF permite crear múltiples aplicaciones finales, especificando sus servidores y la estrategia de balanceo de carga (actualmente una sola) en un archivo YAML. Para cada servidor final se debe especificar su host, puerto y protocolo (puede ser HTTP o HTTPS).
- Los pedidos, una vez procesados por el "CRS-Engine", son enviados a uno de los servidores finales de la aplicación seleccionada por el "Router". Para la elección de dicho servidor, se utiliza una estrategia básica de balanceo de carga: el primer pedido recibido va al primer servidor, el segundo pedido al segundo, el tercero al tercero, y así hasta que no haya más servidores, y se empiece de nuevo (estrategia Round-Robin).
- Previo a enviar un pedido al servidor final, se le asignan los valores correspondientes a las cabeceras "Forwarded", "X-Forwarded-For", "X-Forwarded-Host" y "X-Forwarded-Proto" del mismo.

Es importante destacar que la comunicación HTTPS entre el cliente y el WAF (no entre el WAF y el servidor final) se encuentra en fase experimental. Si bien funciona correctamente en los casos en los que debería funcionar, tiene problemas en el manejo de errores. Por ejemplo, si el cliente no confía en el certificado del servidor, y hay un problema en el *handshake* de la comunicación HTTPS, el servidor deja de funcionar. Esto se debe a que Hyper no ofrece funcionalidades de comunicación HTTPS, y se utiliza una librería llamada "tokio-rustls" que aún no se acopla correctamente con Hyper.

5.3.2 Configuración básica

Para iniciar el WAF, se debe crear un archivo YAML que contenga , al menos, dos entradas: *apps* y *listeners*. Presentamos un ejemplo a continuación.

```
log_file: "waf.log"
lang_log_file: "lang.log"
apps:
    - name: example_application
      lb_strategy: RoundRobin
      servers:
        - port: 443
          protocol: https
          url: example_secure.com
        - port: 80
          protocol: http
```

```
          url: example_not_secure.com

listeners:
    - interface: 127.0.0.1
      protocol: https
      cert_file: certs/server.crt
      key_file: certs/server.key
      port: 443
    - interface: 127.0.0.1
      protocol: http
      port: 80
```

En la entrada *apps* se indican las aplicaciones finales. Recordemos que la primera será la designada por el "Router", mientras que el resto no serán tenidas en cuenta. En un futuro se deberían poder configurar políticas de ruteo más complejas.

Cada aplicación consta de tres entradas:

- La entrada *name* con el nombre de la misma, el cual sirve como identificador.
- La entrada *lb_strategy* con la estrategia de balanceo de carga (actualmente sólo está disponible el valor RoundRobin).
- La entrada *servers* con la lista de servidores asociados a dicha aplicación.

Cada servidor consiste en una dirección, un puerto y un protocolo. Por lo tanto, se deben declarar las siguientes entradas:

- **port**: se corresponde con el puerto.
- **protocol**: se corresponde con el protocolo con el que se establecerá la comunicación con el servidor. Su valor puede ser *http* o *https*.
- **url**: se corresponde con la dirección del servidor y su valor puede ser una URL o una dirección IP.

En la entrada *listeners* se declaran las direcciones donde el WAF estará escuchando por nuevos pedidos. Cada *listener* consiste de una dirección, un puerto, un protocolo y, dependiendo del protocolo, puede necesitar de un certificado y una clave. A continuación se detallan las entradas.

- **interface**: Se corresponde con la dirección IP donde se deberá escuchar.
- **port**: Se corresponde con el puerto donde se deberá escuchar.
- **protocol**: Se corresponde con el protocolo que se deberá utilizar para recibir los pedidos. Sus valores pueden ser *http* o *https*. En caso de ser *https*, se deberán

incluir las siguientes dos entradas:

- **cert_file**: Se corresponde con el certificado necesario para utilizar el protocolo HTTPS, su valor debe ser una ruta indicando la dirección de dicho certificado.
- **key_file**: Se corresponde con la clave privada asociada al certificado anterior, su valor debe ser una ruta indicando la dirección de dicha clave.

Observamos que, opcionalmente, se puede ingresar una entrada *log_file* cuyo valor debe ser la dirección de un archivo en donde se registrará información de la ejecución del WAF. Además, también opcionalmente, se puede ingresar en la entrada *lang_log_file* la dirección de un archivo en donde el intérprete del lenguaje registrará información de la ejecución de las distintas funciones de decisión, funciones de las políticas, etc.

5.3.3 Funcionalidades de seguridad

Además de las funcionalidades básicas del WAF, el mismo cuenta con el "Engine" diseñado en la sección 4.3 para brindar funcionalidades de seguridad. En este caso, no se implementó el manejo de excepciones, aunque sí se implementó el manejo de las distintas variables globales: "env" (junto a la función de configuración) y "client" (junto con su cierre de exclusión mutua).

Dentro de la información de un pedido, listamos a continuación las entradas a las cuales se puede acceder a través de las distintas funciones para pedidos:

Variable	**Implementado/No Implementado**
req → args → combined_size	Implementado
req → args → get	Implementado
req → args → post	Implementado
req → body	Implementado
req → body_processor	Implementado
req → cookies	No Implementado
req → files → combined_size	No Implementado
req → files → form_names	No Implementado
req → files → remote_names	No Implementado
req → headers	Implementado
req → ip	Implementado
req → json → attributes	Implementado
req → json → values	Implementado

req → method	Implementado
req → protocol	Implementado
req → request_line	Implementado
req → url → filename	Implementado
req → url → path	Implementado
req → url → query	Implementado
req → url → raw_url	Implementado
req → xml → attributes	No Implementado
req → xml → nodes	No Implementado

Del mismo modo, listamos a continuación las entradas a las cuales se puede acceder dentro de la información de una respuesta, para las distintas funciones para respuestas:

Variable	**Implementado/No Implementado**
res → body	Implementado
res → status	Implementado

Funcionalidades del lenguaje.

El intérprete del lenguaje fue implementado siguiendo la especificación dada en el apéndice A2. Logramos implementar todo lo descrito en dicho apéndice, a excepción de algunas funciones predefinidas.

De las funciones predefinidas, implementamos:

- Todas las funciones predefinidas sobre objetos de tipo *array*: *find*, *length*, *pop*, *push* y *remove_entry*.
- Todas las funciones predefinidas sobre objetos de tipo *hash*: *clear_hash*, *get_keys*, *is_key* y *remove_key*.
- Todas las funciones sobre objetos de tipo *IP*: *in_subnet*, *ip_v4* y *ip_v6*.

Del resto de las categorías de funciones, definidas en A2.4, no se lograron implementar todas las funciones. A continuación, mostramos una tabla con las funciones sobre objetos de tipo *string* que fueron implementadas y aquellas que no.

Función	**Implementado/No Implementado**
aho_corasick	Implementado
aho_corasick_from_file	Implementado
capture_regex	Implementado

contains	Implementado
ends_with	Implementado
in_byte_range	Implementado
is_sqli	Implementado
is_xss	Implementado
match_regex	Implementado
to_uppercase	Implementado
validate_url_encoding	Implementado
validate_utf8_encoding	Implementado

Las funciones *is_sqli* y *is_xss* fueron implementadas parcialmente: se implementaron pero agregando la condición de que la entrada de ambas funciones debe ser una cadena de caracteres UTF-8. Esto es debido a que se utilizó la librería libinjection-rs, la cual recibe como entradas *strings* de Rust, que son cadenas de caracteres UTF-8. La librería libinjection-rs, simplemente invoca a la librería libinjection de C.

En el caso de las transformaciones, la siguiente tabla muestra cuáles funciones fueron implementadas y cuáles no.

Transformación	**Implementado/No Implementado**
base64_decode	Implementado
cmd_line	Implementado
compress_white_space	Implementado
css_decode	No Implementado
hex_encode	Implementado
html_entity_decode	No Implementado
js_decode	No Implementado
normalize_path	Implementado
normalize_path_win	Implementado
remove_nulls	Implementado
replace_comments	Implementado
utf8_to_unicode	No Implementado
sha1	Implementado
str_length	Implementado
to_lowercase	Implementado
url_decode	Implementado
url_decode_uni	No Implementado
utf8_to_unicode	No Implementado

Las funciones *normalize_path* y *normalize_path_win* fueron implementadas parcialmente: se implementaron pero agregando la condición de que la entrada de ambas funciones debe ser una cadena de caracteres UTF-8.

Por último, la siguiente tabla muestra cuáles funciones se implementaron para la categoría "otras funciones".

Función	**Implementado/No Implementado**
audit	No Implementado
deny	Implementado
drop	Implementado
expire_client_key	No Implementado
geo_lookup	No Implementado
load	Implementado
log	No Implementado
pass	Implementado
random	Implementado

Registro de información.

A pesar de no implementarse las funciones predefinidas *audit* y *log*, el lenguaje provee las funciones *file_log* y *file_log_line*. La primera recibe un objeto cualquiera y registra dicho objeto en un archivo, mientras que la segunda realiza lo mismo agregando un salto de línea al final.

Dichas funciones escribirán en el archivo de registro de información del lenguaje, especificado en la entrada *lang_log_file* del archivo de configuración descrito en la sección 5.3.2.

5.3.4 Configuración del "CRS-Engine"

Para configurar las funcionalidades de seguridad es necesario crear un directorio con la configuración del "CRS-Engine". A continuación, vemos la organización de un directorio con una configuración particular:

```
config_dir
├── main.waf
├── policies.yml
├── req_policies
│   ├── policy1.waf
│   └── policy2.waf
└── res_policies
    ├── policy3.waf
    └── policy4.waf
```

El archivo *main.waf* es el único obligatorio y debe contener la función de configuración y las funciones de decisión. El identificador de la función de configuración es *setup*, mientras que el identificador de la función de decisión para pedidos es *request_decision*, y el de la función de decisión para respuestas es *response_decision*. Un ejemplo de un archivo *main.waf* básico es:

```
#[ver(1.0.0)]
fn setup() {
    env["block_all"] = true;
}

#[ver(1.5.0)]
fn request_decision(req, tx) {
    return pass();
}

#[ver(3.0.1)]
fn response_decision(res, tx) {
    if (env["block_all"]) {
        return drop();
    } else {
        return pass();
    }
}
```

Observamos que se pueden ingresar las versiones de dichas funciones utilizando la sintaxis *#[ver(...)]*. Aún así, dicha información no es obligatoria: en caso de no ingresarse una versión, se les asigna por defecto la versión 1.

Además de dicho archivo, si se desean definir políticas se debe crear un archivo *policies.yml*. El archivo *policies.yml* debe ser un archivo YAML con, como máximo, dos entradas:

- **req_policies**: se corresponde con una lista de identificadores de políticas para pedidos, las cuales serán las políticas a ejecutar por el "CRS-Engine".
- **res_policies**: se corresponde con una lista de identificadores de políticas para respuestas, las cuales serán las políticas a ejecutar por el "CRS-Engine".

Un ejemplo de un archivo *policies.yml* básico es:

```
req_policies:
    - policy1
    - policy2
res_policies:
    - policy3
    - policy4
```

En caso de no contar con, por ejemplo, políticas para pedidos, no es necesario ingresar la entrada *req_policies*. Además, el orden de dichas listas es fundamental debido a que es el orden en el que se ejecutarán las políticas.

Cada política consta de un archivo el cual debe tener como nombre el identificador de la política, y la extensión *waf*. En dicho archivo se debe encontrar la función de la política, cuyo identificador será *main*, y distintas funciones que serán las reglas de dicha política.

Los archivos de las políticas para pedidos deben ir en un directorio de nombre **req_policies**, mientras que los de las políticas para respuestas deben ir en uno de nombre **res_policies**.

A continuación, vemos un ejemplo de un archivo de nombre *args_analysis.waf* que corresponde a una política con dos reglas:

```
#[ver(1.0.3)]
fn main(req, tx) {
    if (req["method"] == "POST") {
        sqli_in_post_params();
    } else {
        if (req["method"] == "GET") {
            xss_in_get_params();
        }
    }
}

#[severity(WARNING)]
#[ver(1.0.7)]
fn xss_in_get_params(req, tx) {
    for (arg, value) in req["args"]["get"] {
        if (is_xss(value)) {
            tx["score"] = tx["score"] + 1;
```

```
        }
    }
}

#[severity(WARNING)]
#[tag(tag1,tag2,tag3)]
#[tag(tag4)]
#[tag(tag5)]
fn sqli_in_post_params(req, tx) {
    for (arg, value) in req["args"]["post"] {
        if (is_sqli(value)) {
            tx["score"] = tx["score"] + 1;
        }
    }
}
```

Observamos que, otra vez, podemos definir la versión de la política (sobre la función *main*) y la de las reglas. Al igual que en los casos anteriores, en caso de no ingresarse una versión, se les asigna la versión 1 por defecto.

Además de la versión y su identificador, la *metadata* de una regla contenía también un conjunto de categorías y una severidad. La severidad debe ser asignada obligatoriamente a cada regla, utilizando la sintaxis *#[severity(...)]* (recordemos que los valores posibles de severidad son EMERGENCY, ALERT, CRITICAL, ERROR, WARNING, NOTICE, INFO y DEBUG).

Las categorías de una regla, pueden ser definidas utilizando la sintaxis *#[tag(...)]*. Para ingresar múltiples categorías, se pueden utilizar múltiples entradas de estas, aunque también se pueden ingresar múltiples categorías en una única entrada al separar dichas categorías con una coma.

Todo esto es análogo para las políticas para respuestas (que se encuentran en el directorio *res_policies*).

5.3.5 Ejecución

Para iniciar el WAF, el mismo recibe dos parámetros:

1. La dirección del archivo de configuración descrito en 5.3.2 con la información de aplicaciones, servidores y *listeners*.
2. La dirección del directorio descrito en 5.3.4 utilizado para configurar el "CRS-Engine".

En el apéndice A4 se pueden observar ejemplos de ejecución, además de las dependencias necesarias para ejecutar el WAF.

5.3.6 Pruebas

Tanto para el WAF implementado, como para la librería Rhodium, se crearon pruebas unitarias y pruebas de integración, logrando alcanzar un cubrimiento del 90.95% del código de ambos. Además de esto, se creó una prueba de concepto con el fin de poder mostrar de mejor forma el funcionamiento del WAF en un escenario más real.

5.4 Prueba de Concepto

Luego de concluir con la implementación y con el objetivo de probar el correcto funcionamiento del proyecto, se realizó una prueba de concepto. Esta etapa consistió en poner el WAF en funcionamiento en un ambiente de prueba, definir algunas políticas y enviar una serie de ataques. De esta manera se obtuvieron resultados estadísticos sobre el comportamiento del WAF implementado.

El ambiente de prueba mencionado anteriormente consistió en correr el WAF en Docker, utilizando un servidor final básico (corriendo también en Docker), el cual siempre retorna una respuesta con estado HTTP 200.

GoTestWaf.

Por otro lado, para enviar los ataques se utilizó el proyecto externo "Go Test WAF" [12] desarrollado por la empresa Wallarm y utilizado por la comunidad para este tipo de pruebas. Este proyecto permite crear una aplicación en Docker, y definir una serie de ataques los cuales son enviados al WAF a evaluar, generando luego un reporte estadístico con los resultados obtenidos.

Además de permitir al usuario definir sus propios ataques, el proyecto tiene incorporados una serie de ataques por defecto. Algunos de estos inspirados en OWASP y otros definidos por la comunidad de usuarios del mismo.

Para decidir si un ataque fue bloqueado por el WAF o no, *gotestwaf* se basa únicamente en el estado de la respuesta HTTP recibida para el pedido en cuestión. En este caso, si el estado es 403 se considera que el ataque fue bloqueado y si el estado es 200 se considera que el mismo no fue bloqueado.

Los ataques son agrupados en distintas categorías, como por ejemplo: *ldap-injection* , *mail-injection*, *shell-injection* y *sql-injection*. Los resultados reportados son mostrados

en una tabla (como se ve más adelante), que indica la cantidad de ataques detenidos por el WAF según la categoría, además de los resultados globales.

Además de las pruebas que vienen incorporadas con *gotestwaf*, agregamos dos conjuntos de pruebas adicionales, basándonos en casos de prueba utilizados por el CRS. En primer lugar, creamos ataques para vulnerabilidades de deserialización insegura en Java, debido a que no existían pruebas para esta categoría en *gotestwaf*. Además, agregamos casos de pruebas para ataques de inyección SQL, también utilizados por el CRS.

Al realizarse muchos ataques simultáneamente, muchas veces *gotestwaf* no espera por una respuesta del WAF y decide catalogar dicho ataque como sin resolver. El problema de esto, es que los resultados obtenidos son dependientes de la ejecución. Por lo tanto, modificamos dicho proyecto para que se espere por todas las respuestas del WAF atacado.

Configuración del WAF.

Para esta prueba de concepto, implementamos políticas basándonos en algunas reglas de severidad crítica del CRS. En particular, implementamos reglas de las políticas para ataques de inyección SQL, XSS, RCE y deserialización insegura en Java. De forma similar al CRS, en la función *setup* de la configuración del WAF se puede configurar el nivel de paranoia. En nuestro caso, utilizamos el nivel de paranoia 3.

Además, como nos basamos únicamente en reglas de severidad crítica, el hecho de que cualquier regla detecte un posible ataque, implica que el pedido correspondiente se bloquee. En caso de que se bloquee el pedido, se retorna una respuesta con estado 403, como espera el *gotestwaf*.

El hecho de no haber implementado todas las funcionalidades descritas en el diseño del WAF, no nos permitió ser totalmente fieles a la definición de las reglas que intentamos recrear. Por ejemplo, las reglas recreadas en general son aplicadas a múltiples variables y, entre estas, muchas veces se inspeccionan las cookies del pedido. Esto no puede ser realizado por nuestro WAF debido a que no implementamos el campo *req["cookies"]*. Otras veces, se aplican múltiples transformaciones, las cuales no todas fueron implementadas. En dicho caso, aplicamos aquellas que sí hayan sido implementadas.

A continuación, podemos ver un ejemplo de una regla del CRS y nuestra implementación para la misma en el nuevo lenguaje.

```
SecRule REQUEST_HEADERS|REQUEST_LINE "@rx ^\(\s*\)\s+{"
  "id:932170,
  phase:1,
  block,
  capture,
  t:none,t:urlDecode,
  msg:'...',
  logdata:'...',
  tag:'application-multi',
  tag:'language-shell',
  tag:'platform-unix',
  tag:'attack-rce',
  tag:'paranoia-level/1',
  tag:'OWASP_CRS',
  tag:'capec/1000/152/248/88',
  tag:'PCI/6.5.2',
  ctl:auditLogParts=+E,
  ver:'OWASP_CRS/3.3.0',
  severity:'CRITICAL',
  setvar:'tx.rce_score=+%{tx.critical_anomaly_score}',
  setvar:'tx.anomaly_score_pl1=+%{tx.critical_anomaly_score}'"
```

Nuestra implementación correspondiente fue la siguiente:

```
#[severity(CRITICAL)]
fn rule_932170(req, tx) {
    regex = "^\(\s*\)\s+\{";

    for (arg, value) in req["headers"] {
        if (match_regex(url_decode(value), regex)) {
            return 1;
        }
    }
    if (match_regex(url_decode(req["request_line"]), regex)) {
        return 1;
    }

    return 0;
```

```
}
```

El valor retornado es sumado, desde la función de la política, a la entrada *tx["score"]* la cual es utilizada por la función de decisión para decidir si bloquear o no el pedido. Además, para simplificar esta prueba de concepto, no incluimos las categorías y versiones de las reglas y políticas.

Del CRS, recreamos las siguientes reglas:

Política	**Reglas**
Ataques RCE	932105, 932130, 932160, 932170, 932101, 932200, 932106 y 932190
Ataques XSS	941100
Ataques SQLi	942100, 942160, 942500, 942101 y 942511
Ataques de deserialización insegura en Java	944100, 944110, 944120, 944130, 944200, 944210, 944240, 944250 y 944300

Resultados.

En la siguiente tabla se muestran los resultados obtenidos al evaluar el WAF en el ambiente descrito previamente. Los casos de prueba del conjunto "owasp-regression", son los que fueron agregados por nosotros (siendo extraídos del conjunto de pruebas del CRS). Omitimos mostrar aquellos casos de prueba para los cuales no se crearon reglas. Aún así, incluyendo dichos casos, el porcentaje de ataques detenidos fue del 64.34%.

Conjunto de Prueba	**Caso de prueba**	**Porcentaje**	**Bloqueados**	**No bloqueados**
community	community-rce	92.86	39	3
community	community-sqli	77.08	37	11
community	community-xss	83.88	255	49
owasp	sql-injection	25.00	8	24
owasp	xss-scripting	21.43	6	22
owasp-regression	java	100.00	16	0
owasp-regression	sqli	100.00	10	0

Tabla 5.1: Resultados de la evaluación del WAF implementado.

Además de los ataques, el *gotestwaf* envía un conjunto de pedidos que no se corresponden con ningún ataque, con el objetivo de verificar la cantidad de falsos

positivos obtenidos por el WAF. En este caso, se obtuvo un único falso positivo de ocho posibles.

De la misma manera, a continuación presentamos los resultados de utilizar ModSecurity en el mismo ambiente, con los mismos casos de prueba y nivel de paranoia (3), y configurado **sólo con las reglas que nosotros recreamos para nuestras políticas**.

Conjunto de Prueba	Caso de prueba	Porcentaje	Bloqueados	No bloqueados
community	community-rce	92.86	39	3
community	community-sqli	77.08	37	11
community	community-xss	83.55	254	50
owasp	sql-injection	25.00	8	24
owasp	xss-scripting	21.43	6	22
owasp-regression	java	100.00	16	0
owasp-regression	sqli	100.00	10	0

Tabla 5.2: Resultados de la evaluación de ModSecurity.

ModSecurity detectó un único falso positivo, y detuvo el 64.51% del total de ataques.

En los resultados mostrados se puede ver que el WAF implementado por nosotros detuvo un ataque más que ModSecurity en la categoría *community-xss*. Esta diferencia puede ser debido a que al realizar más transformaciones sobre los ataques, ModSecurity puede no haber reconocido un ataque que sí fue reconocido por el WAF implementado.

6 | Conclusiones y trabajo a futuro

A pesar de ser ampliamente utilizados, la documentación del WAF ModSecurity y la del conjunto de reglas del CRS no es una documentación rigurosa. En ese sentido, creemos que el informe actual genera un aporte con respecto al entendimiento de la organización del CRS y de algunas características de ModSecurity, como el manejo de concurrencia o la evaluación de colecciones, que son omitidas por la literatura sobre el tema.

De todos modos, el principal aporte de este trabajo es el diseño de un WAF moderno, con un lenguaje más apropiado para la definición de políticas de seguridad, capaz de soportar las distintas reglas del CRS. Además, su diseño es acompañado de una documentación, especialmente sobre el nuevo lenguaje, más exhaustiva.

En cuanto al desarrollo de dicho WAF, creemos que hemos hecho un aporte significativo en su implementación. La utilización del lenguaje de programación Rust, si bien implica una gran complejidad debido a su dificultad, es de gran ayuda a la hora de construir aplicaciones de este tipo. Esto se debe a que brinda soluciones a distintos problemas de memoria, o manejo de hilos, que dan lugar a vulnerabilidades.

Por otro lado, se encuentran pocos proxys inversos de código abierto implementados en Rust, mucho menos proyectos del tamaño del WAF implementado en este trabajo. Por lo tanto, creemos que es un aporte importante en un área que está creciendo dentro de la comunidad de desarrolladores de Rust. Además, la implementación de la librería Rhodium es un aporte importante para el desarrollo de servidores HTTP.

En un futuro, además de la finalización de la implementación del WAF diseñado a lo largo del informe, creemos que se podrían agregar algunas funcionalidades que le darían otro valor a dicho WAF:

- En primer lugar, el lenguaje definido puede ser expandido para soportar aún más instrucciones (por ejemplo, la estructura "while"), operadores, funciones predefinidas, nuevos tipos e incluso incorporar manejo de excepciones.

Además, la optimización del intérprete del lenguaje es fundamental para la eficiencia del WAF. El área de desarrollo de intérpretes, es un área sumamente explorada y en la que se pueden encontrar ideas para la optimización del WAF.

- Otro aporte interesante, sería la posibilidad de poder incorporar librerías de funciones del lenguaje, dando lugar a la implementación de funciones básicas que luego puedan ser invocadas a través de las reglas, políticas, etc. Esto facilitaría el trabajo de los desarrolladores de políticas debido a que podrían utilizar librerías desarrolladas por terceros para implementar sus propias políticas. Más aún, permitiría evitar el código duplicado entre distintas reglas que tienen características similares, lo cual es fundamental para el mantenimiento de un conjunto de políticas.
- Sería interesante también explorar la utilización del CRS-Parser para la conversión automática de las reglas del CRS en el lenguaje de ModSecurity, al lenguaje actual.

Además de lo anterior, con respecto a lo hecho actualmente, creemos que se deberían mejorar los mensajes de error en los casos en que se ingresa una función del lenguaje con una sintaxis incorrecta. Otro aspecto interesante, sería aprovechar la función de *setup* con el fin de asignarle un valor inicial a la variable global *client*. De modo que cuando un cliente realice una primera transacción, el valor de la variable *client* pueda comenzar siendo distinto a un *hash* vacío. Del mismo modo, se podría realizar algo similar con el *hash* que se utiliza para almacenar información de toda la transacción (el que es recibido como segundo parámetro en las distintas funciones).

Continuando con aspectos más técnicos, en la implementación del WAF existen invocaciones a la librería **libinjection**, la cual es implementada en C++, con el fin de obtener ciertas funcionalidades específicas como reconocer un ataque de tipo inyección SQL. Es recomendable que en un futuro se re-implementen estas funcionalidades en Rust, para obtener todos los beneficios que conlleva utilizar este lenguaje de programación.

Por último, creemos que para continuar este proyecto sería fundamental comenzar a utilizar el WAF en ambientes más reales, y exponerlo a la opinión de posibles usuarios del mismo. Entendemos que es sumamente importante encontrar un equilibrio entre ofrecer un lenguaje que permita recrear el CRS de una manera adecuada, pero que al mismo tiempo no limite la utilización del mismo con otros fines. En ese sentido, sería enriquecedor poder recibir otros puntos de vista.

Referencias

[1] *Código fuente del proyecto*. URL: https://gitlab.fing.edu.uy/gsi/wafnextgen.

[2] Christian Folini and Ivan Ristić. *ModSecurity Handbook*. Feisty Duck, 2017. ISBN: 978-1-907117-07-7.

[3] *Reference Manual (v2.x) - SpiderLabs/ModSecurity*. URL: https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual-(v2.x) Visitado: 04/22/2020.

[4] *About OWASP*. URL: https://owasp.org/about Visitado: 05/01/2020.

[5] *OWASP ModSecurity Core Rule Set*. URL: https://owasp.org/www-project-modsecurity-core-rule-set Visitado: 05/01/2020.

[6] *Azure Application Gateway - CRS*. URL: https://docs.microsoft.com/en-us/azure/web-application-firewall/ag/application-gateway-crs-rulegroups-rules?tabs=owasp31 Visitado: 03/03/2021.

[7] *Google Cloud Armor - CRS*. URL: https://cloud.google.com/armor/docs/rule-tuning#preconfigured_rules Visitado: 03/03/2021.

[8] *Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...)* URL: https://swtch.com/~rsc/regexp/regexp1.html Visitado: 11/10/2020.

[9] *Regex Tutorial - Lookahead and Lookbehind Zero-Length Assertions*. URL: https://regular-expressions.mobi/lookaround.html?wlr=1 Visitado: 06/10/2020.

[10] *ModSecurity Advanced Topic of the Week: JSON Support*. URL: https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/modsecurity-advanced-topic-of-the-week-json-support/ Visitado: 03/23/2021.

[11] *regex - Rust*. URL: https://docs.rs/regex/1.4.3/regex/ Visitado: 11/10/2020.

[12] *Go Test WAF*. URL: https://github.com/wallarm/gotestwaf Visitado: 03/13/2021.

[13] *Traefik*. URL: https://docs.traefik.io Visitado: 05/23/2020.

[14] *EntryPoints - Traefik*. URL: https://doc.traefik.io/traefik/routing/entrypoints/ Visitado: 03/06/2021.

[15] *Routers - Traefik*. URL: https://doc.traefik.io/traefik/routing/routers/ Visitado: 03/06/2021.

[16] *nginx documentation*. URL: http://nginx.org/en/docs Visitado: 05/25/2020.

[17] *The Architecture of Open Source Applications (Volume 2): nginx*. URL: https://www.aosabook.org/en/nginx.html Visitado: 05/25/2020.

[18] *The Architecture of Open Source Applications (Volume 2): nginx*. URL: https://aosabook.org/en/nginx.html Visitado: 03/06/2021.

[19] *Architecture overview - envoy 1.15.0-dev-5c5532 documentation*. URL: https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/arch_overview Visitado: 05/27/2020.

[20] *HAProxy Starter Guide*. URL: https://www.haproxy.org/download/2.2/doc/intro.txt Visitado: 06/01/2020.

A1 | Relevamiento de proxys inversos

Con el fin de diseñar la arquitectura del WAF, decidimos estudiar distintas arquitecturas de productos utilizados a gran escala. Relevamos, principalmente, distintas alternativas de proxys inversos y obtuvimos una aproximación a las arquitecturas de estos. En este apéndice se describen las arquitecturas de los cuatro proxys inversos relevados: Traefik, Nginx, Envoy y HAProxy.

A1.1 Traefik

Traefik [13] es un proxy inverso de alto nivel, cuyo principal objetivo es permitir publicar servicios de manera sencilla. Una de sus principales características es que permite descubrir automáticamente los servicios a publicar, re-configurándose a sí mismo en caso de haber algún cambio en los mismos.

Las principales componentes de Traefik son:

- **EntryPoint**: La función más importante de este módulo es recibir pedidos, los cuales serán reenviados al módulo "Router", tal como se puede apreciar en la figura A1.1. Cada "entrypoint" estará escuchando en un puerto determinado, con un protocolo determinado.

  Es importante mencionar que la configuración de estos "entrypoints" es estática, es decir, no varía a lo largo de la ejecución del programa.

- **Router**: Este módulo cumple algunas de las funciones más importantes dentro del proxy. Luego de recibir un pedido reenviado por un "entrypoint", el módulo "Router" procesará el mismo mediante el uso de "middlewares" y lo reenviará a un "Service".

  Cada "Router" tendrá cierta prioridad -un entero mayor a 0- y un conjunto de reglas para decidir si le corresponde o no hacerse cargo de un pedido. Si una de sus reglas es evaluada positivamente, dicho "Router" podrá encargarse del pedido. En caso de que más de un "Router" pueda hacerse cargo de un pedido, se ejecutará el de mayor prioridad. En caso de empate se ejecutará el "Router"

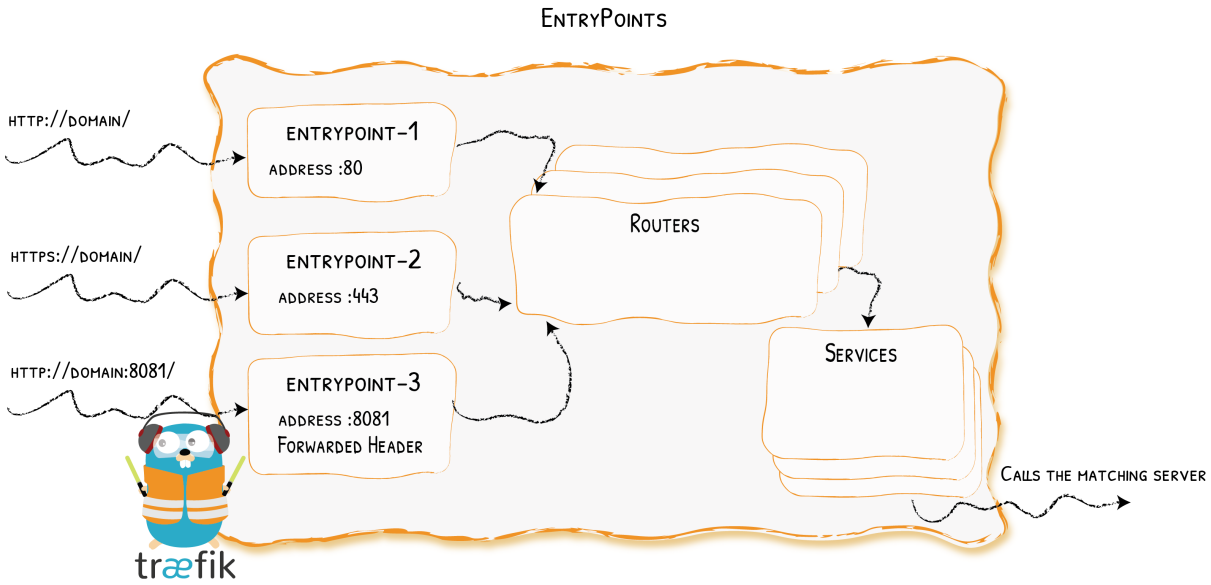

Figura A1.1: Entrada y ruteo de pedidos [14]

que fue declarado primero.

Como se puede apreciar en la figura A1.2, los "middlewares" actúan dentro del módulo "Router" y se ejecutan sobre un pedido, antes de reenviar el mismo a un "Service". Dichos "middlewares" se ejecutarán en cadena, en el orden en que fueron declarados. Pueden modificar diferentes partes del pedido en cuestión, como por ejemplo las cabeceras o también pueden ocuparse de redirigirlo.

Una vez procesado el pedido, se reenvía el mismo a un "Service".

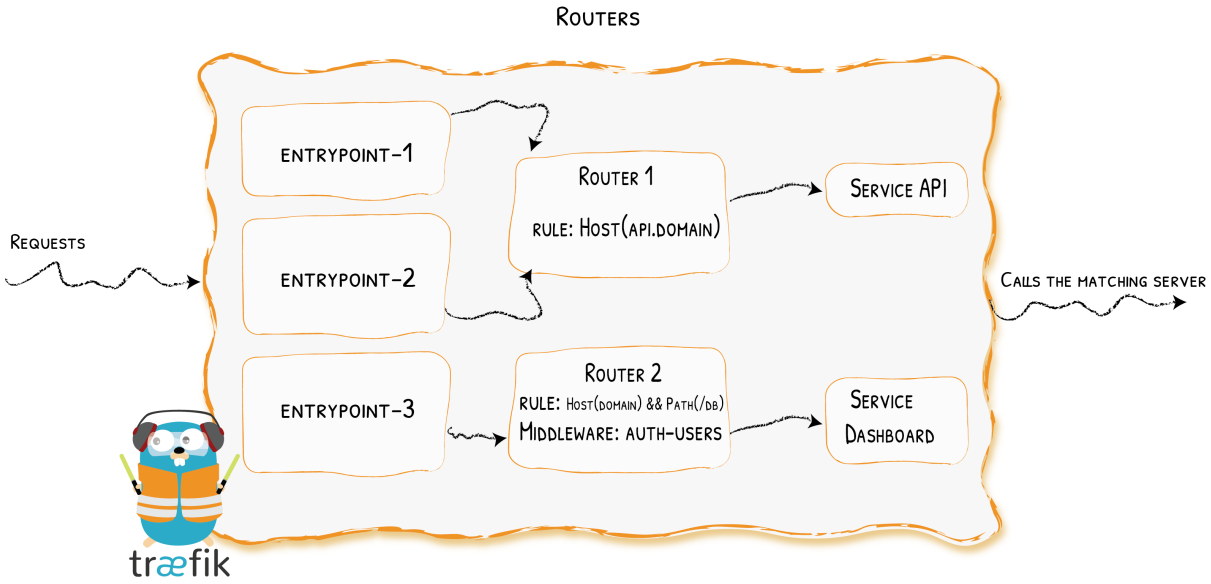

Figura A1.2: Procesamiento y ruteo de pedidos [15]

- **Service**: Este módulo contiene la información sobre dónde se encuentran los servidores que pueden responder el pedido. Por lo tanto, se encarga de reenviar el pedido a uno de estos servidores. En caso de que haya más de un servidor, este módulo se encargará de balancear la carga entre los mismos.
- **Provider**: La principal función de un "provider" es configurar dinámicamente los componentes mencionados anteriormente, con excepción de los "entrypoints".

  Los mismos contienen información sobre los servicios que se encuentran disponibles para usarse. De esta manera, cada cierto tiempo, si se detecta algún cambio en estos servicios, este módulo reconfigura el sistema para que pueda seguir con su correcto funcionamiento.

A1.2 Nginx

El segundo proxy inverso relevado fue Nginx. Nginx [16] es un servidor web y proxy inverso de código abierto. Una de sus principales características es su gran rendimiento, siendo uno de los más rápidos -sino el más rápido- en esta área.

Este proxy inverso tiene una arquitectura muy modularizada [17]. La arquitectura, a grandes rasgos, se puede apreciar en la figura A1.3.

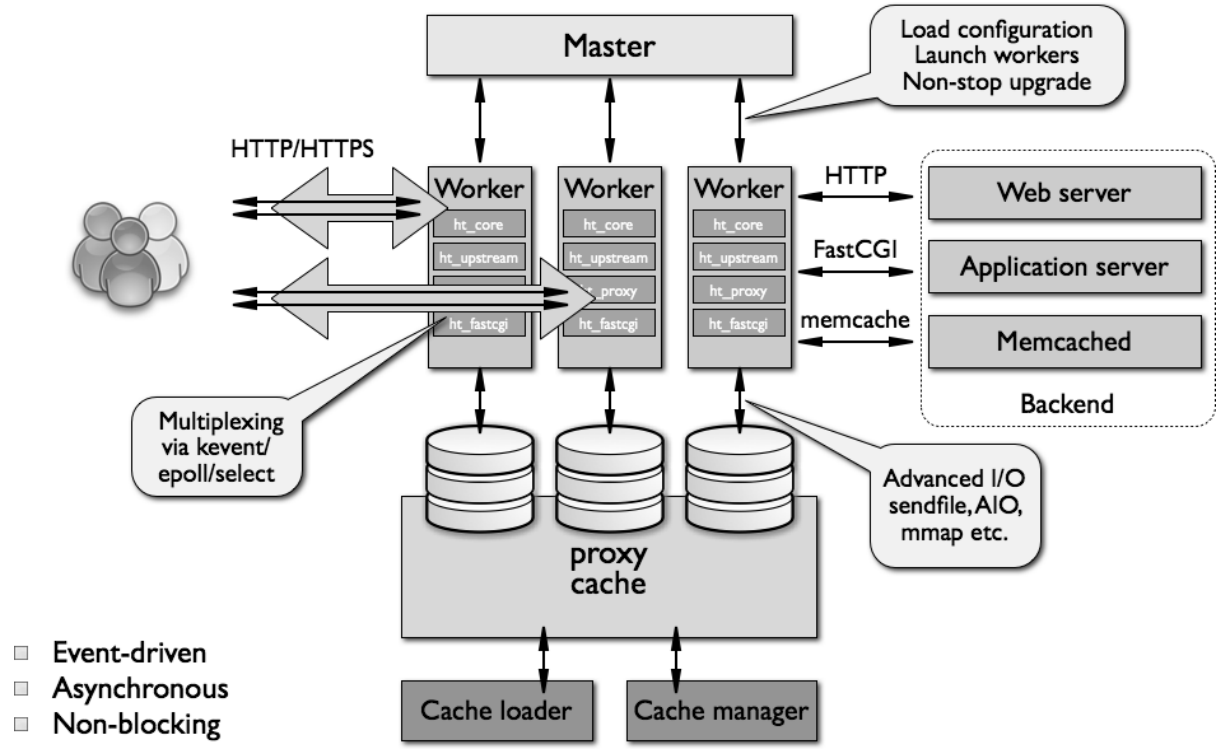

Figura A1.3: Diagrama de arquitectura de Nginx [18]

En la misma se puede apreciar como los "workers" atienden los pedidos que llegan al servidor, al tiempo que se comunican con el caché o con los servidores finales, para devolver una respuesta.

Internamente, Nginx procesa cada conexión a través de una cadena de módulos. En otras palabras, para cada operación hay un módulo que se encarga de realizarla (ej: compresión, modificación de contenido, comunicación con los servidores donde está ejecutando la aplicación, etc). En particular, Nginx cuenta con un conjunto de módulos considerados el "core". Estos módulos son los siguientes:

- **Master**: Este es el módulo encargado de leer el archivo de configuración y crear los "workers" necesarios, así como reconfigurar el servidor automáticamente. También se encarga de crear sockets para la comunicación con el cliente.
- **Workers**: Aceptan y se encargan de procesar las conexiones con los clientes. Es el módulo encargado de conectarse con los servidores finales, ya sean locales o remotos, cumpliendo así con la función de proxy inverso.

  Cada worker ejecuta un bucle no bloqueante donde recibe pedidos y retorna respuestas.
- **Cache Loader**: Es responsable de chequear el caché del disco y llenar la memoria de Nginx con información sobre el caché.
- **Cache Manager**: Es responsable de invalidar el caché cuando lo considere necesario.

Hay un par de módulos de Nginx que se sitúan entre el "core" y el resto de los módulos (llamados módulos funcionales). Estos módulos son el módulo "http" y el módulo "email". Estos dos módulos proveen una capa de abstracción adicional con respecto al "core" de Nginx. En estos módulos, se implementa el manejo de eventos asociados a sus respectivos protocolos de capa de aplicación (HTTP, SMTP, IMAP por ejemplo). Conjuntamente con el "core" de Nginx, estos módulos son responsables de mantener correcto el orden de llamadas a los respectivos módulos funcionales.

A1.3 Envoy

Luego de observar la arquitectura de Nginx, continuamos con la de Envoy. Envoy [19] es un proxy diseñado para grandes arquitecturas orientadas a servicios, desde un punto de vista técnico, es un proceso autocontenido que puede ejecutarse junto con cualquier servidor de aplicaciones. Un conjunto de instancias de Envoy forman una red de comunicación transparente, donde cada aplicación se comunica sin necesidad de conocer la topología de la red, ni si hay otros servidores ejecutando una

instancia de Envoy. Esta arquitectura permite que los servidores puedan estar ejecutando una aplicación en cualquier lenguaje (es muy común en arquitecturas orientadas a servicios utilizar múltiples frameworks y lenguajes).

El núcleo de Envoy es un proxy a nivel de red, lo que permite poder utilizar a Envoy como un proxy inverso de conexiones TCP. Sobre este núcleo, Envoy implementa un filtro que permite tomar decisiones sobre conexiones HTTP.

En la figura A1.4 vemos un diagrama de cómo Envoy maneja cada conexión.

Una misma instancia de Envoy cuenta con uno o más "TCP Listeners", encargados de recibir los pedidos en un puerto e interfaz fija. Además, pueden ser configurados dinámicamente o estáticamente. Cada "TCP Listener" puede tener, opcionalmente, una lista de "Listener filters", que son filtros para los "TCP Listener", y debe tener una o más "Filter chains". Una vez que un "TCP Listener" recibe una conexión, a la misma se le asocia una "Filter Chain".

Cada "Filter Chain" es una pila de "Network Filters", los cuales pueden ser de tres tipos: Read, Write y Read/Write, los de tipo Read procesan segmentos TCP enviados por el cliente, los de tipo Write procesan segmentos TCP enviados por el servidor, y los Read/Write procesan segmentos TCP en ambos casos.

Cuando una conexión recibe un segmento TCP, dicho segmento es procesado primero por los "Listener filters" asociados al "TCP Listener", y luego por cada uno de los "Network Filters" (de tipo Read) de la "Filter Chain" designada a la conexión. Dichos filters pueden modificar el segmento TCP, o incluso bloquearlo.

Dos ejemplos particulares de "Network filters" son TCP Proxy y HTTP Connector Manager. El filtro TCP Proxy se encarga de retransmitir los segmentos recibidos a otro servidor, dependiendo de la configuración del filtro. El filtro HTTP Connector Manager, permite gestionar conexiones HTTP y aplicar filtros a nivel de dicho protocolo. Ambos son de tipo Read/Write.

A su vez, el HTTP Connector Manager cuenta con subfiltros dentro, conocidos como "HTTP Filters". Al igual que los "Network Filters", los "HTTP Filters" se clasifican en 3 tipos: Decoders, encoders, decoder/encoder (definiciones análogas a las de read, write, read/write).

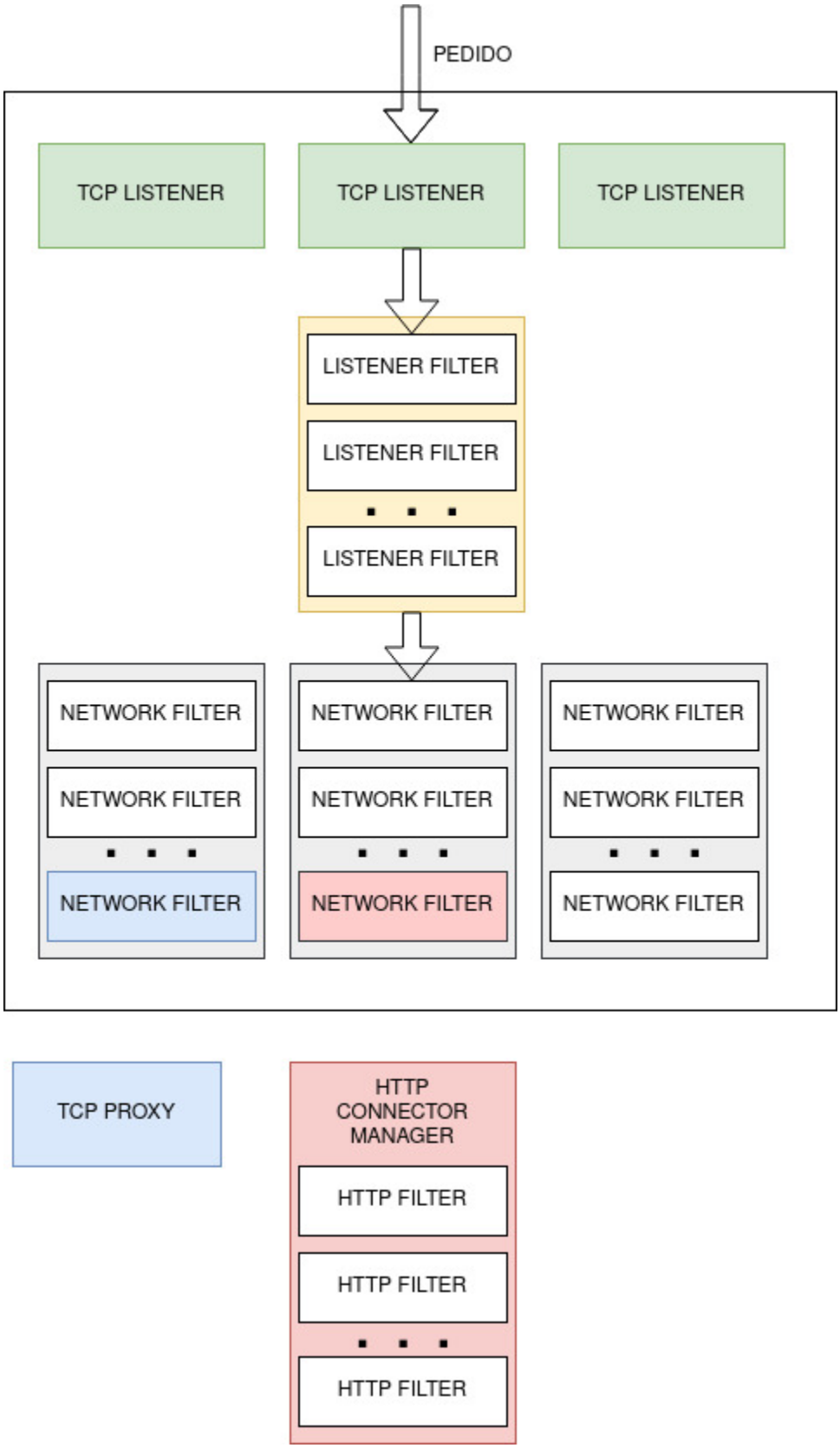

Figura A1.4: Flujo del procesamiento de un pedido en Envoy

A1.4 HAProxy

Por último, relevamos la arquitectura del balanceador de carga HAProxy. HAProxy [20] es un balanceador de carga TCP/HTTP. Actualmente es el más utilizado dentro de las alternativas de código abierto. En la figura A1.5 podemos ver un bosquejo de los distintos componentes de HAProxy que procesan una conexión.

El flujo es bastante sencillo y es el mismo tanto para conexiones TCP como para conexiones HTTP. HAProxy acepta conexiones entrantes a distintos sockets que pertenecen a una componente denominada "frontend", la cual puede escuchar en más de una dirección.

Una vez recibido un segmento TCP o un pedido HTTP desde el cliente, se le aplica las reglas específicas del "frontend" que pueden tener como resultado que se bloquee el segmento/pedido, se modifique y/o se utilice para algún procesamiento interno como puede ser la obtención de estadísticas.

Una vez procesadas las reglas del "frontend", se pasa el mensaje a otra componente conocida como "backend", la cual contiene una lista de servidores y la estrategia de balanceo de carga para los servidores que ejecutan detrás de HAProxy. Si ningún filtro bloquea el mensaje, se decide a qué servidor hay que enviarlo de acuerdo a la estrategia de balanceo de carga y luego se envía.

Para la respuesta, se ejecutan primero las reglas específicas del "backend", y luego las del "frontend". En el caso de conexiones HTTP, queda esperando un nuevo pedido, pero de otro modo cierra la conexión.

Algunas funcionalidades específicas de HAProxy son:

- Posibilidad de escuchar en múltiples direcciones IP y/o múltiples puertos (incluso en un rango de puertos dado).
- El puerto del servidor no tiene por qué ser el mismo que el puerto donde escucha HAProxy.
- Compresión de respuestas HTTP que no fueron comprimidas previamente por el servidor.
- "Health Checks" con el fin de descubrir servidores que no estén respondiendo.

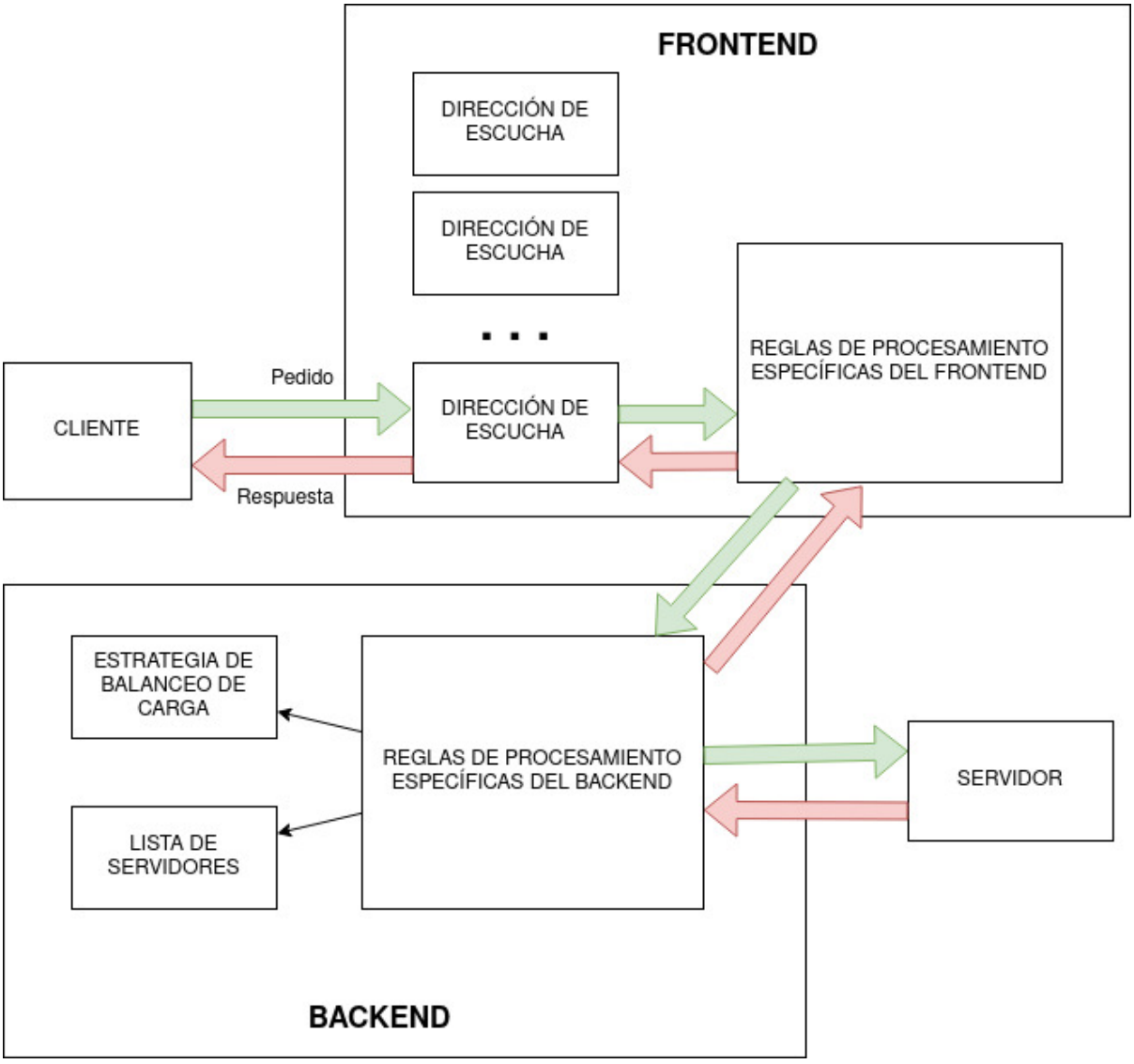

Figura A1.5: Flujo de un pedido y su respuesta en HAProxy

A2 | Especificación del lenguaje nuevo

En este apéndice, describimos formalmente el lenguaje introducido en la sección 4.2, que será utilizado por el WAF implementado para definir reglas y políticas. Comenzamos describiendo su sintaxis, y luego su semántica: empezando por la ejecución de instrucciones y finalizando con el cómputo de expresiones. Por último, describimos las funciones predefinidas del lenguaje.

Es importante, antes de leer este apéndice, haber leído la sección 4.2 en donde se describen conceptos generales del lenguaje, como los tipos del mismo.

A2.1 Sintaxis

La construcción más compleja del lenguaje es una función, la sintaxis de una función queda definida por el BNF dado a continuación.

⟨*function*⟩ ::= '`fn `' ⟨*ident*⟩ '`() {`' ⟨*stat-list*⟩ '`}`'
| '`fn `' ⟨*ident*⟩ '`(`' ⟨*ident-list*⟩ '`) {`' ⟨*stat-list*⟩ '`}`'

⟨*ident-list*⟩ ::= ⟨*ident*⟩ '`,`' ⟨*ident-list*⟩ | ⟨*ident*⟩

⟨*stat-list*⟩ ::= ⟨*statement*⟩ ⟨*stat-list*⟩ | ⟨*empty*⟩

⟨*statement*⟩ ::= ⟨*function-invocation*⟩ '`;`'
| ⟨*return-stat*⟩
| ⟨*if-else-stat*⟩
| ⟨*loop*⟩
| ⟨*assignment*⟩
| '`break;`'
| ⟨*lock*⟩
| ⟨*comment*⟩

⟨*function-invocation*⟩ ::= ⟨*ident*⟩ '`( )`'
| ⟨*ident*⟩ '`(`' ⟨*expr-list*⟩ '`)`'

⟨*expr-list*⟩ ::= ⟨*expr*⟩ ‘`,`’ ⟨*expr-list*⟩ | ⟨*expr*⟩

⟨*return-stat*⟩ ::= ‘`return`’ ⟨*expr*⟩ ‘`;`’

⟨*if-else-stat*⟩ ::= ‘`if (`’ ⟨*expr*⟩ ‘`) {`’ ⟨*stat-list*⟩ ‘`}`’
| ‘`if (`’ ⟨*expr*⟩ ‘`) {`’ ⟨*stat-list*⟩ ‘`} else {`’ ⟨*stat-list*⟩ ‘`}`’

⟨*loop*⟩ ::= ‘`for (`’ ⟨*ident*⟩ ‘`,`’ ⟨*ident*⟩ ‘`) in `’ ⟨*expr*⟩ ‘`{`’ ⟨*stat-list*⟩ ‘`}`’

⟨*assignment*⟩ ::= ⟨*ident*⟩ ‘`=`’ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*index*⟩ ‘`=`’ ⟨*expr*⟩

⟨*lock*⟩ ::= ‘`lock_client {`’ ⟨*stat-list*⟩ ‘`}`’

⟨*comment*⟩ ::= ‘`//`’ [^‘`\n`’] ‘`\n`’

⟨*index*⟩ ::= ‘`[`’ ⟨*expr*⟩ ‘`]`’

⟨*expr*⟩ ::= ⟨*ident*⟩
| ⟨*primitive-type*⟩
| ⟨*array*⟩
| ⟨*hash*⟩
| ⟨*expr-access*⟩
| ⟨*function-invocation*⟩
| ⟨*unary-operation*⟩
| ⟨*binary-operation*⟩
| ‘`undefined`’

⟨*primitive-type*⟩ ::= ⟨*int*⟩
| ⟨*string*⟩
| ⟨*bool*⟩

⟨*array*⟩ ::= ‘`[ ]`’
| ‘`[`’ ⟨*expr-list*⟩ ‘`]`’

⟨*hash*⟩ ::= ‘`{ }`’
| ‘`{`’ ⟨*hash-entries*⟩ ‘`}`’

⟨*hash-entries*⟩ ::= ⟨*expr*⟩ ‘`:`’ ⟨*expr*⟩ ‘`,`’ ⟨*hash-entries*⟩ | ⟨*expr*⟩ ‘`:`’ ⟨*expr*⟩

⟨*expr-access*⟩ ::= ⟨*expr*⟩ ⟨*index*⟩

⟨*unary-operation*⟩ ::= ‘`!`’ ⟨*expr*⟩ | ‘`-`’ ⟨*expr*⟩

⟨*binary-operation*⟩ ::= ⟨*expr*⟩ ⟨*binary-operator*⟩ ⟨*expr*⟩

⟨*binary-operator*⟩ ::= ‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`/`’ | ‘`%`’ | ‘`==`’ | ‘`!=`’ | ‘`||`’ | ‘`&&`’ | ‘`>`’ | ‘`<`’ |
‘`>=`’ | ‘`<=`’

⟨*int*⟩ ::= ⟨*digit*⟩ ⟨*int*⟩ | ⟨*digit*⟩

⟨*string*⟩ ::= `'"'` ⟨*char-sequence*⟩ `'"'`

⟨*char-sequence*⟩ ::= ⟨*char*⟩ ⟨*char-sequence*⟩ | ⟨*empty*⟩

⟨*bool*⟩ ::= `'false'` | `'true'`

⟨*ident*⟩ ::= ⟨*ident-char*⟩ ⟨*ident*⟩ | ⟨*ident-char*⟩

⟨*char*⟩ ::= `[ ^'"' ]` | `'\"'`

⟨*ident-char*⟩ ::= `'a'` | `'b'` | ... | `'z'` | `'A'` | `'B'` | ... | `'Z'` | `'_'` | `'-'` | ⟨*digit*⟩

⟨*digit*⟩ ::= `'0'` | `'1'` | `'2'` | `'3'` | `'4'` | `'5'` | `'6'` | `'7'` | `'8'` | `'9'`

A lo largo del informe se pueden observar ejemplos de funciones del mismo. En este apéndice nos centramos en la definición formal y, por lo tanto, luego de describir la sintaxis, describimos cómo se ejecutan las instrucciones del lenguaje.

A2.2 Ejecución de instrucciones

En esta sección, describimos cómo se ejecutan las distintas instrucciones definidas en la sintaxis del lenguaje.

Invocación de una función.

La sintaxis es:

```
<identificador>(<expresion1>,<expresion2>,...,<expresionn>);
```

En este caso, el identificador debe corresponder al nombre de una función, y el resto de las expresiones serán los argumentos utilizados para invocar dicha función. La cantidad de argumentos brindados debe ser igual a la cantidad de parámetros en la definición de la función.

Al ejecutarse una instrucción de esta forma, primero se computa el primer argumento, luego el segundo, y así sucesivamente. Por último, se ejecuta la función para los argumentos dados.

Retorno de una expresión.

La sintaxis es:

```
        return <expresion>;
```

Al ejecutar esta instrucción, primero se computa la expresión (dicho valor será el retornado por la función) y luego se finaliza la ejecución de la función en la que se encuentra.

Si finaliza la ejecución de una función, y no se ejecutó esta instrucción, entonces la función retorna el valor *undefined*.

Bloque if-else.

Estas instrucciones son de la forma:

```
if (<expresion>) {
    //instrucciones
}
```

O de la forma:

```
if (<expresion>) {
    //instrucciones
} else {
    //instrucciones
}
```

Para ejecutar este tipo de instrucciones, primero se computa la expresión, y luego se evalúa el valor de verdad del valor computado. Si es *true* se ejecutan, secuencialmente, las instrucciones dentro del bloque if. En caso de ser *false*, se ejecutan las instrucciones dentro del bloque else (en caso de estar presente).

Los valores de verdad para los distintos tipos son los siguientes:

Tipo	**Valor de verdad**
int	Todos los valores evalúan a *true* a excepción del valor 0, que evalúa a *false*.
string	Todos los valores evalúan a *true* a excepción de la cadena vacía, que evalúa a *false*.
bool	*true* evalúa a *true*, y *false* evalúa a *false*
undefined	*false*
array	*true*
hash	*true*
IP	*true*
action	*true*

Bucle.

La sintaxis de un bucle es la siguiente:

```
for (<identificador1>, <identificador2>) in <expresion> {
    //instrucciones
}
```

Para ejecutar dicha instrucción, primero se computa la expresión, la cual debe computar un valor de tipo *hash* o de tipo *array*.

En caso de que compute un valor de tipo *hash*: Se recorren las entradas del *hash* referenciado por el valor (los pares (*clave*, *valor*)) sin un orden determinado. Para cada par:

1. Se agrega una variable con identificador igual al identificador$_1$ y con el valor de *clave* al ambiente.
2. Se agrega una variable con identificador igual al identificador$_2$ y con el valor de *valor* al ambiente.
3. Se ejecutan secuencialmente las instrucciones del cuerpo del bucle.

En caso de que compute un valor de tipo *array*: Se recorren las entradas del *array* referenciado por el valor (los pares (*índice*, *valor*)), ordenados de menor a mayor *índice*. Para cada par:

1. Se agrega una variable con identificador igual al identificador$_1$ y con el valor de *índice* al ambiente.
2. Se agrega una variable con identificador igual al identificador$_2$ y con el valor de *valor* al ambiente.
3. Se ejecutan secuencialmente las instrucciones del cuerpo del bucle.

Cuando se habla de "agregar una variable" al ambiente, podría pasar que ya exista una variable con dicho identificador en el ambiente, en ese caso sólo se modificaría el valor de dicha variable.

Asignación a una variable.

La sintaxis de una asignación a una variable puede ser:

```
        <identificador> = <expresion>;
```

En este caso, si el identificador corresponde a una variable del ambiente, se modifica el valor de dicha variable con el computado por la expresión. En caso de que no exista tal variable, se agrega una variable al ambiente con dicho identificador, y cuyo valor es el computado por la expresión.

Por otro lado, la asignación a una variable puede tener la siguiente sintaxis:

```
        <expresion1> [expresion2] = <expresion3>;
```

En este caso, primero se computa la primera expresión, la cual debe computar un valor de tipo *hash* o un valor de tipo *array*. Si computa un valor de tipo *hash*:

1. Se computa la segunda expresión, la cual debe computar un valor de tipo primitivo.
2. Se computa la tercera expresión, y se modifica el *hash* referenciado por el valor de la primera expresión, agregándole un par (*clave*, *valor*) utilizando el valor de la segunda expresión como *clave*, y el de la tercera como *valor*. En caso de ya existir otro par con dicha *clave*, el mismo se sobreescribe.

Si la primera expresión computa un valor de tipo *array*:

1. Se computa la segunda expresión, la cual debe computar un valor de tipo *int*, mayor o igual a cero.
2. Se computa la tercera expresión, y se modifica el *array* referenciado por el valor de la primera expresión, agregándole un par (*índice*, *valor*) utilizando el valor de la segunda expresión como *índice*, y el de la tercera como *valor*. En caso de ya existir otro par con dicho *índice*, el mismo se sobreescribe.
3. Si no existía un par con dicho *índice*: para todo *i* entero mayor o igual al largo del *array*, y menor al valor computado por la segunda expresión, se agregan los pares (*i*, *undefined*) al *array* referenciado por la primera expresión.

Break de un bucle.

La sintaxis es "break;". Esta instrucción se debe ejecutar dentro del cuerpo de un bucle y finaliza la ejecución de dicho bucle. Si la instrucción es ejecutada dentro del cuerpo de más de un bucle (en el caso de que haya bucles anidados), se finaliza la ejecución del que haya empezado a ejecutar último (el más "chico").

Cierre de exclusión mutua.

La sintaxis de un cierre de exclusión mutua es la siguiente:

```
    lock_client {
        //instrucciones
    }
```

Para ejecutar dicha instrucción, se ejecutan las instrucciones de su cuerpo.

Comentario.

Los comentarios inician con el dos barras ("//") y finalizan con un salto de línea. Los mismos son ignorados por el intérprete del lenguaje.

A2.3 Cómputo de expresiones

En esta sección, que complementa la anterior, describimos cómo se computan las distintas expresiones definidas en la sintaxis del lenguaje.

Identificador.

En el caso que la expresión sea un identificador, se busca si hay alguna variable con dicho identificador en el ambiente. En caso de haberlo, se computa el valor de dicha variable. En caso que no se encuentre, se computa el valor *undefined*.

Tipo primitivo.

En estos casos, la computación es trivial:

- *false* computa el valor *false* de tipo *bool*.
- *true* computa el valor *true* de tipo *bool*.
- Una secuencia de dígitos computa dicho valor (visto como un natural) como valor de tipo *int*, siempre y cuando dicho valor se encuentre en el rango de números representables por el tipo *int*.
- Una cadena de caracteres entre comillas dobles computa dicha cadena de caracteres como valor de tipo *string* (utilizando su codificación UTF-8). En la cadena de caracteres, se remplaza la cadena de dos caracteres \", por el carácter ".

Array.

La sintaxis de estas expresiones es:

$$\texttt{[expresion}_1\texttt{, expresion}_2\texttt{, } \cdots \texttt{ , expresion}_n\texttt{]}$$

Este tipo de expresiones computan un valor de tipo *array*: primero se computa la primera expresión, luego la segunda, y así sucesivamente. La expresión computa una

referencia a un *array* con las entradas $(i, val\_expresion_{i+1})\ \forall i : 0 \leq i \leq n-1$ (donde $val\_expresion_{i+1}$ es el valor computado por la expresión $expresion_{i+1}$).

Hash.

La sintaxis de estas expresiones es:

```
{ expresion1.1: expresion1.2, ··· , expresionn.1: expresionn.2 }
```

Este tipo de expresiones computan un valor de tipo *hash*: si n es cero, la expresión computa una referencia a un *hash* vacío. Si $n > 0$ primero se computa la expresión

```
{ expresion1.1: expresion1.2, ··· , expresionn−1.1: expresionn−1.2 }
```

Luego se computa la expresión $expresion_{n.1}$ (la cual debe computar un valor de tipo primitivo), y por último la expresión $expresion_{n.2}$. El valor computado, entonces, es la referencia computada recursivamente, agregándole el par ($val\_expresion_{n.1}$, $val\_expresion_{n.2}$) al *hash*. Si ya existe un par con dicha clave, se sustituye el mismo.

Acceso a una expresión.

La sintaxis en este caso es

```
        <expresion1>[<expresion2>]
```

Para computar una expresión de este tipo, primero se computa la primera expresión la cual debe computar un valor de tipo *array* o *hash*.

En caso que la primera expresión compute un valor de tipo *array*, se computa la segunda expresión la cual debe computar un valor de tipo *int*, mayor o igual a cero. En caso de que el *array* referenciado contenga un par (*índice*, *valor*) donde *índice* coincida con el valor computado por la segunda expresión, el valor computado es *valor*. En otro caso, el valor computado es *undefined*.

En caso que la primera expresión compute un valor de tipo *hash*, se computa la segunda expresión la cual debe computar un valor de tipo primitivo. En caso de que el *hash* referenciado contenga un par (*clave*, *valor*) donde *clave* coincida con el valor computado por la segunda expresión, el valor computado es *valor*. En otro caso, el valor computado es *undefined*.

Precedencia de operadores.

Previo a describir cómo se computan las distintas operaciones, presentamos la tabla con la precedencia (de mayor a menor) de cada operador.

Precedencia	Operadores
7	!, - (unario)
6	*, /, %
5	+, - (binario)
4	<, <=, >, >=
3	==, !=
2	&&
1	||

Además, todos los operadores binarios asocian a la derecha.

Operación unaria.

La sintaxis de este tipo de expresiones es:

```
<operador> <expresion>
```

En este caso, el cómputo de la expresión depende del operador:

- !: En este caso, se computa la expresión y se obtiene su valor de verdad. Luego, se computa el negado de dicho valor de verdad.
- -: En este caso, se computa la expresión la cual debe computar un valor de tipo *int*. Luego, se computa el opuesto a dicho valor (siempre y cuando dicho valor se encuentre en el rango de números representables por el tipo *int*).

Operación binaria.

La sintaxis de este tipo de expresiones es:

```
<expresion1> <operador> <expresion2>
```

En este caso, el cómputo de la expresión depende del operador:

- Operadores booleanos:
  - &&: Se realiza la evaluación mediante circuito corto: se computa la primera expresión y se obtiene su valor de verdad. En caso de ser *false*, se computa *false*. Sino, se computa el valor de verdad de la segunda expresión y dicho valor es el valor computado por la expresión.

  - ||: Se realiza la evaluación mediante circuito corto: se computa la primera expresión y se obtiene su valor de verdad. En caso de ser *true*, se computa *true*. Sino, se computa el valor de verdad de la segunda expresión y dicho valor es el valor computado por la expresión.

- Operadores numéricos: Para estos operadores siempre se computa la primera expresión, la cual debe computar un valor de tipo *int*. Luego, se computa la segunda expresión la cual debe computar un valor de tipo *int*. Utilizando ambos valores, se computa la expresión dependiendo del operador.
  - -: El valor computado es el valor de tipo *int* dado por la resta del primer valor menos el segundo.
  - *: El valor computado es el valor de tipo *int* dado por la multiplicación de ambos valores.
  - /: En este caso, el segundo valor debe ser distinto a cero. El valor computado es el valor de tipo *int* dado por la división entera del primer valor dividido el segundo.
  - %: En este caso, el segundo valor debe ser distinto a cero. El valor computado es el valor de tipo *int* dado por el resto de la división entera del primer valor dividido el segundo.
  - >: El valor computado es un valor de tipo *bool*: *true* si el primer valor es mayor al segundo, y *false* en caso contrario.
  - <: El valor computado es un valor de tipo *bool*: *true* si el primer valor es menor al segundo, y *false* en caso contrario.
  - >=: El valor computado es un valor de tipo *bool*: *true* si el primer valor es mayor o igual al segundo, y *false* en caso contrario.
  - <=: El valor computado es un valor de tipo *bool*: *true* si el primer valor es menor o igual al segundo, y *false* en caso contrario.
- Operador +: Este operador depende del tipo de las expresiones. Primero se computa la primera expresión:
  - Si dicho valor es de tipo *int*: Se computa la segunda expresión la cual debe retornar un valor de tipo *int*. Luego, el valor computado es el valor de tipo *int* dado por la suma de los dos valores.
  - Si dicho valor es de tipo *string*: Se computa la segunda expresión la cual debe retornar un valor de tipo *string*. Luego, el valor computado es el valor de tipo *string* dado por la concatenación del primer valor con el

segundo valor (en ese orden).

- Operador ==: Este operador define la igualdad de dos valores. Para esto primero se computa la primera expresión, y luego la segunda. En caso de que los tipos de los dos valores sean distintos, retorna *false*. En caso de que sean iguales:
  - Si el tipo es *bool*: Retorna *true* si ambos valores son *true* o ambos valores son *false*. Sino retorna *false*.
  - Si el tipo es *int*: Retorna *true* si ambos valores representan el mismo entero. Sino retorna *false*.
  - Si el tipo es *string*: Retorna *true* si ambos valores representan la misma cadena de bytes. Sino retorna *false*.
  - Si el tipo es *IP*: Retorna *true* si ambos valores representan la misma dirección IP. Sino retorna *false*.
  - Si el tipo es *action*: Retorna *true* si ambos valores representan la misma acción. Sino retorna *false*.
  - Si el tipo es *array*: Retorna *true* si ambos valores son iguales como conjunto de pares (*índice*, *valor*). Un par ($índice_1$, $valor_1$) es igual a ($índice_2$, $valor_2$) si, y sólo si, $índice_1$ == $índice_2$ computa *true* y $valor_1$ == $valor_2$ computa *true*.
  - Si el tipo es *hash*: Retorna *true* si ambos valores son iguales como conjunto de pares (*clave*, *valor*). Sino retorna *false*.
  - Si el tipo es *undefined*: Retorna *true* (ambos valores son el valor *undefined*).
- Operador !=: Retorna el *booleano* opuesto al retornado por el operador ==.

Para aquellos operadores que computan un valor de tipo *int*, es importante recordar que es necesario que dicho valor se encuentre en el rango de números representables por el tipo *int*.

Invocación de una función.

La sintaxis es:

```
<identificador>(<expresion1>, <expresion2>, ... , <expresionn>)
```

En este caso, el identificador debe corresponder al nombre de una función, y el resto de las expresiones serán los argumentos utilizados para invocar dicha función. La cantidad de argumentos brindados debe ser igual a la cantidad de parámetros en la

definición de la función.

Al computarse una expresión de esta forma, primero se computa el primer argumento, luego el segundo, y así sucesivamente. Por último, se ejecuta la función para los argumentos dados, y el valor retornado por la función será el valor computado.

undefined.

La expresión con sintaxis "undefined" computa el valor especial *undefined*.

A2.4 Funciones predefinidas

Para ejecutar todas las operaciones necesarias a la hora de implementar el CRS en este lenguaje, deben ser implementadas algunas funciones predefinidas. Estas son funciones previamente implementadas a bajo nivel, que están disponibles para que el usuario las utilice en cualquier momento. Si bien el fin de algunas de estas funciones es facilitar la tarea del usuario al utilizar el lenguaje, muchas de estas agregan funcionalidades al lenguaje que de otra manera no se podría conseguir.

En esta sección presentamos las distintas funciones predefinidas, junto a su descripción correspondiente, separadas en seis categorías: funciones sobre objetos de tipo *array*, de tipo *hash*, de tipo *IP*, de tipo *string*, transformaciones y otras funciones.

Funciones sobre objetos de tipo *array*.

- *find(arr, item)*: Recibe un objeto de tipo *array* como primer parámetro y un objeto de cualquier tipo como segundo parámetro. Retorna un objeto de tipo *int* indicando la primera posición en la que se encuentra el objeto *item* en *arr*. En caso de no encontrarse en *arr*, retorna el valor -1.
- *length(arr)*: Recibe un objeto de tipo *array* y retorna el largo del mismo.
- *pop(arr)*: Recibe un objeto de tipo *array*. Remueve el último objeto de *arr* y retorna su valor. En caso de ser un *array* vacío retorna *undefined*.
- *push(arr, item)*: Recibe un objeto de tipo *array* como primer parámetro y un objeto de cualquier tipo como segundo parámetro. Inserta el objeto *item* al final de *arr*.
- *remove_entry(arr, i)*: Recibe un objeto de tipo *array* como primer parámetro y un objeto de tipo *int*, mayor o igual a cero, como segundo parámetro:

  - Si *i* es mayor o igual al largo del *array*, no hace nada.
  - Si *i* es igual al largo del *array* menos uno, remueve el último objeto del *array* (disminuyendo su largo).
  - Si *i* es menor al largo del *array*, es equivalente a haber realizado la asignación *arr[i] = undefined* (no disminuye su largo).

Funciones sobre objetos de tipo *hash*.

- *clear_hash(h)*: Recibe un objeto de tipo *hash* y sobreescribe el valor de *h* por un *hash* vacío.
- *get_keys(h)*: Recibe un objeto de tipo *hash* y retorna un objeto de tipo *array* conteniendo todas las claves del mismo.
- *is_key(h, k)*: Recibe un objeto de tipo *hash* como primer parámetro y un objeto de tipo primitivo como segundo parámetro. Retorna *true* si existe una clave igual a *k* en *h*. De lo contrario retorna *false*.
- *remove_key(h, k)*: Recibe un objeto de tipo *hash* como primer parámetro y un objeto de tipo primitivo como segundo parámetro. En caso de que el *hash* cuente con una entrada con dicha clave, la misma es eliminada.

Funciones sobre objetos de tipo *IP*.

- *in_subnet(ip1, ip2, mask)*: Recibe tres objetos de tipo *IP*. Retorna *true* si la dirección *ip1* se encuentra en la subred dada por la dirección *ip2* y la máscara *mask*. De lo contrario retorna *false*.
- *ip_v4(a, b, c, d)*: Recibe cuatro objetos de tipo *int* y retorna un objeto de tipo *IP* que representa la dirección IPv4 dada por los cuatro parámetros. El primer parámetro se interpreta como los primeros 8 bits, el segundo parámetro se corresponde con los siguientes 8 bits, etc.
- *ip_v6(a, b, c, d, e, f, g, h)*: Recibe ocho objetos de tipo *string* y retorna un objeto de tipo *IP* que representa la dirección IPv6 dada por los ocho parámetros. El primer parámetro se interpreta como los caracteres hexadecimales de los primeros 16 bits, el segundo parámetro se corresponde con los siguientes 16 bits, etc.

Funciones sobre objetos de tipo *string*.

- *aho_corasick(s, arr)*: Recibe un objeto de tipo *string* y un objeto de tipo *array* el cual debe estar compuesto de objetos de tipo *string*. Retorna *true* si *s* se

encuentra en *arr*, de lo contrario retorna *false*. Utiliza el algoritmo Aho-Corasick para dicha búsqueda.

- *aho_corasick_from_file(s, file)*: Recibe dos objetos de tipo *string*. El segundo debe corresponder a la dirección de un archivo, el cual debe contener una lista de cadenas de caracteres separadas por saltos de líneas. Retorna *true* si *s* se corresponde a una cadena de dicho archivo, de lo contrario retorna *false*. Utiliza el algoritmo Aho-Corasick para dicha búsqueda. La dirección del archivo debe ser una cadena de caracteres UTF-8.
- *capture_regex(s, re)*: Recibe dos objetos de tipo *string*. Retorna un objeto de tipo *array* con todas las capturas de la expresión regular *re* al evaluar la cadena *s*. La primera entrada del *array* retornado se corresponderá con el área donde la expresión regular coincidió. En caso de que la expresión regular no coincida con ninguna subcadena de *s*, retorna un *array* vacío. La expresión regular debe ser una cadena de caracteres UTF-8.
- *contains(s, p)*: Recibe dos objetos de tipo *string*. Retorna *true* si *s* contiene una subcadena igual a *p*. De lo contrario retorna *false*.
- *ends_with(s, p)*: Recibe dos objetos de tipo *string*. Retorna *true* si *s* termina con una subcadena igual a *p*. De lo contrario retorna *false*.
- *in_byte_range(s, arr)*: Recibe dos objetos, uno de tipo *string* y otro de tipo *array*. Las entradas del segundo pueden corresponder a valores de tipo *int* o valores de tipo *array* de largo dos, cuyas dos entradas sean valores de tipo *int* (los pensamos como pares *(min, max)*). Retorna *true* si todo byte de la cadena *s* cumple una de las siguientes dos condiciones:
  - pertenece a *arr*.
  - el byte se encuentra dentro del rango dado por una entrada *(min, max)* de *arr*.

  De lo contrario retorna *false*.
- *is_sqli(s)*: Recibe un objeto de tipo *string*. Retorna *true* si se detecta, utilizando la librería **libinjection**, un ataque de inyección SQL en *s*. De lo contrario retorna *false*.
- *is_xss(s)*: Recibe un objeto de tipo *string*. Retorna *true* si se detecta, utilizando la librería **libinjection**, un ataque de tipo *XSS* en *s*. De lo contrario retorna *false*.
- *match_regex(s, re)*: Recibe dos objetos de tipo *string*. El segundo parámetro representa una expresión regular. Retorna *true* en caso de que la expresión

regular coincida con alguna subcadena de *s*. De lo contrario retorna *false*. La expresión regular debe ser una cadena de caracteres UTF-8.

- *to_uppercase(s)*: Recibe un objeto de tipo *string*. Retorna un *string* igual a *s* pero con todos sus caracteres ASCII en mayúscula.
- *validate_url_encoding(s)*: Recibe un objeto de tipo *string*. Retorna *true* si los caracteres URL-codificados de *s* son correctos. De lo contrario retorna *false*
- *validate_utf8_encoding(s)*: Recibe un objeto de tipo *string*. Retorna *true* si es una cadena UTF-8 válida. De lo contrario retorna *false*

Transformaciones.

- *base64_decode(s)*: Recibe un objeto de tipo *string* con un valor codificado en base64 y retorna la decodificación del mismo. Retorna *undefined* en caso de que no sea una codificación válida.
- *cmd_line(s)*: Recibe un objeto de tipo *string*. Retorna el resultado de aplicarle a *s* las siguientes transformaciones:
  - Se remueven los siguientes caracteres: ', ", \, ^, (, [,) y].
  - Se remplazan las comas (,) y semicomas (;) por espacios.
  - Se reemplazan espacios múltiples y saltos de línea por espacios simples.
  - Se convierten todos los caracteres ASCII a minúsculas.
- *compress_white_space(s)*: Recibe un objeto de tipo *string* y retorna un *string* igual a *s* pero reemplazando todos los espacios múltiples por espacios simples.
- *css_decode(s)*: Recibe un objeto de tipo *string* y retorna un *string* igual a *s* pero decodificando todos los caracteres codificados utilizando las reglas de codificación CSS 2.x (`https://www.w3.org/TR/CSS2/`). Retorna *undefined* en caso de que no sea una codificación válida.
- *hex_encode(s)*: Recibe un objeto de tipo *string* y retorna un *string* donde se codifica cada byte de *s* en hexadecimal. Cada byte se convierte en dos bytes con los caracteres ASCII de su codificación hexadecimal.
- *html_entity_decode(s)*: Recibe un objeto de tipo *string* y retorna un *string* donde se decodifica cada carácter codificado como una entidad HTML. Retorna *undefined* en caso de que no sea una codificación válida.
- *js_decode(s)*: Recibe un objeto de tipo *string* y retorna un *string* donde se decodifican las secuencias de escape de JavaScript. Retorna *undefined* en caso de

que no sea una codificación válida.

- *normalize_path(s)*: Recibe un objeto de tipo *string* que representa una ruta y retorna el resultado de aplicarle a *s* las siguientes transformaciones:
  - Se reemplazan todas las barras (/) múltiples por barras simples.
  - Se remueven todas referencias al directorio actual (./).
  - Se remueven todas las referencias hacia atrás (../) a no ser que se encuentren al principio de *s* (preservando el sentido de la ruta, por ejemplo se convierte *folder/../file.txt* en *file.txt*).
- *normalize_path_win(s)*: Recibe un objeto de tipo *string*. Se comporta igual que *normalize_path*, pero primero convierte todas las barras inversas (\) en barras (/).
- *remove_nulls(s)*: Recibe un objeto de tipo *string* y remueve todos sus bytes nulos.
- *replace_comments(s)*: Recibe un objeto de tipo *string*. Se retorna el resultado de reemplazar todos los comentarios *C-style* (/* ... */) por espacios simples.
- *sha1(s)*: Recibe un objeto de tipo *string*. Se retorna el resultado de aplicar el algoritmo *sha1* sobre *s*.
- *str_length(s)*: Recibe un objeto de tipo *string* y retorna el largo de dicha cadena de bytes.
- *to_lowercase(s)*: Recibe un objeto de tipo *string*. Retorna un *string* igual a *s* pero con todos sus caracteres ASCII en minúscula.
- *url_decode(s)*: Recibe un objeto de tipo *string* y retorna un *string* igual a *s* pero donde se decodifican los caracteres URL-codificados. Retorna *undefined* en caso de que no sea una codificación válida.
- *url_decode_uni(s)*: Análoga a *url_decode* pero con soporte para la codificación %u específica de Microsoft.
- *utf8_to_unicode(s)*: Recibe un objeto de tipo *string* y retorna un *string* igual a *s* pero donde se convierten todas las secuencias de caracteres UTF-8 a Unicode.

Otras funciones.

- *audit(conf)*: Esta función recibe un *array* con valores de tipo *string* que indican qué partes de la transacción deben ser registradas en el archivo de auditoría al finalizar el procesamiento de la transacción. En caso de recibir el *string* "req_headers", entonces se auditarán las cabeceras del pedido. Esto es análogo

para el cuerpo del pedido (con el *string* "req_body"), las cabeceras de la respuesta (con el *string* "res_headers") y el cuerpo de la respuesta (con el *string* "res_body").

- *deny(status)*: Recibe un objeto de tipo *int*, el cual debe corresponder con un estado válido para una respuesta HTTP. Retorna un objeto de tipo *action* que, en caso de ser retornado por una función de decisión, finaliza el procesamiento de la transacción y retorna una respuesta con dicho estado al cliente.
- *drop()*: Retorna un objeto de tipo *action* que, en caso de ser retornado por una función de decisión, finaliza el procesamiento de la transacción y no retorna una respuesta al cliente.
- *expire_client_key(key, time)*: Recibe un objeto de tipo primitivo, y otro objeto de tipo *int*, mayor o igual a cero. Esta función indica que una entrada de la variable global *client* debe ser eliminada después del tiempo (en segundos) dado por *time*. En caso de que dicha entrada ya tuviera un tiempo de expiración, el mismo se actualiza.
- *geo_lookup(ip, database)*: Recibe un objeto de tipo *IP* y un objeto de tipo *string* el cual debe ser la dirección de un archivo. Realiza una búsqueda de geolocalización usando la dirección *ip*, en el archivo *database*. En caso de no encontrar la ubicación de dicha dirección, retorna *undefined*. Si encuentra dicha dirección, retorna un *hash* con claves de tipo *string*, donde el contenido de cada entrada es el especificado a continuación:
  - **COUNTRY_CODE:** El código del país en dos caracteres. Ejemplos: US, GB.
  - **COUNTRY_CODE3:** El código del país en hasta tres caracteres.
  - **COUNTRY_NAME:** El nombre completo del país.
  - **COUNTRY_CONTINENT:** El continente del país en dos caracteres. Ejemplo: EU.
  - **REGION:** El nombre de la ciudad en caso de poder obtenerse.
  - **POSTAL_CODE:** El código postal en caso de poder obtenerse.
  - **LATITTUDE:** La latitud en caso de poder obtenerse.
  - **LONGITUDE:** La longitud en caso de poder obtenerse.
  - **DMA_CODE:** El código de área metropolitana en caso de poder obtenerse.
- *load(s)*: Recibe un objeto de tipo *string* que debe corresponder a la ruta de un

archivo. Se retorna un objeto de tipo *array*, donde cada entrada del mismo se corresponde con una línea del archivo.

- *log(s)*: Recibe un objeto de tipo *string* y registra el mensaje recibido en el archivo de registro. Al igual que dicho mensaje, se almacena la información de la regla/política/función de decisión/excepción que invocó la función, y un identificador único de la transacción.
- *pass()*: Retorna un objeto de tipo *action* que, en caso de ser retornado por una función de decisión, no interviene en el procesamiento de la transacción.
- *random(min, max)*: Recibe dos objetos de tipo *int* y retorna un entero aleatorio entre ambos (inclusives).

A3 | Relevamiento de librerías y proyectos

Para el desarrollo del WAF, fue fundamental realizar un relevamiento de librerías que podíamos utilizar en un proyecto como este, y el estado de las mismas. Si bien nuestra intención fue siempre evitar reinventar la rueda, sabíamos que construir nuestro proyecto basándonos en una librería no adecuada nos podía llevar a realizar re-trabajo en una etapa más avanzada del proyecto.

Si bien Rust es un lenguaje joven, existe una gran comunidad de desarrolladores, lo que inexorablemente implica una gran cantidad de librerías. Rust utiliza un gestor de paquetes llamado **cargo**, el cual permite instalar paquetes o librerías junto con sus correspondientes dependencias. En la URL `www.crates.io` se pueden encontrar distintos proyectos y librerías de código abierto, que pueden ser incluidas utilizando **cargo**.

En este apéndice, describimos proyectos similares que fueron relevados, al igual que las librerías que podían ser utilizadas para la implementación del WAF.

A3.1 Proyectos similares

En un primer momento nos pareció relevante buscar proyectos de similares características, tanto proxys inversos como WAFs, implementados en Rust. En ese sentido encontramos cinco proxys inversos de código abierto que nos parecieron interesantes, pero ningún WAF. Los proxys inversos encontrados, listados de mayor a menor "tamaño", fueron los siguientes:

- **Sozu (`https://github.com/sozu-proxy/sozu`):** Sozu es un proxy inverso muy completo y configurable, el énfasis de los desarrolladores está en crear un proxy inverso que nunca falle, que no tenga problemas de memoria, y que ni siquiera sea necesario reiniciarlo cuando se modifique su configuración.

  Algunas ventajas de Sozu son que tiene mantenimiento actualmente, soporta

HTTPS y ofrece una gran cantidad de funcionalidades.

Una característica muy particular de Sozu, es que prácticamente no utiliza librerías externas, ni siquiera para el manejo de mensajes HTTP, está todo implementado dentro del proyecto. Esto hace realmente difícil de entender su código ya que la documentación que hay actualmente es sobre su uso y configuración, no hay una documentación (al menos pública) de su arquitectura.

- **KatWebX (`https://github.com/katattakd/KatWebX`):** KatWebX es un proxy inverso y, al mismo tiempo, un servidor web estático. Una de sus principales ventajas es su velocidad, aunque ofrece múltiples funcionalidades como soporte para HTTPS, autenticación HTTP y compresión de archivos (al funcionar como servidor de archivos). KatWebX está implementado utilizando la librería Actix.

  Como desventaja, KatWebX continúa en desarrollo, y en el propio repositorio se indica que no se recomienda utilizarlo en producción (principalmente debido a que no está testeado correctamente).

- **Limitation Proxy (`https://github.com/fnichol/limitation`):** Limitation Proxy es un proxy inverso que tiene la particularidad de que realiza un "rate limiting" en todos los pedidos que pasan a través de él. El "rate limiting" es una variante de la estrategia "fixed window", y utiliza Redis para la persistencia temporal de los pedidos.

  En la figura A3.1 se puede ver el diagrama que se encuentra en la documentación del proxy inverso, donde se muestra cómo Limitation Proxy utiliza Redis para almacenar pedidos con el fin de no saturar lo que se encuentra "detrás" del proxy (usualmente un servidor). Este proyecto no soporta HTTPS, y está implementado sobre la librería Actix.

- **ProxyBoi (`https://github.com/svenstaro/proxyboi`):** ProxyBoi es un proxy inverso simple, sin ninguna funcionalidad destacada. Permite levantar un proxy inverso en una interfaz de red dada, eligiendo el dominio de la aplicación web que será la aplicación final. El proxy inverso creado puede configurarse para recibir pedidos HTTPS. Este proyecto fue desarrollado utilizando la librería Actix.

- **Hyper Reverse Proxy (`https://github.com/felipenoris/hyper-reverse-proxy`):** Este es un proyecto muy pequeño, pero que muestra un simple ejemplo de cómo crear un proxy inverso utilizando la librería Hyper. No ofrece soporte para HTTPS.

Figura A3.1: Limitation Proxy

Como dato adicional, en el ámbito privado ThreatX afirma que su WAF y su "central analysis engine" fueron programados en Rust. Si bien no nos aportó demasiado, sí fue bueno saber que ya habían aplicaciones de este tipo implementadas en Rust.

A3.2 Librerías para comunicaciones HTTP

Una vez que estudiamos esos proyectos, relevamos qué librerías eran utilizadas por los mismos y qué alternativas a ellas eran utilizadas también. Previamente, comentamos que KatWebX, Limitation Proxy y Proxyboi están implementadas utilizando la librería Actix, mientras que Hyper Reverse Proxy está implementada utilizando la librería Hyper. Dichas librerías son librerías que permiten crear clientes y servidores HTTP, y en este tipo de librerías nos centramos a continuación.

No sólo se analizaron las librerías Actix y Hyper, sino alternativas a ellas. Tomamos en cuenta qué servicios ofrecían, la opinión de la comunidad sobre las mismas, la utilización de estas a nivel de producción, el estado en que se encontraban y verificamos que utilizaran la última versión de Rust y fueran compatibles con las últimas funcionalidades de asincronismo. Era fundamental conocer en qué estado se encontraban las librerías ya que podían haber dejado de ser mantenidas.

Particularmente se requería una librería con funcionalidades sobre HTTP que permitiera:

- Comunicarse con un cliente a través de HTTP (HTTPS)
- Comunicarse con un servidor a través de HTTP (HTTPS)

Las librerías relevadas fueron las siguientes:

- **Actix (`https://github.com/actix/actix`):** Actix es una librería de alto nivel, fue diseñada con el fin de permitir desarrollar aplicaciones web. Además, es la librería más popular en proyectos que ofrecen servicios web.

  Actix está construida sobre la librería Hyper y, aunque permite crear servidores HTTP, no permite la creación de clientes HTTP.

  Actualmente, el estado de la misma es incierto. La comunidad ha tenido varias discusiones sobre su performance, su seguridad, y el futuro de la misma. En diciembre del 2019 uno de los desarrolladores más importantes del proyecto decidió abandonar el mismo debido a la gran cantidad de críticas. Las críticas se basaban, principalmente, en la cantidad código "unsafe".

- **Iron (`https://github.com/iron/iron`):** Iron es un framework web configurable y extensible. Está construido sobre Hyper, se puede ver como un framework para crear servidores utilizando dicha librería de una manera más sencilla. Básicamente, permite crear una pila de lo que el framework denomina "middlewares".

  Los "middlewares" se catalogan en dos tipos: "before middlewares", y "after middlewares". Los "before middlewares" son, a grandes rasgos, funciones que procesan un pedido (reciben un pedido y retornan un pedido). Los "after middlewares" son, análogamente, funciones que procesan una respuesta (reciben una respuesta y retornan una respuesta).

  El programador sólo debe crear los "before middlewares", los "after middlewares", establecer en qué orden se van a ejecutar, y crear un un "handler" (una función que recibe un pedido y retorna una respuesta). Iron se encarga de crear un servidor que, dado un pedido, ejecuta el flujo descrito en la figura A3.2.

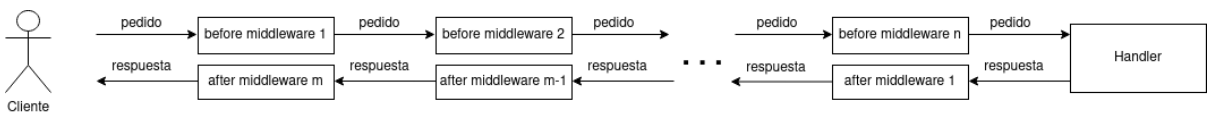

Figura A3.2: Flujo al utilizar Iron

Una gran desventaja de esta librería es que aún no ha actualizado su código

para ser compatible con las directivas async/await. Esto no nos permite, por ejemplo, crear un "Handler" que ejecute un procesamiento asíncrono (como sería retransmitir el pedido a otro servidor, y retornar la respuesta).

- **Hyper (`https://github.com/hyperium/hyper`):** Hyper es una librería de más bajo nivel que Actix, y provee funcionalidades para crear servidores y clientes HTTP. Es una librería muy popular, y es ampliamente utilizada en producción por la comunidad.

  Además, hay distintas librerías que permiten extender funcionalidades de Hyper como, por ejemplo, hyper-tls (creada por el mismo desarrollador) que permite crear clientes que se conecten utilizando el protocolo HTTPS con un servidor.

  Esta librería permite un manejo más directo de pedidos y respuestas, sin entrar en construcciones más complejas como el manejo de sesiones web.

- **H2 (`https://github.com/hyperium/h2`):** H2, al igual que Hyper, es una librería de bajo nivel que brinda funcionalidades para crear servidores y clientes HTTP. Una de las desventajas de H2, además de ser menos utilizada que Hyper, es que sólo permite conexiones utilizando la especificación del protocolo HTTP/2.0.

  Dentro de los desarrolladores de H2, se encuentra el principal desarrollador de Hyper. En la documentación de la misma, se especifica que esta librería es utilizada por Hyper y se afirma que Hyper brindará todas las funcionalidades de H2.

La utilización de la librería Hyper en este proyecto se basó en la posibilidad de trabajar con pedidos y respuestas HTTP a bajo nivel, además de ser ampliamente utilizada y ser compatible con la última versión de Rust. Además, la librería Iron inspiró la librería Rhodium, la cual ofrece funcionalidades similares, pero permitiendo funciones asíncronas en el manejo de pedidos y respuestas.

A4 | Configuración y ejecución

En esta sección, describimos cómo ejecutar tanto el CRS-Parser como el WAF implementado. Para el desarrollo de ambas aplicaciones, se utilizó la última versión de Rust (al momento de escribir este informe). Por lo tanto, para ejecutar las mismas es necesario tener instalado Cargo, el administrador de paquetes de Rust, en su versión 1.50.

A4.1 CRS-Parser

Dentro del repositorio del proyecto, el CRS-Parser se encuentra en el directorio **crs-parser**. Para ejecutar la aplicación, es necesario situarse en dicho directorio y ejecutar el siguiente comando:

```
cargo run <file>
```

Donde <*file* > debe ser un archivo de configuración de ModSecurity. En el directorio **crs-parser/examples** se pueden encontrar las reglas del CRS en su versión 3.3.0. Por lo tanto, un ejemplo de ejecución es el siguiente:

```
cargo run examples/REQUEST-932-APPLICATION-ATTACK-RCE.conf
```

Además, para ejecutar las pruebas del mismo, se debe ejecutar el comando:

```
cargo test
```

Si además se desea obtener el porcentaje de código cubierto por dichas pruebas, se debe utilizar el siguiente comando (no es necesario haber ejecutado antes *cargo test*).

```
cargo tarpaulin
```

Diferenciamos *cargo tarpaulin* de *cargo test* debido a que, una vez ejecutado *cargo run* o *cargo test*, se crearán archivos que harán que la siguiente ejecución de ambos sea más rápida (por ejemplo, si se ejecuta *cargo run* luego *cargo test* se ejecutará más rápido).

Aún así, *cargo tarpaulin* no utiliza dichos archivos, e incluso los elimina.

Por último, en el directorio **crs-parser**, se puede encontrar un archivo *regexes_con_problemas.txt* con las expresiones regulares de ModSecurity, que no son admitidas por la librería "regex" de Rust (separadas según el motivo).

A4.2 WAF

En el caso del WAF implementado, el mismo se encuentra en el directorio **waf**. Para ejecutarlo, es necesario tener instalada la librería **libinjection**. En el caso de contar con una distribución basada en Debian, los siguientes comandos instalan las dependencias necesarias:

```
apt-get install -y libclang-dev
apt install -y llvm-dev libclang-dev clang
```

Una vez instalada dicha librería, para ejecutar el WAF basta con ejecutar el siguiente comando:

```
cargo run <config-file> <policies-directory>
```

Donde <*config-file* > es el archivo de configuración descrito en 5.3.2. Y <*policies-directory* > es la dirección del directorio descrito en 5.3.4.

El proyecto viene con una configuración básica, que inicia el WAF escuchando en el puerto 3000 del servidor donde se esté ejecutando. En dicha configuración, el servidor final es `proygrado-juice-shop.herokuapp.com`. Dicha configuración se puede ejecutar utilizando el comando:

```
cargo run assets/config.yml assets/basic-example
```

Es importante mencionar que la url `proygrado-juice-shop.herokuapp.com` puede tardar en responder al primer pedido que se le envíe.

Para ejecutar las pruebas del proyecto, basta ejecutar el comando:

```
cargo test --workspace
```

Mientras que si se quiere también obtener el porcentaje de código cubierto por las mismas, se debe ejecutar

```
cargo tarpaulin --workspace
```

A4.2.1 Prueba de concepto

Para ejecutar la prueba de concepto, es necesario tener Docker instalado. En un principio, se deben ejecutar los siguientes comandos (desde el directorio **waf**):

```
git submodule update --init
cd POC && sh setup.sh && cd ..
```

Dichos comandos deben ser ejecutados una única vez y aplican los cambios que se realizaron sobre el proyecto *gotestwaf*.

Una vez que hayan finalizado los comandos anteriores, se puede ejecutar la prueba de concepto. Para esto, se debe invocar el siguiente comando (la primera vez puede tomar algunos minutos):

```
docker-compose up
```

En el momento en que el *gotestwaf* haya finalizado sus pruebas, se imprimirá en consola una tabla con los resultados. Además, se creará un pdf en el directorio **waf/POC/wafnextgen/reports** con los mismos.

La configuración utilizada se encuentra en el directorio **waf/assets/crs-based-security**.

Aún cuando haya finalizado el *gotestwaf*, el WAF seguirá ejecutando y podrá ser accedido a través del servidor actual, en el puerto 8080. Aún así, recordamos que en el caso de la prueba de concepto, el servidor final no hace más que retornar respuestas HTTP con estado 200.

De la misma manera, se puede utilizar el siguiente comando para aplicar las pruebas del *gotestwaf* sobre ModSecurity. Los resultados, en este caso, se crearán en el directorio **waf/POC/modsecurity/reports**.

```
docker-compose -f docker-compose-modsec.yml up
```

Las reglas del CRS utilizadas para esta prueba, se encuentran en **waf/POC/modsecurity/rules**.