UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

WACE: Un Integrador de Clasificadores de Ataques Web

Informe de Proyecto de Grado presentado por

Agustín de León y Tobias Iroa

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisores

Juan Diego Campo
Felipe Zipitría

Usuario Responsable

Rodrigo Martínez

Montevideo, 16 de diciembre de 2024

Agradecimientos

Nos gustaría expresar nuestro más sincero agradecimiento a los docentes Juan Diego Campo, Felipe Zipitría, Rodrigo Martínez y Gustavo Betarte por sus enseñanzas, apoyo y motivación a lo largo de este proyecto. Sin su constante dedicación, dicho proyecto no habría sido posible.

Queremos agradecer a los miembros del tribunal, Aiala Rosá, Facundo Benavides y Marcelo Rodríguez por su esfuerzo y dedicación en la evaluación del proyecto.

A nuestras familias y amigos, queremos darles las gracias por su paciencia, motivación y acompañamiento incondicional a lo largo de estos años.

Por último nos gustaría expresar nuestro agradecimiento a todos aquellos que contribuyeron a nuestra formación.

Resumen

Este proyecto aborda las tradicionales limitaciones de los Firewalls de Aplicación Web (WAF por sus siglas en inglés) tales como su incapacidad para detectar ataques desconocidos y la generación de falsos positivos al aplicar reglas genéricas. El objetivo principal del trabajo es integrar la tecnología de WAF OWASP Coraza con la tecnología denominada Web Attack Classification Engine (WACE) para potenciar las capacidades de detección y respuesta ante amenazas en aplicaciones web.

Se presenta un análisis exhaustivo de los conceptos fundamentales de Coraza, así como una evaluación de las diversas opciones de integración. Coraza, un firewall escrito en lenguaje Go compatible con OWASP CRS, se integra a WACE, un marco tecnológico diseñado para incorporar modelos de aprendizaje automático, enriqueciendo así la protección proporcionada por las reglas estáticas de OWASP CRS.

WACE mejora la protección de aplicaciones al añadir análisis predictivo haciendo uso de modelos de aprendizaje automático. La solución que se propone en este proyecto incluye un sistema de comunicación eficiente entre WACE y Coraza, con modos de ejecución tanto síncronos como asíncronos, además de herramientas para la recolección y visualización de métricas las que facilitan la monitorización y el análisis del rendimiento del sistema. Estas métricas no solo permiten evaluar el desempeño del sistema, sino que también ofrecen información crítica para su ajuste y mejora continua.

Palabras clave: Seguridad en aplicaciones Web, Web Application Firewalls, Aprendizaje Automático

Índice general

1. Introducción **1**
- 1.1. Objetivos . . . 2
- 1.2. Estructura del Documento . . . 2

2. Estado del Arte **3**
- 2.1. Web Application Firewall . . . 3
  - 2.1.1. HTTP . . . 4
  - 2.1.2. Reverse Proxy . . . 4
  - 2.1.3. ModSecurity . . . 4
  - 2.1.4. OWASP CRS . . . 8
  - 2.1.5. Coraza . . . 10
  - 2.1.6. Cloudflare WAF . . . 12
- 2.2. Proyectos relacionados . . . 12
  - 2.2.1. Uso de modelos de aprendizaje automático . . . 12
  - 2.2.2. Web Attack Classification Engine . . . 13
- 2.3. Métricas . . . 14
  - 2.3.1. OpenTelemetry . . . 15
  - 2.3.2. Prometheus . . . 16
- 2.4. Sistema de mensajes . . . 17

3. Análisis **19**
- 3.1. Problemática a abordar . . . 19
  - 3.1.1. Definición de excepciones . . . 20
  - 3.1.2. Modo de ejecución asincrónico . . . 20
  - 3.1.3. Métricas . . . 20
- 3.2. Requisitos . . . 21
  - 3.2.1. Requisitos funcionales . . . 21
  - 3.2.2. Requisitos no funcionales . . . 22
- 3.3. Punto de partida . . . 22
- 3.4. Alcance . . . 23
- 3.5. Alternativas de integración . . . 23
  - 3.5.1. Despliegue de WACE . . . 24
  - 3.5.2. Comunicación entre WACE y Coraza WAF . . . 25
  - 3.5.3. Mecanismos para la toma de decisión . . . 29
  - 3.5.4. Análisis de compatibilidad entre alternativas . . . 30
- 3.6. Configuración de excepciones . . . 31
  - 3.6.1. Definir un nuevo mecanismo . . . 31
  - 3.6.2. Adaptar la integración para utilizar Seclang . . . 32
- 3.7. Recolección de métricas . . . 36
- 3.8. Modelos asincrónicos . . . 36

4. Diseño **37**
4.1. Arquitectura . . . 37
4.1.1. Componentes . . . 38
4.2. Comunicación entre Servidor Web y Coraza WAF . . . 40
4.3. Integración con OWASP CRS . . . 41
4.4. Model Plugins . . . 41
4.4.1. Definición de interfaz de modelos . . . 42
4.4.2. Modos de ejecución . . . 43
4.5. Sistema de mensajes . . . 44
4.6. Flujo de análisis . . . 44
4.7. Flujo de chequeo . . . 45
4.8. Métricas . . . 46
4.9. Arquitectura de despliegue . . . 47

5. Implementación **49**
5.1. Lenguaje de programación . . . 49
5.2. Capa intermedia . . . 49
5.2.1. Compatibilidad con servidor Caddy . . . 51
5.2.2. Excepciones de modelos . . . 51
5.2.3. Integración con OWASP CRS . . . 51
5.3. WACE . . . 52
5.3.1. Plugins de modelo . . . 53
5.3.2. Plugins de decisión . . . 53
5.4. Integración con NATS . . . 54
5.5. Métricas . . . 54

6. Experimentación **57**
6.1. Ambiente . . . 57
6.2. Escenario de prueba . . . 58
6.3. Selección de métricas . . . 58
6.4. Resultados de ejecuciones . . . 59

7. Conclusiones y Trabajo Futuro **65**

A. Discusión del cumplimiento de requisitos en las distintas integraciones **71**
A.1. Despliegue de WACE . . . 71
A.1.1. WACE como librería . . . 72
A.1.2. WACE como servicio independiente . . . 72
A.2. Comunicación entre WACE y Coraza WAF . . . 72
A.2.1. Modificar el core de Coraza . . . 72
A.2.2. Generar operadores . . . 73
A.2.3. Modificar conectores . . . 74
A.2.4. Capa intermedia entre los conectores y Coraza . . . 75
A.3. Mecanismos para la toma de decisión . . . 76
A.3.1. Utilizar un operador . . . 76
A.3.2. Modificar el core de Coraza . . . 77
A.3.3. Utilizar variables y Seclang . . . 77
A.3.4. Utilizar los datos accesibles mediante la interfaz de Coraza y los resultados de los modelos . . . 77

Índice de figuras

2.1. Ejemplo de protocolo HTTP . . . 4
2.2. Concepto actual de fases . . . 5
2.3. Niveles de paranoia . . . 9
2.4. Flujo de ejecución de Coraza WAF . . . 11
2.5. Diagrama de fases Cloudflare WAF . . . 12
2.6. Arquitectura heredada de WACE . . . 13
2.7. Arquitectura de Prometheus . . . 16

3.1. Componentes actuales . . . 23
3.2. Flujo de ejecución esperado . . . 24
3.3. Arquitectura parcial de Coraza . . . 26
3.4. Modificar conectores . . . 27
3.5. Capa intermedia . . . 28
3.6. Ejecutar todos los modelos y sólo considerar un subconjunto. . . . 33
3.7. Concepto de estado y conjunto de reglas . . . 34
3.8. Mecanismo de excepciones con dos instancias de WAF . . . 35

4.1. Diagrama general de la solución . . . 38
4.2. Componentes de WACE . . . 39
4.3. Plugin de modelo sincrónico . . . 43
4.4. Plugin de modelo asincrónico . . . 44
4.5. Diagrama de flujo de la funcionalidad Analyze . . . 45
4.6. Diagrama de flujo de la funcionalidad Check . . . 46
4.7. Comunicación mediante OpenTelemetry Exporter . . . 46
4.8. Diagrama de despliegue . . . 47

5.1. Diagrama de los componentes implementados en la capa intermedia . . . 50

6.1. Tiempo de respuesta en función de tiempo de la prueba . . . 60
6.2. Aproximación logarítmica del tiempo de respuesta en función de tiempo de la prueba 61
6.3. Dashboard de métricas para una carga con 10 usuarios utilizando plugins de modelo con NATS . . . 62

Índice de cuadros

2.1. Puntajes de cada severidad 9

3.1. Análisis de compatibilidad entre alternativas de comunicación y mecanismos para la toma de decisión. 30

5.1. Métricas WACE_WAF 54
5.2. Métricas WACE Core 54

6.1. Ambiente de pruebas 57
6.2. Esquema de pruebas 58
6.3. Promedio de tiempos de respuesta 59
6.4. Percentiles 59

Glosario

API Application Programming Interface. 24, 54

FIFO First In, First Out. 36

gRPC Google Remote Procedure Calls. 23, 24, 39

GSI Grupo de Seguridad Informática. 1

HTTP Hypertext Transfer Protocol. 3, 4, 22, 27, 38, 40, 41

ModSecurity Tecnología de WAF open source más utilizada en el mercado. 1, 3–6, 8, 10–13, 34

NATS Neural Autonomic Transport System. 44, 45, 49, 54, 57, 58, 60, 66

OWASP Coraza WAF Tecnología de WAF open source desarrolla en el lenguaje Go. 1–3, 10, 65

OWASP CRS Conjunto genérico de reglas de detección de ataques web de OWASP. v, 1–3, 8–10, 13, 19–22, 27, 29, 30, 34, 35, 37–41, 48, 49, 51–53, 57, 65, 66, 74–77

WACE Web Attack Classification Engine. 1, 2, 19, 21–31, 35–43, 48, 49, 51, 52, 54, 65, 66, 71, 72

WAF Web Application Firewall. 1–4, 6–8, 19–23, 25–29, 31–36, 38, 41, 45, 46, 51, 72

WSL Windows Subsystem for Linux. 57

XSS Cross Site Scripting. 58

Capítulo 1

Introducción

En la actualidad, las aplicaciones web se han convertido en un elemento central en la vida cotidiana de personas y organizaciones, siendo fundamentales en redes sociales, plataformas de entretenimiento y en servicios críticos como sistemas financieros y de salud. A medida que la importancia de estas aplicaciones aumenta, también lo hace el interés de actores maliciosos por vulnerar dichos sistemas.

En este contexto, los Firewalls de Aplicación Web (WAF [40]) representan una primera línea de defensa crucial, al interceptar el tráfico web mediante reglas que identifican y bloquean comportamientos sospechosos. Aunque esta tecnología resulta esencial en la protección de estos sistemas, la mayoría de los WAF open source utilizados actualmente dependen de reglas estáticas, por lo que la detección que ofrecen está limitada por la capacidad de generar reglas de detección efectivas que posea el equipo de expertos de cada organización protegida.

Para abordar sistemáticamente las amenazas a las que se encuentran expuestas las aplicaciones web surge OWASP CRS [37], un conjunto de reglas de detección diseñado para ofrecer protección contra una amplia variedad de ataques conocidos. A pesar de su eficacia en la protección de aplicaciones, las reglas de OWASP CRS no capturan las particularidades del tráfico web específico de cada organización, por lo que su uso da lugar a falsos positivos. Además, la naturaleza rígida de estas reglas limita su capacidad para detectar ataques desconocidos, lo que representa un riesgo significativo ante nuevas amenazas emergentes.

En el ámbito de los WAF open source que utilizan OWASP CRS, ModSecurity [38] es el estándar de facto. Sin embargo, el anuncio de la organización encargada de su soporte hasta el presente año (Trustwave) [50] sobre el cese del soporte oficial para este WAF ha llevado a la búsqueda de alternativas. La tecnología OWASP Coraza WAF [11], una implementación moderna en lenguaje Go que es compatible con OWASP CRS, se está posicionando como un sólido reemplazo dentro del ecosistema de WAF open source.

El presente proyecto está enmarcado dentro de la línea de investigación WAFMind [23] del Grupo de Seguridad Informática (GSI) del Instituto de Computación (InCo) de la Facultad de Ingeniería de la Universidad de la República. En el contexto de WAFMind se han desarrollado diversos proyectos enfocados en mejorar las capacidades de detección de los WAF utilizando técnicas de aprendizaje automático. En nuestro proyecto, se continúa el trabajo realizado en el proyecto ModSecIntl [1], que culminó en la creación de un Producto Mínimo Viable (MVP) del Web Attack Classification Engine (WACE) [34]), una tecnología que posibilita la integración de modelos de aprendizaje automático para complementar la protección ofrecida por OWASP CRS. El objetivo de WACE es no sólo detectar patrones de ataques conocidos, sino también identificar ataques emergentes que no están contemplados por las reglas estáticas, mientras se busca reducir los falsos positivos mediante un análisis contextual del tráfico de la organización. Los modelos que se integran en WACE ejecutan en paralelo al análisis de las reglas de OWASP CRS y se utiliza la información de ambas fuentes para tomar una decisión de bloqueo.

WACE busca aprovechar los beneficios de los modelos de aprendizaje automático, pero sin perder de vista que los modelos también cuentan con falsos positivos, por ello en el contexto de este trabajo se desea que WACE cuente con un mecanismo que permita exceptuar la ejecución de determinados modelos. Además se tiene en cuenta que dichos modelos pueden tener un alto tiempo de análisis, para lo que se propone el modo de ejecución asíncrono.

Considerando que estas soluciones pueden proteger sistemas críticos, es fundamental contar con herramientas de monitorización activa y recolección de métricas. Por esta razón, uno de los propósitos de este trabajo es que WACE recopile métricas detalladas en tiempo real sobre su desempeño en la protección de aplicaciones y el estado del WAF.

1.1. Objetivos

El objetivo principal de este proyecto es contribuir al desarrollo de WACE, enfocándose en los siguientes aspectos:

- **Integración con OWASP Coraza WAF:** Establecer mecanismos de comunicación entre WACE y el WAF open source Coraza, garantizando la sincronización entre ambos componentes y permitiendo una toma de decisiones conjunta basada en los resultados de Coraza y los modelos de aprendizaje automático.
- **Flexibilidad en los criterios de decisión:** Facilitar la creación y configuración de algoritmos de decisión que integren los resultados de OWASP CRS con los modelos de aprendizaje automático, permitiendo a cada usuario ajustar los criterios según sus necesidades específicas.
- **Eficiencia:** Diseñar e implementar una solución que optimice el uso de recursos y minimice el tiempo de procesamiento.
- **Recolección y visualización de métricas:** Desarrollar un sistema de recolección de métricas que provea información sobre el rendimiento de la herramienta e incorpore herramientas para visualizar estos datos de forma efectiva.
- **Modos de ejecución de modelos:** Crear mecanismos que permitan la ejecución de modelos de aprendizaje automático tanto en tiempo real como de manera asincrónica en relación con el procesamiento de las solicitudes gestionadas por el WAF.

1.2. Estructura del Documento

El resto del documento está organizado de la siguiente manera: el capítulo Estado del Arte presenta una revisión del contexto actual del proyecto, abordando las tecnologías, enfoques relevantes y sus limitaciones. Posteriormente, en el capítulo Análisis, se describe el problema a resolver junto con los requisitos funcionales y no funcionales necesarios, considerando diversas opciones y detallando las decisiones tomadas para asegurar una integración eficiente y escalable. El capítulo Diseño presenta la arquitectura propuesta de la solución, describiendo los componentes principales, sus interacciones y flujos definidos. Más adelante en el capítulo Implementación se detallan las decisiones técnicas adoptadas durante el desarrollo, abarcando la integración de componentes y la estructura interna del sistema. El capítulo Experimentación describe las pruebas realizadas sobre la herramienta y analiza sus resultados. Finalmente, en el capítulo Conclusiones y Trabajo Futuro se resumen los objetivos alcanzados, así como los problemas enfrentados y se discuten las líneas de trabajo futuro que podrían desarrollarse a partir de este proyecto.

Capítulo 2

Estado del Arte

En este capítulo se presenta una revisión del estado del arte relacionado con los WAFs, antecedentes que motivan y fundamentan el proyecto actual, así como tecnologías para la gestión de rendimiento de aplicaciones y sistemas de mensajería para sistemas distribuidos. En cada sección se abordarán tanto las tecnologías específicas y sus fundamentos, como su relevancia en el contexto del trabajo. El objetivo es establecer una base sólida para comprender las herramientas a utilizar y los desafíos que se presentan en cada área.

2.1. Web Application Firewall

En esta sección se abordará el concepto de Web Application Firewall (WAF [40]), para poder comprender este tema se introducirá también conceptos base como lo son el protocolo HTTP utilizado por las aplicaciones web y las nociones de Proxy, Reverse Proxy y Virtual Host que facilitarán la comprensión del escenario estándar en el que se despliegan los WAF. Por otra parte se presentarán las principales implementaciones de código abierto, como ModSecurity y una alternativa moderna como lo es OWASP Coraza WAF. Además, se explicará el proyecto OWASP CRS en detalle, cuyo conjunto de reglas es ampliamente utilizado en las soluciones mencionadas anteriormente. Finalmente, se explorarán herramientas comerciales, como Cloudflare WAF, destacando sus enfoques innovadores en funcionalidades específicas.

Un WAF es una herramienta que intercepta y analiza las solicitudes HTTP/HTTPS de un cliente a un servidor web y sus respuestas, con el objetivo de filtrar aquellos que pueden potencialmente presentar un riesgo para la seguridad del sistema que protegen. Operan mediante un conjunto de reglas que tienen el fin de proteger contra vulnerabilidades en la aplicación al filtrar el tráfico malicioso. Una gran ventaja de este tipo de herramientas radica, en parte, en la velocidad y facilidad con que se puede generar una protección contra vulnerabilidades conocidas, permitiendo una rápida respuesta ante diversos vectores de ataque. Esto se debe a que no es necesario modificar el código de las aplicaciones, lo que implica no tener que pasar por el proceso de control de cambios definido para la aplicación. Esta técnica se conoce como “Virtual Patching”.

Por otro lado, la administración de las reglas de filtrado puede volverse compleja, dado que se deben generar reglas que sean lo suficientemente genéricas para filtrar los distintos tipos de ataque pero al mismo tiempo es necesario que se adapten a las aplicaciones que están protegiendo, con el fin de no generar bloqueos ante tráfico no malicioso (falsos positivos).

Como parte de comprender la administración de un WAF en general, y en particular de sus reglas de filtrado, en la subsección 2.1.1 se describe las características principales del protocolo HTTP, que es el protocolo que utilizan las aplicaciones web.

2.1.1. HTTP

El protocolo HTTP [17] utiliza mensajes para llevar a cabo la comunicación entre cliente y servidor. Cada uno de los mensajes debe seguir un formato, tanto para las solicitudes como para las respuestas. En la figura 2.1 se presenta un ejemplo de intercambio de mensajes HTTP, a la izquierda se presenta una solicitud que utiliza el método GET. Este método indica que se debe recuperar la información asociada al recurso. A la derecha se presenta la respuesta asociada a solicitud, primero notar que el código de respuesta es 200 OK, lo que quiere decir que la solicitud fue procesada exitosamente.

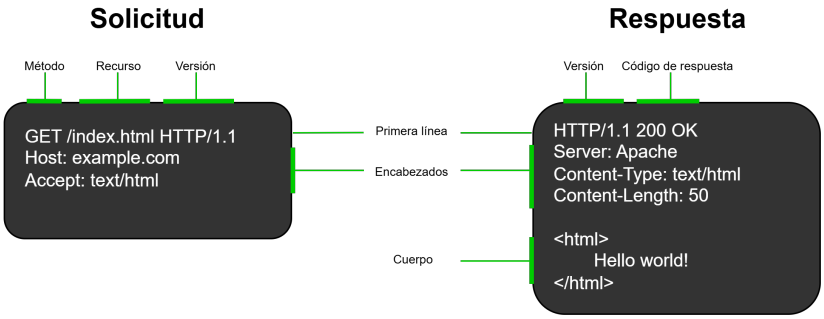


Figura 2.1: Ejemplo de protocolo HTTP

Así como las solicitudes procesadas exitosamente tienen asociado el código de respuesta 200, las solicitudes que generan algún tipo de error en su procesamiento tienen un código de respuesta asociado, en particular, el código de respuesta 403 indica que el acceso a un recurso está prohibido. Este código es comúnmente utilizado en las respuestas HTTP cuando el WAF toma la decisión de bloquear una transacción.

En la próxima sección se introducirá el concepto de Reverse Proxy, que permitirá entender el escenario en el que se suelen utilizar los WAFs.

2.1.2. Reverse Proxy

Cuando se habla de redes o tráfico web, un proxy es un dispositivo que se encarga de realizar solicitudes a nombre de otro equipo, el proxy se sitúa entre el cliente y el servidor. En particular un Reverse Proxy[5] se coloca frente a un servidor, siendo uno de sus propósitos el realizar controles de seguridad antes de que las solicitudes sean reenviadas al servidor destino. En particular en el contexto de aplicaciones web y WAFs, se aprovecha el concepto de reverse proxy para interceptar los pedidos web de los clientes, realizar el análisis del WAF y posteriormente enviar dicho pedido al servidor destino (si no corresponde bloquearlo). A este último servidor que posee desplegada la aplicación web cuyos recursos son solicitados, se le denomina servidor backend.

En este tipo de proxies es común definir lo que se conoce como “Virtual Host”[6], que es una técnica que permite que un mismo equipo físico aloje más de un sitio web en la misma dirección IP y el mismo puerto, el proxy define qué virtual host debe atender el pedido utilizando el nombre de dominio que se le solicitó en el pedido HTTP.

El servidor HTTP Apache [12] es un ejemplo de una tecnología que puede ser utilizada como reverse proxy, junto al WAF open source ModSecurity permiten realizar análisis de solicitudes y respuestas HTTP.

2.1.3. ModSecurity

ModSecurity es el estándar de facto en lo que refiere a WAF open source [38]. Originalmente estaba diseñado para funcionar como un módulo del servidor HTTP de Apache. Actualmente ha

evolucionado para proveer filtrado de solicitudes y respuestas HTTP en distintas plataformas, incluyendo Apache HTTP Server, Microsoft IIS y Nginx. Es de uso libre bajo la "Apache license 2.0"[2] y continúa siendo el motor de WAF open source más utilizado por empresas, organizaciones gubernamentales y proveedores de internet según [42].

A continuación se presentarán los principales conceptos que implementa ModSecurity: en primera instancia se hablará sobre las Fases que son los momentos en los cuales se va a realizar el análisis de las distintas partes del contenido de los paquetes del protocolo HTTP [17]. Posteriormente se introduce el lenguaje ModSecurity Rule Language (Seclang), que es el lenguaje en el que se definen las reglas de la política de filtrado.

2.1.3.1. Transacción

Una transacción se refiere a cada pedido de solicitud y respuesta HTTP entre un cliente y el servidor web.

En ModSecurity existe el concepto de fase, que hace referencia a una etapa del análisis sobre una transacción HTTP. Las fases se definen según la disponibilidad que se tiene de los datos de una transacción. En la figura 2.2 se ilustran las primeras cuatro fases del flujo y los datos que cada una de ellas posee disponible para procesar. Se enumeran las fases presentes en la figura junto con una quinta fase no dedicada al análisis de la transacción:

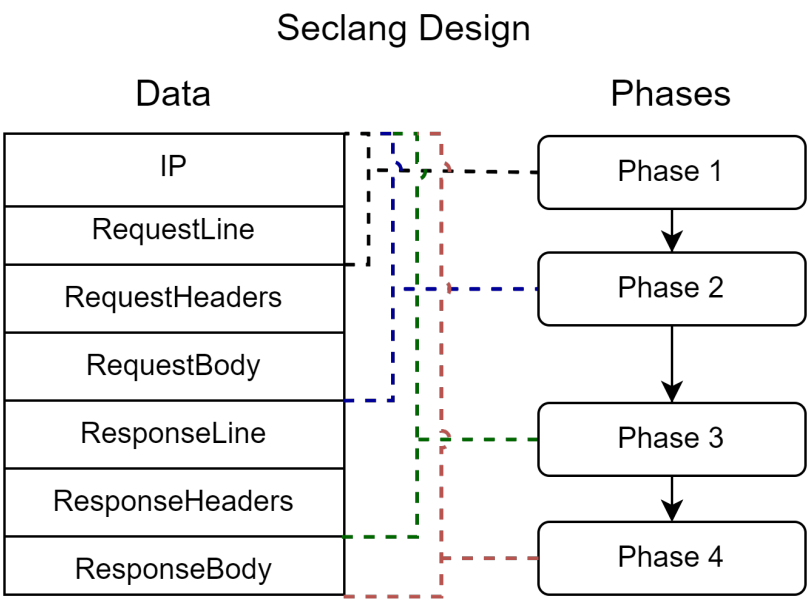


Figura 2.2: Concepto actual de fases

- La fase 1 está asociada con el análisis de los encabezados del pedido HTTP.
- La fase 2 está asociada con el análisis del contenido del cuerpo del pedido HTTP.
- La fase 3 está asociada con el análisis de los encabezados de la respuesta HTTP.
- La fase 4 está asociada con el análisis del contenido del cuerpo de la respuesta HTTP.
- La fase 5 está asociada al proceso de registrar los logs del análisis anterior.

2.1.3.2. ModSecurity Rule Language

Como se mencionó previamente, ModSecurity implementa el ModSecurity Rule Language [41] (también denominado Seclang) que es un lenguaje específico de un dominio (DSL por sus siglas en inglés) [13] especializado para trabajar con los datos de transacciones HTTP. Este lenguaje está basado en directivas, se tienen dos grandes tipos de directivas: aquellas enfocadas en la configuración del WAF y las directivas que realizan el filtrado de las transacciones (se destaca en particular la directiva SecRule), estas últimas hacen uso de variables, operadores y acciones:

- **Variables:** son elementos de la transacción a analizar, como lo pueden ser el recurso o los encabezados de un pedido.
- **Operadores:** son funciones booleanas que son ejecutadas sobre las variables. Por ejemplo, es posible detectar la presencia de una determinada cadena de caracteres. A continuación, se muestra un ejemplo utilizando el lenguaje de una regla que detecta ".php" en cualquier parte de la línea de un pedido HTTP:

```
SecRule REQUEST_LINE "@contains .php" "id,phase,deny"
```

- **Acciones:** Las directivas de filtrado poseen una lista de acciones que en combinación definen su comportamiento. Existen distintos tipos de acciones[30], estos tipos son:
  - Disruptivas
  - Metadata
  - No disruptivas
  - Flujo
  - Datos

  Si bien se cuenta con cinco tipos de acciones, se entrará en detalle sobre las primeras dos debido a que son las relevantes para el proyecto.

  Las acciones disruptivas indican la acción a tomar en caso de que la condición del operador en la regla sea evaluada positivamente sobre la transacción (se produjo un match de la regla). Estas acciones pueden ser:

  - **allow:** detiene la evaluación de las reglas de filtrado sobre la transacción y permite que continúe con su ejecución normal.
  - **block:** realiza la acción disruptiva definida en la directiva **SecDefaultAction**, la cual se describe en detalle más adelante.
  - **deny:** detiene el procesamiento de la transacción y devuelve un código de error al cliente.
  - **pass:** continúa procesando la siguiente regla a pesar de la coincidencia exitosa de la regla.
  - **redirect:** detiene el procesamiento de las reglas y envía una redirección al cliente.

  Las acciones de metadata indican información asociada a la regla como por ejemplo: un identificador único (ID) o la fase de ejecución.

Presentados los componentes de las directivas se desarrollará sobre aquellas que se consideran con mayor relevancia en el contexto del proyecto:

- **SecRuleEngine:** Define el comportamiento del motor de reglas. Pertenece a las directivas que regulan la configuración del WAF.

```
SecRuleEngine On|Off|DetectionOnly
```

Los posibles valores son:

- **On**: Evalúa las reglas normalmente.
- **Off**: No evalúa las reglas definidas.
- **DetectionOnly**: Evalúa las reglas pero no ejecuta ninguna acción disruptiva.

- **SecDefaultAction:** Define una lista de acciones por defecto para una fase en particular. Pertenece a las directivas que regulan la configuración del WAF.

```
SecDefaultAction "phase,disruptiveAction,action3,..."
```

Toda directiva de este tipo debe contener una fase de ejecución y una acción disruptiva. Además, **no** puede contener otras acciones de metadata, como por ejemplo la acción "id". Cada regla cuya fase coincida con la de una SecDefaultAction tendrá incluida la lista de acciones que define la acción por defecto, a menos que la mencionada regla sobrescriba alguna acción en su propia lista.

- **SecAction:** ejecuta incondicionalmente la lista de acciones recibidas como parámetro:

```
SecAction "id,phase,action3,..."
```

Toda directiva SecAction debe tener definido un id.

- **SecRule:** Directiva que crea una regla de filtrado que ejecutará un conjunto de acciones si se cumple una expresión que está dada por una lista de variables y un operador:

```
SecRule variable|variable|... operator
↪  "id,phase,disruptiveAction,action4,..."
```

Toda regla de filtrado debe tener definido un id. Además, cada regla debe proveer una o más variables, junto al operador que será usado para inspeccionarlas. Si no se proveen acciones se utilizará la lista de acciones por defecto (siempre hay una lista de acciones por defecto, incluso cuando no se definió explícitamente con la directiva "SecDefaultAction"). Si hay acciones especificadas en una regla, serán tomadas junto con la lista por defecto para formar la lista final de acciones que se ejecutarán en dicha regla, se debe tener en cuenta que se pueden sobreescribir acciones. Las acciones que se pueden sobreescribir son aquellas que sólo pueden aparecer una vez en cada regla, como por ejemplo las acciones disruptivas. Por otro lado, aquellas que pueden tener múltiples apariciones, serán insertadas al final de la lista.

- **SecRuleUpdateActionById:** Modifica la lista de acciones de la regla que se especifica mediante un identificador único que deben tener presentes todas las reglas:

```
SecRuleUpdateActionById id "action1,action2,action3,..."
```

La lista final de acciones a llevar adelante finalmente para la regla modificada, sigue el mismo mecanismo descrito para la directiva SecRule.

- **SecRuleRemoveById:** Elimina de la lista de reglas a ejecutar a la regla cuyo identificador se especifica mediante "id":

```
SecRuleRemoveById id
```

2.1.4. OWASP CRS

OWASP CRS [37] es un conjunto de reglas de detección de ataques genéricos para utilizar con ModSecurity o con WAFs compatibles. El OWASP CRS tiene como objetivo proteger a las aplicaciones web de un amplio rango de ataques, incluyendo parte del OWASP Top Ten[39], con un mínimo de falsos positivos. El OWASP Top Ten define mediante un amplio consenso diez categorías en las cuales las aplicaciones web presentan mayor riesgo de seguridad. Actualmente, el ultimo ranking corresponde a 2021, y las primeras tres categorías son las siguientes:

1. Problemas de control de acceso: el control de acceso aplica políticas que impiden que los usuarios actúen fuera de los permisos previstos. Las fallas suelen provocar la divulgación no autorizada de información, la modificación o la destrucción de todos los datos.
2. Fallas criptográficas: fallas relacionadas con la criptografía que a menudo conducen a la exposición de datos confidenciales o al compromiso del sistema.
3. Inyección: algunas de las inyecciones más comunes son SQL, NoSQL, comandos del sistema operativo, mapeo relacional de objetos (ORM), LDAP o Cross-site Scripting (XSS).

A lo largo de esta sección se presentará el funcionamiento de OWASP CRS en la protección de aplicaciones web.

Funcionamiento

OWASP CRS cuenta con dos modos de ejecución, el *modo tradicional* y el *modo de puntuación de anomalías*. El modo de detección tradicional[8] es el antiguo modo operativo por defecto, es además el modo de funcionamiento más básico, en el que todas las reglas son "autónomas". En este modo no se comparte información entre las reglas y cada regla no conoce sobre las coincidencias de reglas anteriores. Es decir, si una regla se dispara, ejecutará la acción disruptiva que tenga asociada.

Por otro lado, el *modo de puntuación de anomalías* (anomaly scoring mode) es un modo de ejecución que asigna un valor numérico a las transacciones HTTP que representa que tan anómalas aparentan ser. Este valor se corresponde con la suma de la severidad asignada a cada una de las reglas que coincidieron, próximamente se profundizará sobre este concepto.

Mediante el modo de puntuación de anomalías es posible detectar comportamientos sospechosos en las solicitudes HTTP y sus respuestas, sin realizar bloqueo en el momento que una regla de filtrado se evalúa positivamente. OWASP CRS es capaz de identificar cientos de patrones de ataque, utilizando sus reglas como mecanismo de detección e inspección, dichas reglas acumulan puntajes que son utilizados para decir si realizar el bloqueo de la transacción.

Severidad y umbrales de bloqueo

Cada regla de OWASP CRS está asociada a un nivel de severidad, que determina cuánto incrementa el puntaje de anomalía al activarse. Los niveles de severidad y sus puntajes predeterminados pueden observarse en la tabla 2.1.

Esto significa que una regla crítica podría, por sí sola, acercar el puntaje de anomalía al umbral de bloqueo. Sin embargo, si por ejemplo se aumenta el umbral por defecto de 5 a un umbral de 7, requeriría más de una regla para activar el bloqueo.

Severidad	Puntaje
CRITICAL	5
ERROR	4
WARNING	3
NOTICE	2

Tabla 2.1: Puntajes de cada severidad

OWASP CRS contiene cientos de reglas divididas en diferentes fases y niveles de paranoia, que aumentan o disminuyen la sensibilidad de la detección. La complejidad de OWASP CRS radica en que maneja un gran número de patrones de ataque, y cada regla se activa según la configuración elegida, sumando al puntaje de anomalía total cada vez que identifica un patrón sospechoso.

Niveles de paranoia (Paranoia levels)

El nivel de paranoia (PL) [7] permite definir qué tan agresivo es el OWASP CRS. El nivel de paranoia 1 (PL1) proporciona un conjunto de reglas que casi nunca generan falsos positivos (idealmente nunca, aunque puede suceder, dependiendo de la configuración local). El PL2 proporciona reglas adicionales que detectan más ataques que PL1 (estas reglas incluyen las reglas de PL1 y agregan nuevas), pero existe la posibilidad de que estas reglas adicionales también generen coincidencias sobre transacciones legítimas.

En PL3 se agregan más reglas a PL1 y PL2, en particular para ciertos ataques especializados. Esto lleva a un aumento de falsos positivos. Luego, en PL4, las reglas son tan agresivas que detectan casi todos los posibles ataques, aunque también marcan una gran cantidad de tráfico legítimo como malicioso.

En la siguiente figura se presenta un diagrama de Venn que representa los conjuntos de reglas de cada nivel de paranoia y como a medida que se aumenta el nivel, es de esperar que aumenten los falsos positivos.

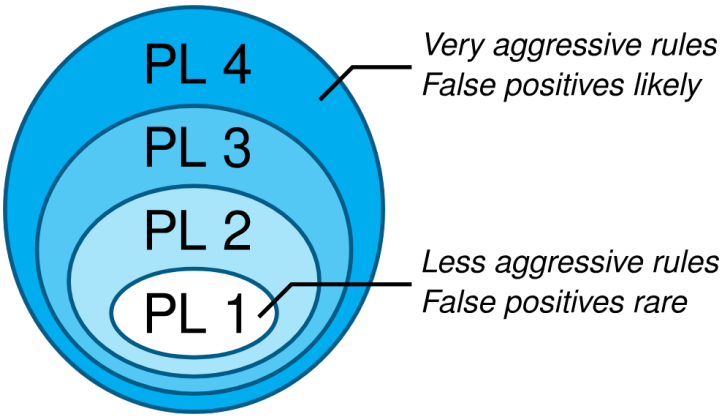


Figura 2.3: Niveles de paranoia
Imagen extraída de: [7]

Introducidos los principales conceptos utilizados por OWASP CRS, se explicará el flujo estándar de ejecución.

Flujo de ejecución del modo de puntuación de anomalías

OWASP CRS define un conjunto de variables donde se va acumulando los resultados del análisis de las reglas, al contenido de éstas variables se les denomina "puntaje". Se tiene el siguiente flujo para aplicar el mecanismo de puntuación.

1. **Ejecución de reglas de solicitud:** Cada solicitud se somete a las reglas de OWASP CRS que están configuradas para detectar patrones de ataque. Cada coincidencia en una regla aumenta el puntaje de anomalía.
2. **Evaluación de bloqueo de solicitudes:** Si el puntaje de anomalía supera el umbral de puntuación de anomalía de entrada ("tx.inbound_anomaly_score_threshold"), el WAF bloquea la solicitud.
3. **Ejecución de reglas de respuesta:** Las reglas de OWASP CRS también analizan las respuestas enviadas al cliente.
4. **Evaluación de bloqueo de respuestas:** Si el puntaje de anomalía supera el umbral de puntuación de anomalía de salida ("tx.outbound_anomaly_score_threshold"), el WAF impide que la respuesta llegue al cliente.

Configuración y manejo de falsos positivos

Uno de los desafíos del OWASP CRS es minimizar falsos positivos. Los falsos positivos ocurren cuando el conjunto detecta un patrón en una solicitud legítima que coincide con una regla de ataque. Para evitar bloqueos innecesarios, se pueden configurar excepciones específicas para estos casos:

- **Ajuste de Umbrales:** se pueden ajustar los umbrales de puntaje para hacer el OWASP CRS más o menos sensible según el tipo de aplicación y los patrones esperados de tráfico.
- **Ajuste de Niveles de Paranoia:** un nivel de paranoia más bajo reduce la cantidad de reglas activas, lo que puede disminuir los falsos positivos.
- **Excepciones de Reglas (Whitelisting):** se pueden crear excepciones específicas en OWASP CRS para permitir que ciertas solicitudes pasen sin activar una regla determinada. Esto se logra mediante reglas personalizadas que omiten reglas para ciertos parámetros, encabezados o URI particulares.

Opción de bloqueo temprano (Early Blocking)

OWASP CRS en su versión 4 permite un bloqueo temprano (Early Blocking) de manera opcional, que se puede configurar para bloquear solicitudes o respuestas en fases tempranas si ya han alcanzado el umbral de anomalía. Esto ahorra recursos, ya que evita la ejecución de reglas adicionales en fases posteriores, pero también tiene la desventaja de que puede ocultar información adicional sobre el tráfico que podría haberse registrado en las fases finales de la evaluación.

Además de ModSecurity existen otras implementaciones de WAF que son compatibles con el conjunto de reglas de OWASP CRS, un ejemplo es Coraza WAF, sobre el que se desarrollará la próxima sección.

2.1.5. Coraza

OWASP Coraza WAF [36] es un framework de WAF escrito en el lenguaje de programación Go, que soporta el mecanismo de configuración Seclang de ModSecurity presentado anteriormente y es 100 % compatible con el OWASP CRS. Puede ser importado como una librería o integrarse mediante conectores para ser utilizado con servidores web como Caddy[4].

Se analizará en particular el flujo de ejecución de Coraza y sus capacidades de extensión.

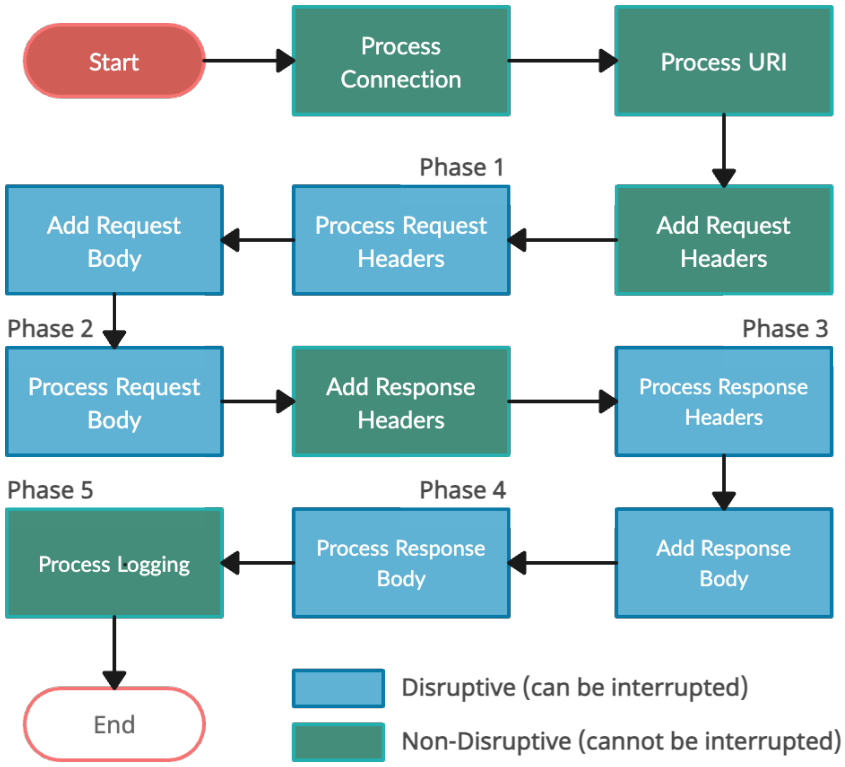


Figura 2.4: Flujo de ejecución de Coraza WAF.
Imagen extraída de [36].

2.1.5.1. Flujo de ejecución

Coraza posee incluido en su flujo de funcionamiento las mismas fases de análisis que ModSecurity. Además de eso, en Coraza se procesa la conexión y la URI previo a la fase 1, y entre fases se agregan los datos para poder ser analizados en la fase posterior. Por otro lado, existen otros WAFs que poseen distintas fases de ejecución como es el caso del WAF de Cloudflare, el cual es descrito en la sección 2.1.6.

Volviendo a Coraza, cada transacción procesada atraviesa el flujo que se muestra en la figura 2.4. Existen distintas etapas de la transacción que pueden ser interrumpidas mediante una acción disruptiva, en el esquema se presentan en color celeste, y otras que no pueden serlo, presentadas en color verde.

2.1.5.2. Posibilidades de extensión

Dentro de Coraza existe el concepto de plugin. Los plugins permiten la extensión de las funcionalidades de Coraza[35]. Actualmente, se permite extender las funcionalidades mediante la generación de operadores, acciones y transformaciones. Dichos componentes poseen la misma semántica que fue descrita en la sección 2.1.3.2.

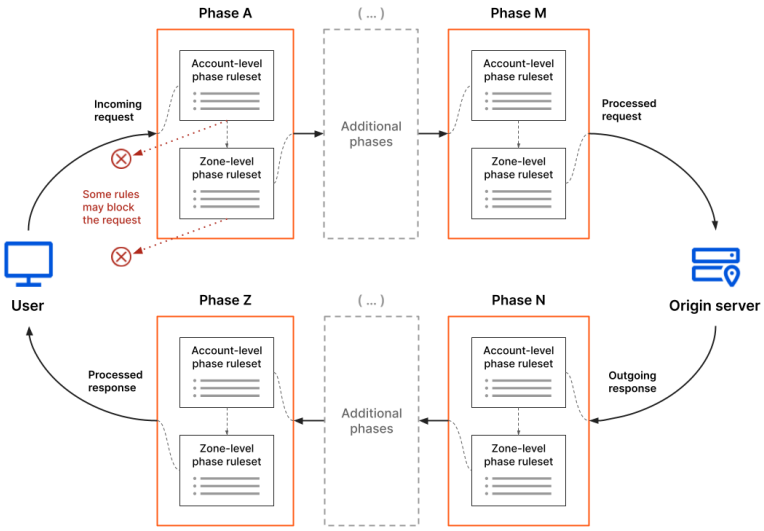


Figura 2.5: Diagrama de fases Cloudflare WAF.
Imagen extraída de [21].

2.1.6. Cloudflare WAF

Cloudflare es un proveedor servicios de CDN (Content Delivery Network), Ciberseguridad en la nube, protección contra denegación de servicio y más.

Entre sus productos se encuentra un WAF [22] que es la "piedra angular" dentro de las soluciones de seguridad que ofrecen. Posee un motor de reglas que es utilizado por distintos productos de la compañía, permitiendo configurar varios productos utilizando la misma sintaxis. Este motor define fases [21], que representan una etapa en el procesamiento de la transacción y "Rulesets" que, como lo indica su nombre, definen un conjunto de reglas con una versión asociada y que ejecutan en una fase en particular.

Se presenta en la figura 2.5 la estructura general de las fases de Cloudflare WAF. Como se evidencia en dicha figura, las fases difieren del esquema de ModSecurity y Coraza en cantidad y distribución. Además el usuario sólo puede ingresar reglas personalizadas en ciertas fases, en contraposición al otro esquema donde el usuario tiene control total de las reglas. Por otra parte las fases existen en dos niveles, a nivel "account" y a nivel "zone". Para la misma fase, las reglas definidas en el nivel "account" son evaluadas antes que las definidas en el nivel "zone".

2.2. Proyectos relacionados

Ya introducidos los conceptos y tecnologías principales que permiten comprender y contextualizar a los WAFs, se introducirán proyectos de investigación relacionados a la temática del presente trabajo. Primero se introducirán aquellos proyectos que fundamentan la utilización de modelos de aprendizaje automático, que son una importante motivación al proyecto. Posteriormente se presentan los proyectos que han tenido participación en el desarrollo de WACE.

2.2.1. Uso de modelos de aprendizaje automático

Anteriormente se han realizado trabajos de investigación en los que se ha explorado el uso de modelos de aprendizaje automático para la detección de ataques web en diferentes escenarios

dependiendo de los datos con los que se cuente[3] y por otra parte trabajos donde se ha profundizado en explorar nuevas técnicas de aprendizaje automático[31] como el uso de *transformers*.

En ambos trabajos se logró mejorar los resultados de clasificación de ataques con respecto al uso de ModSecurity junto con OWASP CRS, particularmente en la disminución de falsos positivos (tráfico legítimo) y también en el incremento de detección de verdaderos positivos (ataques). Para adaptarse y disminuir dichos falsos positivos estos modelos se pueden entrenar con registros históricos de tráfico válido de la organización, este tráfico es fácil de generar y obtener dado que se obtiene al realizar un uso normal de las aplicaciones web.

Resulta entonces de interés el contar con herramientas que permitan la integración de modelos de aprendizaje automático con tecnologías de WAF.

2.2.2. Web Attack Classification Engine

Web Attack Classification Engine (WACE) es un software con el objetivo principal de construir un WAF asistido por mecanismos de aprendizaje automático para la identificación, análisis y prevención de ataques en aplicaciones web. La idea central es combinar el poder de clasificación de los modelos de aprendizaje automático con el conocimiento integrado en la especificación del OWASP CRS utilizado con por los WAFs con el objetivo de detectar ataques, mientras que se reducen los falsos positivos.

En su origen [34] la herramienta fue desarrollada en el lenguaje C y luego extendida en [49], donde se cambió también a una implementación en el lenguaje Go.

En la figura 2.6 se presenta la arquitectura a alto nivel de la solución, se puede apreciar además la posibilidad de interaccionar con distintos WAFs a través de una única instancia. A continuación se describirá brevemente la arquitectura y las características principales de la solución.

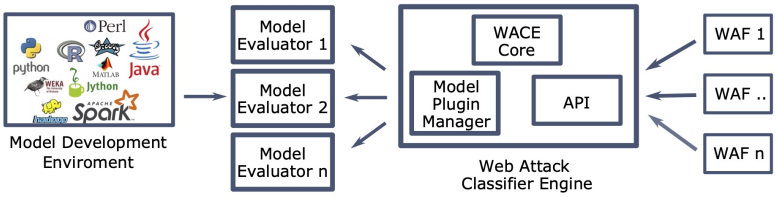


Figura 2.6: Arquitectura heredada de WACE.
Imagen extraída de [49].

2.2.2.1. Arquitectura

WACE posee una arquitectura basada en plugins. Este tipo de arquitectura permite la integración de diferentes modelos de aprendizaje automático y criterios de decisión mediante la implementación de plugins. Esto facilita que se agreguen modelos nuevos de clasificación o criterios de decisión sin la modificación del código del core de WACE.

2.2.2.2. Plugins

WACE cuenta con dos tipos de plugin:

- Los plugins de modelo que implementan la integración con modelos de aprendizaje automático.
- Los plugins de decisión que implementan el algoritmo que toma la decisión de bloquear la transacción en base a los distintos resultados obtenidos en el análisis por parte de OWASP CRS y los modelos.

2.2.2.3. Integración con WAFs

La implementación de WACE está desacoplada de la de sus WAFs clientes, para lo que WACE implementa un protocolo de comunicación. La idea central es que cualquier cliente que respete los mensajes definidos en el protocolo de comunicación sin importar su implementación pueda comunicarse y hacer uso de WACE. El mecanismo de comunicación definido es implementado mediante gRPC [16] y Protobuf [14].

2.3. Métricas

En esta sección se abordará la cuestión asociada con las métricas, que son un insumo fundamental para evaluar el estado de un sistema.

En el contexto de este proyecto, serán una pieza clave para conocer el rendimiento de la herramienta y las aplicaciones que ésta protege. Por otro lado, son un insumo para la mejora de la defensa realizada, ya que se puede utilizar para conocer el tiempo que tarda cada componente en procesar, lo que permite detectar puntos de mejora. En esta línea, existen herramientas que permiten recolectar métricas que brindan información acerca de las aplicaciones. Algunas de estas herramientas están concebidas desde su diseño para no solo brindar información acerca del estado de la aplicación, sino además de la infraestructura subyacente. Cuando se pone foco únicamente en el rendimiento de las aplicaciones por lo general se habla de herramientas de gestión de rendimiento de aplicaciones (APM por sus siglas en inglés), mientras que cuando se trata además la infraestructura subyacente se introduce el concepto de observabilidad. En este contexto, entraremos en detalle sobre la solución de APM.

APM es el proceso de utilizar herramientas de software y datos de telemetría para monitorear el rendimiento de las aplicaciones. APM ayuda a mantener el nivel de servicio esperado, ofreciendo información en tiempo real del estado de salud de las aplicaciones. Con esta información, los equipos encargados de dar soporte a las aplicaciones pueden detectar, analizar y solucionar de manera eficiente los problemas que puedan surgir.

APM utiliza métodos, técnicas y herramientas para realizar monitoreo del estado de salud de las aplicaciones que están en el alcance de la plataforma, así como detectar, diagnosticar y resolver problemas relacionados al rendimiento de la aplicaciones.

Para detectar los problemas mencionados anteriormente, APM desarrolla cuatro actividades principales, recolección, extracción, visualización e interpretación de datos, las cuales serán descritas más adelante.

Se presenta una lista que contiene las cuatro actividades que se realizan al momento de evaluar el rendimiento de las aplicaciones.

- **Recolección de datos de rendimiento:** abarca múltiples sistemas y capas, desde datos de hardware hasta componentes de software. Dado que las aplicaciones actuales en general son distribuidas y multi-capa, es crucial definir de dónde se extraen los datos y cómo se realiza esta captura. Existen dos enfoques principales: el activo, que simula pedidos para evaluar la respuesta del sistema en condiciones específicas, y el pasivo, que recolecta datos de eventos reales, como logs y trazas de ejecución, reflejando el comportamiento real del sistema.

- **Extraer la información relevante:** para su extracción, los datos recolectados se organizan en estructuras, siendo las más comunes las series de tiempo. Estas hacen referencia a secuencias cronológicas de puntos de datos que muestran la evolución del rendimiento del sistema, con estadísticas como promedios, mínimos, máximos y percentiles en periodos específicos. Estas series son esenciales para detectar patrones y tendencias en el rendimiento de las aplicaciones.

- **Visualización:** esta actividad suele incluir dashboards con gráficos e indicadores de estado, mostrando datos clave de manera accesible para la gestión y operación. Estas métricas

permiten un análisis continuo y detallado del rendimiento de la aplicación, facilitando la identificación de patrones y posibles problemas.

- **Interpretación de datos y uso:** permite monitorear la salud del sistema, anticipar problemas y facilitar ajustes o nuevos despliegues.

Por otro lado se describe OpenTelemetry que permite realizar mediciones de los datos, definiendo y utilizando un estándar, y también Prometheus y Grafana que son un estándar de facto en el almacenamiento y visualización de métricas respectivamente[51].

2.3.1. OpenTelemetry

OpenTelemetry [33] es una iniciativa de código abierto enfocada en proporcionar un estándar para la recolección de datos de observabilidad, tales como métricas, trazas y registros (logs).

2.3.1.1. Arquitectura

OpenTelemetry se organiza en varios componentes modulares que permiten recolectar y exportar datos desde aplicaciones distribuidas. Los tres tipos principales de datos que maneja son:

- **Trazas (Traces):** capturan el recorrido de una solicitud a través de servicios distribuidos, proporcionando visibilidad sobre tiempos de respuesta y cuellos de botella.
- **Métricas (Metrics):** registran información sobre el rendimiento de los sistemas, como el uso de recursos y la latencia de las operaciones.
- **Registros (Logs):** recopilan eventos detallados que ocurren dentro de los sistemas para facilitar la depuración y auditoría.

La arquitectura de OpenTelemetry incluye las siguientes capas principales:

- **Instrumentación:** bibliotecas que se integran en el código de la aplicación para recolectar automáticamente datos de trazas y métricas.
- **SDK:** proporciona control sobre la recolección y configuración de datos, como las políticas de muestreo y la exportación.
- **Exporters:** permiten enviar los datos recolectados a herramientas externas, como Prometheus (la cual se aborda más adelante) [44] o Jaeger [25].

2.3.1.2. Características Principales

- **Soporte Multiplataforma:** OpenTelemetry ofrece bibliotecas para múltiples lenguajes de programación, incluyendo Go, Java, Python, y más.
- **Interoperabilidad:** facilita la integración con herramientas populares de monitoreo y trazabilidad, como Prometheus y Jaeger.
- **Muestreo y Agregación:** incluye soporte para técnicas de muestreo (forma de seleccionar una muestra representativa de una población [9]) que ayudan a gestionar la cantidad de datos recolectados.
- **Recolección Automática:** las bibliotecas de OpenTelemetry pueden detectar y recolectar automáticamente trazas y métricas sin necesidad de instrumentación manual extensa.

2.3.2. Prometheus

Prometheus[44] es un conjunto de herramientas de alerta y monitorización de sistemas open source creado originalmente en SoundCloud[47]. Desde su creación en 2012, muchas empresas y organizaciones lo han adoptado, y el proyecto cuenta con una comunidad de desarrolladores y usuarios muy activa. Actualmente es un proyecto open source independiente de cualquier empresa. Prometheus recopila y almacena sus métricas como datos de series temporales, es decir, la información de las métricas se almacena con la marca de tiempo en la que se registró, junto con pares clave-valor opcionales denominados etiquetas.

2.3.2.1. Arquitectura

En este punto se abordan los principales componentes que posee la herramienta y se describe brevemente el propósito de los que se consideran relevantes para el proyecto.

En la siguiente imagen se presenta todo el ecosistema de Prometheus y como se relacionan sus componentes, dentro de los cuales varios son opcionales.

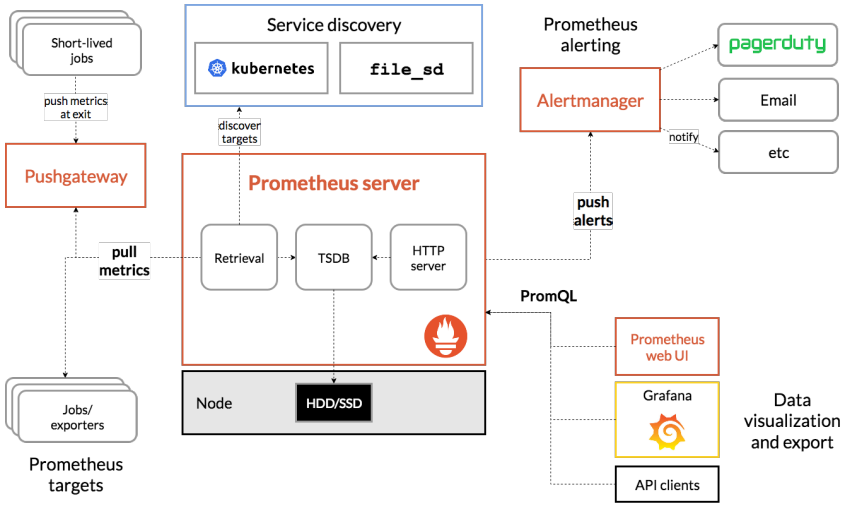


Figura 2.7: Arquitectura de Prometheus
Imagen extraída de [44]

En el contexto del proyecto, se pondrá atención sobre los siguientes componentes: Exporters, Prometheus server y Grafana. En particular, Grafana será abordado en la siguiente sección debido a la importancia para el proyecto.

- **Exporters:** son objetivos (*targets*) de Prometheus y se encargan de exponer las métricas mediante una API, la cual es consumida por el servidor de Prometheus a través del protocolo HTTP.
- **Prometheus server:** se encarga de recolectar métricas de los objetivos configurados cada ciertos intervalos de tiempo, evaluar reglas, mostrar resultados y poder generar alertas cuando se cumplen determinadas condiciones[45].
- **Grafana:** Grafana[15] es un sistema open source que permite consultar, visualizar, alertar y explorar métricas, registros (logs) y trazas desde cualquier ubicación. Dentro del proyecto, nos centraremos en la posibilidad de generar visualizaciones amigables al usuario, que en este caso son los administradores de WAFs, además, de la generación de alertas. Grafana posee varios plugins que le permiten consultar datos de distintas fuentes, en particular, una integración relevante es con la herramienta Prometheus.

2.4. Sistema de mensajes

En esta sección se introduce el concepto de sistema de mensajes, utilizado en el proyecto para realizar la comunicación con los modelos de aprendizaje automático en ciertos casos que se abordan en capítulos posteriores.

Un sistema de mensajes permite la transferencia de datos entre dos procesos o aplicaciones. Este tipo de sistemas permite que las aplicaciones estén débilmente acopladas mediante la comunicación asincrónica, lo cual también hace que la comunicación sea más confiable, ya que ambas aplicaciones no necesitan estar ejecutándose al mismo tiempo. Por otro lado, el sistema de mensajes es responsable de transferir los datos de una aplicación a otra, de modo que las aplicaciones pueden poner su foco en qué datos necesitan intercambiar en lugar de cómo hacerlo.

Existen distintos productos que brindan servicios de este tipo [48]. Por las características de este proyecto donde se busca que la solución ejecute en tiempo real de manera eficiente, se decide poner atención sobre la solución de NATS. En el documento [27] se describen otros productos aparte de NATS.

NATS[32] es un sistema de mensajería ligero y de alto rendimiento, diseñado para la comunicación en tiempo real entre aplicaciones distribuidas. Se ha convertido en una opción popular para arquitecturas de microservicios, IoT, y sistemas con requisitos de baja latencia.

La arquitectura de NATS se caracteriza por su simplicidad y eficiencia. Implementa un modelo basado en *publish/subscribe* (pub/sub), permitiendo una comunicación directa entre productores y consumidores. Esto simplifica la configuración y reduce la latencia.

Los productores (publishers) envían mensajes a un tema (*subject*), y los consumidores (subscribers) los reciben si están suscritos a dicho tema.

Entre las principales características se destacan:

- **Alta Disponibilidad:** NATS puede configurarse en un clúster de múltiples nodos, ofreciendo alta disponibilidad y tolerancia a fallos.
- **Escalabilidad Horizontal:** La arquitectura ligera permite escalar el sistema fácilmente mediante la adición de nodos sin necesidad de configuraciones complejas.
- **Baja Latencia:** Está diseñado para soportar miles de mensajes por segundo con una latencia extremadamente baja, ideal para aplicaciones en tiempo real.
- **Protocolos Simples:** Utiliza un protocolo basado en texto que facilita la integración y depuración.
- **Soporte para JWT:** NATS incluye opciones avanzadas de autenticación y autorización mediante *JSON Web Tokens* (JWT).

Capítulo 3

Análisis

En este capítulo se abordará el análisis del problema para el cual el presente trabajo se propone contribuir.

Se comienza presentando la parte central del proyecto que refiere a obtener una integración funcional de Coraza WAF con WACE, quien permitirá la integración con modelos de aprendizaje automático. Posteriormente se expondrán los distintos aspectos que complementan la herramienta a construir (excepciones, métricas y modos de ejecución).

En secciones siguientes se definirá la lista de requisitos que guiarán el desarrollo del proyecto. Además se ahondará en el punto desde el que parte el proyecto, es decir, las características principales de los componentes que se tiene en el punto de partida del proyecto y el flujo de ejecución que se espera tenga la solución una vez estén integrados los componentes mencionados anteriormente. Una vez completado el desarrollo de estas últimas dos secciones se procederá a definir el alcance del proyecto.

En las secciones finales del capítulo se explorará sobre las posibles alternativas de resolución en alto nivel de las distintas problemáticas a abordar y se comentará cuál es la opción elegida en el contexto del proyecto y por qué.

Durante todo el capítulo se hablará de “herramienta” y “solución” haciendo referencia al software que se irá construyendo a lo largo del proyecto. Además se hablará de “puntaje” haciendo referencia a los puntajes que define OWASP CRS y que fueron introducidos en la sección 2.1.4.

3.1. Problemática a abordar

Como se ha mencionado en secciones anteriores, las tecnologías de WAF open source utilizadas en la industria actualmente basan su configuración en la declaración de reglas estáticas. Dichos WAFs hacen uso de las reglas definidas por OWASP CRS, sin embargo, dichas reglas son genéricas dado que deben poder detectar ataques en cualquier organización sin que eso implique el bloqueo de su tráfico legítimo. Teniendo esto en mente se busca tener una herramienta que permita integrar modelos de aprendizaje automático de una manera sencilla para los usuarios (administradores del WAF) y que sea eficiente en comparación con el análisis normal que hace el WAF sobre las transacciones. Como se vio en la sección 2.2.1, la integración de dichos modelos puede permitir un aumento en el desempeño de la tecnología de WAF, en particular, pueden ser entrenados con registros del tráfico de una organización para adaptarse y disminuir la cantidad de falsos positivos.

Se desea que la integración se realice con la tecnología de WAF OWASP Coraza que fue anteriormente descrita en la sección 2.1.5. Esto en primera instancia implica comprender el flujo de ejecución, arquitectura y posibilidades de extensión de dicho WAF. Por otra parte es importante analizar si los mecanismos de comunicación ya existentes en WACE brindan la mayor eficiencia a la solución o si conviene tener otro mecanismo de comunicación para esta tecnología de WAF.

Otro punto a tener en cuenta durante el análisis es la usabilidad de la herramienta desde el punto de vista de un administrador, con respecto a esta variable lo que interesa es que la configuración sea simple y en lo posible centralizada, es decir, que las configuraciones de la herramienta estén ubicadas en la menor cantidad de archivos posibles y en una misma ubicación.

En las próximas secciones se expondrá con mayor detalle sobre los distintos aspectos que complementan la herramienta a construir. Como se mencionó anteriormente, uno de los beneficios de los modelos de aprendizaje automático es la disminución de los falsos positivos generados por el WAF, sin embargo, los modelos de aprendizaje automático no están exentos de generar falsos positivos, por ello en la sección 3.1.1 se presentarán los problemas asociados a la definición de excepciones. En la sección 3.1.2 se abordarán las consecuencias que tiene la ejecución de los modelos sobre el rendimiento del WAF. Teniendo este último punto en cuenta también se discutirá en la sección 3.1.3 la importancia de contar con métricas de rendimiento de la herramienta.

3.1.1. Definición de excepciones

Trabajando con WAFs, al detectar qué pedidos legítimos están siendo bloqueados es común tener que definir excepciones sobre determinadas reglas para evitar que se bloquee dicho tráfico legítimo.

Como se verá en secciones siguientes, al agregar la ejecución de los modelos al flujo, se agrega complejidad dado que ante un falso positivo puede que se tenga que agregar excepciones tanto para las reglas del WAF como para la ejecución de los modelos. Es deseable que el mecanismo para agregar excepciones de modelos utilice el mismo lenguaje que se utiliza para generar excepciones para las reglas de OWASP CRS, por una cuestión de usabilidad para el administrador del WAF.

Al mismo tiempo se debe tener en cuenta que la ejecución de los modelos debe ser paralela a la ejecución del WAF con el conjunto de reglas de OWASP CRS. Para resolver este problema se deben analizar distintos puntos, entre ellos el mencionado anteriormente respecto a la usabilidad desde el punto de vista de un administrador. Por otra parte se debe tener en cuenta el rendimiento y la optimización de recursos de cómputo ya que, en caso de ser definidas, las excepciones serán evaluadas por todas las transacciones que el WAF procesa con el fin de saber si corresponde que se deba exceptuar el análisis de algún modelo.

3.1.2. Modo de ejecución asincrónico

Un desafío clave de este proyecto es contar con buen rendimiento por parte de la solución. Al integrar modelos de aprendizaje automático para el análisis de transacciones, se agrega el tiempo de procesamiento correspondiente al tiempo de inferencia de los modelos y de comunicación entre los modelos y la herramienta. Sucede que los tiempos de inferencia de algunos modelos son lo suficientemente altos como para causar un tiempo de procesamiento inaceptable para una transacción. Surge la necesidad de tener un modo de ejecución asincrónico con respecto a la transacción. Evidentemente, como dichos modelos ejecutarán de manera asincrónica y potencialmente se obtenga el resultado de su análisis posterior a la finalización de la evaluación de la transacción, no se utilizará su resultado en la decisión de bloqueo, sino que dicho resultado será registrado. Dichos resultados pueden ser utilizados, por ejemplo, para generar alertas si la transacción no fue bloqueada y los modelos con mayor precisión indican que se trataba de un ataque.

3.1.3. Métricas

Es importante contar con métricas y herramientas de monitoreo que permitan a los administradores del WAF detectar velozmente problemas de rendimiento que pueden estar asociados tanto al análisis tradicional del WAF como al análisis de los modelos. En particular, disponer de métricas relacionadas con componentes clave de la solución, medidas como el tiempo consumido en cada uno o los resultados del análisis de transacciones, facilita la identificación de problemas de rendimiento. Esto permite, por ejemplo, detectar de manera más precisa cuál componente impacta en

mayor medida el desempeño general. Utilizando dichas métricas como insumo, es posible ajustar los parámetros de decisión utilizados.

3.2. Requisitos

Anteriormente se presentaron los temas a abordar y los principales desafíos que se tiene para llevar adelante el proyecto. En esta sección se describen los requisitos funcionales y no funcionales que se deben satisfacer. Es importante destacar que los requisitos definidos toman parte de los definidos en proyectos anteriores, adaptándolos al contexto del proyecto actual en el que se pone foco en Coraza WAF.

3.2.1. Requisitos funcionales

1. Se debe definir un mecanismo que permita la comunicación desde Coraza hacia WACE, de manera que éste pueda recibir la información de la transacción y otra información asociada con ésta que fuera necesaria para su clasificación. Se debe permitir el envío de los datos de la transacción acumulados en el WAF al inicio de cada fase (2.1.5.1), esto quiere decir que la transacción no tiene que necesariamente ser enviada por completo a WACE en el mismo momento, sino que pueden enviarse diferentes partes de la transacción en distintos momentos.

2. WACE debe permitir la integración de un criterio de decisión que especifique las condiciones que se deben cumplir para determinar que una transacción es maliciosa o no. El criterio de decisión debe poder obtener por parte de la herramienta la información resultante del análisis de las reglas de OWASP CRS y por otro lado de los modelos de aprendizaje automático. Se debe permitir la implementación de nuevos criterios de decisión, además debe ser posible seleccionar el criterio de decisión que desea utilizar para cada aplicación. Por defecto se debe utilizar el criterio de suma ponderada.

3. La herramienta debe inferir la fase en la que hay que ejecutar un modelo en función de los datos de la transacción que va a utilizar. Los datos que va a utilizar cada modelo deben ser un parámetro de configuración.

4. La herramienta debe permitir definir excepciones y configuraciones particulares, indicando que determinadas transacciones:

   a) Deben ser ignoradas totalmente por el motor de clasificación y no deben ser analizadas.

   b) Deben ser analizadas pero ninguna acción de bloqueo se debe llevar a cabo, solamente el registro en los logs del análisis.

   c) Deben ser analizadas, pero solamente por determinados modelos de aprendizaje automático y no por todos.

   Estas configuraciones deben poder ser realizadas según distintos parámetros presentes en la transacción, como lo puede ser el recurso solicitado o encabezados definidos por el protocolo HTTP.

5. Se debe poder exportar distintas métricas desde la solución de manera que se pueda saber:

   a) La cantidad de transacciones procesadas.

   b) El tiempo de procesamiento de una transacción por parte de la herramienta en su conjunto.

   c) El tiempo de análisis de una transacción que insume cada componente relevante de la solución.

d) Datos de los resultados del análisis de las transacciones, diferenciando los resultados de OWASP CRS, de los modelos y el resultado final.

Además es deseable contar con un mecanismo que permita al usuario generar métricas personalizadas.

6. Se debe integrar una herramienta que permita la visualización de las distintas métricas generadas por la herramienta.

7. Se debe proveer la posibilidad de configurar modos de ejecución para los modelos de aprendizaje automático. Dichos modos de ejecución pueden ser de dos tipos: modo sincrónico o asincrónico.

3.2.2. Requisitos no funcionales

1. La herramienta debe ser eficiente: su uso no debe afectar el uso normal en lo que respecta a las aplicaciones web que se pretenden proteger. Se debe contar con un tiempo de procesamiento.

2. La herramienta debe ser sencilla de instalar y actualizar, definiendo procedimientos claros para ello y analizando la posibilidad de automatizar dichos procesos. Además debe ser sencillo de replicar la instalación teniendo parámetros de configuración centralizados, permitiendo generar una nueva instancia de la herramienta a partir de la configuración de una instancia ya existente.

3. La ejecución del análisis por parte del WAF y de WACE debe realizarse en paralelo. Esto quiere decir que no debe analizarse primero la transacción por el WAF y luego por WACE sino que debe hacerse simultáneamente permitiendo así un mejor aprovechamiento del tiempo.

4. Es deseable que se puedan agrupar varias transacciones para ser procesadas por los modelos asincrónicos, de forma de paralelizar la ejecución y optimizar el uso de hardware. Muchos modelos de aprendizaje automático tienen mejor desempeño al procesar a la vez un conjunto grande de datos que hacerlo individualmente.

5. Se debe proveer casos de prueba que alcancen una cobertura de al menos 80 % del código.

6. Se deben proveer casos de prueba de *benchmark*, con el fin de evitar la degradación del rendimiento del sistema.

7. Debe ser extensible de forma de adaptarse a nuevos requisitos que surjan.

8. Se debe proveer documentación que facilite la experiencia de los usuarios con la herramienta, facilitando el cumplimiento de los requisitos anteriores.

3.3. Punto de partida

En esta sección se presentan las características de los componentes que serán integrados en el estado que se encontraban antes de comenzar el proyecto. Dichos componentes son Coraza WAF y WACE.

A la izquierda de la figura 3.1 es posible visualizar cómo se despliegan las soluciones en las cuales se utiliza únicamente a Coraza como WAF. Se puede ver el componente Reverse Proxy, el cual es encargado de redirigir el tráfico HTTP con destino al servidor de backend. Además, se cuenta con un conector cuya función es comunicar y permitir la integración entre el reverse proxy con Coraza WAF, utilizando la interfaz que provee Coraza. A su vez, este componente implementa cierta lógica para el procesamiento de las transacciones. Por último, se cuenta con el propio WAF de Coraza que se encarga de analizar las transacciones y devolver un resultado.

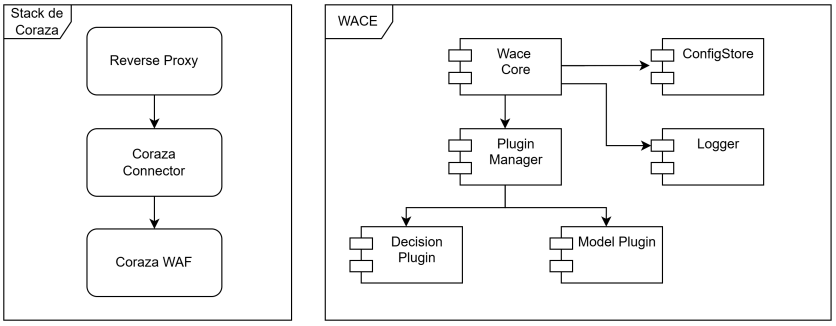


Figura 3.1: Componentes actuales

A la derecha de la figura se presenta brevemente a WACE, el cual tiene una arquitectura basada en plugins. Actualmente, WACE es un servidor que recibe transacciones o parte de ellas desde distintos WAFs a través de gRPC[16]. A su vez, se utilizan los plugins nativos del lenguaje Go para incorporar modelos y tomar la decisión sobre la transacción.

Presentados los componentes que se busca integrar, en el diagrama de la figura 3.2 se presenta el flujo deseado de ejecución de Coraza WAF integrado con WACE. Toma como base el flujo de ejecución de Coraza presentado en 2.4, al que se le agregan las invocaciones a WACE. La elección de los puntos en los cuales se realizan las invocaciones de WACE responde principalmente al momento en cual se disponibilizan los datos, aprovechando que las invocaciones tanto de Coraza como de WACE esperan los mismos datos.

3.4. Alcance

En esta sección se define el alcance del proyecto. Con respecto al análisis, se define que se evalúen todas las alternativas de solución que se consideren adecuadas con respecto a los requisitos del proyecto. Sin embargo, para las etapas de diseño e implementación la modificación de los componentes internos de Coraza WAF estará por fuera del proyecto, esto significa que si la alternativa de solución que presenta un mejor cumplimiento de los requisitos implica la modificación de los componentes internos de Coraza WAF, se deberá optar por otra alternativa para continuar con el diseño e implementación. Esto se debe principalmente a que no se tiene garantías de que la estructura interna de Coraza WAF se mantendrá estable y que no se implementarán cambios que impliquen que la solución se vuelva obsoleta. Esto es distinto al uso de Coraza WAF de manera externa, utilizando su interfaz, dado que se tiene garantías de compatibilidad dentro de la misma versión “MAJOR” [46].

3.5. Alternativas de integración

En esta sección se presentan los distintos ejes de discusión en el análisis de la integración de Coraza WAF con WACE.

El primer eje de discusión abordará la conveniencia de que WACE sea un servicio independiente (implementación actual) en comparación con la posiblidad de que WACE sea una librería.

El segundo eje tratará sobre la comunicación entre Coraza y WACE. Se detallará en particular qué componentes de la figura 3.1, deberán ser modificados para obtener el flujo deseado presentado en la figura 3.2.

Por último, se analizará en mayor profundidad dónde y con qué información se debe realizar la toma de decisión de bloqueo sobre una transacción (posterior a su análisis). Se deben plantear distintos mecanismos que permitan responder a estas cuestiones. En definitiva, lo que se necesita

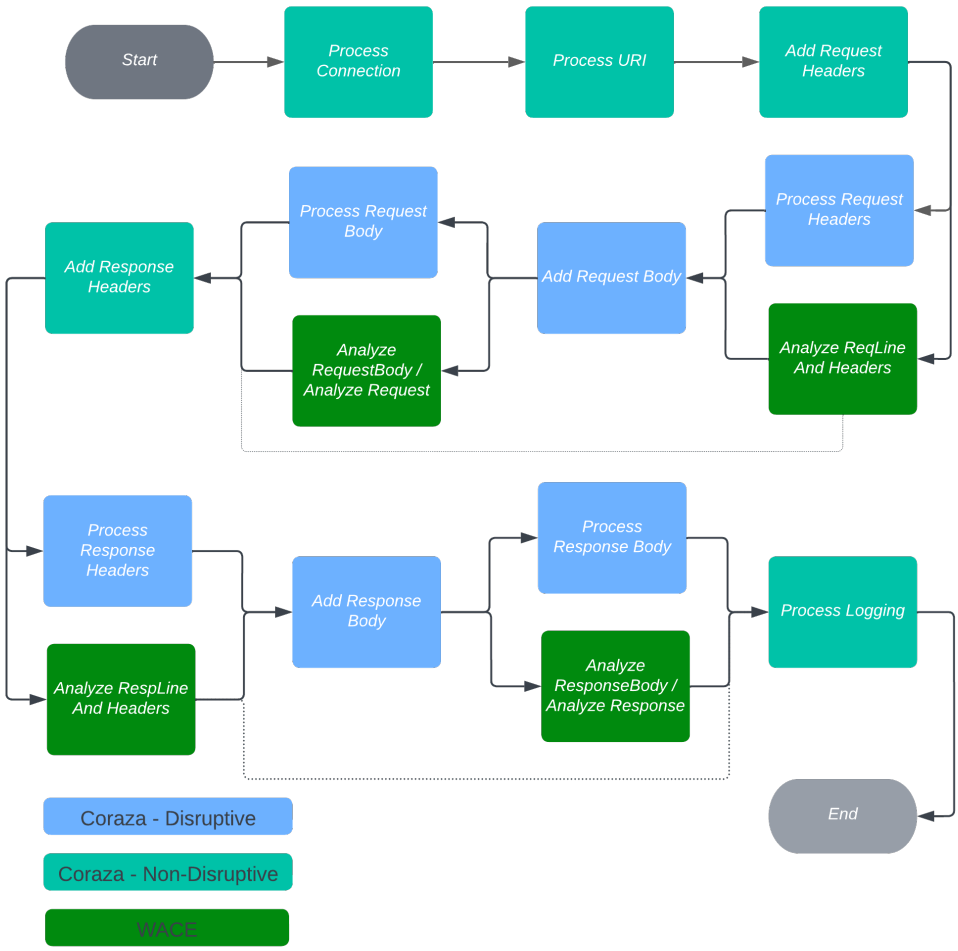


Figura 3.2: Flujo de ejecución esperado

es tener los resultados tanto de los modelos como de Coraza en un mismo punto para poder tomar una decisión, se desea definir cuál es dicho "punto". Este eje es relevante en este contexto, dado que no todos los mecanismos de toma de decisión serán compatibles con todas las opciones para comunicar Coraza y WACE.

En las siguientes secciones se presentarán las distintas opciones dentro de cada eje, destacando la opción que se llevará a cabo finalmente.

3.5.1. Despliegue de WACE

En esta sección se analizarán las alternativas para el despliegue de WACE, esto es, si WACE funcionará como un servicio independiente o por el contrario si será una librería.

3.5.1.1. WACE como servicio

Esta posibilidad refiere a que WACE ejecute como un servicio que exponga una API donde estén las funcionalidades para invocar el análisis de los modelos. La implementación actual de WACE responde a este esquema, en particular dicha implementación utiliza gRPC y Protobuf

para la comunicación con los distintos WAFs.

La ventaja de esta opción es que permite que WAFs de distintos proveedores tecnológicos que estén desplegados en distintos puntos puedan comunicarse con una única instancia (servidor) de WACE. Por otro lado presenta las siguientes desventajas: restringe a que todas las comunicaciones sean mediante la red, por lo que presenta un menor rendimiento; que la configuración no sea centralizada, dado que tanto el WAF que realiza los pedidos de análisis como WACE deben compartir la configuración; el proceso de instalación puede resultar más complejo.

3.5.1.2. WACE como librería

Esta posibilidad refiere a que WACE sea una librería que posea las funcionalidades para invocar el análisis de los modelos y ejecute en el mismo entorno de ejecución que el WAF.

Las ventajas de esta opción son: permitir una comunicación más directa con el WAF y evitar la red para ese propósito, por ende una mayor eficiencia; tener una integración sencilla con Coraza WAF que está implementado en el mismo lenguaje que WACE; permitir centralizar la configuración en un único punto; facilitar la instalación. Por otro lado, la principal desventaja es que si se tienen WAFs distribuidos, no se tendrá un único punto donde centralizar la comunicación a los modelos, aunque dependiendo del arquitectura de red de una organización, esto puede no ser un problema.

Esta opción será la que se utilizará durante el proyecto, dadas las ventajas con respecto a la opción "WACE como servidor" presentada en la sección anterior.

3.5.2. Comunicación entre WACE y Coraza WAF

Como se introdujo anteriormente, esta sección abordará las posibilidades de comunicar WACE con Coraza. Partiendo del diagrama presentado en 3.1, se debe decidir qué componentes modificar en pos de poder comunicar correctamente todos los datos necesarios tanto a Coraza como a WACE. Cada una de las posibilidades que se presentarán posee diferentes niveles de intervención en los componentes de Coraza e impacto en el proceso de configuración y mantenimiento de la herramienta.

3.5.2.1. Modificar el core de Coraza

Esta opción implica la modificación de los componentes internos de Coraza. Desde una visión de alto nivel, para el funcionamiento de WACE se necesita cargar y almacenar la configuración de la herramienta, y por otro lado se necesita realizar los llamados a los modelos.

En la figura 3.3, puede observarse la arquitectura parcial que se dedujo a partir de la documentación y la implementación de Coraza. En los siguientes párrafos se hará referencia a los componentes que se visualizan en ella.

Dentro de Coraza, el componente *WAF Engine* es el encargado de mantener configuraciones estáticas que no se espera se modifiquen en tiempo de ejecución. Dicha responsabilidad nos indica que se lo debe modificar para cargar y almacenar las configuraciones estáticas de WACE.

Por otra parte, el componente *Transaction* de Coraza es el encargado de accionar las funcionalidades de análisis de las reglas para cada una de las fases. Dicho análisis se realiza utilizando el componente *RuleGroup* que se encarga de almacenar grupos de reglas (en particular almacena las reglas asociadas a cada fase en distintos grupos) y realiza la evaluación de las reglas a pedido de la *Transaction*, éste último componente accede a los *RuleGroup* a través del *WAF Engine*. Por lo tanto, los llamados a los modelos de parte de WACE se podrían implementar en cualquiera de dichos componentes, desde un punto de vista de la filosofía de los componentes de Coraza y pensando en encapsular las funcionalidades de WACE, parece ser adecuado su implementación dentro del componente *Transaction*.

En resumen esta opción implica generar una nueva versión de Coraza WAF donde se agreguen las funcionalidades de WACE. Aunque esta opción parece ser la más adecuada en cuanto simpleza y eficiencia, complejiza las actualizaciones y el mantenimiento de la herramienta, ya que ante una

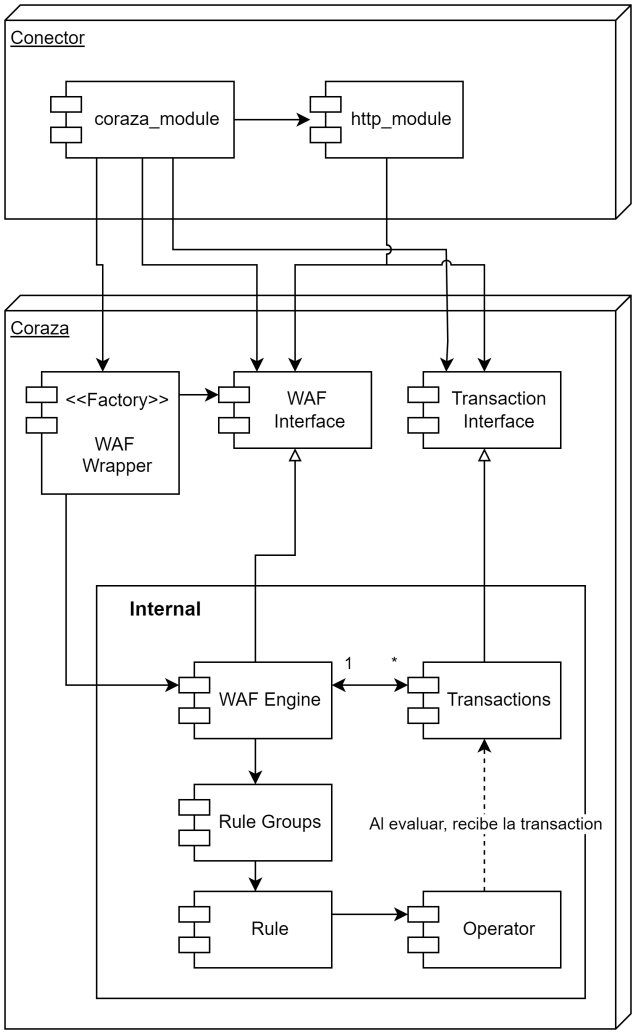


Figura 3.3: Arquitectura parcial de Coraza

nueva actualización de Coraza se deben analizar los cambios antes de impactarlos en el código de la herramienta, esto se debe a que se estarían modificando los componentes internos de Coraza y no se tiene garantía de que los nuevos cambios no impliquen la pérdida de funcionalidad en la integración. Por lo tanto la actualización se vuelve compleja.

Además de los componentes internos de Coraza se debe modificar el conector con el objetivo de que utilice la librería nueva generada a partir de Coraza. Estos cambios son mínimos por lo que ante una actualización del conector original no se generarían conflictos importantes.

3.5.2.2. Generar operadores

Esta forma de integración propone utilizar los mecanismos de extensión que posee Coraza WAF. en particular, se utilizaría la capacidad de generar operadores personalizados. De esta manera, al invocar desde el WAF uno de estos operadores personalizados, se realiza la invocación a WACE y éste realiza el llamado a los modelos. Con este mecanismo de integración no se modifica ningún componente del WAF, lo que simplifica el mantenimiento de la herramienta a desarrollar.

A continuación se presenta un ejemplo de cómo sería la invocación a WACE mediante un

operador llamado “wace” para el análisis del cuerpo de una solicitud HTTP:

```
SecRule REQUEST_BODY "@wace" "id,phase:2,pass,..."
```

Por otra parte, la configuración de WACE resulta compleja y poco intuitiva, dado que se le debe comunicar a WACE su configuración utilizando el lenguaje del WAF (Seclang). Además de esto, la generación de métricas implica una implementación compleja, por ejemplo para obtener el tiempo que consume el análisis de una transacción por parte de las reglas de OWASP CRS no se tiene un punto natural donde realizar las mediciones como podrían ser los puntos donde se realizan las invocaciones de las funcionalidades de Coraza, sino que se tendría que agregar una funcionalidad, también mediante operadores, que realice dichas mediciones.

3.5.2.3. Modificar conectores

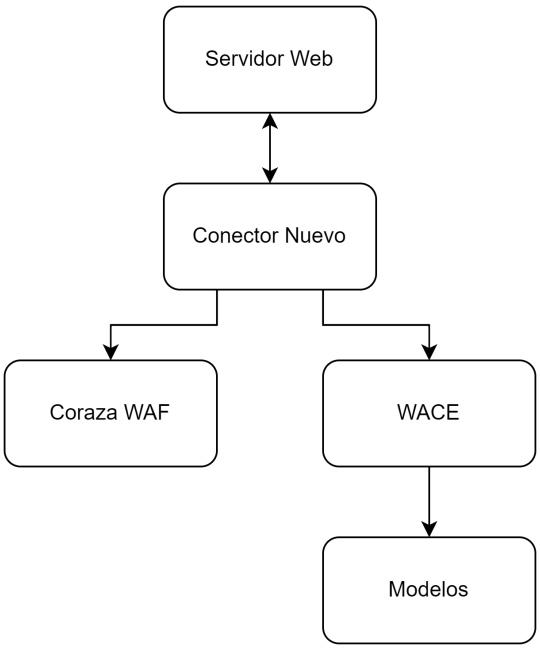


Figura 3.4: Modificar conectores

Esta opción implica la modificación de todos los programas que implementen la conexión entre algún tipo de servidor web y Coraza, desde donde se gestiona los llamados al WAF, en esos lugares se debería agregar los llamados a WACE.

En la figura 3.4 se presenta un esquema donde se puede observar las comunicaciones entre los componentes en esta alternativa. Dicha implementación implica crear una versión nueva de cada conector, pero como se observa en la figura mencionada anteriormente, la solución es independiente de la implementación de los componentes internos de Coraza, dado que el conector sólo utiliza la interfaz.

Estas implementaciones corren el riesgo de quedar desactualizadas, dado que ante un cambio en el conector original se debe evaluar qué impacto posee en la nueva implementación que integra la comunicación con WACE. Pero por otra parte, estos componentes no son actualizados con una frecuencia muy alta y son considerablemente más simples que Coraza WAF, por lo que el riesgo que presenta la alternativa no es muy alto. Si bien el costo de implementación por conector

es relativamente bajo, esto se debe impactar en cada uno de los conectores que existan. En la actualidad eso no es un problema dado que la cantidad es reducida.

Esta alternativa permite tener una configuración centralizada y un punto donde centralizar la medición de métricas.

La integración es simple y presenta buena accesibilidad a las partes de la transacción dado que el conector es quien envía a Coraza estos datos.

3.5.2.4. Capa intermedia entre los conectores y Coraza

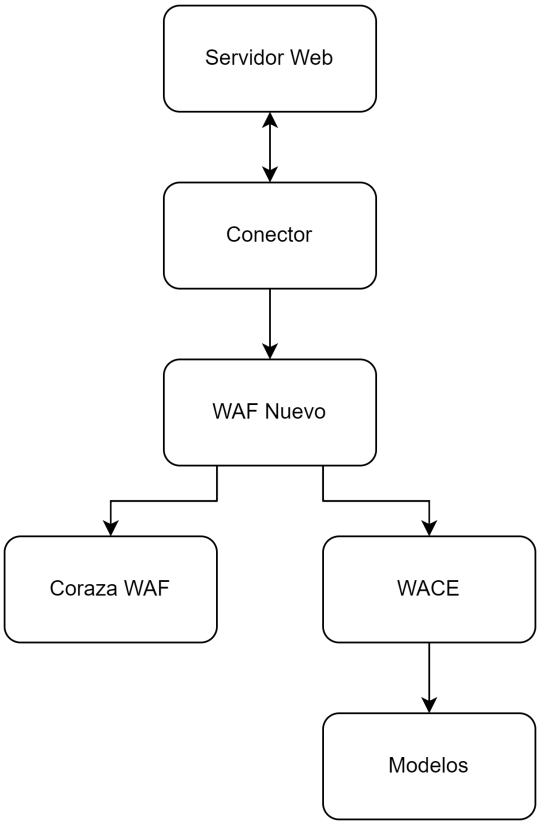


Figura 3.5: Capa intermedia

Esta opción implica la generación de una capa intermedia entre el conector de un servidor web y Coraza WAF, donde se gestionen los llamados a dicho WAF y al mismo tiempo a WACE, conlleva una modificación del conector. Dicha modificación puede ser mínima si se mantienen los mismos llamados y se respeta la estructura que ya poseen los tipos de Coraza. En la figura 3.5 se puede observar un esquema general de las comunicaciones entre componentes. Desde los conectores se instancia un objeto que cumple la interfaz `WAF` y este despacha objetos que cumplen la interfaz `Transaction`. Finalmente las transacciones realizan el procesamiento de los pedidos. Por ello se debería generar una nueva estructura de `WAF`, desde ahora `WACE_WAF`, que posea las funcionalidades de WACE. Para ello `WACE_WAF` debe realizar llamados a la interfaz del `WAF` ya existente de Coraza, teniendo en cuenta que a su vez se debe re-implementar los métodos que se consideren necesarios para la integración de las funcionalidades de WACE al conector. Esto es esencialmente despachar una nueva estructura de transacción, que realice llamados a la interfaz de transacción antigua pero re-implemente las funciones de procesamiento, llamando al procesamiento habitual de Coraza y

agregando ahora el procesamiento de WACE.

Además, el resultado de la decisión que tome WACE se puede comunicar de la misma forma que lo hace Coraza: se puede devolver un objeto de tipo `Interruption` en caso de que WACE considere que se debe realizar una acción disruptiva sobre la transacción. Se debe tener en cuenta que según el diseño de Coraza, una vez que se produce una interrupción, este evento es almacenado internamente en la transacción y evita que dicha transacción ejecute cualquier otro procesamiento. Esto no puede ser modificado desde la interfaz. Por ello se debe tener sumo cuidado con la configuración de Coraza que se utilice, posiblemente teniendo que modificar ésta a nivel de la capa intermedia.

Por ende el costo de implementación es moderado, se mantiene una clara separación de responsabilidades y no es necesario modificar el código fuente de Coraza, haciendo que el código sea mantenible. Se estima que las modificaciones en los conectores son mínimas y simplemente refieren al cambio del uso de la interfaz de la capa intermedia por sobre la interfaz de Coraza directamente, generando transparencia tanto para el conector como para Coraza. Esta alternativa permite tener una configuración global y centralizada, además presenta un punto central desde donde se puede realizar la medición de la mayoría de las métricas. En contraposición, se estima que puede ser levemente menos eficiente en el uso de tiempo que las opciones anteriores.

Se utilizará esta opción en el contexto del proyecto dado que es la opción más balanceada entre eficiencia, configuración centralizada, y el costo de mantenimiento sobre el proyecto. Aún así, se destaca que la opción de modificar los componentes internos de Coraza, descrita en 3.5.2, sería la elegida si sólo se tienen en cuenta los aspectos técnicos requeridos.

3.5.3. Mecanismos para la toma de decisión

En esta sección se explorarán las distintas posibilidades en cuanto al componente encargado y a la información que se utilizará para tomar la decisión de si una solicitud es un ataque o no.

3.5.3.1. Utilizar un operador

Esta opción se basa en utilizar un operador como medio para comunicar al WAF si una transacción debe ser bloqueada. Previamente WACE debe tener acceso a los resultados de los modelos y a los puntajes de OWASP CRS, cuando WACE retorne el valor de la decisión, el operador transmitirá el resultado del chequeo al WAF. Este mecanismo aprovecha la implementación del manejo de la interrupción ya implementado por Coraza.

A continuación se presenta un ejemplo de como se podría realizar esto:

```
SecRule TX:BLOCKING_INBOUND_ANOMALY_SCORE "@wacecheck
↪  %{tx.inbound_anomaly_score_threshold}" "id,phase,deny,..."
```

En este caso se envía el puntaje acumulado hasta el momento y el valor del umbral al operador “wacecheck”, la implementación de este último debe poder acceder a los resultados de los modelos. En caso de que el operador retorne el valor booleano *True*, la acción disruptiva “deny” generará que se bloquee la transacción.

Se pueden agregar llamados a distintos criterios de decisión generando más operadores y agregando reglas para dicha evaluación. En cuanto a la configuración, se tiene que realizar mediante los mecanismos de Coraza, es decir, Seclang. Por lo tanto, para realizar la configuración se deben agregar reglas que permitan tomar la decisión.

3.5.3.2. Modificar el core de Coraza

Esta opción se basa en agregar al core de funcionalidades de Coraza, un mecanismo que integre los resultados de WACE en la decisión que se toma para interrumpir la transacción. Se debe modificar el mecanismo nativo que utiliza Coraza para tomar la decisión, que es la ejecución de

aquellas reglas de OWASP CRS que comparan los puntajes acumulados hasta el momento contra el umbral definido. Se debe pasar a un enfoque donde se obtengan dichos puntajes y se envíe a WACE para que tome la decisión junto a los resultados del análisis de los modelos. Nuevamente esto implica un mayor costo de mantenimiento de la herramienta.

3.5.3.3. Utilizar variables y Seclang

Esta opción se basa en tomar la decisión utilizando las directivas de Seclang. Para que esto pueda suceder, previamente se debe tener en variables de Seclang (introducidas anteriormente en la sección 2.1.3.2) los resultados de los modelos. Esto implica tener acceso al componente *Transaction* de Coraza.

Este mecanismo permite a los administradores definir criterios de decisión sencillos con mecanismos a los que ya están adaptados. Dichos criterios tienden a ser más simples, como reglas donde se compare el resultado de un modelo contra cierto umbral. Sin embargo, mediante el encadenamiento de reglas podrían generarse criterios más complejos, aunque esto está limitado por el lenguaje.

3.5.3.4. Utilizar la interfaz de Coraza

Esta opción se basa en utilizar los datos accesibles mediante la interfaz de Coraza. Esto significa que se pueden utilizar dos tipos de datos: las interrupciones que son devueltas si una transacción fue interrumpida (este es un valor booleano, que indica si la transacción fue bloqueada o no) y por otra parte la lista de reglas que evaluaron positivamente. Estas interrupciones no contienen el resultado de los puntajes acumulados por OWASP CRS, pero dichos puntajes sí están presentes en forma de texto dentro de las reglas que coincidieron. Este mecanismo implica que se deba extraer los puntajes deseados a partir de los datos brindados por Coraza.

La alternativa que se acaba de describir junto con "Modificar el core de Coraza" expuesto en 3.5.3.2, son los mecanismos más simples y al mismo tiempo con mayor expresividad de los presentados. Por una cuestión de alcance y compatibilidad con la alternativa elegida para el eje "Comunicación entre WACE y Coraza WAF", se utilizará el mecanismo descrito en la presente sección.

3.5.4. Análisis de compatibilidad entre alternativas

En esta sección se desarrollará la compatibilidad entre las distintas alternativas presentadas en cada eje de discusión planteado. Las opciones de WACE como librería o servicio independiente son compatibles con todas las alternativas de comunicación y mecanismos para la toma decisión. En la tabla 3.1 puede observarse el cruzamiento entre las alternativas para la comunicación y los mecanismos para la toma de decisión.

Mec. Decisión / Alt. Comunicación	Utilizar operador	Desde el Core de Coraza	Variables y Seclang	Interfaz de Coraza
Modificar Core de Coraza	✓	✓	✓	✓
Llamado desde operadores	✓	✓	✓	X
Modificar conectores	X	X	X	✓
Capa intermedia	X	X	X	✓

Tabla 3.1: Análisis de compatibilidad entre alternativas de comunicación y mecanismos para la toma de decisión.

Si se utiliza una arquitectura de integración que modifique el *core* de Coraza es posible utilizar todos los mecanismos para la toma de decisión, ya que cualquier modificación necesaria para poder

integrar el mecanismo se puede realizar. Si bien este es un aspecto positivo, no se debe perder de vista que según el mecanismo de decisión que se elija, dependerá la cantidad de cambios a realizar. Por ejemplo, utilizar un operador para tomar la decisión implica tener la capacidad de poder invocar el operador luego de que se evalúe la transacción por parte de los modelos y Coraza, lo que en principio no implicaría un gran cambio, si solo se considera agregar esta invocación de la manera descrita. En cambio, si se lo compara con el mecanismo de tomar la decisión internamente en Coraza, esto puede tener un impacto bastante alto, ya que se van a modificar componentes internos de Coraza, lo que es un problema para el mantenimiento.

Para la alternativa de comunicación que implica generar operadores de Coraza, se tiene una situación similar que para el caso analizado anteriormente respecto a los mecanismos de decisión. La diferencia radica en que como cada operador es invocado durante el flujo de evaluación de reglas de la transacción, no se puede obtener información correspondiente a si la transacción fue interrumpida o no, ya que si lo fuera, la invocación al operador correspondiente nunca ocurriría.

Para las dos restantes arquitecturas el análisis es muy similar, ya que ambas carecen de acceso a los componentes de Coraza y solamente utilizan sus interfaces, de manera que la única opción para tomar la decisión en estas arquitecturas debe ser un mecanismo que utilice información de las interfaces de Coraza y la información que se puede obtener de WACE.

Por un análisis mayor del cumplimiento de los requisitos de cada alternativa se pueden ver más detalles en el anexo A.

3.6. Configuración de excepciones

Con el fin de abordar la problemática planteada en la sección 3.1.1, en esta sección se analizarán los caminos viables que permiten generar o configurar excepciones para los modelos de aprendizaje automático.

Se observa que las alternativas de comunicación que se apoyan en el uso de Seclang para accionar las funciones de WACE proveen el mecanismo nativo del WAF para la definición de excepciones, dado que se pueden exceptuar las reglas que invocan a los modelos. Esto permite entonces que se definan las excepciones en un mismo lenguaje tanto para el análisis que realiza el WAF, como para la invocación de los modelos.

En contraposición, la arquitectura de capa intermedia debe tener otras consideraciones para poder realizar las excepciones dado que la invocación de los modelos está totalmente desacoplada de la ejecución del WAF, existen dos alternativas a alto nivel:

- La primera alterativa se basa en definir desde cero un mecanismo de chequeo de excepciones que cumpla con los requisitos de excepciones deseados.
- La segunda alternativa es generar un mecanismo que permita la utilización de Seclang para la definición de excepciones y obtener la excepción a través de la interfaz de Coraza, mediante las reglas que hicieron match.

En las siguientes secciones se profundizará el análisis de cada una de las alternativas.

3.6.1. Definir un nuevo mecanismo

En esta sección se explorará la alternativa de definir un nuevo mecanismo de excepciones. Con este propósito, es necesario definir el alcance de éstas excepciones, lo que implica definir qué datos, operaciones y potencialmente transformaciones deben ser implementadas. Mínimamente se debería poder predicar sobre los siguientes elementos de la transacción:

- IP del cliente
- Request Line, Headers y parámetros de GET o POST.

- Request Body.
- Response Line y Headers.
- Response Body.

Con el fin de definir operaciones sobre los anteriores elementos a éstos se les debe asignar un tipo. Por ejemplo "IP del cliente" debería ser de tipo IP y el valor del encabezado "Content-Length"[18] que indica la cantidad de *bytes* que posee del cuerpo del mensaje HTTP, debería ser un valor entero. Estos elementos deben poseer un tipo de datos definido con el objetivo de definir operaciones que los relacionen con otros valores. Dichos tipos son String, Enteros, Reales e IP y deben poseer las siguientes operaciones:

- String: igualdad, inclusión de un string, o matcheo de expresión regular.
- Enteros y Reales: igualdad, $>$, $<$, $\geq$, $\leq$.
- IP: igualdad o pertenencia a subrango.

Además se deben proveer transformaciones típicas como URL encode/decode, lowercase, entre otras.

En suma esto representa la definición de un nuevo lenguaje, lo que puede implicar un gran costo de implementación. A esto se le debe agregar que el administrador de WAF debe manejar dos lenguajes distintos para definir excepciones, lo que presenta un problema de usabilidad. Por lo que no se utilizará esta alternativa.

3.6.2. Adaptar la integración para utilizar Seclang

Como se mencionó anteriormente, esta alternativa lo que busca es generar un mecanismo que permita la utilización de Seclang como lenguaje para definir excepciones para los modelos.

Esta opción presenta problemas de coordinación entre la ejecución de la fase de reglas y la ejecución de aquellos modelos que se supone deberían haber sido exceptuados. Dado que si un modelo ejecuta en fase 1 y las excepciones son definidas también en esa fase, no es posible obtener el resultado de la evaluación de las excepciones hasta que termine de ejecutar la evaluación de la fase por parte del WAF. No se puede solucionar ejecutando de manera secuencial el WAF con los modelos porque se tiene el requisito de que la ejecución sea en paralelo.

Ante esta realidad surgen las siguientes opciones para remediar el problema:

Lanzar la ejecución de todos los modelos y sólo considerar aquellos que no fueron exceptuados Esta posibilidad se basa en agregar las reglas de excepción lo antes posible según el dato sobre el cuál se desee predicar, es decir en la fase más temprana posible. En la figura 3.6 se presenta un diagrama de secuencia que permite visualizar el flujo que se propone en esta opción. En aquellos casos donde se definan las excepciones en fases anteriores a la fase donde se comienza a ejecutar el modelo, el flujo no tendría problemas de coordinación.

Sin embargo, en aquellos casos que se define la excepción en la misma fase que se va a lanzar la ejecución del modelo, se tendría el problema anteriormente comentado sobre el requisito de ejecución paralela.

Ante esta situación, se podría evitar que los modelos que fueron ejecutados sin que las excepciones se ejecutaran previamente, no sean considerados al momento de la evaluación. Aunque esto solucionaría el problema de coordinación, posee la desventaja de que se realiza procesamiento innecesario de los modelos, que se espera sean la parte del flujo que implica una mayor demora, en comparación con la evaluación de reglas del WAF.

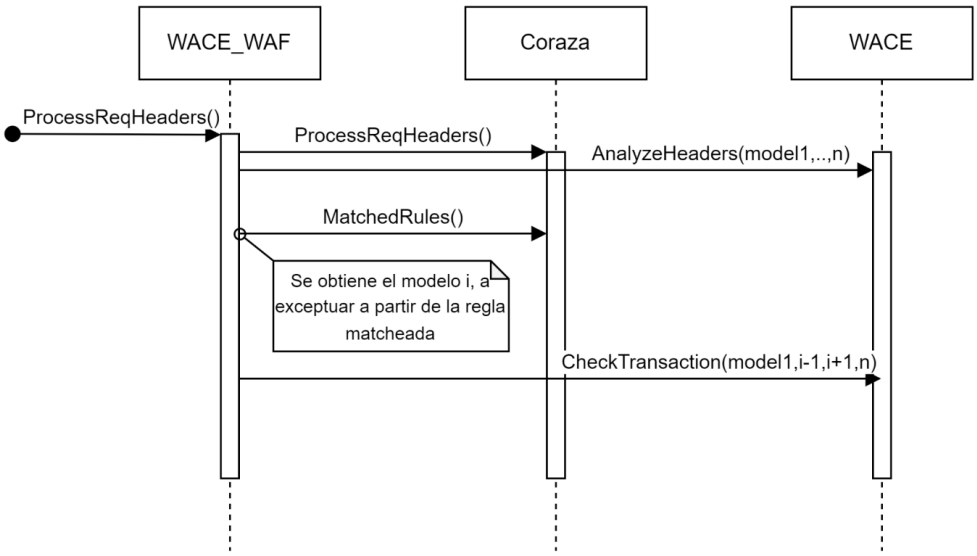


Figura 3.6: Ejecutar todos los modelos y sólo considerar un subconjunto.

Modificar las fases existentes para adecuar el flujo de las excepciones de modelos El concepto actual de fase, que fue introducido en la sección 2.1.5.1 y detallado en la figura 2.2, está atado a la disponibilidad de los datos y a motivos históricos de su diseño original en Apache. Se definen cinco fases, las primeras cuatro de ellas están asociadas a Request Line y Headers; Request Body; Response Line y Headers; Response Body. Finalmente hay una fase extra para realizar el proceso de logging.

No todos los servidores web disponibilizan los datos de la misma forma que lo hace Apache. Se puede dar el caso, por ejemplo, de que la función encargada de accionar la fase 1 (Request Line y Headers) tenga disponible todos los datos de la Request y aún siga el flujo habitual de fases. Esto tiene dos consecuencias que se pueden apreciar a simple vista:

- Si se bloquea la transacción con datos disponibles en fase 1, como no se llegó a realizar la extracción de los datos del Request Body entonces se evitaría el consumo de tiempo y recursos de manera innecesaria.
- Por el contrario, si se está en un caso de Virtual Patching (donde se generan reglas con el objetivo de bloquear solicitudes que explotan vulnerabilidades conocidas en las aplicaciones protegidas, como se introdujo en la sección 2.1) y las reglas de bloqueo predican sobre contenidos del Request Body, se estaría ejecutando el conjunto de reglas de fase 1 innecesariamente, resultando en un camino no óptimo para minimizar el tiempo de procesamiento del WAF por transacción, dado que las reglas de Virtual Patching podrían alcanzar para bloquear los pedidos sin ese procesamiento asociado innecesario.

Teniendo lo anterior en cuenta, se analizará la posibilidad de modificar la definición actual de fase y su migración al concepto de estado y conjunto de reglas (ruleset).

El esquema general propuesto puede visualizarse en la figura 3.7. El estado del WAF representa el conjunto de datos disponible para el análisis hasta el momento, dado que los datos disponibilizados son acumulativos, un estado n del WAF contiene todos los datos que tiene un estado m, con $m < n$.

Por otra parte un ruleset es, como su nombre lo indica, un conjunto de reglas que comparten una semántica. Cada regla predicará sobre algún dato, por lo tanto, si se realiza la unión de los datos sobre los que predica cada regla de un ruleset, se obtendrá un conjunto de datos que permitirá

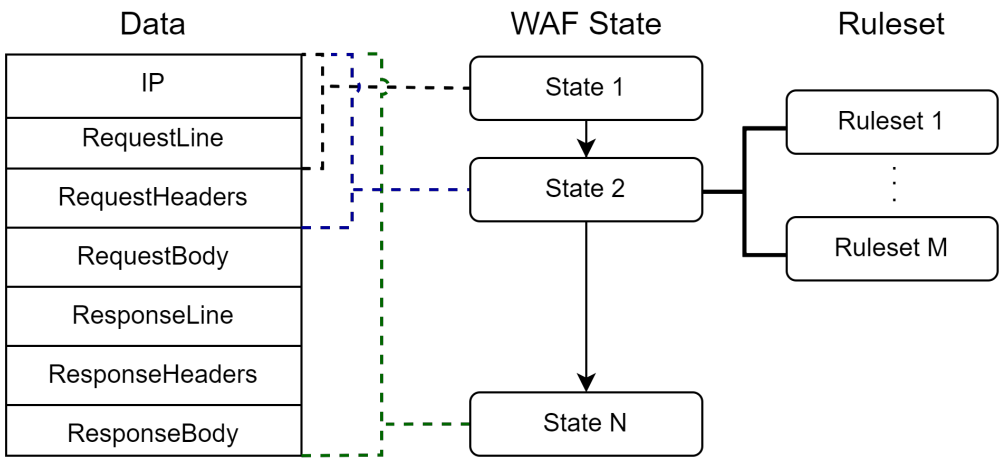


Figura 3.7: Concepto de estado y conjunto de reglas

definir un estado mínimo sobre el cuál se podrá ejecutar el ruleset. Sin embargo, el usuario que defina estas configuraciones debería poder definir explícitamente si desea ejecutar en un estado posterior. Al mismo tiempo, debería poder definir el orden de ejecución de los ruleset dentro de los estados.

Esto permitiría definir distintos estados que se adapten a flujos de disponibilización de datos para los distintos servidores donde se ejecute el WAF. Por otra parte permite definir conjuntos reducidos de reglas que se adapten a servicios particulares que protegen de una manera más simple que con el esquema de configuración de Seclang en el que se deben definir etiquetas y saltos (GOTO) entre dichas etiquetas. Esto con el objetivo de evitar el análisis innecesario de tecnologías que no se están protegiendo, es decir, si se está protegiendo un sitio con una base de datos tiene sentido utilizar reglas que protejan contra inyección SQL, en otro caso simplemente representa una penalización en el rendimiento. Se cree que este mecanismo presenta mayor usabilidad que la actual implementación, tanto de Coraza como de ModSecurity, en lo que a orden de evaluación respecta. Esto se debe a que en el esquema propuesto el usuario configura el orden de evaluación de los conjuntos de reglas dentro de un estado, mientras que en la actualidad, el orden de las reglas dentro de una fase está determinado por el orden alfabético de los archivos en el sistema de archivos desde el cuál las reglas son cargadas. Por ejemplo, dada una fase i, si se tienen reglas de dicha fase en el archivo llamado “REQUEST-900” y reglas en otro archivo llamado archivo llamado “REQUEST-901”, las reglas del primer archivo ejecutarán antes que las reglas del segundo dentro de la fase i.

Otra ventaja, es que proporciona un lugar intuitivo para definir un punto de control de acceso basado en la semántica de las reglas. Es decir, se tiene por ejemplo un conjunto de reglas que corresponden a “Virtual patching” y se puede definir un punto de control de acceso (punto donde se verifique si corresponde que la transacción continúe su procesamiento o sea rechazada) para esas reglas de una manera más intuitiva y usable que con la definición actual de fases.

Generar otro WAF dedicado a evaluar las excepciones Este mecanismo se apoya en la separación de responsabilidades entre las reglas que se definen en el flujo de la ejecución de OWASP CRS y aquellas cuyo objetivo es modificar el flujo de la ejecución de los modelos.

Para esto se deben inicializar dos instancias distintas de `WAF` cada una con uno de los conjuntos de reglas mencionado anteriormente. En la figura 3.8 se pueden apreciar los componentes y el flujo de comunicación esperado entre ellos. En la mencionada imagen se puede visualizar como se realizan los mismos llamados a dos instancias de `WAF` distintas, donde una instancia realiza el

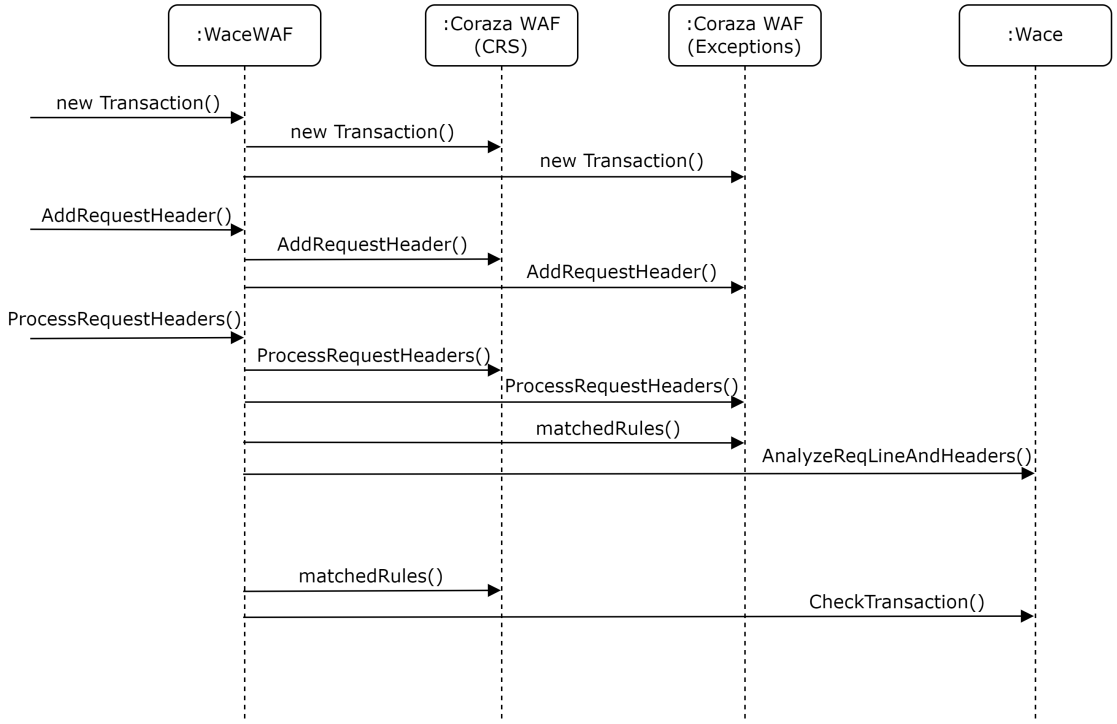


Figura 3.8: Mecanismo de excepciones con dos instancias de WAF

análisis asociado a las reglas del OWASP CRS mientras que la otra realiza el análisis asociado a las reglas de excepción de modelos, cuando ésta última instancia finaliza, se invoca en WACE el análisis de aquellos modelos que resultan activos posterior a la evaluación de las excepciones.

Este mecanismo permitiría que la ejecución de OWASP CRS sea independiente y en paralelo a la evaluación de excepciones y posterior ejecución de los modelos como se puede apreciar en la mencionada figura.

Sin embargo se hace un uso subóptimo de recursos al generar dos WAFs y posteriormente dos instancias de Transacción para cada transacción real que se reciba. El costo principal está asociado al uso de memoria duplicado y al tiempo que se consume extrayendo y almacenando los mismos datos en dos instancias de `WAF` distintas.

Con todo lo anterior en mente, este es el mecanismo a utilizar en el contexto del proyecto, si bien se estima que no es el mecanismo con mejor rendimiento, es el que aprovecha mejor los recursos y además se encuentra comprendido dentro del alcance discutido en la sección 3.4. Seguidamente se explica resumidamente por qué las otras alternativas fueron descartadas:

- Modificar las fases existentes implica modificar la interfaz y los componentes internos de Coraza, por lo que está fuera de alcance.

- La opción de lanzar la ejecución de todos los modelos y sólo considerar aquellos que no fueron exceptuados se estima puede tener un rendimiento peor dado que se considera que los modelos son el componente del flujo que consume más tiempo.

En un contexto sin esas restricciones, se cree que el mejor mecanismo sería el de modificar el concepto de fase presentado anteriormente.

3.7. Recolección de métricas

Como se introdujo en la sección 2.3, a nivel de la solución interesa recolectar métricas que permitan analizar el desempeño de la aplicación en tiempo real, en pos del cumplimiento de los requisitos que imponen que la herramienta debe ser eficiente y que debe contar con un tiempo de procesamiento adecuado.

En este sentido resulta evidente la necesidad de tener mediciones sobre el tiempo promedio de procesamiento de las transacciones, así como el tiempo que demora el procesamiento de cada uno de los componentes de la solución. Esto es, el tiempo que demora Coraza, el tiempo de WACE propiamente dicho, el tiempo insumido por la arquitectura de integración que se decida utilizar y por otra parte el tiempo de procesamiento insumido por los modelos de aprendizaje automático. Estas métricas resultan fundamentales para comprender los cuellos de botella presentes en la aplicación así como permitir la mejora de los componentes que sean mayormente responsables del consumo de tiempo.

Por otro lado, se requiere tener información acerca de los resultados que están teniendo las aplicaciones que están siendo protegidas por la herramienta. En esta línea, interesa tener la cantidad de transacciones procesadas, diferenciando según el código de respuesta, permitiendo saber si la transacción fue bloqueada o no. Dentro de esto, además de contar con información sobre la decisión global tomada por la solución, conocer qué decisión tomaron cada uno de los componentes individuales. Esto permite conocer cuál sería el resultado de una transacción en caso de que el servidor web no esté protegido por un WAF, así como el resultado si dicho WAF no cuenta con la información de los modelos de aprendizaje automático. Por otra parte, esta es información clave para la mejora de los algoritmos de decisión.

3.8. Modelos asincrónicos

Como se menciona en la sección 3.1.2, interesa tener dos modos de ejecución para los modelos, uno que ejecuta de manera sincrónica con respecto a la transacción, ayudando a la toma de decisión en tiempo real, y otro modo para modelos que realicen un análisis mas profundo de las transacciones de manera asincrónica. En esta sección se entrará en detalle respecto a este último modo.

La arquitectura propuesta debe ser capaz de permitir la coordinación entre los pedidos que el WAF desea clasificar por los modelos “lentos” y la capacidad de procesamiento que posean los modelos. De esta manera, cuando la carga de pedidos generada por el WAF es mayor a la capacidad de procesamiento de los modelos asincrónicos, se pueden retener y enviar los pedidos, con algún criterio como lo puede ser FIFO, y posteriormente los modelos pueden realizar la clasificación cuando tengan disponibilidad. Además de un sistema que permita la coordinación, se requieren de mecanismos de comunicación, que permitan tanto la comunicación desde WACE hacia los modelos como desde los modelos hacia WACE.

A su vez, una vez realizada la clasificación, se requiere mantener un registro de los resultados.

Es deseable además que el modo de ejecución de cada modelo pueda ser configurable de manera transparente, es decir, que un modelo que ejecuta sincrónicamente pueda ejecutar asincrónicamente sin cambiar la implementación del código que comunica a WACE con el modelo.

Capítulo 4

Diseño

En el capítulo previo se presentó el problema, se detallaron los requisitos y se propusieron alternativas para la resolución de los distintos aspectos que complementan la herramienta deseada.

El objetivo principal de este capítulo es diseñar tanto la arquitectura como los componentes que la integran, utilizando como entrada el análisis y las decisiones tomadas en el capítulo anterior. En este sentido, se comenzará el capítulo sintetizando las decisiones tomadas en el capítulo anterior y presentando un diagrama general de los componentes que tendrá la solución, así como una descripción de estos. En las secciones posteriores se profundizará en el diseño de algunos componentes y se expondrán aquellos flujos importantes de los que participen.

En la sección 4.3 se explicarán los desafíos técnicos presentes en la integración con las reglas de OWASP CRS, y la solución propuesta para dichos desafíos. Posteriormente se profundizará sobre los plugins de modelo, donde se presentará la interfaz que deben implementar y sus modos de ejecución.

En la sección 4.5 se presentará el componente "Sistema de mensajes", su rol en la solución y la tecnología a utilizar. Una vez realizado esto se estará en condiciones de presentar los flujos de análisis y chequeo de una transacción dentro de WACE.

En la sección 4.8 se presentará el diseño para la recolección de métricas y las tecnologías involucradas en el proceso.

Se finalizará el capítulo presentando la arquitectura de despliegue de la solución, donde se englobarán las propuestas expuestas hasta el momento.

4.1. Arquitectura

La arquitectura que se eligió desarrollar parte de la opción de comunicación que propone una capa intermedia (WACE_WAF) entre Coraza y su conector, dicha capa intermedia implementará la interfaz que brinda Coraza, pero agregando a WACE como parte de la arquitectura de una manera transparente para el conector y Coraza. Esto minimiza el costo de mantenimiento. Principalmente por cuestiones de rendimiento se decidió utilizar WACE como librería para evitar la disminución de rendimiento que agrega la conexión mediante la red.

Por otra parte el mecanismo elegido para la toma de decisión es aquel que utiliza los datos accesibles mediante la interfaz de Coraza, donde nuevamente se evita la modificación de los componentes internos de Coraza.

En cuanto a la definición de excepciones de modelos, se decidió utilizar el opción de tener una instancia de Coraza WAF dedicada al procesamiento de excepciones para los modelos, es decir, que se utilizará el mismo lenguaje (Seclang) para definir excepciones tanto a nivel de OWASP CRS como de los modelos de aprendizaje automático.

La arquitectura propuesta contempla el envío de métricas desde un único punto centralizado en la capa intermedia.

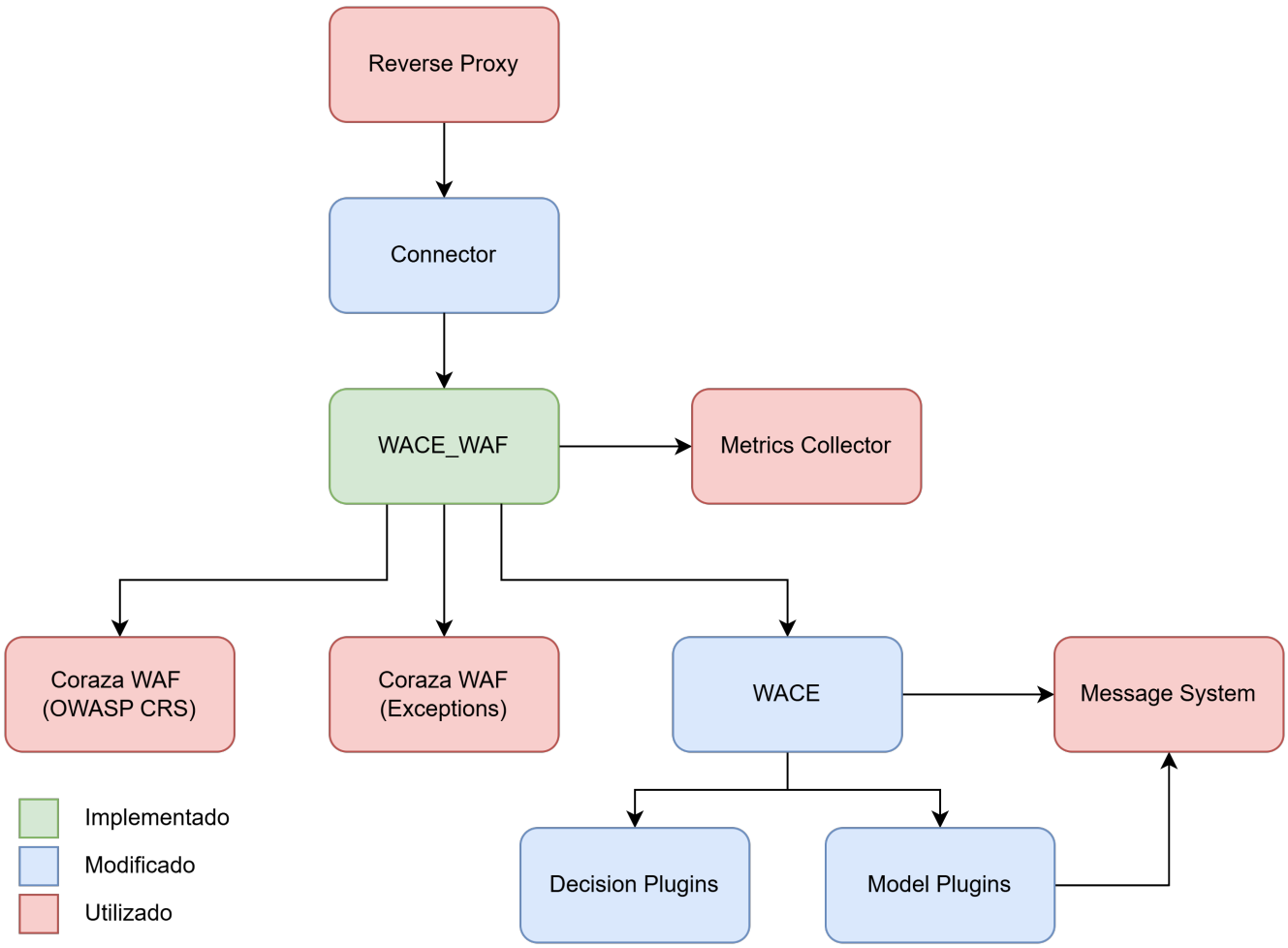


Figura 4.1: Diagrama general de la solución

4.1.1. Componentes

En esta sección se describirán de manera general los componentes principales de la solución, en la figura 4.1 se puede observar un diagrama de alto nivel destacando en cada caso si deben ser construidos desde cero, modificados o simplemente se los utilizará sin modificarlos. A continuación se listarán aquellos componentes relativos a la integración:

- **Reverse Proxy:** encargado de redirigir los pedidos HTTP de los clientes hacia el servidor de backend. Es el encargado de inicializar el conector de Coraza, además de cargar los archivos de configuración.
- **Connector:** componente encargado de integrar Coraza WAF al servidor web que funciona de Reverse Proxy. Genera un WAF por cada virtual host y se encarga de realizar los llamados para el análisis de los pedidos y respuestas HTTP. La modificación de este componente es mínima, implica simplemente cambiar la librería que se utiliza para invocar las funcionalidades de Coraza para en cambio invocar las funcionalidades de "WACE_WAF".
- **WACE_WAF:** componente encargado de implementar la interfaz de Coraza WAF y administrar los pedidos a: Coraza WAF (OWASP CRS), Coraza WAF (Exceptions) y WACE.

Además de comunicarse con el colector de métricas.

- **Coraza WAF (OWASP CRS):** componente encargado de evaluar cada transacción con el conjunto de reglas de OWASP CRS y con aquellas reglas agregadas por el usuario.
- **Coraza WAF (Exceptions):** componente encargado de determinar qué modelos deben ser ejecutados por WACE, aplicando las excepciones de modelos definidas.
- **Metrics Collector:** se encarga de recibir las métricas generadas por la solución y exportarlas en caso que sea necesario que se consuman por otros servicios para su posterior visualización.
- **Message System:** se encarga de comunicar WACE con ciertos modelos que sean configurados para utilizar este mecanismo. En el caso de los modelos asincrónicos siempre se utiliza este sistema para la comunicación, mientras en el caso de los modelos sincrónicos se define opcionalmente mediante un parámetro de configuración.

Teniendo una noción general de los componentes presentes en la solución, se dará una descripción de los componentes internos de WACE. Estos últimos pueden ser observados en la figura 4.2 donde se puede observar además si deben ser modificados en el contexto del proyecto o simplemente deben ser utilizados.

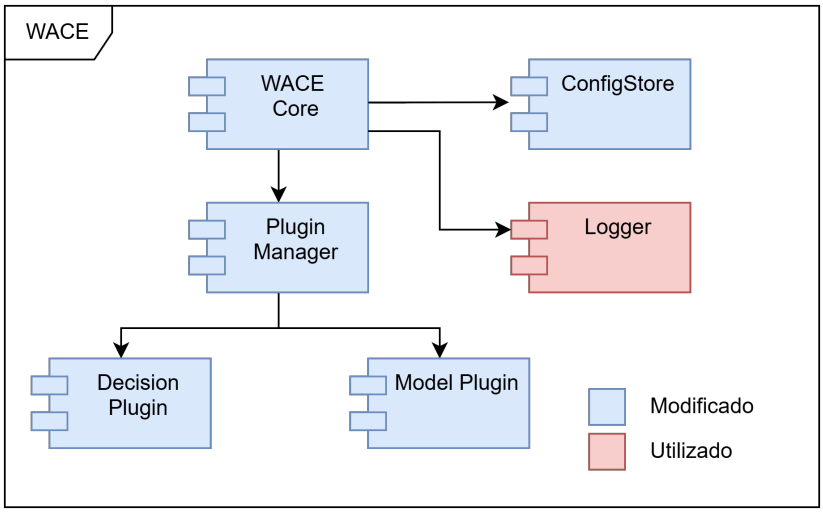


Figura 4.2: Componentes de WACE

- **WACE Core:** componente encargado de recibir los contenidos de la transacción a analizar, en las distintas fases. Además, implementa la función que retorna los resultados de la evaluación de la transacción. Se encarga de inicializar los restantes componentes de WACE.

  Entre los cambios a realizar en este componente se encuentra la eliminación de las funcionalidades asociadas a la comunicación con gRPC. Esto a se debe al análisis presentado en el capítulo anterior, donde se explicó la conveniencia de que WACE sea una librería. Por otra parte se debe ajustar las funciones expuestas por la librería, con el fin de que en un futuro sea adaptable a fases dinámicas. Esto último implica el pasaje de una función por tipo de modelo a una única función parametrizada.
- **Plugin Manager:** Este componente se encarga de cargar, gestionar y ejecutar los plugins de modelo así como los de decisión. Dichos plugins extienden la funcionalidad de una aplicación sin necesidad de modificar ni volver a compilar el programa principal.

Este componente proporciona amplia flexibilidad para agregar modelos dado que abstrae a la herramienta de la implementación de su comunicación con los modelos.

Para la compatibilidad entre el Plugin Manager y los plugins se define una interfaz a ser implementada por éstos últimos. Dicha interfaz se detallará en la sección de implementación.

Los cambios a realizar sobre este componente implican nuevamente adaptar las funciones para que sean compatibles con el uso de fases dinámicas, al igual que en el componente anterior implica parametrizar las funcionalidades actuales. Por otra parte se debe adaptar este componente a la nueva interfaz que se defina para los plugins. Por último se debe generar el soporte para el nuevo modo de ejecución (modo asincrónico).

- **ConfigStore:** Este componente es el encargado de almacenar la configuración de los modelos y decisiones existentes. Le corresponde además validar la configuración. Deberá ser adaptado a los nuevos parámetros de configuración (modo de ejecución en modelos, parámetros para la conexión con el sistema de mensajería, etc.). Por otra parte, se decidió que este componente deje de ser el encargado de cargar la configuración, por lo que se debe retirar esta funcionalidad.

- **Logger:** Este componente se encarga realizar el logging configurado por el usuario, donde se define un nivel de registro y un archivo en donde se realiza la escritura de logs. El nivel de registro posee cuatro opciones de ejecución que son: ERROR, WARN, INFO y DEBUG.

- **Decision Plugin:** este tipo de plugin permite agregar a WACE un algoritmo de decisión, dicho algoritmo puede utilizar los resultados de los modelos, los puntajes de OWASP CRS y otros parámetros definidos por el usuario con el objetivo de tomar decisiones de bloqueo. Este componente deberá ser modificado de acuerdo con la nueva interfaz que se defina.

- **Model Plugin:** este tipo de plugin implementa la comunicación entre WACE y un modelo de aprendizaje automático. Al igual que en el caso del componente anterior, se deben modificar los plugins de modelo de acuerdo con la nueva interfaz que se defina.

Habiendo descrito los componentes de la solución y en particular aquellos que integran a WACE, en la próxima sección se trasladará el foco a la integración con Coraza WAF.

4.2. Comunicación entre Servidor Web y Coraza WAF

En esta sección se profundizará en la comunicación estándar entre un servidor web y Coraza WAF. Se hablará haciendo referencia a la integración existente con Caddy, que actualmente es la única que se encuentra estable. Como se introdujo en la sección 3.3 esta comunicación es responsabilidad de un "conector" de coraza.

Teniendo en cuenta la figura 3.3 se puede observar que hay 3 componentes visibles de Coraza para el conector: `WAF Wrapper`, la interfaz `WAF` y la interfaz `Transaction`.

- **WAF Wrapper**: Este componente es el único que tiene la capacidad de crear estructuras `WAF Engine` de manera externa a Coraza. La funcionalidad para obtener una de estas estructuras es llamada cuando se inicializa el servidor, se genera un `WAF Engine` por cada virtual host definido. Dichas estructuras `WAF Engine` cumplen la interfaz `WAF`, es decir que implementan las funciones que define dicha interfaz.

- **Interfaz WAF**: Esta interfaz posee la funcionalidad de crear transacciones, dicha funcionalidad es invocada cuando se recibe una nueva solicitud HTTP y retorna una estructura que cumple la interfaz `Transaction`.

- **Interfaz Transaction**: La interfaz `Transaction` posee todas las funcionalidades asociadas a el envío de datos a Coraza, como por ejemplo "AddRequestHeader" donde se envía un encabezado de la solicitud HTTP a Coraza. Por otra parte también posee todas las funcionalidades de análisis para cada una de las fases, por ejemplo: "ProcessRequestHeaders".

Para poder realizar la integración con la capa intermedia se debe aprovechar este esquema para implementar estructuras propias que cumplan las interfaces mencionadas pero además realicen las funcionalidades asociadas a WACE. El conector sólo posee visibilidad sobre la interfaz y no su implementación, por lo que esta integración es transparente para él.

Además de las interfaces mencionadas también existe la interfaz `WAFConfig` que almacena la configuración del `WAF` hasta que éste es creado. Dicha interfaz también debe ser implementada, pero por tener menor relevancia no se brindará una descripción de sus funcionalidades.

En esta sección se ha profundizado sobre los componentes que participan en la comunicación entre el servidor web y Coraza WAF, se abordó de manera tal que se dilucidaron los cambios que se necesita introducir en el esquema de comunicación para obtener los flujos esperados. De la misma forma, en la siguiente sección se expondrán los aspectos que el diseño de la solución debe tener en consideración para la correcta integración con OWASP CRS.

4.3. Integración con OWASP CRS

Para utilizar las reglas que define OWASP CRS se debe contemplar que este conjunto de reglas está diseñado para evaluar transacciones y tomar decisiones de bloqueo sobre éstas. Además, se debe recordar que Coraza una vez que recibe una acción disruptiva por parte de la evaluación de reglas, no continúa con la ejecución de reglas para la transacción. Esto difiere del comportamiento deseado en la integración, por ejemplo, si se tiene una transacción donde OWASP CRS indica que se debe bloquear y por otra parte la decisión de WACE indica que no se debe bloquear, se debe respetar esta última decisión y el WAF debería poder continuar con el análisis de las reglas de las siguientes fases de manera normal.

Además del uso de las reglas de OWASP CRS, se debe tener en cuenta el uso de Virtual Patching. Como se introdujo en la sección 2.1, es una técnica necesaria en la operativa de WAFs donde se busca evitar que se exploten vulnerabilidades conocidas. A diferencia de las acciones disruptivas que generan las reglas de OWASP CRS, donde se espera que la decisión de bloqueo sea tomada en conjunto con WACE, los bloqueos por reglas de Virtual Patching deben detener la ejecución del análisis del WAF y WACE.

Lo que se propone en esta etapa es que el componente "WACE_WAF" modifique las reglas de decisión provistas por OWASP CRS para que no contengan acciones disruptivas, permitiendo que se evalúe todo el conjunto de reglas sin que se puedan generar interrupciones.

En caso de que se defina una regla de "Virtual Patching" esto generará una interrupción (a diferencia de la evaluación de reglas de OWASP CRS donde se desactivaron las acciones disruptivas), estas interrupciones deben detener el análisis de la transacción por parte de WACE.

Por otro lado, para que WACE pueda tomar la decisión se requiere de los puntajes calculados por OWASP CRS, para lograr esto se define agregar una regla en Seclang que obtenga estos puntajes permitiendo exportarlos y obtenerlos a través de la interfaz de Coraza.

Con lo expuesto en esta sección y la anterior se han tratado los desafíos técnicos presentes en la integración de WACE a la arquitectura de Coraza WAF y al flujo de funcionamiento que poseen las reglas de OWASP CRS. Las restantes secciones del capítulo se centrarán en el diseño de los componentes y flujos de WACE.

4.4. Model Plugins

Como se mencionó anteriormente, este componente se encarga de realizar la comunicación hacia el modelo de aprendizaje automático. A dicho plugin se le asociará un tipo, que actualmente puede

ser una de las siguientes opciones: RequestHeaders, RequestBody, AllRequest, ResponseHeaders, ResponseBody, AllResponse y Everything. Dichos tipos corresponden de alguna manera, a las fases de ejecución actuales que define Seclang e implementa Coraza. El tipo asociado indica simplemente qué datos se le enviarán al plugin, la interfaz que deben cumplir éstos últimos no está atada al tipo, por lo que esta configuración (el tipo de un plugin) puede ser modificada sin que implique volver a implementar el plugin nuevamente. Esto permitiría en un futuro generar nuevos tipos en caso de que se definan nuevas fases como se analizó en el capítulo anterior.

A continuación se hablará sobre sobre la interfaz de los plugins de modelo y posteriormente se ahondará sobre los modos de ejecución presentados en el capítulo anterior.

4.4.1. Definición de interfaz de modelos

Para la correcta comunicación entre WACE y los modelos se debe definir una interfaz en los plugins de modelo que permita el llamado para realizar la evaluación de las transacciones y al mismo tiempo devuelva el resultado de éstas. Es deseable además que la interfaz permita a quiénes desarrollen plugins de modelo, diseñar y enviar sus propias métricas mediante la herramienta.

Para definir dicha interfaz se debe tener en cuenta qué se desea realizar con los resultados de los modelos en la decisión. Se tienen los siguientes enfoques con respecto a los datos que retornan los plugins de modelo:

- Cada plugin de modelo es responsable de implementar una decisión basada en su conocimiento del modelo, en otras palabras, el plugin de modelo retorna un valor booleano indicando si considera que la transacción es un ataque. El algoritmo de decisión toma en cuenta las decisiones de cada plugin de modelo.
  - Fácil de integrar más modelos.
  - La configuración de la decisión es más simple. No se debe programar para integrar modelos.
- Cada plugin de modelo devuelve información que WACE en principio no sabe interpretar, se debe construir un algoritmo en un plugin de decisión que reciba dicha información y la procese apropiadamente para tomar una decisión.
  - Se gana generalidad, ya que en principio un modelo puede devolver cualquier tipo de información, el plugin de decisión debe conocer el funcionamiento de cada uno de los plugins de modelo que utilice, cuando hay más de un modelo se torna difícil de configurar y mantener.
- Un enfoque intermedio donde se definan distintos tipos de modelo, en base al tipo de información que retornan. Se debe adaptar los algoritmos de decisión para que sean capaces de manejar los tipos de modelos definidos.
  - Permite mayor riqueza en la información que brindan los plugins de modelos y al mismo tiempo permite que los plugins de decisión puedan adaptar la decisión según el tipo de modelo. Tipos de modelo pueden ser:
    - Modelos que retornan si una transacción es un ataque o no.
    - Modelos que retornan la probabilidad de ataque.
    - Modelos que retornen información de distintas categorías de ataque.
    - Modelos que aporten información contextual del tipo de transacción que se esta analizando. Por ejemplo confidencialidad de los datos.

    Se debe destacar que con el enfoque de que los plugins devuelvan información de distintas categorías se podrían implementar los otros enfoques. Para realizar esto se deben generar ciertas restricciones sobre el tipo de categorías que se deben agregar obligatoriamente. Teniendo esto en cuenta, se definió que los plugins de modelo puedan devolver

información de distintas categorías pero obligatoriamente devuelvan la categoría "probabilidad de ataque".

4.4.2. Modos de ejecución

Los plugins de modelo poseen dos modos de ejecución, uno en tiempo real respecto al procesamiento de la transacción (modo sincrónico) y otro modo donde el análisis de los modelos no es tomado en cuenta para la toma de decisión (modo asincrónico). Sin embargo, ésto es una configuración de los plugins de modelo y no altera la comunicación entre dicho plugin y su modelo, por lo que la configuración es intercambiable sin alterar el correcto funcionamiento del sistema. Se debe tener en cuenta que todo el procesamiento de los plugins de modelo es realizado en paralelo con respecto a las demás funcionalidades de WACE, incluso es paralelo entre distintos plugins de modelo.

4.4.2.1. Sincrónicos

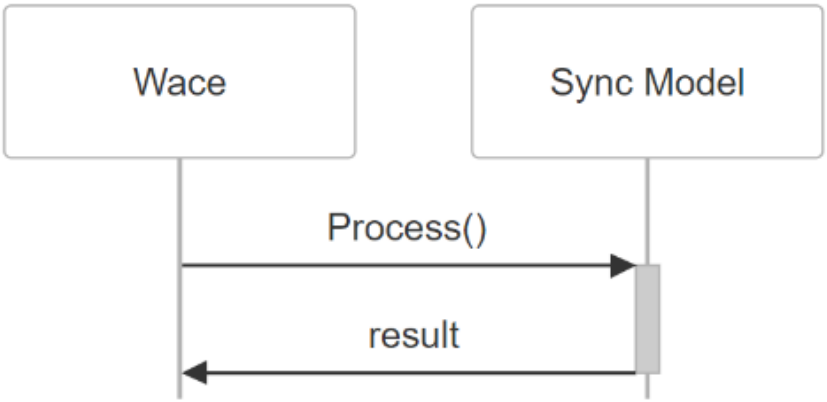


Figura 4.3: Plugin de modelo sincrónico

Como se puede visulizar en la figura 4.3, el modo de ejecución sincrónico indica que ante un llamado de evaluación a WACE para una transacción, dicho llamado debe esperar por los resultados del análisis de todo aquel plugin que ejecute en este modo.

Por defecto los plugins reciben invocaciones de análisis desde el Plugin Manager, en forma de llamados a una librería haciendo uso de la arquitectura de plugins, pero también pueden comunicarse con WACE a través del sistema de mensajes. La decisión de que se mantengan las dos opciones de comunicación para los modelos sincrónicos se tomó en base a mantener la mayor eficiencia y a la vez tener un diseño con una flexibilidad tal que permita en un futuro generar clientes de modelo remotos, que sólo se comuniquen con WACE haciendo uso del sistema de mensajes.

4.4.2.2. Asincrónicos

La finalidad del modo de ejecución asincrónico es la de comunicar modelos con mayor precisión pero que cuyo tiempo de análisis sea superior al tiempo máximo de procesamiento de una transacción para una aplicación particular.

Este tipo de plugin se comunica con WACE para realizar el análisis a través del sistema de mensajes. En la figura 4.4 se puede observar el flujo simplificado que se espera tengan los plugins en este modo de ejecución. En contraposición con el modo sincrónico, la función de chequeo de los resultados para una transacción no se bloquea en espera de los resultados de los plugins que

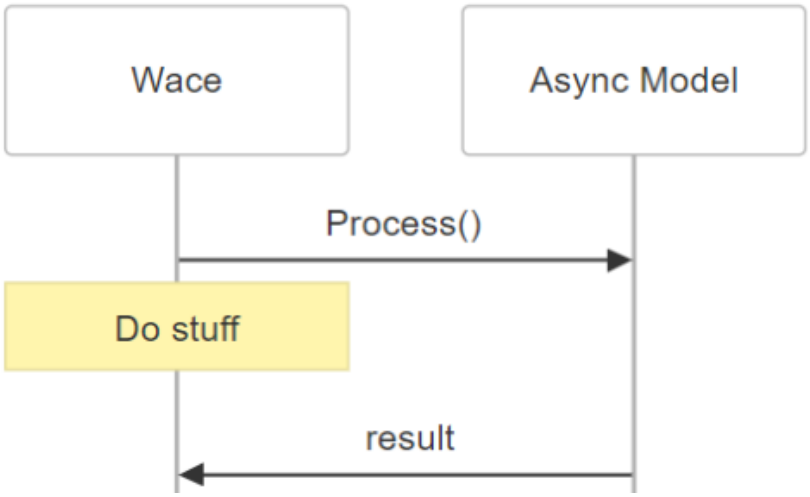


Figura 4.4: Plugin de modelo asincrónico

ejecutan en modo asincrónico, sino que su objetivo es el de envío de alertas en caso de que se detecte que un ataque no fue bloqueado.

Como se mencionó anteriormente, un componente fundamental para el funcionamiento de este modo es el sistema de mensajes, por lo que a continuación se explicará el funcionamiento de la tecnología a utilizar.

4.5. Sistema de mensajes

La motivación principal para este componente es la de comunicar y almacenar de manera temporal los pedidos que deben procesar los plugins de modelos. Por otro lado, también permitiría a futuro generar una vía de comunicación para clientes remotos.

La tecnología a utilizar es NATS, dado que es un sistema de mensajería y colas altamente optimizado y es fácilmente integrable en la herramienta a desarrollar.

NATS permite generar clientes, a quienes se les define una función a ejecutar (*callback*). Cuando el servidor le notifica y envía el mensaje al cliente, se ejecuta esa función.

En particular este componente se utilizará en el flujo de la funcionalidad de análisis, donde recibirá tanto mensajes de transacciones a procesar como resultados del procesamiento de los modelos.

4.6. Flujo de análisis

En la figura 4.5 se presenta el flujo de ejecución de la función definida para realizar la invocación a los modelos de aprendizaje automático con el fin de que realicen el análisis de la transacción. El tipo de modelos a invocar dependerá de la etapa en la que se encuentre la transacción.

Una vez invocado el llamado para el análisis en WACE Core, este se encarga de realizar el llamado al Plugin Manager para cada uno de los modelos. Al mismo tiempo, se deben generar los mecanismos para la recepción de los resultados, discriminando entre el modo de ejecución de dichos modelos.

El componente Plugin Manager actúa como un dispatcher, encargándose de gestionar la información sobre los plugins de modelo y su modo de ejecución. Dependiendo de este modo, el Plugin Manager determina el flujo adecuado para asegurar que los plugins reciban la información de la forma esperada. En el caso de los plugins sincrónicos esto se resume a un simple llamado a la

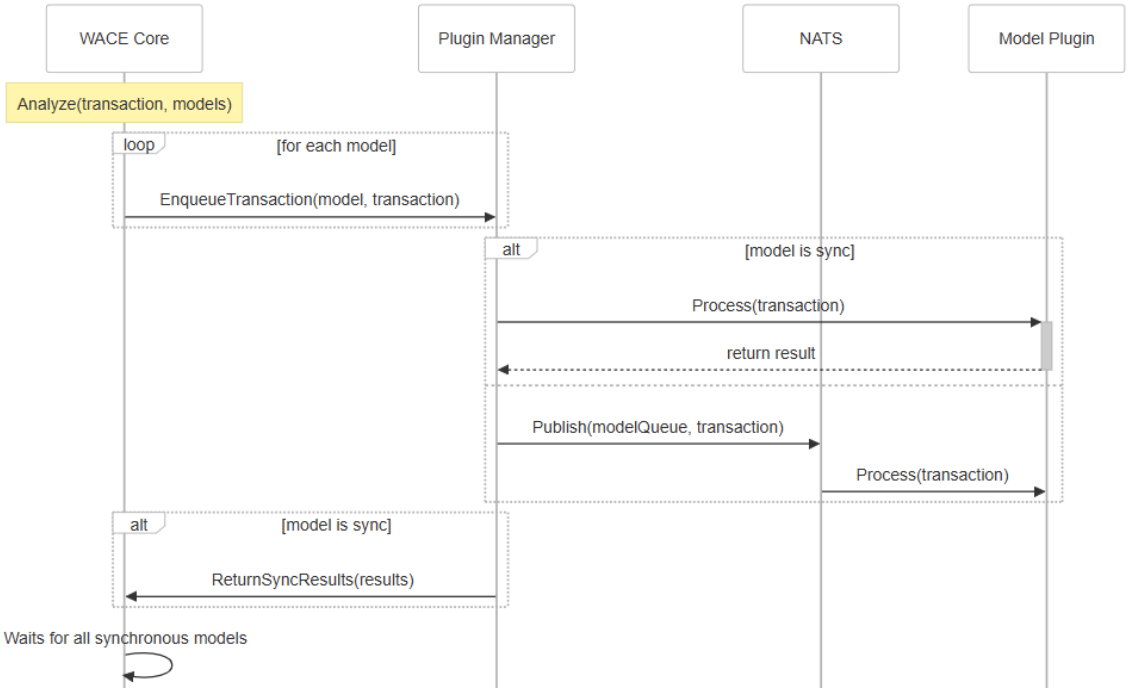


Figura 4.5: Diagrama de flujo de la funcionalidad Analyze

función de Process del plugin, pero para el modo asincrónico esto implica insertar los datos de la transacción en la cola asociada al modelo en el sistema NATS, desde el cuál el plugin de modelo recibe mensajes para procesar. Una vez realizado el procesamiento por los plugins asincrónicos, éstos ingresan los resultados del análisis en una cola de resultados asociada al modelo. Se debe tener la transacción identificada tanto en los pedidos a analizar como los resultados del análisis, dado que no está garantizado que los modelos procesen los mensajes en el orden que son ingresados en la cola.

4.7. Flujo de chequeo

En esta sección se aborda el flujo de la función de chequeo de transacciones implementada por WACE.

En la figura 4.6 se presenta el flujo de ejecución para la función encargada de realizar la toma de decisión de bloqueo sobre la transacción con los resultados de los modelos de aprendizaje automático.

Esta funcionalidad devuelve el resultado de la ejecución del algoritmo de decisión elegido, con los datos acumulados de los modelos hasta el momento y los parámetros del WAF que se le envíen.

Al comienzo del flujo de la funcionalidad de chequeo se debe esperar que finalicen todos los llamados previos a la función “Analyze”, teniendo en cuenta el flujo presentado en la sección anterior esto implica indirectamente que esta función espera a que todos los plugins de modelo en modo sincrónico devuelvan un resultado. Se debe tener en cuenta que esta función puede ser llamada en múltiples ocasiones, dado que el WAF va acumulando más información a medida que avanza en el análisis de la transacción y más modelos se ejecutan. Tiene sentido por lo tanto que se retorne la decisión de bloquear una transacción si el algoritmo de decisión así lo determina, aún cuando no se haya analizado toda la transacción por el WAF. El plugin manager debe mantener los resultados de los modelos hasta que WACE Core haga explícito que una transacción finalizó.

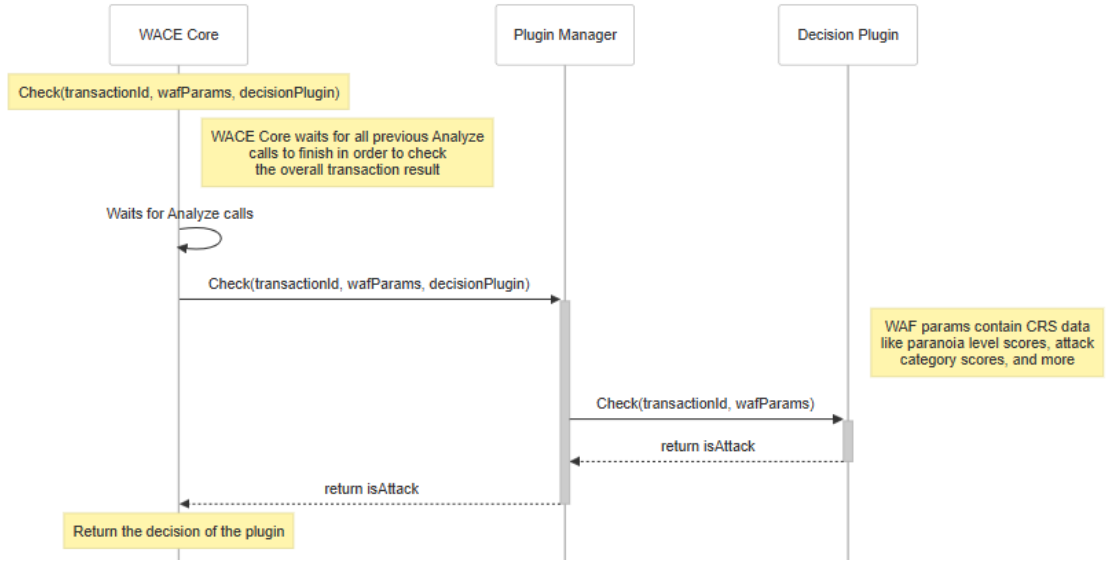


Figura 4.6: Diagrama de flujo de la funcionalidad Check

Esto es accionado sobre WACE Core por parte de WACE_WAF cuando la transacción finaliza su procesamiento por parte del WAF.

4.8. Métricas

En esta sección se presentará el diseño para la resolución de los distintos aspectos que presenta la problemática de las métricas y por otra parte se justificará el uso de las tecnologías elegidas.

Existen cuatro aspectos que la solución debe permitir: recolectar métricas en distintas partes del código; almacenarlas; permitir su visualización; y finalmente se debe permitir la configuración de alertas.

Las métricas serán recolectadas y enviadas con OpenTelemetry, tecnología que como se comentó anteriormente, no restringe el mecanismo por el cuál las métricas deben ser consumidas, sino que por el contrario dichas métricas son enviadas a un colector de métricas que es agnóstico en cuanto al procesamiento y receptor final de dichas métricas.

Otra posibilidad para la recolección de métricas es utilizar Prometheus Client que expone un puerto para que un Servidor de Prometheus pueda realizar *pull* de las métricas. Sin embargo el uso de Prometheus Client restringe que el almacenamiento de métricas sea Prometheus.

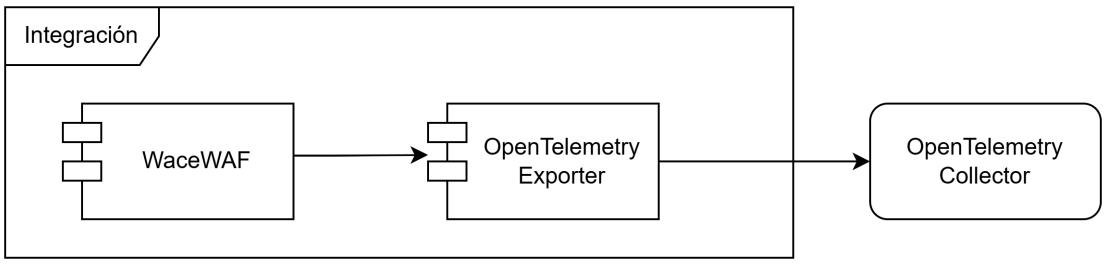


Figura 4.7: Comunicación mediante OpenTelemetry Exporter

En contraposición, el mecanismo para recolectar métricas con OpenTelemetry propuesto utiliza un "OpenTelemetry Exporter" para enviar las métricas a un colector. La comunicación propuesta puede observarse en la figura 4.7. En el colector se puede realizar procesamiento para permitir la integración con Prometheus, pero esto se realiza de manera externa a la herramienta desarrollada, sin reducir el espectro de posibilidades tecnológicas que se tiene para el almacenamiento y visualización de las métricas.

Para el almacenamiento de las métricas se utilizará Prometheus, que como se comentó en la sección 2.3.2 es el estándar de facto para el almacenamiento de métricas, es sencillo de desplegar y es muy utilizado en el ecosistema cloud. Esta tecnología es muy utilizada para la configuración de alertas. Por lo que es la tecnología a utilizar para el almacenamiento y la configuración de alertas. La integración de este componente a la solución resulta sencilla, ya que como se mencionó anteriormente, un OpenTelemetry Collector puede ser configurado de forma tal que exponga un servicio donde el servidor Prometheus haga *pull* de las métricas.

Aunque Prometheus permite generar visualizaciones y dashboards, la tecnología Grafana expuesta en la sección 2.3.2.1 posee una interfaz gráfica considerablemente más cómoda y personalizable, además de una integración muy sencilla con Prometheus. Por lo que será la tecnología que se utilizará para la generación de *dashboards* y visualización de métricas.

4.9. Arquitectura de despliegue

En esta sección se presentará el diagrama de despliegue de la solución diseñada que permite visualizar la integración con todos los componentes expuestos en este capítulo. Dicho diagrama puede observarse en la figura 4.8.

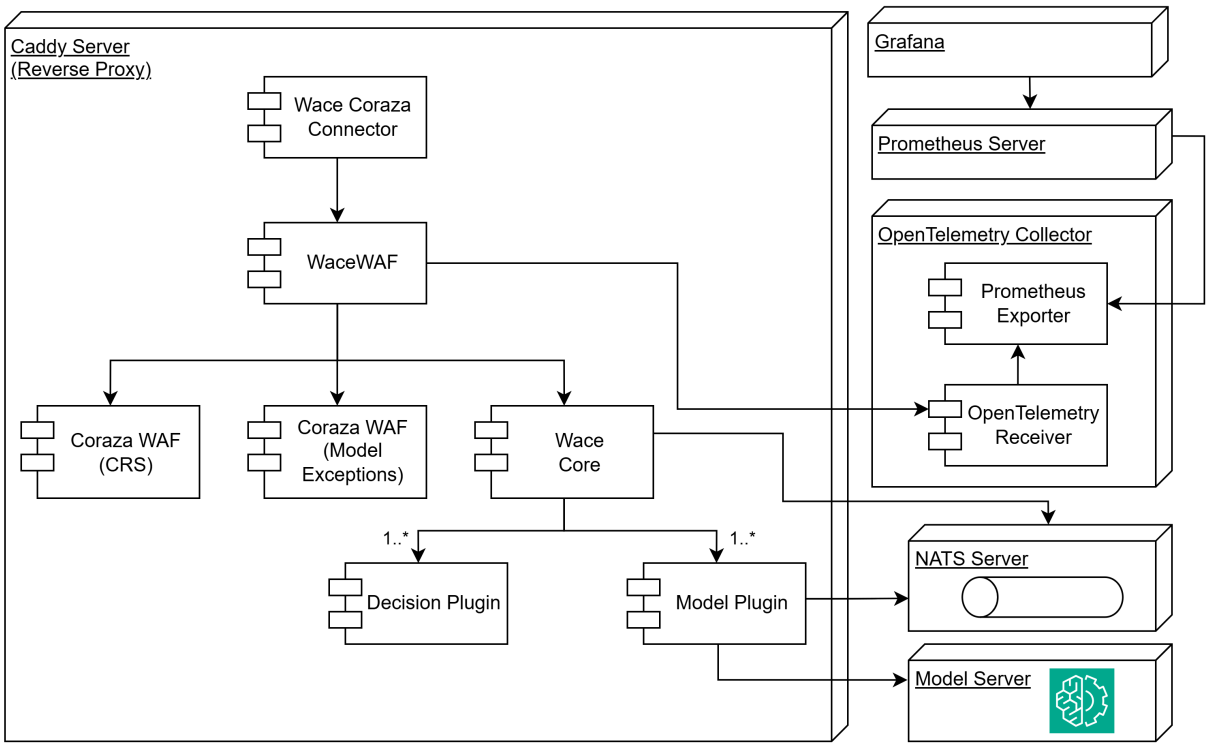


Figura 4.8: Diagrama de despliegue

En el mencionado diagrama pueden visualizarse la organización de los componentes en distintos

"servicios", que pueden o no ser ejecutados en un mismo *hardware*. El flujo operacional del sistema comienza en el servicio Caddy Server, donde se recibe el tráfico web dirigido a las aplicaciones web que protege. Dicho tráfico es analizado por Coraza WAF utilizando las reglas de OWASP CRS y por otra parte es enviado a WACE, quien invoca a los modelos de aprendizaje automático. El servicio de NATS es consumido por los plugins de modelo cuando estos ejecutan de manera asincrónica.

Las métricas generadas por el sistema son enviadas y a un colector de OpenTelemetry, que las recibe y las procesa de manera tal que puedan ser consumidas por el servicio Prometheus. Dichas métricas son visualizadas en Grafana.

Este diseño contempla distintos aspectos que le aportan robustez a la solución. En primera instancia tolerancia a fallos, ya que los componentes se encuentran desplegados en distintos servicios, lo que implica que no se tiene un único punto de falla. Esto permite que ante la indisponibilidad de alguno de los componentes, no se afecte a los demás. Por ejemplo, si se utilizan modelos en modo asincrónico mediante el servidor NATS y este fallara, su inactividad no afectaría el funcionamiento normal de la solución para la evaluación de transacciones en tiempo real.

En el contexto del proyecto se generará un ambiente de despliegue con docker compose[19] donde se definirán contenedores[20] para cada uno de los servicios expuestos en el diagrama presentado en esta sección. Esto responde a los requisitos de facilidad de instalación y replicación de componentes. El servicio debe estar disponible para su funcionamiento una vez los contenedores sean desplegados, esto implica que se debe generar la configuración que interconecte los componentes de manera previa al despliegue.

Capítulo 5

Implementación

En este capítulo se describirán los desafíos y decisiones técnicas tomadas durante el desarrollo de la solución, detallando cómo se implementaron componentes clave y las integraciones necesarias entre WACE y Coraza WAF. Se iniciará explicando el lenguaje de programación a utilizar para la implementación de la herramienta.

En la sección 5.2 se explicará con mayor profundidad las funcionalidades implementadas a nivel de la capa intermedia, así como desafíos técnicos que se presentaron durante el proceso de desarrollo y las soluciones encontradas.

Posteriormente se hablará de los cambios realizados en WACE, se presentarán las interfaces definidas tanto para la comunicación con WACE Core como para la implementación de nuevos plugins de modelo o decisión.

En la sección 5.4 se hablará de como se integró y utilizó NATS en el contexto de los plugins de modelo. Y se finalizará el capítulo explicando como se implementó la recolección de métricas y cuales fueron definidas y exportadas al *collector* de OpenTelemetry

5.1. Lenguaje de programación

Con el fin de cumplir con los requisitos planteados, el lenguaje de programación que utilice la solución debe ser adecuado, por ello debe ser eficiente, orientado al rendimiento y además es deseable que sea compatible con las tecnologías a integrar. En ese sentido el lenguaje Go es la opción que más se adecúa a la situación, dada su simplicidad, eficiencia y soporte nativo para concurrencia y sincronización, sin mencionar el hecho de que Coraza, Caddy y la versión previa de WACE están implementados en este lenguaje.

5.2. Capa intermedia

En esta capa se implementan las interfaces definidas por Coraza, en particular se implementa: la firma de la invocación que genera un nuevo `WAF`; la interfaz `WAF`; la interfaz `Transaction`; y la interfaz `WAFConfig`. En la figura 5.1 pueden verse los componentes que implementan la interfaz de Coraza, al mismo tiempo se puede apreciar las diferencias con respecto a la arquitectura original de Coraza que fue presentada en la figura 3.3. A continuación se dará una descripción de cada una de las interfaces que se implementan:

- La función que genera un nuevo `WAF` debe generar una nueva instancia de Coraza WAF para OWASP CRS, otra para excepciones si corresponde y finalmente la configuración de WACE para la aplicación web para la que se está creando el `WAF`. Se retorna un `WACE_WAF`.

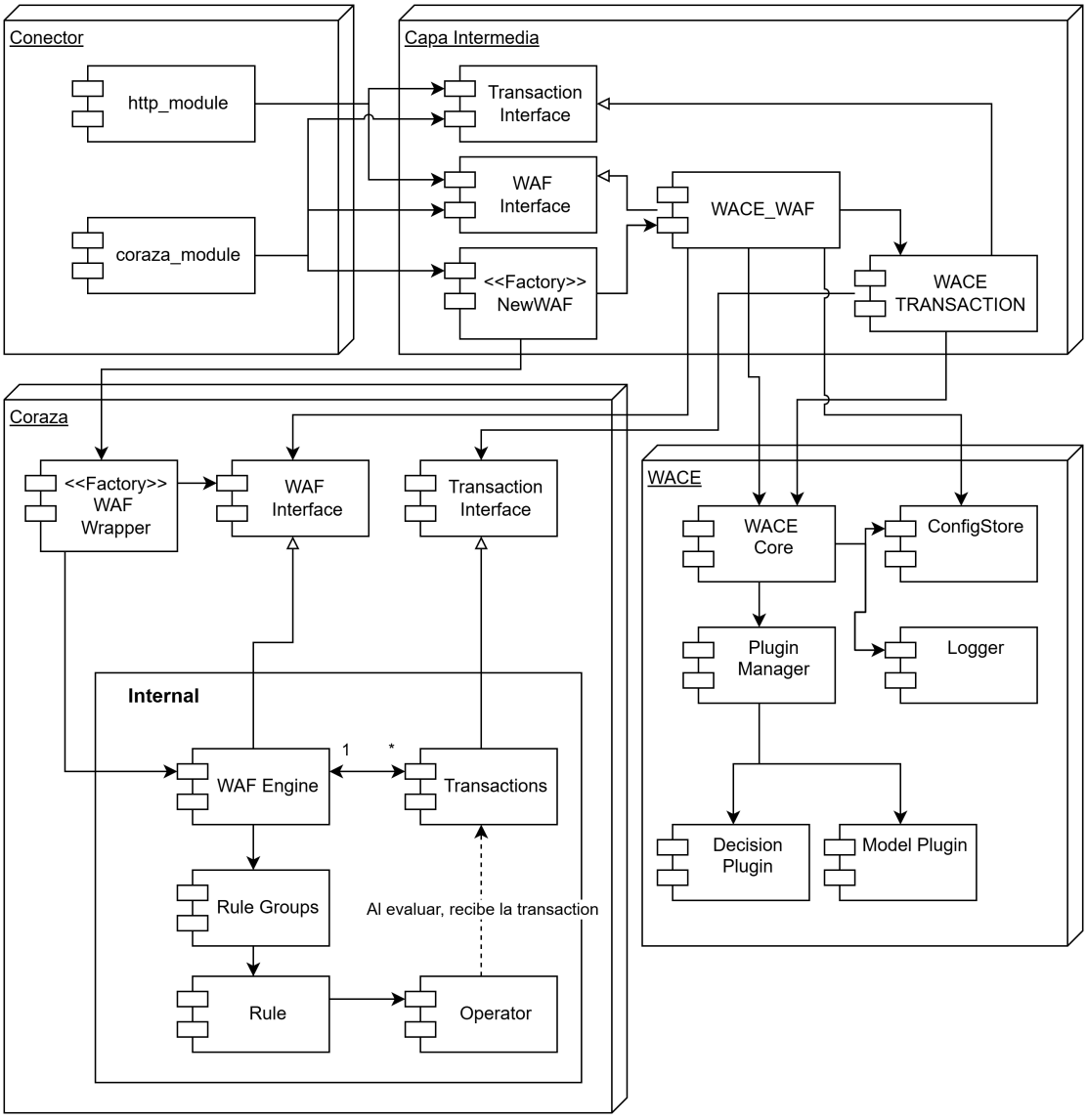


Figura 5.1: Diagrama de los componentes implementados en la capa intermedia

- La interfaz del `WAF` posee el llamado para generar nuevas transacciones. Se generó la estructura `WACE_WAF` que implementa dicha interfaz, en él se almacenan los `WAF`s de Coraza para la aplicación web protegida y se guardan las configuraciones de modelos que deben usarse para invocar a WACE. Este componente genera transacciones de tipo `WACE_Transaction`.
- Se generó el tipo `WACE_Transaction` que implementa la interfaz de las transacciones de Coraza. Dicha interfaz posee, para cada fase de Seclang, una función para agregar datos y otra para accionar el análisis del `WAF`. En ese sentido, `WACE_Transaction` envía los datos a todos los componentes en las funciones de agregar datos y acciona el análisis de todos los componentes en la función de análisis.
- Se generó el tipo `WACE_WAFConfig` que implementa las funcionalidades de la interfaz `WAFConfig`, donde se almacenan las configuraciones del `WAF`. En nuestro caso, con esta interfaz se almacenan también las configuraciones de WACE y el `WAF` de excepciones. Se agregó lógica basada en el nombre de los archivos, si el nombre del archivo contiene el string "waceconfig.yml" el componente almacena el contenido como configuración general de WACE, si contiene el string "waceappconfig.yml" se almacenan el contenido como configuraciones particulares de la aplicación y finalmente si contiene el nombre "waceexceptions.conf" se almacenan como configuraciones dirigidas al `WAF` de excepciones

5.2.1. Compatibilidad con servidor Caddy

Como se mencionó en la sección 2.1.5, Coraza posee una integración funcional con el servidor Caddy que permite utilizarlo como un reverse proxy y añadir Coraza para el análisis de transacciones. Utilizando esta integración que implementa un conector entre Coraza y Caddy, se modificó el punto donde se invoca a Coraza para que invoque a la solución desarrollada.

Con este fin, se crearon dos nuevos repositorios de código, uno que es un *fork* del repositorio de la integración de Coraza con Caddy y que contiene las modificaciones necesarias, y el otro repositorio contiene el proyecto en lenguaje Go necesario para poder compilar el servidor Caddy.

5.2.2. Excepciones de modelos

Para la configuración de las excepciones de modelos se estableció que se escriban en el mismo lenguaje con el que hoy funciona el WAF y en el que los administradores de WAF deberían escribir excepciones para OWASP CRS. En este sentido, desde la capa intermedia se crea un WAF dedicado al análisis de las excepciones, se inicializa una variable de Seclang por cada identificador de modelo, dichas variables se inicializan en **true** lo que indica que el modelo por defecto está activo y un usuario debe modificar el valor de la variable a **false** para exceptuar a dicho modelo de la ejecución de la transacción actual. Para comunicar si un modelo debe ser exceptuado o no, en la capa intermedia se cargan además un conjunto de reglas de resumen, cuyos rangos de IDs de Seclang son configurables por el usuario. Dichas reglas de resumen muestran el valor actual de las variables asociadas a cada modelo, dependiendo de la fase en la que se está ejecutando, en un formato conocido y fijo para la integración. De forma que los valores de dichas variables pueden ser extraídos de manera correcta por la capa intermedia.

5.2.3. Integración con OWASP CRS

En la etapa de diseño se consideró la posibilidad de poder implementar una integración de la solución con OWASP CRS en cualquiera de sus versiones, 2, 3 y 4. En esta línea, se definió una configuración a nivel de WACE_WAF donde es especificada dicha versión. Por otro lado, dado que Coraza funciona únicamente con la versión 4 de OWASP CRS, la implementación a nivel de la integración fue realizada únicamente para dicha versión.

En concreto, para poder evitar las reglas de evaluación de OWASP CRS que realizan las acciones de bloqueo, se agregaron reglas en Seclang con el siguiente formato:

```
SecRuleRemoveById id
```

Donde *id* representa los ID de las reglas de evaluación, estas reglas son las responsables de tomar las decisiones de bloqueo en OWASP CRS basadas en los puntajes acumulados por el análisis de las reglas sobre la transacción y los umbrales definidos. Se consideró realizar la implementación con la directiva SecRuleUpdateActionById pero esta no se encuentra implementada aún por Coraza. Esto hubiera permitido que se conserven las otras acciones realizadas por las reglas de evaluación aparte de la acción disruptiva de bloqueo.

Por otro lado, los puntajes acumulados por las reglas de OWASP CRS para cada transacción deben ser obtenidos por WACE_WAF para poder tomar la decisión. Lo que se realizó fue agregar una regla por cada fase de la transacción, que obtenga todos los puntajes. Las reglas utilizadas presentan el siguiente formato:

```
SecAction "id,phase,pass,noauditlog,msg:'SCORES'"
```

Donde SCORES se sustituye por todos los puntajes que utiliza el CRS en su versión 4, ya que se trata de muchos puntajes se omite su detalle en este punto. Se presenta uno de los puntajes a modo de ejemplo, donde se tiene los puntajes asociados a las reglas que coincidieron en fase 1 o 2 para cada uno de los niveles de paranoia (estos niveles se presentaron en la subsección 2.1.4):

```
inbound_per_pl=%{tx.inbound_anomaly_score_pl1}...%{tx.inbound_anomaly_score_pl4}
```

Por otro lado, cuando se quieren obtener los puntajes, se accede al campo *msg* de las reglas que evaluaron positivamente para la transacción, el cual es posible acceder desde la interfaz de Coraza y así poder exportar dichos puntajes.

5.3. WACE

Habiendo descrito la Capa Intermedia en la sección anterior, ahora se pondrá foco sobre el componente encargado de invocar a los modelos de aprendizaje automático, WACE.

La arquitectura de WACE estaba pensada para ser ejecutada como un servicio independiente, en el nuevo diseño esto fue cambiado, dejando al componente WACE Core como una librería. Se debió adaptar el componente para tener dicho comportamiento, en la misma línea se adaptó el componente ConfigStore para que sólo almacene la configuración y no se encargue de cargarla.

En la versión anterior de WACE, además, se tenía una función de análisis por cada tipo de modelo existente. Esto es contrario al diseño realizado que debe ser adaptable a fases dinámicas. A continuación se presentan las funciones nuevas adaptadas al nuevo diseño.

La función definida para realizar el análisis de una transacción en una fase dada según el tipo de modelos se denomina *Analyze* y posee la siguiente firma:

```
Analyze(modelType string, transactionId string, payload string, modelIds
↪  []string)
```

- Donde modelType indica el tipo de modelo que se va a ejecutar, este valor es de tipo string dado que se espera que en un futuro los tipos sean dinámicos y configurables por el usuario.
- TransactionId indica el valor único que identifica cada transacción a nivel de todo el sistema. Se asociarán a este valor todos los resultados, así como registros de logging y métricas.

- Payload es el contenido que será analizado por los modelos de aprendizaje automático. Modelos que son indicados utilizando el parámetro modelIds.

Por otro lado, para poder obtener los resultados de decisión sobre la transacción, se debe invocar a la función CheckTransaction:

```
CheckTransaction(transactionId string, decisionPluginId string, wafParams
↪  map[string]string)
```

En dicha función se indica:

- De que transacción se va a evaluar el resultado mediante el parámetro TransactionId.
- Se indica el algoritmo a utilizar mediante un identificador al plugin.
- Finalmente se envían a la función todos aquellos parámetros del WAF que se consideren necesarios.

5.3.1. Plugins de modelo

La interfaz de los plugins de modelo posee las siguientes funciones:

```
Init(params map[string]string, meter metric.Meter) error

Process(transactionId, payload string) (map[string]float64, error)
```

Donde la función Init se utiliza para inicializar el plugin, con los parámetros que el usuario considere e ingrese en la configuración. En dicha función se le comparte al plugin de modelo un medidor (`meter`) en el que se pueden generar métricas personalizadas.

Por otro lado el plugin posee la función Process, que dado un identificador de transacción y `payload` a procesar, retorna un map de categorías (que contiene la probabilidad de ataque) y potencialmente un error.

5.3.2. Plugins de decisión

La interfaz de los plugins de decisión posee las siguientes funciones:

```
Init(params map[string]string, meter metric.Meter) error

CheckResults(transactionId string, modelRes map[string]map[string]float64,
                WAFdata map[string]string) (bool, error)
```

Al igual que en los plugins de modelo, la función Init se utiliza para inicializar el plugin, con los parámetros que el usuario considere e ingrese en la configuración. En dicha función se le comparte al plugin de decisión un medidor (`meter`) en el que se pueden generar métricas personalizadas.

Por otro lado la función CheckResults, es donde se define el algoritmo de decisión, toma como entrada los resultados de los modelos y de OWASP CRS para una transacción y retorna si se debe bloquear o no la transacción.

La implementación por defecto para la toma de decisión utiliza una suma ponderada con parámetros de pesos cargados utilizando la función Init, dichos parámetros se configuran dentro de la configuración del plugin de decisión.

5.4. Integración con NATS

En esta sección se describirán los detalles de la implementación de la integración de WACE con el sistema NATS.

Para la comunicación con NATS se definió utilizar subcripciones asincrónicas de NATS, esto permite generar un *callback* a ejecutar cuando se recibe un mensaje en la cola donde se está subscrito. Este mecanismo se utilizó tanto dentro del plugin manager para esperar por mensajes en las colas de resultados, así como para la recepción de mensajes con *payload* a procesar por los clientes. En particular, se generó una función (InitNATS) que ejecuta la función Init de los plugins de modelo y posteriormente define una función de *callback* que utiliza la función Process de los plugins de modelos, logrando así que las funciones a implementar para clientes sincrónicos como asincrónicos sea la misma. Cabe destacar que InitNats hoy se definió a nivel del plugin de modelo, sin embargo, el objetivo es que esta funcionalidad se distribuya para la implementación de clientes remotos en un futuro.

5.5. Métricas

En esta última sección se profundizará en la implementación de la recolección de métricas y se listarán las métricas generadas.

Para la generación de métricas se utiliza la API que provee OpenTelemetry para la tarea mencionada. En primera instancia se crea un Meter Provider donde se le define la frecuencia con la que las métricas serán exportadas hacia el colector de OpenTelemetry y la URL donde este se localiza.

La implementación de cada una de las métricas se realiza en distintos componentes. A continuación se listan los componentes con sus respectivas métricas.

WACE_WAF

http.client.request.processed.total	Total de transacciones procesadas
http.client.request.processed.duration.nanoseconds	Tiempo que demora en procesarse la transaccion
http.client.request.processed.CRSExecTime.nanoseconds	Tiempo de procesamiento insumido por el CRS
http.client.integration.processed.duration.nanoseconds	Tiempo de procesamiento insumido por WACE_WAF
http.exceptions.duration.nanoseconds	Tiempo de procesamiento insumido para evaluar excepciones de modelo
http.client.request.blockedp1.total	Total de transacciones bloqueadas en fase 1
http.client.request.blockedp2.total	Total de transacciones bloqueadas en fase 2
http.client.request.blockedp3.total	Total de transacciones bloqueadas en fase 3
http.client.request.blockedp4.total	Total de transacciones bloqueadas en fase 4

Tabla 5.1: Métricas WACE_WAF

WACE Core

wace.client.request.blocked.total	Total de transacciones bloqueadas por WACE
wace.model.duration.nanoseconds	Tiempo insumido en una invocacion al modelo con identificador ModelID

Tabla 5.2: Métricas WACE Core

Para la elección de nombres se siguió la guía de buenas prácticas en la nomenclatura de métricas de Prometheus[43], que es la herramienta utilizada para almacenar las métricas en el contexto del proyecto.

Capítulo 6

Experimentación

En este capítulo se documentarán las pruebas realizadas para evaluar el rendimiento y funcionamiento de la solución desarrollada. Se llevaron a cabo pruebas orientadas a medir el desempeño, en busca de conocer el rendimiento que tiene utilizar la herramienta desarrollada en comparación al uso exclusivo de Coraza WAF junto con OWASP CRS. Además, se experimentó con la ejecución de la solución utilizando únicamente la memoria del sistema en comparación a la solución utilizando únicamente el sistema de mensajería NATS para la comunicación con los plugins de modelo. A fin de realizar un análisis riguroso, se aplicó el mismo conjunto de pruebas en ambos modos y bajo un ambiente controlado.

Con respecto a la organización del capítulo, primero se presenta la descripción detallada del ambiente de pruebas, incluyendo la configuración del hardware y el software empleados. Luego, se definen y describen las pruebas realizadas, incluyendo los casos de prueba específicos y las métricas clave analizadas.

El propósito de las pruebas que se presentan en este capítulo es identificar y analizar las diferencias de rendimiento entre la herramienta y Coraza WAF, y por otra parte las diferencias entre la utilización o no de NATS, determinando los factores que impactan en el tiempo de procesamiento y evaluando según el tipo de tráfico y las características de la aplicación protegida.

6.1. Ambiente

En esta sección se describe el ambiente utilizado para realizar las pruebas. Además, se detallan las configuraciones de la herramienta que serán utilizadas.

En primer lugar, el dispositivo utilizado para las pruebas posee las siguientes características.

CPU	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz [24]
Memoria RAM	16 GB
Sistema Operativo	Windows 11 Home 24H2

Tabla 6.1: Ambiente de pruebas

A su vez, es importante aclarar, que debido a que los plugins del lenguaje Go solamente tienen soporte para sistemas operativos Linux, las pruebas se realizaron sobre un WSL [29] instalado en el dispositivo descrito anteriormente, lo que implica que los recursos disponibles fueron inferiores a los descritos en la tabla, en particular se cuenta con 8GB de RAM. Por otro lado, para ejecutar los servicios correspondientes a métricas y sistema de mensajes, se utilizó la tecnología Docker [10]. Como servidor de backend se utilizó WordPress [52]. Por más detalles de cómo fue desplegado el ambiente, se puede seguir la documentación en el siguiente repositorio de la solución [26].

Una configuración que se utilizó a lo largo de todas las pruebas, fue utilizar plugins de modelo que implementan la interfaz en forma trivial, retornando una respuesta inmediatamente después de su invocación. Esto se realiza con el objetivo de poder evaluar el funcionamiento y rendimiento de la solución desarrollada sin tener de por medio el procesamiento de modelos que están por fuera del alcance del proyecto. De esta manera, se puede medir y evaluar en forma específica los componentes relevantes.

6.2. Escenario de prueba

En la presente sección se detallará el escenario definido para los distintos experimentos. Para llevar adelante las pruebas se define un escenario en el cual se tiene dos perfiles de usuario. Uno intenta simular trafico válido de la aplicación, mientras que el otro intenta ser un atacante.

En el caso del usuario válido se define el acceso a la página de inicio de la aplicación (`/`) y la creación de `posts`, a través del recurso `/wp-json/wp/v2/posts`.

Para el usuario atacante se define un intento de ataque de XSS sobre el parámetro de búsqueda en la página de inicio, para esto, el atacante intentará el acceso al siguiente recurso: `/?s=<script>alert('XSS')</script>`. Además, se define un ataque fuerza bruta sobre el recurso de inicio de sesión `/wp-login.php`. Los intentos de este ataque nunca consiguen obtener las credenciales válidas de acceso, solo se utilizan los intentos para realizar las pruebas.

Por otro lado, se establece que la cantidad de pruebas en un periodo sea mayor para usuarios válidos que para atacantes, esto en línea con intentar simular el tráfico real que puede recibir una aplicación.

En concreto, se utilizará el siguiente esquema de pruebas:

Carga sostenida con 2 usuarios en paralelo	Se realizan los pedidos en las pruebas definidas anteriormente durante 5 minutos. Se utiliza un usuario de cada tipo definido.
Carga sostenida con 10 usuarios en paralelo	Se realizan los pedidos en las pruebas definidas anteriormente durante 5 minutos. Se utilizan cinco usuarios de cada tipo definido.
Carga sostenida con 25 usuarios en paralelo	Se realizan los pedidos en las pruebas definidas anteriormente durante 5 minutos. Se utilizan doce y trece usuarios de cada tipo.
Carga sostenida con 50 usuarios en paralelo	Se realizan los pedidos en las pruebas definidas anteriormente durante 5 minutos. Se utilizan veinticinco usuarios de cada tipo definido.

Tabla 6.2: Esquema de pruebas

Para el esquema definido en la Tabla 6.2 se define realizar pruebas: sobre el servidor de backend directamente, a través de Coraza WAF, a través de la herramienta utilizando los plugins de modelo en memoria y utilizando los plugins de modelo mediante NATS. De lo mencionado se desprende un conjunto de 16 pruebas a realizar. Es importante destacar que para la ejecución de estas pruebas se utiliza la herramienta Locust[28].

6.3. Selección de métricas

Para el análisis de los resultados de las pruebas se van a utilizar las métricas de promedio, percentiles 95 % y 99 % correspondientes al tiempo de respuesta de cada pedido realizado. La

selección de las métricas es un punto clave para realizar el análisis, por lo que daremos una breve justificación de por qué considerar cada una de las métricas mencionadas en este contexto.

El promedio proporciona un valor medio del tiempo de respuesta, dando una idea global del rendimiento del sistema. Se debe considerar que esta métrica es sensible a valores extremos.

Un percentil indica el tiempo de respuesta por debajo del cual se encuentra un determinado porcentaje de las solicitudes. La elección de los percentiles 95 y 99 responde a identificar el rendimiento típico de la mayoría de los usuarios, excluyendo los valores extremos que pueden distorsionar la visión general, ademas, el percentil 99 es una métrica mas estricta ya que considera una mayor cantidad de datos, solo excluyendo los valores más extremos.

6.4. Resultados de ejecuciones

En esta sección se presentan los resultados obtenidos de las pruebas realizadas. Se comienza con la Tabla 6.3 que contiene los resultados correspondientes a los experimentos todos los datos presentados corresponden a tiempos de respuesta medidos en milisegundos (ms).

	WordPress	Coraza WAF	WACE WAF (Memoria)	WACE WAF (NATS)
Carga sostenida con 2 usuarios en paralelo	46	82	84	84
Carga sostenida con 10 usuarios en paralelo	72	107	111	116
Carga sostenida con 25 usuarios en paralelo	89	205	215	220
Carga sostenida con 50 usuarios en paralelo	260	499	535	573

Tabla 6.3: Promedio de tiempos de respuesta

	WordPress		Coraza WAF		WACE WAF (Memoria)		WACE WAF (NATS)	
Percentil	95	99	95	99	95	99	95	99
Carga sostenida con 2 usuarios en paralelo	80	100	180	190	180	190	190	200
Carga sostenida con 10 usuarios en paralelo	130	200	250	310	260	300	280	330
Carga sostenida con 25 usuarios en paralelo	200	280	560	660	570	690	590	710
Carga sostenida con 50 usuarios en paralelo	550	690	1500	1700	1600	1900	1600	1900

Tabla 6.4: Percentiles

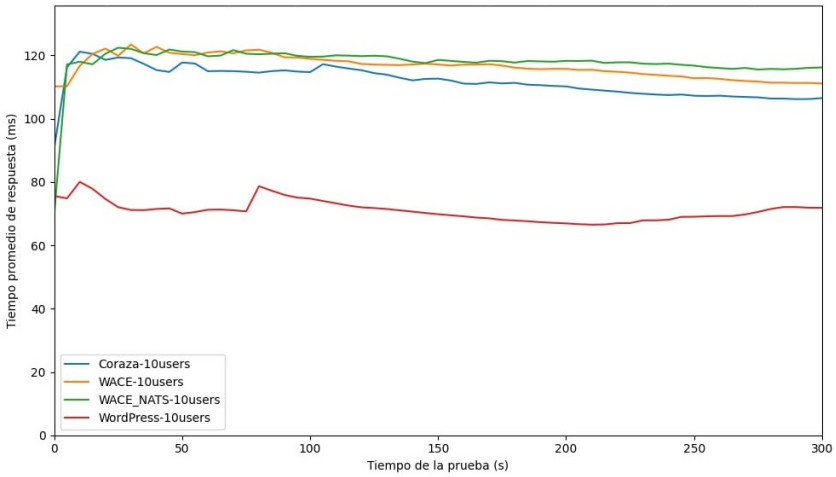
En líneas generales se puede ver que accediendo directamente al servidor de backend, los tiempos de respuesta son mucho menores que haciéndolo mediante uno de los WAFs.

Por otro lado, para la métrica promedio es posible visualizar un aumento sostenido a medida que aumenta la cantidad de usuarios de la prueba, esto es esperable ya que se aumenta la carga del sistema mientras los recursos permanecen fijos. Respecto a los percentiles se aprecia que el tiempo

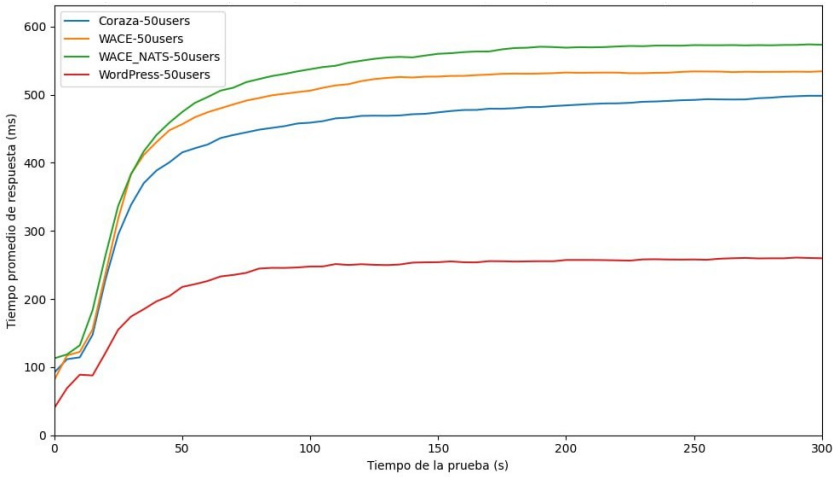
de respuesta de Coraza WAF con respecto a la herramienta desarrollada es prácticamente igual, en el caso que se utiliza NATS se tienen tiempos mayores.

Si se analiza el peor caso obtenido para los tiempos de respuesta, se puede observar que Coraza WAF agrega una cantidad considerable de tiempo que asciende a 239ms en la prueba con 50 usuarios. Si bien este tiempo es elevado, se puede explicar por la cantidad limitada de recursos del ambiente de prueba. Además, si se compara Coraza WAF contra la herramienta WACE_WAF, se puede visualizar que el *overhead* agregado por WACE_WAF es bajo, en particular para el escenario más demandante (50 usuarios en paralelo) el aumento del tiempo promedio entre Coraza WAF y WACE_WAF con los modelos en memoria es del 7 %. En algunos casos se puede observar una tendencia a que estos valores converjan, como se puede apreciar en la figura 6.2 para el caso (a).

A continuación, se presentan gráficas que muestran el comportamiento en los tiempos de respuesta.



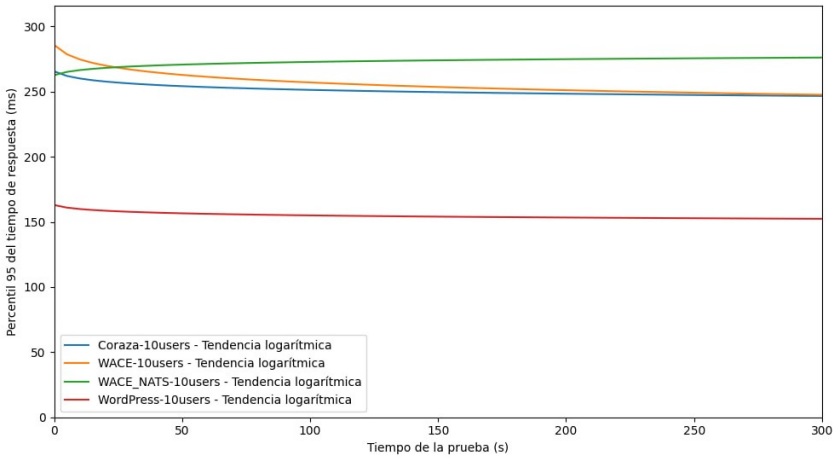
(a)



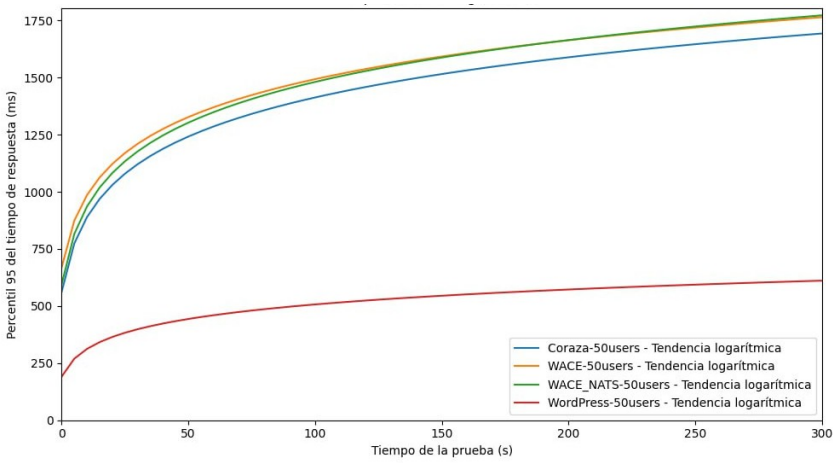
(b)

Figura 6.1: Tiempo de respuesta en función del tiempo de la prueba, dividido por caso de prueba, para (a) 10 usuarios y (b) 50 usuarios

Por otra parte, se analizarán los resultados de utilizar NATS como parte de la solución. Como se puede observar en la tabla 6.3, la utilización de NATS para el caso de una carga sostenida de 25



(a)



(b)

Figura 6.2: Percentil 95 de la aproximación logarítmica del tiempo de respuesta en función del tiempo de la prueba, dividido por caso de prueba, para (a) 10 usuarios y para (b) 50 usuarios

usuarios en paralelo, representa un incremento del promedio de los tiempos de procesamiento de las transacciones de 15ms más con respecto a Coraza y 5ms más con respecto a la solución utilizando la memoria. En general, es posible observar que cuando se utiliza NATS los tiempos son superiores si se lo compara con la ejecución a través de la memoria. Se debe tener en cuenta que los plugins de modelo que utilizan la memoria del sistema, en un caso realista se espera tengan conexiones remotas para invocar a los modelos, mientras que en el enfoque con NATS ya se está midiendo en estas pruebas la comunicación por la red. Por lo que en una situación realista se espera que la diferencia en el tiempo de ejecución con las dos alternativas sea más reducida.

Además, se realiza una comparación entre las estadísticas exportadas por la herramienta Locust y las generadas en el presente proyecto. Se utiliza como referencia el resultado de Locust para la misma prueba con 10 usuarios para el caso WACE_WAF (NATS) presentada en la tabla 6.3, y se compara contra las métricas generadas por WACE_WAF durante el transcurso de la prueba. En la figura 6.3 se presentan las visualizaciones que contiene el *dashboard* de Grafana que utiliza las métricas exportadas por la herramienta.

Una correcta implementación de la medición del tiempo de procesamiento insumido por WA-

Figura 6.3: Dashboard de métricas para una carga con 10 usuarios utilizando plugins de modelo con NATS

CE_WAF debería ser levemente inferior al tiempo percibido por Locust, dado que el tiempo que no es medido por WACE_WAF abarca el tiempo correspondiente a la comunicación entre el cliente y el servidor web.

Como es posible ver en la figura 6.3, el promedio del tiempo de procesamiento de las transacciones[1] es de 113ms. Este tiempo comprende desde que la transacción ingresa a WACE_WAF hasta que finaliza su procesamiento. Recordando el valor de la tabla 6.3 para esta prueba (10 usuarios, WACE_WAF NATS), el tiempo de respuesta promedio es de 116ms, este tiempo engloba desde que se envía el pedido desde la herramienta Locust hasta que se recibe la respuesta. Por lo tanto, se puede decir que la medición obtenida está de acuerdo a lo esperado, dado que en la prueba analizada la diferencia entre el tiempo percibido por el cliente y el medido por WACE_WAF es de 3ms.

Continuando con el análisis de la métricas exportadas en el contexto del proyecto, se analiza el tiempo insumido para evaluar las excepciones de modelos. En este sentido, se configuraron dos excepciones sobre modelos para probar cómo afectan al rendimiento, para medir esto, se toma el tiempo que tarda el procesamiento del WAF dedicado a evaluar excepciones. Se puede observar que el tiempo absoluto de procesamiento de excepciones[2] para la prueba presentada en la figura 6.3 no supera los 0.21ms milisegundos, lo que indica que aunque en el capitulo 3 (Análisis) se planteó que esta no era la solución ideal, se obtienen resultados favorables, ya que el tiempo insumido por este componente representa solamente el 0,18 % del tiempo total de la transacción (116ms).

En resumen, los resultados obtenidos están dentro de los rangos esperados, con datos prometedores respecto al desempeño de Coraza. Además, de la experiencia realizada se desprende que el uso de NATS puede ser muy beneficioso, ya que es un servicio que ofrece baja latencia en la comunicación, lo que lo hace atractivo para el despliegue de modelos que no se encuentren en el

[1] El promedio del tiempo de procesamiento de las transacciones corresponde al *widget* denominado *Avg transaction process time*.

[2] El promedio del tiempo de procesamiento de excepciones de modelos corresponde al *widget* denominado *Model exceptions process time*

mismo entorno de ejecución que la solución, sin añadir una carga significativa al tiempo de procesamiento de las transacciones. Sin embargo, es importante considerar el tamaño de los mensajes intercambiados por la aplicación protegida. Si la aplicación maneja un gran volumen de mensajes pequeños, el uso de NATS podría sobrecargar la comunicación y añadir un tiempo de procesamiento innecesario. En estos casos, podría ser más adecuado emplear los plugins directamente a través de la memoria del sistema.

Capítulo 7

Conclusiones y Trabajo Futuro

En este trabajo se ha desarrollado una solución integral que facilita la integración de modelos de aprendizaje automático en un WAF, con el objetivo de mejorar la precisión en la detección de amenazas y minimizar la tasa de falsos positivos. La herramienta implementada, denominada WACE_WAF, se diseñó para ser flexible y eficiente, permitiendo una fácil integración con Coraza WAF. La arquitectura de plugins implementada en WACE no solo permite la incorporación de diversos modelos de aprendizaje automático, sino también la ejecución de un proceso de toma de decisiones en tiempo real, lo cual optimiza la capacidad del WAF para identificar amenazas en entornos web de manera ágil y precisa.

Entre los principales beneficios logrados con esta solución destaca la opción de definir excepciones sobre los modelos que permiten un control más granular sobre el análisis de las transacciones, evitando sobrecargar el sistema y favoreciendo un rendimiento óptimo. La herramienta permite seleccionar criterios de decisión personalizados y, a su vez, soporta el uso de diferentes fases de ejecución, de modo que los datos de las transacciones se analicen en el momento oportuno y según el modelo que mejor se adapte a cada escenario.

Además, se integraron mecanismos de visualización y reporte de métricas que permiten a los usuarios monitorear en tiempo real el rendimiento y el estado del sistema, diferenciando entre los resultados del análisis mediante reglas de OWASP CRS y los provenientes de modelos de aprendizaje automático.

En términos de usabilidad, se puso énfasis en la facilidad de instalación y actualización de la herramienta utilizando docker-compose, permitiendo replicar la instalación y configuración de la herramienta en diferentes entornos.

Pasando a los objetivos específicos, a continuación se tratarán las conclusiones para cada uno de ellos.

- **Integración con OWASP Coraza WAF:** se ha logrado una integración efectiva entre WACE y Coraza WAF, permitiendo que ambos sistemas se comuniquen de manera fluida y compartan información para la toma de decisiones. La integración permite una toma de decisión en tiempo real y con base en múltiples fuentes de información.

- **Flexibilidad en los criterios de decisión:** la solución desarrollada provee una arquitectura flexible, capaz de adaptarse a distintas necesidades y contextos de seguridad. La posibilidad de configurar y personalizar los criterios de decisión según los requerimientos específicos amplía el alcance y la utilidad del sistema, permitiendo el uso de algoritmos que combinan tanto los resultados del conjunto de reglas de OWASP CRS como los modelos de aprendizaje automático, como por ejemplo, el algoritmo de suma ponderada utilizado por defecto.

- **Eficiencia:** el enfoque en la eficiencia ha resultado en una solución adecuada en el tiempo de procesamiento, lo que permite que el WAF funcione sin introducir latencias significativas en el tráfico de la red en comparación con Coraza WAF. Por otro lado, en cuanto al uso de

recursos, se tiene el problema de duplicar los datos de cada transacción para poder ejecutar por un lado OWASP CRS y por otro las excepciones de modelos.

- **Recolección y visualización de métricas:** la inclusión de un módulo de métricas (OpenTelemetry) permite una supervisión del rendimiento de la herramienta y sus componentes, ofreciendo datos clave para analizar su efectividad en tiempo real. Además, permite la generación de métricas personalizadas para quienes implementan plugins de modelo. La integración de herramientas de visualización facilita la interpretación de estos datos, lo que contribuye a una mejor comprensión del rendimiento. Esto brinda a los administradores de WAFs una mayor transparencia sobre el comportamiento de la herramienta. Otro aspecto relevante es que la recolección de métricas utiliza el estándar de OpenTelemetry por lo que está desacoplado de la herramienta de visualización de métricas.
- **Modos de ejecución de modelos:** se logró implementar mecanismos que permiten ejecutar los modelos de aprendizaje automático tanto en tiempo real como de manera asincrónica. Esta capacidad dual de ejecución permite que el sistema responda rápidamente a amenazas inmediatas en tiempo real y, al mismo tiempo, realice análisis más profundos en modo asincrónico, optimizando la carga de trabajo del WAF y mejorando la capacidad de respuesta ante diferentes tipos de ataques. Una línea de trabajo futura sobre estos mecanismos es la conexión mediante clientes remotos, cuya interacción con WACE se dé únicamente utilizando el sistema de mensajería NATS.

En conclusión, la integración de WACE con Coraza cumple satisfactoriamente con los requisitos funcionales y no funcionales planteados, alcanzando los objetivos definidos. Sin embargo, quedaron fuera del alcance del proyecto ciertas mejoras adicionales que se espera en un futuro puedan ser incorporadas en Coraza, como la implementación de un nuevo concepto de fase y la posibilidad de trasladar la integración con WACE a los componentes internos de Coraza WAF, mejorando la eficiencia ya que en dicho caso no sería necesario la implementación de una capa intermedia como fue realizado en este proyecto. Además, como se respetaron las interfaces de Coraza, los cambios a introducir para trasladar la integración no difieren notoriamente de la implementación realizada en el presente trabajo.

Bibliografía

[1] Gustavo Betarte et al. *ModSecIntl*. https://gitlab.fing.edu.uy/gsi/wafmind/-/blob/master/Articles/WP/modsecintl-overview.pdf. Último acceso: Noviembre 2024. 2024.

[2] Apache. *Apache License Version 2.0*. https://www.apache.org/licenses/LICENSE-2.0. Último acceso: Junio 2024. 2004.

[3] Gustavo Betarte, Álvaro Pardo y Rodrigo Martínez. «Web Application Attacks Detection Using Machine Learning Techniques». En: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 2018, págs. 1065-1072. DOI: 10.1109/ICMLA.2018.00174.

[4] Caddy. *Caddy server*. https://caddyserver.com/. Último acceso: Setiembre 2024. 2024.

[5] Cloudflare. *Reverse Proxy*. https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy/. Último acceso: Octubre 2024. 2024.

[6] IBM Corporation. *Virtual Host*. https://www.ibm.com/docs/en/was-nd/8.5.5?topic=hosts-virtual. Último acceso: Noviembre 2024. 2024.

[7] OWASP CRS. *Paranoia Levels*. https://coreruleset.org/docs/concepts/paranoia_levels/index.html. Último acceso: Octubre 2024. 2024.

[8] OWASP CRS. *Self-Contained Mode*. https://coreruleset.org/docs/miscellaneous/self_contained_mode/index.html. Último acceso: Noviembre 2024. 2024.

[9] Neus Canal Díaz. *Técnicas de muestreo. Sesgos más frecuentes*. https://www.revistaseden.org/files/9-CAP%209.pdf. Último acceso: Noviembre 2024. 2024.

[10] Docker. *Docker*. https://docs.docker.com/desktop/. Último acceso: Noviembre 2024. 2024.

[11] Christian Folini. *Talking about ModSecurity and the new Coraza WAF*. https://coreruleset.org/20211222/talking-about-modsecurity-and-the-new-coraza-waf/. Último acceso: Noviembre 2024. 2024.

[12] The Apache Software Foundation. *Apache HTTP Server*. https://httpd.apache.org/. Último acceso: Noviembre 2024. 2024.

[13] Martin Fowler. *Domain Specific Language (DSL)*. https://martinfowler.com/dsl.html. Último acceso: Noviembre 2024. 2024.

[14] Google. *Protobuf*. https://protobuf.dev/. Último acceso: Agosto 2024. 2024.

[15] Grafana. *Grafana*. https://grafana.com/docs/grafana/latest/. Último acceso: Octubre 2024. 2024.

[16] gRPC. *gRPC*. https://grpc.io/. Último acceso: Setiembre 2024. 2024.

[17] IEEE. *HTTP*. https://datatracker.ietf.org/doc/html/rfc2616. Último acceso: Octubre 2024. 2024.

[18] IEEE. *HTTP: Content-Length*. https://datatracker.ietf.org/doc/html/rfc7230#section-3.3.2. Último acceso: Noviembre 2024. 2024.

[19] Docker Inc. *Docker Compose*. https://docs.docker.com/compose/. Último acceso: Noviembre 2024. 2024.

[20] Docker Inc. *What is a Container?* https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-container/. Último acceso: Noviembre 2024. 2024.

[21] Cloudflare Inc. *Cloudflare Ruleset Engine Phases*. https://developers.cloudflare.com/ruleset-engine/about/phases/. Último acceso: Junio 2024. 2024.

[22] Cloudflare Inc. *Cloudflare Web Application Firewall*. https://developers.cloudflare.com/waf/. Último acceso: Junio 2024. 2024.

[23] InCo. *WAFMind*. https://www.fing.edu.uy/inco/proyectos/wafmind/. Último acceso: Noviembre 2024. 2024.

[24] Intel. *Intel Core i7 CPU*. https://www.intel.la/content/www/xl/es/products/sku/122589/intel-core-i78550u-processor-8m-cache-up-to-4-00-ghz/specifications.html. Último acceso: Noviembre 2024. 2024.

[25] Jaeger. *Jaeger*. https://www.jaegertracing.io/. Último acceso: Noviembre 2024. 2024.

[26] Agustín de León y Tobias Iroa. *Caddy WACE integration*. https://gitlab.fing.edu.uy/gsi/pgrado-wace/caddy_wace. Último acceso: Noviembre 2024. 2024.

[27] Agustín de León y Tobias Iroa. *Investigación del estado del arte en Web Application Firewalls*. Facultad de Ingeniería, Universidad de la República, 2024.

[28] Locust. *Locust*. https://locust.io/. Último acceso: Noviembre 2024. 2024.

[29] Microsoft. *WSL*. https://learn.microsoft.com/es-es/windows/wsl/install. Último acceso: Noviembre 2024. 2024.

[30] ModSecurity. *SecLang Actions*. https://github.com/owasp-modsecurity/ModSecurity/wiki/Reference-Manual-%28v3.x%29#actions. Último acceso: Setiembre 2024. 2024.

[31] Nicolás Montes et al. «Web Application Attacks Detection Using Deep Learning». En: *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*. Ed. por João Manuel R. S. Tavares, João Paulo Papa y Manuel González Hidalgo. Cham: Springer International Publishing, 2021, págs. 227-236. ISBN: 978-3-030-93420-0.

[32] NATS. *What is NATS*. https://docs.nats.io/nats-concepts/what-is-nats. Último acceso: Octubre 2024. 2024.

[33] OpenTelemetry. *OpenTelemetry*. https://opentelemetry.io/docs/what-is-opentelemetry/. Último acceso: Octubre 2024. 2024.

[34] Cuttica E. Outeda F. «WACE: Un integrador de clasificadores de ataques web». En: *Proyecto de grado, Instituto de Computación, Facultad de Ingeniería, Universidad de la República* (2021).

[35] OWASP. *Extending Coraza*. https://coraza.io/docs/reference/extending/. Último acceso: Agosto 2024.

[36] OWASP. *OWASP Coraza*. https://owasp.org/www-project-coraza-web-application-firewall/. Último acceso: Junio 2024.

[37] OWASP. *OWASP CRS*. https://owasp.org/www-project-modsecurity-core-rule-set/. Último acceso: Junio 2024.

[38] OWASP. *OWASP ModSecurity*. https://owasp.org/www-project-modsecurity/. Último acceso: Junio 2024.

[39] OWASP. *OWASP Top Ten*. https://owasp.org/www-project-top-ten/. Último acceso: Noviembre 2024. 2024.

[40] OWASP. *Web Application Firewall*. https://owasp.org/www-community/Web_Application_Firewall. Último acceso: Noviembre 2024. 2024.

[41] OWASP ModSecurity. *ModSecurity Reference Manual*. https://github.com/owasp-modsecurity/ModSecurity/wiki/Reference-Manual-%28v3.x%29. Último acceso: Octubre 2024. 2024.

[42] OWASP ModSecurity. *ModSecurity Wiki*. https://github.com/owasp-modsecurity/ModSecurity/wiki. Último acceso: Junio 2024. 2024.

[43] Prometheus. *Metric and label naming*. https://prometheus.io/docs/practices/naming/. Último acceso: Octubre 2024. 2024.

[44] Prometheus. *Prometheus*. https://prometheus.io/docs/introduction/overview/. Último acceso: Octubre 2024. 2024.

[45] Prometheus. *Prometheus server*. https://github.com/prometheus/prometheus. Último acceso: Setiembre 2024. 2024.

[46] Semver.org. *Semantic Versioning 2.0.0*. https://semver.org/. Último acceso: Noviembre 2024. 2024.

[47] SoundCloud. *SoundCloud*. https://soundcloud.com/. Último acceso: Agosto 2024. 2024.

[48] Sharvari T y Sowmya Nag K. «A study on Modern Messaging Systems- Kafka, RabbitMQ and NATS Streaming». En: *CoRR* abs/1912.03715 (2019). arXiv: 1912.03715. URL: http://arxiv.org/abs/1912.03715.

[49] Tilsor SA, Universidad de la República, Universidad Católica del Uruguay. *Web Attack Classification Engine (WACE)*. https://github.com/tilsor/ModSecIntl_wace_core. Último acceso: Julio 2024. 2022.

[50] Inc Trustwave Holdings. *About us*. https://www.trustwave.com/en-us/resources/security-resources/software-updates/end-of-sale-and-trustwave-support-for-modsecurity-web-application-firewall/. Último acceso: Noviembre 2024. 2021.

[51] Uptrace. *Prometheus vs Grafana*. https://uptrace.dev/blog/grafana-vs-prometheus.html#grafana-vs-prometheus-conclusion. Último acceso: Octubre 2024. 2024.

[52] WordPress. *WordPress*. https://wordpress.com/es/. Último acceso: Noviembre 2024. 2024.

Anexo A

Discusión del cumplimiento de requisitos en las distintas integraciones

Dentro del análisis de las posibilidades de arquitectura para la herramienta e integración, se presenta las distintas alternativas en los distintos ejes de discusión. Posteriormente se analiza la compatibilidad entre las opciones de los distintos ejes. Estos ejes son:

1. Despliegue de WACE
    a) WACE como un servicio independiente
    b) WACE como librería
2. Comunicación entre WACE y Coraza WAF
    a) Modificar el core de Coraza
    b) Generar operadores
    c) Modificar los conectores
    d) Capa intermedia entre los conectores y Coraza
3. Mecanismo para la toma de decisión
    a) Utilizar un operador.
    b) Modificar el core de coraza
    c) Agregar variables y Seclang
    d) Utilizar la interfaz de Coraza

A.1. Despliegue de WACE

Ambas posibilidades de despliegue soportan la comunicación entre Coraza y WACE en distintas formas, así como distintas alternativas para la toma de decisión. Los requisitos correspondientes a la definición de excepciones pueden ser llevados a cabo en ambas arquitecturas de la misma forma, ejecutando procedimientos o verificaciones previo a la invocación de WACE.

A.1.1. WACE como librería

Cumplimiento requisitos funcionales Con respecto al requisito de configuración, la librería se puede centralizar agregándola a la configuración de Coraza o se puede agregar un archivo de configuración nuevo. Cualquiera de estas implementaciones implica modificar la función que inicializa el WAF. Una posible implementación sin la modificación de componentes implica que la ruta donde está el archivo de configuración sea fija.

En cuanto a métricas, WACE como librería, brinda la posibilidad de centralizar las mediciones y la recolección de los datos de las métricas.

Cumplimiento requisitos no funcionales Con respecto a la eficiencia, se puede decir que se evita el uso de la red y se aprovecha el entorno de ejecución del WAF. Además se evitan los procesos de serialización, comunicación entre hosts y deserialización.

Con respecto a la instalación, replicación y actualización, estos procesos dependen enormemente de la arquitectura de integración que se utilice. En particular, para la instalación, se puede decir que el uso de WACE como librería implica que se descargue el software desde el repositorio, y se lo importe donde se desea utilizar, un proceso relativamente sencillo.

Se elimina un posible punto de falla que puede retrasar y detener la ejecución del sistema.

A.1.2. WACE como servicio independiente

Cumplimiento requisitos funcionales En esta alternativa no es posible centralizar la configuración del WAF y WACE, ya que se tendrán dos servicios independientes que deberán estar configurados para poder ejecutrar. Además, se debe conocer la información de los modelos, esto es, su modo de ejecución (sincrónico o asincrónico), que tipo de transacciones puede procesar (Request, Response, etc), ya que al llegar una solicitud de evaluación, se debe validar que la solicitud es posible de ejecutar. Por lo tanto, en la mayoría de los casos, se tendrán configuraciones duplicadas, tanto en WACE como en el WAF, haciendo que no se cumpla con el requisito de una configuración simple y centralizada. Con respecto a la obtención de datos necesarios para la generación de métricas, se puede decir que se deben obtener de distintos puntos, ya que se WACE se encuentra independiente del WAF e interesa realizar mediciones en ambos lugares.

Cumplimiento requisitos no funcionales Como se trata de un servicio independiente, el rendimiento se ve afectado ya que se debe atravesar la red para llegar del WAF a WACE, lo que tiene un impacto negativo sobre el requisito de eficiencia. Con respecto a la instalación, se agrega una mayor complejidad si se compara con la librería, ya que se debe de realizar la instalación del propio servidor sobre una determinada infraestructura, sumado a la instalación en el WAF de los componentes necesarios para la interacción desde Coraza hacia WACE.

A.2. Comunicación entre WACE y Coraza WAF

En general todas las arquitecturas de integración soportan la definición de excepciones. Esto se traduce en que por ejemplo, se pueden tomar diferentes acciones dependiendo de la URI, una determinada IP de origen o destino, etc. La posibilidad de utilizar los mecanismos nativos del WAF dependerá de la alternativa analizada.

A.2.1. Modificar el core de Coraza

Impacto en componentes a integrar La implementación de esta integración implica la modificación de los componentes internos de Coraza, desde una visión de alto nivel, para el funcionamiento

de WACE se necesita cargar y almacenar la configuración de la herramienta y por otro lado se necesita realizar los llamados a los modelos.

Dentro de Coraza, el componente `WAF` es el encargado de mantener configuraciones estáticas que no se espera se modifiquen en tiempo de ejecución, esto nos indica que se debe modificar dicho componente para cargar y almacenar las configuraciones de WACE.

Por otra parte, el componente `Transaction` de Coraza es el encargado de implementar las funcionalidades de procesamiento de cada una de las fases, se realiza utilizando el componente `RuleGroup`, que se encarga de almacenar grupos de reglas de cada uno de los componentes y al mismo tiempo realiza la evaluación de las reglas a pedido de la `Transaction`. Por lo tanto, los llamados a los modelos de parte de WACE se podrían implementar en cualquiera de dichos componentes, desde un punto de vista de la filosofía de los componentes de Coraza y desde un punto de vista de encapsular las funcionalidades de WACE, parece ser adecuado su implementación dentro del componente `Transaction`.

Cumplimiento requisitos funcionales Con respecto al cumplimiento de los requisitos funcionales, la arquitectura permite tener una configuración global y acceso a los datos de la transacción. La configuración de WACE podría ser implementada para cargarse directamente en el componente WAF o ingresarse directamente con la configuración de Coraza. De esta manera, se puede especificar que elementos de la transacción se desea enviar a los modelos y a su vez tener acceso a estos. Es posible enviar parte de la transacción a WACE, ya sea los encabezados, el cuerpo HTTP e incluso la solicitud completa.

En cuanto a métricas, la arquitectura permite generar todas las métricas requeridas. Esta arquitectura permite centralizar en un lugar la medición de muchos de los datos, dado que se van a modificar los componentes de Coraza y dentro de ellos se van a realizar los llamados a WACE, teniendo acceso a los tiempos y resultados de cada uno de los procesamientos, lo que permite fácilmente su medición de tiempo y conteo, ya sea de manera genérica o condicional por alguno de sus atributos.

Cumplimiento requisitos no funcionales Con respecto a la eficiencia, se estima que esta arquitectura no verá afectado el rendimiento de la aplicación con respecto a otras arquitecturas de integración disponibles, dado que se reducen los intermediarios al mínimo.

Con respecto a la instalación y replicación, el proceso es considerablemente sencillo, implica cambiar los conectores para que utilicen la nueva versión de Coraza que incluye la integración con WACE.

Como se hace uso de los componentes de Coraza y éstos se ven modificados, no se permite la actualización de manera independiente, es decir, ante una nueva actualización de Coraza, esto tiene que analizarse en la versión que contiene a WACE para poder integrarse. Por lo tanto la actualización se vuelve compleja.

Con respecto al conector los cambios son mínimos y no presentan complejidad de ser necesarios rehacerlos ante una actualización del conector original. Por lo que esta arquitectura cumple parcialmente dichos objetivos.

A.2.2. Generar operadores

Impacto en componentes a integrar Esta forma de integración propone utilizar los mecanismos de extensión que posee Coraza WAF, en particular, se utilizaría la capacidad de generar operadores personalizados. De esta manera, al invocar desde el WAF uno de estos operadores personalizados, se realiza la invocación a WACE y éste realiza el llamado a los modelos. Con este mecanismo de integración no se modifica ningún componente del WAF, lo que simplifica el mantenimiento de la herramienta a desarrollar.

El proceso de sincronización entre la ejecución de reglas por parte del WAF y de la evaluación de los modelos, se vuelve considerablemente complejo.

Cumplimiento requisitos funcionales La configuración es una desventaja de la arquitectura, ya que siempre se carga con cada transacción procesada, lo cual es ineficiente, además ésta debe ser cargada con directivas de Seclang que no están pensadas para esta tarea.

Con esta arquitectura se puede aprovechar el funcionamiento de SecRule para generar excepciones, como ya se realiza con las reglas del OWASP CRS en el WAF, permitiendo por ejemplo, que se puedan agregar reglas que deshabiliten la invocación a los modelos de WACE, esto quiere decir, que no se ejecuten las reglas que accionan los operadores asociados a la evaluación de WACE.

La obtención de los datos para exportar métricas implica una implementación compleja, ya que no se pueden acceder a las invocaciones de Coraza sino que se tendría que agregar una funcionalidad, también mediante operadores, que realice dichas mediciones.

Cumplimiento requisitos no funcionales Esta arquitectura posee un intermediario entre el WAF y WACE que está dada por la implementación del operador. Además el comienzo de la evaluación de los modelos depende de las evaluaciones de las reglas, no se puede comenzar totalmente en paralelo. Por lo que la eficiencia es relativamente menor que en otras arquitecturas.

Con respecto a la instalación, es relativamente más compleja a las demás alternativas de integración, porque se debe de instalar todos los operadores que se requieran de utilizar y el proceso de configurarlos es sumamente complejo.

En cuanto a la actualización, es una arquitectura que permite integrar fácilmente las actualizaciones de Coraza, ya que no se modifica ninguno de sus componentes, haciendo que el proceso de actualizar y mantener la integración sea simple.

A.2.3. Modificar conectores

Impacto en componentes a integrar Esta implementación implica la modificación de todos los paquetes que implementen la conexión entre algún tipo de servidor web y Coraza, desde donde se gestiona los llamados al WAF, en ese punto se debería agregar los llamados a WACE.

Dicha implementación implica crear una versión nueva de cada conector, pero de esta manera la solución es independiente de la implementación de Coraza.

Estas implementaciones corren el riesgo de quedar desactualizadas, dado que ante un cambio en el componente original se debe evaluar que impacto posee en la nueva implementación que integra la comunicación con WACE. Pero por otra parte, estos componentes no son actualizados con una frecuencia muy alta.

Si bien el costo de implementación por conector es relativamente bajo, esto se debe impactar en cada uno de los conectores que hay. En la actualidad esto no es un problema dado que la cantidad de conectores existentes es reducida.

Cumplimiento requisitos funcionales Con respecto al cumplimiento de los requisitos funcionales, la arquitectura permite tener una configuración global y además acceso a la interfaz de la transacción. La configuración de WACE podría ingresarse directamente en el conector. De esta manera, se puede especificar que elementos de la transacción se desea enviar a los modelos y a su vez tener acceso a estos. Es posible enviar parte de la transacción a WACE, ya sea los encabezados, el cuerpo HTTP e incluso la solicitud completa, dado que el conector tiene acceso a todos estos recursos.

La interfaz de la transacción posee la capacidad de brindar información de todas las reglas que evaluaron positivamente en una transacción, con ello se puede obtener las reglas de resumen que contienen los puntaje de OWASP CRS.

En cuanto a métricas, la arquitectura permite generar todas las métricas requeridas. Esta arquitectura permite centralizar en un lugar la medición de muchos de los datos, dado que se convierte en un despachador tanto de las funcionalidades de Coraza como de WACE, teniendo acceso a los tiempos y resultados de cada uno de los modelos, también se puede saber si Coraza planeaba rechazar la transacción o no.

Cumplimiento requisitos no funcionales Con respecto a la eficiencia, se estima que esta arquitectura no verá afectado el rendimiento de la aplicación con respecto a otras arquitecturas de integración disponibles, sí con respecto a la ejecución únicamente de Coraza con OWASP CRS, dado que la evaluación de los modelos poseen un mayor tiempo de procesamiento que las reglas de OWASP CRS, pero esto evidentemente depende de la eficiencia de dichos modelos.

El proceso de instalación no agrega cambios significativos a una instalación común de Coraza ni es más complejo que en las otras arquitecturas de integración. Simplemente se debe utilizar el nuevo conector implementado.

Con respecto a la actualización, el proceso se vuelve más complejo, dado que se si se actualiza el conector original esto no se puede ver impactado inmediatamente en la arquitectura, sino que se debe analizar el impacto en el nuevo conector implementado y posteriormente generar una nueva versión de éste. Las actualizaciones de Coraza en contraposición sí son independientes y pueden ser realizadas sin procesamiento adicionales, a menos de una actualización "MAJOR" donde se cambien las interfaces o lógica de procesamiento de Coraza.

A.2.4. Capa intermedia entre los conectores y Coraza

Impacto en componentes a integrar La implementación de una capa intermedia entre el conector de un servidor web y Coraza WAF, donde se gestione los llamados a dicho WAF y al mismo tiempo a WACE, conlleva una modificación del conector. Dicha modificación puede ser mínima si se mantienen los mismos llamados y se respeta la estructura que ya poseen los tipos de Coraza.

Desde los conectores se hace uso de la interfaz `WAF` y de la interfaz `Transaction`, donde se instancia un objeto que cumple la interfaz `WAF` y este despacha objetos que cumplen la interfaz `Transaction`, finalmente las transacciones realizan el procesamiento de los pedidos. Por ello se debería generar una nueva estructura de `WAF`, desde ahora `WACE_WAF`, que posea las funcionalidades de WACE. Para ello `WACE_WAF` debe realizar llamados a la interfaz del `WAF` ya existente de Coraza, teniendo en cuenta que a su vez se debe reimplementar los métodos que se consideren necesarios para la integración de las funcionalidades de WACE al conector. Esto es esencialmente despachar una nueva estructura de transacción, que realice llamados a la interfaz `Transaction` de Coraza pero reimplemente las funciones de procesamiento, llamando al procesamiento habitual de Coraza y agregando ahora el procesamiento de WACE.

El resultado de la decisión que tome WACE se puede comunicar de la misma forma que lo hace Coraza, se puede devolver un objeto de tipo `Interruption` en caso de que WACE considera que se debe realizar una acción disruptiva sobre la transacción. Se debe tener en cuenta que según el diseño de Coraza, una vez que se produce una `Interruption`, este evento es almacenado internamente en la `Transaction` y evita que dicha transacción ejecute cualquier otro procesamiento. Esto no puede ser modificado desde la interfaz. Por ello se debe tener sumo cuidado con la configuración de Coraza que se utilice, posiblemente teniendo que modificar las reglas de OWASP CRS a nivel de la capa intermedia para que no generen interrupciones no deseadas.

En suma se puede decir que el costo de implementación es moderado, se mantiene una clara separación de responsabilidades y no es necesario modificar el código fuente de Coraza, generando que el código sea mantenible. Se estima que las modificaciones en los conectores son mínimas y simplemente refieren al cambio del uso del componente que implementa la interfaz de Coraza, pasando de la implementación de Coraza a la implementación de la capa intermedia.

Cumplimiento requisitos funcionales Con respecto al cumplimiento de los requisitos funcionales, la arquitectura permite tener una configuración global y acceso a los llamados de la interfaz de la transacción. La configuración de WACE podría ser implementada para cargarse directamente en la capa intermedia o ingresarse directamente en el conector con la configuración de Coraza. De esta manera, se puede especificar que elementos de la transacción se desea enviar a los modelos

y a su vez tener acceso a estos. Es posible enviar parte de la transacción a WACE, ya sea los encabezados, el cuerpo HTTP e incluso la solicitud completa.

En cuanto a métricas, la arquitectura permite generar todas las métricas requeridas. Esta arquitectura permite centralizar en un lugar la medición de muchos de los datos, dado que se convierte en un despachador tanto de las funcionalidades de Coraza como de WACE, teniendo acceso a los tiempos y resultados de cada uno de los procesamientos, lo que permite fácilmente su medición de tiempo y conteo, ya sea de manera genérica o condicional por alguno de sus atributos.

Cumplimiento requisitos no funcionales Teniendo en cuenta que se agrega una capa intermedia, se puede decir respecto a la eficiencia que se estima que esta arquitectura podría afectar mínimamente el rendimiento de la aplicación con respecto a otras arquitecturas de integración disponibles.

Con respecto a la instalación, replicación y actualización, se considera que simplifica considerablemente el procedimiento con respecto a otras arquitecturas, ya que la instalación inicial agrega a una instalación de Coraza una leve modificación del conector y la descarga del software que implementa la capa intermedia. Como se hace uso de los componentes de Coraza pero éstos no se ven modificados, se permite la actualización de Coraza de manera independiente. Con respecto al conector los cambios son mínimos y no presentan complejidad.

Esta alternativa deja un único componente independiente cuya actualización no se ve alterada por los demás componentes. Por lo que cumple en gran medida los requisitos no funcionales.

A.3. Mecanismos para la toma de decisión

A.3.1. Utilizar un operador

Esta opción se basa en utilizar un operador como medio para comunicar al WAF si una transacción debe ser bloqueada, este operador debe contar tanto con la información de los modelos como de los puntajes acumulados por las reglas de OWASP CRS.

Impacto en componentes a integrar El impacto sobre los componentes ya existentes es bajo, dado que los operadores pueden agregarse como extensiones de Coraza. Para que WACE pueda definir la decisión de bloqueo a tomar, debe tener acceso a los los puntajes de OWASP CRS, por ello a este operador debe obtener esta información como entrada.

Por otra parte, este mecanismo aprovecha la implementación del manejo de la interrupción ya implementado por Coraza.

Cumplimiento requisitos funcionales Se pueden agregar distintos criterios de decisión generando más operadores y agregando reglas para dicha evaluación. O agregando implementaciones distintas a un mismo operador, donde al momento de evaluar el operador pueda acceder a información que le permita saber que criterio de decisión se va a utilizar.

En cuanto a la configuración, se tiene que realizar mediante los mecanismos de Coraza, que se corresponde Seclang. Por lo tanto, para realizar la configuración se deben agregar reglas que permitan tomar la decisión.

Cumplimiento requisitos no funcionales Con respecto a la instalación y actualización este criterio no interfiere en dichos procesos, dado que es un mecanismo de extensión que brinda el propio Coraza. La instalación requiere que se agregue el operador como un plugin y que se agregue una regla a la configuración que permita llamarlo. Por su parte la actualización de los componentes puede realizarse de manera independiente sin perjudicar el funcionamiento del sistema.

A.3.2. Modificar el core de Coraza

Esta opción se basa en agregar al core de funcionalidades de Coraza, un mecanismo que integre los resultados de WACE en la decisión que se toma para interrumpir la transacción.

Impacto en componentes a integrar El impacto sobre los componentes existentes es medio/alto, ya que debe modificar el mecanismo nativo que utiliza para Coraza para tomar la decisión.

Cumplimiento requisitos funcionales Se pueden agregar distintos criterios de decisión mediante el mecanismo de plugins del lenguaje Go.

Los criterios de decisión a utilizar pueden ser especificados mediante un archivo de configuración permitiendo una configuración centralizada de WACE.

Cumplimiento requisitos no funcionales Este mecanismo dificulta el mantenimiento de la herramienta, ya que al modificar componentes de Coraza, hace que la actualización se dificulte, teniendo que analizar si la solución implementada es compatible con los cambios introducidos.

A.3.3. Utilizar variables y Seclang

Esta opción se basa en utilizar las directivas de Seclang para realizar los algoritmos de decisión, previamente a la toma de la decisión se debe tener en variables de la transacción los resultados de los modelos. Permitiendo a los operadores definir criterios sencillos con mecanismos a los que ya están adaptados.

Impacto en componentes a integrar El impacto sobre los componentes ya existentes es bajo, dado que las variables de la transacción pueden ser agregadas sin la modificación de los componentes ya existentes.

Cumplimiento requisitos funcionales Los criterios de decisión tienden a ser más simples, como reglas donde se compare el resultado de un modelo contra cierto umbral. Sin embargo, mediante el encadenamiento de reglas podrían generarse criterios más complejos. Se pueden agregar más criterios de decisión agregando distintas reglas.

Cumplimiento requisitos no funcionales Al igual que con la toma de decisiones mediante operadores nuevos, este mecanismo no interfiere en la instalación y actualización, dado que este mecanismo se implementa con las directivas que el propio Coraza es compatible.

A.3.4. Utilizar los datos accesibles mediante la interfaz de Coraza y los resultados de los modelos

Esta opción se basa en utilizar los datos accesibles de manera externa a Coraza, esto es por un lado las interrupciones se que devuelven si una transacción fue interrumpida, debido a qué regla y por otra parte la lista de reglas que evaluaron positivamente. Dichas interrupciones no contienen los puntajes acumulados de OWASP CRS, pero dichos puntajes si están presentes en forma de texto en la lista de las reglas que hicieron match que es accesible mediante la interfaz de Coraza. Además de dichos puntajes se utilizaría, evidentemente, los resultados de los modelos.

Impacto en componentes a integrar Este mecanismo implica que se deba extraer los puntajes deseados a partir de los datos brindados por Coraza. Esto puede implementarse en cualquier lugar que tenga acceso al resultado de las funciones de procesamiento.

Cumplimiento requisitos funcionales Puede aprovecharse el mecanismo de plugins del lenguaje Go para agregar criterios de decisión.

No es posible configurar la decisión mediante Seclang ya que en el punto donde se toma la decisión ya fueron procesadas las reglas.

Cumplimiento requisitos no funcionales Para la instalación se debe tener en cuenta que se deben compilar los plugins de decisión de cierta manera para que la integración funcione. Con respecto a lo demás, como este mecanismo iría incluido en la implementación que se realice para la integración, el impacto está contemplado en el análisis de dicha integración.