Facultad de Ingeniería - Universidad de la República

Proyecto de grado

Carrera Ingeniería en Computación

WACE: Un integrador de clasificadores de ataques web

Informe ejecutivo

Estudiantes:

Elias Cuttica

Fernando Outeda

Tutores:

Gustavo Betarte

Rodrigo Martínez

Marcelo Rodríguez

Julio 2021

Resumen

Desde principios de los años 90 cuando se desarrolló por primera vez la idea de un firewall de aplicación (WAF por sus siglas en inglés), hasta la actualidad donde existen un gran número de WAFs protegiendo muchas de las aplicaciones utilizadas en internet, el uso de esta tecnología ha aumentado en gran medida y es una herramienta más que ayuda a proteger, en conjunto con otras, a las aplicaciones web. Dentro de las soluciones que existen actualmente hay un gran número de implementaciones comerciales, pero la más importante y reconocida en el ambiente *open-source* es ModSecurity[23].

La motivación de este trabajo es mejorar los resultados que se obtienen cuando se utiliza ModSecurity junto con el CRS de OWASP[26] en modo de detección por acumulación de puntajes, desarrollando una herramienta que permita integrar modelos de aprendizaje automático con ModSecurity. Se buscan detectar las transacciones maliciosas utilizando ambos criterios de manera complementaria, el de ModSecurity y el de los modelos de aprendizaje automático, principalmente para reducir el alto número de falsos positivos que pueden surgir del uso de ModSecurity en niveles de paranoia mayores a 1 [4], sin aumentar la tasa de falsos negativos significativamente.

Índice

1. Introducción **6**

1.1. Motivación 6

1.2. Objetivo 6

1.3. Estructura del documento 6

2. Estado del Arte **8**

2.1. ModSecurity WAF 8

2.2. Formas de despliegue 8

2.3. Ciclo de vida de una transacción 9

2.4. Lenguaje 10

2.5. Core Rule Set 11

2.6. Paranoia levels 12

2.7. Modos de funcionamiento 13

2.7.1. Modo de detección tradicional 13

2.7.2. Modo de detección de anomalías 14

2.8. Trabajos académicos relacionados 15

2.8.1. Modelos para la verificación 17

2.9. PMML 17

2.10. Herramientas para servir modelos de aprendizaje automático 17

3. Análisis **19**

3.1. Herramienta a desarrollar 19

3.2. Requerimientos 20

3.2.1. Requerimientos funcionales 20

3.2.2. Requerimientos no funcionales 20

3.3. Actores 20

3.4. Análisis de integración 21

3.4.1. Esquema del sistema 21

3.4.2. Comunicación entre WACE y ModSecurity 22

3.5. Flexibilidad de WACE 24

3.6. Decisión 25

3.6.1. Resultados de ModSecurity y modelos de aprendizaje automático 25

3.6.2. Variables para la decisión 26
3.6.3. Fórmulas de decisión 28
3.6.4. Integración de resultados con ModSecurity 29
3.6.5. Flexibilidad en los criterios de decisión 30
3.6.6. Balance 30
3.7. Procesamiento de transacciones 30

4. Diseño **32**
4.1. Arquitectura de Microkernel 32
4.1.1. Ventajas y desventajas 32
4.2. Arquitectura del sistema 34
4.2.1. Descubrimiento de plugins 36
4.2.2. Comunicación entre el core y los plugins 36
4.2.3. Comunicación entre el core y mod_wace 36
4.2.4. Protocolo de comunicación 38
4.2.5. Diagrama de secuencia 41
4.2.6. Configuración 42
4.2.7. Estructuras de datos 44
4.2.8. Logging 45

5. Implementación **46**
5.1. Lenguaje de programación 46
5.2. Componentes 47
5.2.1. Core del sistema 47
5.2.2. Plugins 49
5.2.3. mod_wace 52
5.2.4. api_wace 53

6. Pruebas del sistema **54**
6.1. Datos de prueba 54
6.2. Pruebas de desempeño 55
6.2.1. Resultados utilizando paranoia level 4 57
6.2.2. Resultados utilizando paranoia level 3, 2 y 1: 59
6.2.3. Conclusiones de las pruebas de desempeño 60
6.3. Tiempos de procesamiento 62

7. Trabajo a Futuro **64**

8. Conclusiones **66**

9. Apéndices **72**

9.1. Instalación y despliegue 72

9.1.1. Instalación de WACE 72

9.1.2. Elementos adicionales a WACE 72

9.2. Función de inicialización 74

9.3. Plugin de decisión 74

1. Introducción

1.1. Motivación

Los firewalls de aplicación web (WAF por sus siglas en inglés) son una muy buena herramienta para proteger aplicaciones web, pero en ocasiones, pueden presentar un alto número de falsos positivos lo cual afecta la usabilidad de la aplicación que protege. Esto puede causar que clientes legítimos no puedan acceder o se encuentren con errores al intentar utilizar al servicio. Dichos problemas son relevantes ya que pueden llevar a que la organización desactive reglas, o deje de utilizar el WAF para evitar estos falsos positivos. Esto va en detrimento de la seguridad y no es lo deseado.
Hoy día la mejor manera de reducir estos falsos positivos es con el refinado de las reglas, esta tarea puede ser efectiva, pero presenta cierta complejidad lo cual implica la necesidad de contar con personal capacitado y con experiencia que detecte el falso positivo y modifique las reglas de manera acorde sin permitir un mayor acceso que el necesario. Por esto la motivación de este trabajo es la integración de modelos de aprendizaje automático que formen parte de la decisión de bloqueo del WAF, en pos de una obtención de mejores resultados.

1.2. Objetivo

El objetivo principal de este trabajo es analizar, diseñar e implementar una solución que permita realizar la integración entre ModSecurity y uno o más modelos de aprendizaje automático de manera eficiente. El uso de dicha solución no debe enlentecer de manera significativa el funcionamiento de la aplicación que se pretende proteger.

1.3. Estructura del documento

A continuación se describe la estructura del informe y una breve descripción de cada parte del mismo,

- Estado del Arte: Se brinda un resumen del documento confeccionado en la investigación del estado del arte.

- Análisis: Se define el problema, se establecen los requisitos del proyecto y se discuten y analizan distintas formas de organizar la integración del sistema.
- Diseño: Se define como se van a desarrollar las decisiones tomadas en la etapa de análisis.
- Implementación: Se detallan aspectos relacionado a las tecnologías utilizadas, lenguaje de programación, estructuras de datos y una breve descripción de cómo se implementaron algunos componentes importantes del sistema.
- Resultado de pruebas: Se muestran los resultados de diferentes pruebas realizadas.
- Trabajo a Futuro: Se plantean algunas funcionalidades interesantes que se podrían agregar al prototipo pero que quedaron fuera del alcance del proyecto.
- Conclusiones: Se plantea un resumen del proyecto haciendo principal énfasis en si los resultados cumplen o no con los objetivos.
- Apéndice: Se agrupa información complementaría y más detallada sobre la implementación.

2. Estado del Arte

2.1. ModSecurity WAF

Un WAF (Web Application Firewall) es una herramienta que intercepta e inspecciona todo el tráfico entre un servidor web y sus clientes en busca de ataques dentro del contenido de cada paquete HTTP. Se implementan con el objetivo de establecer una capa de seguridad externa que detecte y prevenga ataques antes de que lleguen a las aplicaciones web.
ModSecurity[24] es un WAF de código abierto regido bajo la licencia de Apache 2.0[2] que se viene desarrollando activamente por la comunidad desde el año 2002, cuando comenzó como un módulo de Apache, hasta el día de hoy que se encuentra en la versión 3.0.4 (abril del 2020) con un enfoque distinto al inicialmente desarrollado, con una arquitectura despegada de Apache.

Inicialmente fue desarrollado en C, hasta la versión 3 cuando se rediseño y reescribió completamente, pasando a utilizar principalmente C++. Ambas versiones la 2[24] y la 3[23] conviven mutuamente y todavía no se ha logrado una adopción total de la versión nueva, por esto ModSecurity 2 sigue en desarrollo y es mantenido en un repositorio aparte al mismo tiempo que se desarrolla la versión 3.

2.2. Formas de despliegue

ModSecurity cuenta con dos maneras distintas de ser utilizado:

- **Embedded-mode Deployment**: En este modo ModSecurity se instala en el servidor web donde se hospeda la aplicación que se desea proteger. Respecto al otro modo de despliegue, este modo puede traer aparejado como ventajas, que no se deben realizar cambios en la red, que no existe un único punto de falla, entre otros elementos.

- **Network-based Deployment**: En este modo ModSecurity es desplegado como parte de un proxy reverso que se posiciona por delante de las aplicaciones que se deseen proteger. La principal ventaja que se puede identificar

en este modo es que se pueden proteger varios servidores de aplicaciones utilizando una sola instalación de ModSecurity.

2.3. Ciclo de vida de una transacción

Todas las transacciones recibidas por ModSecurity pasan por cinco etapas. En cada una de estas etapas, se hace al comienzo algún preprocesamiento, luego se invocan las reglas que pertenecen a esa etapa y luego opcionalmente se realiza algo al final de la etapa. Esta separación en cinco etapas se da porque hay ataques que se deben detectar antes de que lleguen al servidor analizando la *request* (un intento de SQL Injection[22] por ejemplo), o hay ataques que se pueden detectar solamente al capturar la *response* (por ejemplo, una fuga de información a través de un mensaje de error). Por este motivo es que ModSecurity realiza la separación en las siguientes etapas:

1. ***Request Headers***: El principal objetivo de esta etapa es permitir que existan reglas que analicen el cabezal de una *request* antes que se dé el (costoso) procesamiento del cuerpo de la *request*. Si se pueden detectar en el cabezal elementos que configuren un ataque que determine que la *request* deba ser rechazada, se puede ahorrar mucho tiempo evitando que se haga todo el procesamiento del cuerpo e inclusive evitando que la petición llegue al servidor. Además, si se quieren realizar cosas antes de procesar el cuerpo, ésta es la etapa adecuada para realizarlo.

2. ***Request Body***: Luego que se recibe y procesa el cuerpo de la *request*, se entra en esta etapa. Las reglas que se encuentran aquí tienen toda la información de la *request* a su disposición.

3. ***Response Headers***: Cuando el cabezal de la *response* queda disponible y antes que se genere el cuerpo de la *response*, se está en esta etapa. Las reglas que deben decidir si inspeccionan o no el cuerpo de una response deberían ejecutarse en esta etapa.

4. ***Response Body***: En esta etapa el cuerpo de la respuesta ya va a haber sido procesado y se van a tener todos los datos de la transacción disponibles para las reglas que se ejecuten en esta fase.

5. ***Logging***: Esta es una etapa especial, cuando se llega a esta fase la transacción ya finalizó. Se utiliza para registrar lo sucedido con la transacción. En este paso final las reglas se limitan a definir cómo se va a registrar el evento en los logs.

En la figura 1 se puede ver gráficamente que partes de la transacción son analizadas en cada etapa.

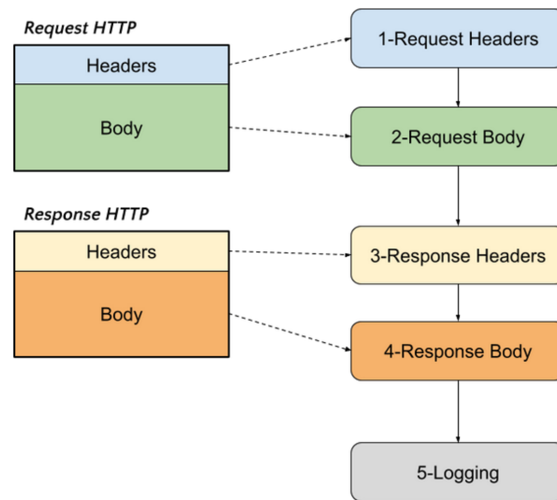

Figura 1: Etapas de ModSecurity

2.4. Lenguaje

Una parte esencial de un firewall de aplicación es su capacidad de ser configurado a través de reglas que permitan detectar diferentes tipos de ataques y especifiquen las acciones a tomar frente a una detección positiva.
ModSecurity cuenta con un lenguaje propio[21] para la implementación de reglas de detección y para la configuración de su funcionamiento.
El lenguaje define varias directivas, dentro de las cuales se encuentra una de las más importantes, *SecRule*. Esta directiva es la que permite definir una regla.

Todas las definiciones de reglas generalmente tienen la siguiente estructura:

SecRule *Variables Operadores Funciones-de-transformación Acciones*

- ***Variables:*** Las variables permiten seleccionar partes de la consulta HTTP, determinan qué parte de la transacción se va a analizar. Se puede definir más de una.

- ***Operadores:*** Los operadores establecen como se van a analizar las variables seleccionadas. El operador más común es el de expresiones regulares, que permite analizar los valores de las variables según una expresión regular. Por cada regla puede haber un solo operador.

- ***Funciones-de-transformación:*** Es una lista de funciones de transformación que indican como se va a convertir cada variable antes de ser analizada por los operadores. Este bloque es opcional si se define una regla sin ninguna función de transformación se van a considerar las establecidas por defecto.

- ***Acciones:*** Con las acciones se especifica qué hacer si la regla tiene una coincidencia positiva.

2.5. Core Rule Set

ModSecurity sin reglas que detecten el tráfico malicioso no tiene mucha utilidad, su funcionamiento se basa en un poderoso lenguaje de reglas que le permite tener la flexibilidad suficiente como para indicar exactamente de qué es lo que se quiere proteger y en qué momento se desean aplicar las reglas[3]. La implementación de estas reglas de cero no es un trabajo muy simple, se requieren conocimientos en el lenguaje y en seguridad informática para llegar a un conjunto de reglas adecuado que minimice al máximo la cantidad de falsos positivos teniendo una buena tasa de verdaderos positivos.
Para intentar minimizar este problema es que se creó el Core Rule Set[26] (o CRS por sus siglas en inglés) un proyecto open-source el cual no es más que un conjunto de reglas de ModSecurity implementadas por expertos en la materia y la comunidad, con el objetivo de tener una forma de protección para cualquier aplicación web sin tener que implementar las reglas de cero. El CRS no es perfecto y puede requerir la modificación de sus reglas o el agregado de otras para personalizar el WAF a la aplicación web que se pretenda proteger, ya que en una instalación por defecto de CRS se pueden tener un porcentaje considerable de falsos positivos[4].

Las reglas del CRS tratan de evitar algunos de los ataques más comunes en aplicaciones web implementando un modelo negativo, es decir permiten pasar todo el tráfico bloqueando eventualmente las transacciones que hayan sido marcadas como maliciosas explícitamente por alguna regla.
Las reglas del Core Rule Set se agrupan en diferentes archivos que refieren a clases de ataques distintos.

2.6. Paranoia levels

El nivel de paranoia es un parámetro de configuración que permite indicar qué reglas considerar en el procesamiento de las transacciones. Existen cuatro niveles, según aumente el nivel de paranoia se consideran más reglas en el procesamiento de las transacciones, brindando mayor seguridad. Pero como aspecto negativo, al incrementar el nivel de paranoia también puede generar que se bloquee tráfico legitimo por la ocurrencia de falsos positivos. Para mitigar este problema al usar niveles altos de paranoia probablemente sea necesario modificar algunas reglas, o agregar reglas de exclusión para algunas aplicaciones que reciban entradas complejas como parte de su tráfico normal.

- **Paranoia level 1 (PL1)**: Es el nivel por defecto, incluye la mayoría de las reglas del CRS. Es raro que se generen falsos positivos con este nivel de paranoia.

- **Paranoia level 2 (PL2)**: En este nivel se incluyen reglas extras, por ejemplo se añaden diversas reglas para la protección contra *SQL Injections*[22] y *Cross Site Scripting*[27], y se mejora la protección contra *Code Injections*[32]. Hay más probabilidad de que se generen algunos falsos positivos.

- **Paranoia level 3 (PL3)**: En este nivel se incluyen más reglas que cubren ataques menos comunes. Se agregan a las reglas listas de caracteres raros que permiten detectar ataques desconocidos. Como en el punto anterior este nivel de paranoia puede llevar a tener más falsos positivos.

- **Paranoia level 4 (PL4)**: En este nivel se aumenta más la lista de caracteres especiales y reglas. Aquí se puede llegar a tener una mayor cantidad de falsos positivos y es donde más se precisaría ajustar las reglas para minimizar las detecciones erróneas.

2.7. Modos de funcionamiento

Dentro del archivo de configuración *crs-setup.conf* se puede configurar el CRS para que funcione en alguno de sus dos modos de funcionamiento, el modo de detección tradicional y el modo de detección de anomalías (*Anomaly Scoring Mode*). A continuación, se describen ambos modos[35].

2.7.1. Modo de detección tradicional

Este es el modo en el que funcionaba inicialmente el CRS, en este caso la forma de operación es más básica que en el modo siguiente. Las reglas son autocontenidas, en el sentido que no comparten información entre ellas. Esto implica que, si una regla obtiene una detección, simplemente se van a ejecutar las acciones especificadas en la regla.
Como principales ventajas de utilizar este modo de funcionamiento se podría mencionar,

- Su facilidad para comprender su funcionamiento.
- Su mejor desempeño, ya que cuando se encuentre la primera regla que evalúe positivamente y tenga una acción disruptiva, o no tenga ninguna acción establecida (en cuyo caso se considera la acción establecida por defecto) se va a cortar el procesamiento.

Dentro de las desventajas pueden considerarse aspectos como,

- Solamente la primera regla que evalúe positivamente va a ser registrada en el log pudiendo haber otras reglas que puedan también evaluar positivamente y no van a ser evaluadas ni registradas en los logs.
- Puede suceder que reglas de menor severidad bloqueen la transacción, aumentando la probabilidad de causar falsos positivos.
- Puede que una regla de menor severidad no amerite bloquear la ejecución de la transacción, pero puede que muchas evaluaciones positivas en estas reglas de menor severidad hagan que sea necesario bloquear. En este modo esto no es posible de realizar.

2.7.2. Modo de detección de anomalías

A partir del CRS versión 3 este modo es el que viene activado por defecto. Cuando se opta por esta opción la funcionalidad de bloqueo se desacopla de las reglas. Las reglas individualmente se evalúan como en el modo anterior, permitiendo la detección, pero cuando se evalúa positivamente alguna regla no se realiza una acción de bloqueo sino que la detección suma a un puntaje de anomalía (*Anomaly Score*). Adicionalmente se almacenan metadatos con información sobre cada regla que evaluó positivamente, para ser registrada luego en el log.
Cada regla que evalúe positivamente no va a bloquear la ejecución de la transacción sino que va a sumar determinado valor al puntaje de anomalía con la directiva de ModSecurity *setvar* que permite sumar a la variable *tx.anomaly_score* (donde se acumula el puntaje obtenido) el puntaje de anomalía que le corresponda a la regla.
Los valores que suman las reglas varían dependiendo de la severidad de la regla. Luego que se obtiene el puntaje de anomalía acumulado, este se compara con un umbral. Si este umbral es superado por el puntaje, la transacción es bloqueada. Este puntaje se evalúa en dos lugares, en la etapa 2 (Ver sección 2.3), luego que se termina de analizar la *request* y al final de la etapa 4 cuando se termina de analizar la *response*.
Dentro de las ventajas de utilizar este modo se pueden mencionar,

- Una mayor confianza al realizar el bloqueo. Esto debido a que, se consideran más factores, más reglas, para tomar la decisión de bloquear una transacción o no.
- Una mayor capacidad de configuración al permitir modificar los umbrales.
- Muchos eventos de poca severidad pueden desembocar en una acción disruptiva.

Como desventaja se puede decir que este modo puede ser más complejo de entender y configurar para el usuario promedio.

2.8. Trabajos académicos relacionados

Hoy en día existen algunos estudios con respecto a la utilización de modelos de aprendizaje automático aplicados a los WAF[5][6]. Lo que buscan estos trabajos académicos es mediante técnicas de aprendizaje automático mejorar las capacidades de detección de los WAF (por ej: Modsecurity), dando especial importancia a la tarea de disminuir los falsos positivos generados por esta herramienta cuando está configurada para proteger una aplicación web sin reducir la tasa de verdaderos positivos.
Algunos mecanismos de aprendizaje automático utilizados para la detección de anomalías se basan en primero preprocesar la *request* HTTP para extraer información (*features*, *tokens*, etc) y luego en base a la información extraída se entrena el modelo de aprendizaje automático para poder clasificar la request como valida o no, en base a determinada probabilidad. En el artículo[5] se describen tres modelos aplicados a distintos escenarios:

- **sc1:** es el escenario ideal donde se dispone de tráfico real, diferenciando el tráfico normal y el anormal (ataques)
- **sc2:** en el escenario sc2, también se dispone de tráfico real (obtenido de requests válidas a la aplicación) y las request clasificadas como ataques son un conjunto de solicitudes que se sabe que son maliciosas, pero no específicamente para realizar un ataque (estas request son sacadas desde un HoneyPot).
- **sc3**: en el escenario sc3 solo se cuentan con request válidas, sin clasificar. Es el escenario más realista.

Cada modelo se basan en la siguiente arquitectura[5]:

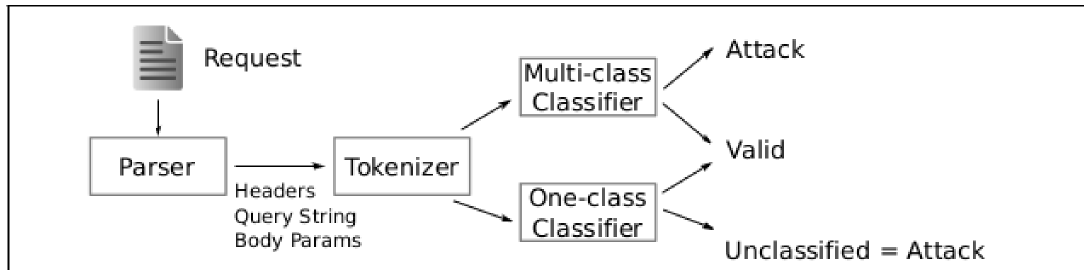

Figura 2: Arquitectura

En el primer paso se *parsea* cada request HTTP para decodificar la información que está en formato *URL encoded* y también para filtrar información que no sirve para inferir el comportamiento de la aplicación (ej. cookies). Luego se lleva a cabo el proceso de *tokenizacion* el cual consiste en separar la información de determinada manera para que sirva como entrada para los algoritmos de aprendizaje automático. Se utilizan varias técnicas, como la de *bag of words*, enfoque *expert-assisted*, etc. Finalmente se llega a la etapa de clasificación en la que se aplican varios modelos:

- ***Multi-class information retrieval:*** Este modelo se aplicó al **sc1**. Luego de tokenizar cada request se transforma en un vector aplicando el Term Frequency Inverse Document Frequency (TF-IDF) para calcular el peso de cada termino en la request, luego se prueba con varios algoritmos, por ejemplo, Support Vector Machine (SVM), K-nearest neighbours (K-NN) y Random Forest.

- ***Multi-class expert-assisted:*** Este modelo se utiliza en **sc2**. Es similar al anterior, pero se diferencia en el proceso de tokenizacion, en este modelo se define mediante un experto (una tabla con valores), no se utiliza el TF-IDF.

- ***Anomaly detection expert-assisted:*** Este modelo se utiliza en **sc3**. Se utiliza un enfoque de *one-class clasification*, donde hay instancias disponibles de una clase y ninguna o muy pocas muestras de la otra. Los clasificadores propuestos organizan muestras de la clase objetivo (las requests http válidas)

en clusters y luego se utiliza la distancia a estos clusters como una medida de anomalía; las muestras alejadas de los clusters se clasifican como anomalías.

2.8.1. Modelos para la verificación

Otra línea en la que se trabaja el uso de los modelos de aprendizaje automático es en la prueba de WAFs para ver si sus reglas se han configurado correctamente para determinado tipo de ataque, por ejemplo, en [7] se puede ver como el uso de estos métodos probó ser una buena estrategia para testear la efectividad de un WAF (ModSecurity fue uno de los utilizados en el trabajo mencionado) frente a la protección a ataques de SQL Injection[22].

2.9. PMML

PMML (Predictive Markup Model Language) es un estándar basado en XML concebido para el intercambio de modelos predictivos entre aplicaciones que utilizan algoritmos de minado de datos (*data mining*) o de aprendizaje automático (*machine learning*).
Los modelos predictivos son modelos matemáticos que utilizan la probabilidad y la estadística para aprender patrones en grandes volúmenes de datos. Estos permiten, luego que son entrenados, utilizar el conocimiento adquirido para predecir un resultado a partir de una determinada entrada de datos.
PMML es el estándar adoptado por muchas organizaciones para representar estos modelos y permite que puedan ser compartido entre varios sistemas.

2.10. Herramientas para servir modelos de aprendizaje automático

- **OpenScoring[25]:** es un servicio web REST que se utiliza para trabajar con modelos PMML. Como se mencionaba en la sección anterior PMML se utiliza para estandarizar los modelos predictivos. Utilizando OpenScorig se pueden consultar dichos modelos y realizar análisis en base a los resultados obtenidos.

- **Flask-RESTFul API[9]:** es una extensión de Flask[8] que permite de manera sencilla implementar una API REST. Es una opción recomendada para utilizar a la hora de servir modelos de aprendizaje automático, principalmente si están implementados en Python.

- **TensorFlow Serving[10]:** es un sistema que permite servir modelos de aprendizaje automático, de manera flexible y con un buen desempeño. TensorFlow Serving facilita desplegar nuevos algoritmos, manteniendo la misma arquitectura de servidor y API. Está orientado a proveer este servicio a modelos desarrollados con TensorFlow, pero se puede adaptar para servir otros tipos de modelos y datos.

3. Análisis

En esta sección se abordan diferentes aspectos, tales como, definición de requerimientos, análisis de mecanismos de integración y los factores a tener en cuenta a la hora de desarrollar un prototipo operativo. En base a estos aspectos se pretende resolver el problema principal del proyecto el cual consiste en la integración funcional de ModSecurity con modelos entrenados usando técnicas de aprendizaje automático, combinando los resultados de ambos. Es relevante aclarar que en este trabajo cuando se refiere a los resultados de ModSecurity luego de analizar una transacción, se esta refiriendo más precisamente al puntaje de anomalía obtenido mediante el uso de ModSecurity junto con el Core Rule Set en modo de detección de anomalías.
El entrenamiento y construcción de los modelos de aprendizaje automático queda por fuera del alcance de este proyecto y para el prototipo se utilizó un modelo PMML ya implementado y entrenado[12].
Posterior al desarrollo del prototipo se lleva a cabo un análisis del desempeño de la herramienta frente al uso de ModSecurity por sí solo, tanto en lo que respecta al tiempo de procesamiento como a la eficiencia en la detección de transacciones maliciosas, analizando los valores de falsos positivos, falsos negativos, verdadero negativo y verdadero positivo, los cuales se definen de la siguiente manera:

- **Falso positivo (fp):** La transacción no es maliciosa y se detectó un ataque.
- **Falso negativo (fn):** La transacción es maliciosa y no se detectó un ataque.
- **Verdadero negativo (tn):** La transacción no es maliciosa y no se detectó un ataque.
- **Verdadero positivo (tp):** La transacción es maliciosa y se detectó un ataque.

3.1. Herramienta a desarrollar

La herramienta a desarrollar se denomina *WACE* (**W**eb **A**ttack **C**lassification **E**ngine). La misma será responsable de coordinar el procesamiento de las transacciones en tiempo real entre ModSecurity y los modelos de aprendizaje automático,

con el objetivo de detectar la mayor cantidad de comportamientos maliciosos o no permitidos, evitando una alta tasa de falsos positivos.

3.2. Requerimientos

A continuación, se describen los requerimientos funcionales y no funcionales que debe satisfacer la herramienta a desarrollar.

3.2.1. Requerimientos funcionales

1. Establecer un mecanismo de comunicación desde WACE hacia los modelos de aprendizaje automático. Esta comunicación debe permitir que los modelos reciban los datos de la transacción necesarios para realizar su clasificación y a su vez puedan retornar su resultado a WACE.

2. Desarrollar un esquema que posibilite la comunicación con más de un modelo de aprendizaje automático y permita a futuro agregar nuevos.

3. Se deben clasificar las transacciones que arriban y salen del servidor web utilizando ModSecurity, WACE debe acceder a ese resultado de clasificación.

4. Determinar con algún criterio definido y configurable, cuándo se bloquea una transacción.

3.2.2. Requerimientos no funcionales

1. Se debe contar con un tiempo de procesamiento adecuado. El desempeño de la aplicación web a proteger no debe verse afectado notoriamente impidiendo su uso normal.

3.3. Actores

A continuación, se describen los actores que van a interactuar con WACE:

- **ModSecurity:** ModSecurity va a analizar las transacciones e informar a WACE sobre el resultado. Adicionalmente teniendo en cuenta el resultado de WACE, ModSecurity bloqueará la transacción si es que así se determina por parte de WACE.

- **Servidor Web:** El servidor web va a hospedar la aplicación web y atender las peticiones de los clientes.
- **Modelos de Aprendizaje Automático:** Los modelos entrenados previamente, van a consumir la información de la transacción provista por WACE para luego retornar un resultado de clasificación.

3.4. Análisis de integración

En esta sección se plantea un análisis del esquema utilizado para la integración de WACE con ModSecurity.

3.4.1. Esquema del sistema

El siguiente diagrama representa el esquema general del sistema, en el cual se muestra como WACE interactúa y se integra con los demás actores.

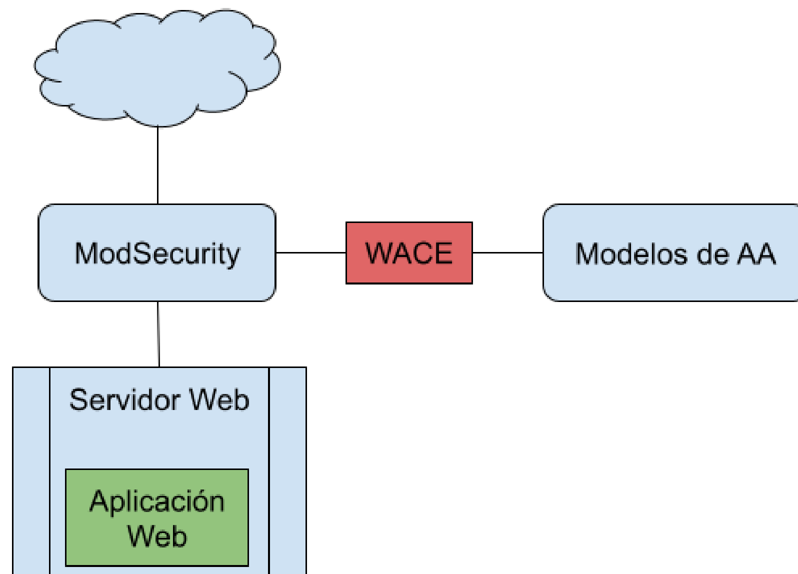

Figura 3: WACE detras de ModSecurity

Como se aprecia en el diagrama ModSecurity es el encargado de transmitir la información que WACE necesita para tomar la decisión de bloquear o no una

transacción. Dentro de esta información se encuentra la transacción, la cual WACE envía a los modelos de aprendizaje automático para su procesamiento y el resultado acumulado de puntaje de anomalías provisto por el CRS luego de ser analizada la transacción por ModSecurity.
En base a los resultados de ModSecurity y de los modelos de aprendizaje automático, WACE determina si la transacción es maliciosa o no. Dicha decisión se transmite a ModSecurity el cual le indica al Servidor Web que debe bloquear o no la misma.

3.4.2. Comunicación entre WACE y ModSecurity

En base al esquema planteado anteriormente la forma en la que WACE se comunica con ModSecurity es utilizando un mecanismo de extensión provisto por Apache, mediante el cual desarrollando un módulo de Apache se puede extender el funcionamiento de ModSecurity [13] sin modificar su código fuente. De esta manera la funcionalidad de WACE puede ser implementada y luego invocada haciendo uso del lenguaje de reglas. Más precisamente este mecanismo permite extender el funcionamiento de ModSecurity de tres maneras: desarrollando un operador, desarrollando una variable o desarrollando una función de transformación[1].
Para entender este método de una mejor manera, a continuación se van a analizar los aspectos de Apache que permiten hacer esta forma de extensión posible.

ModSecurity 2 funciona utilizando el APR (Apache Run Time), en pocas palabras ModSecurity termina siendo un módulo más de Apache que se introduce en el ciclo de procesamiento de transacciones por parte del servidor web.
Apache implementa (a partir de la versión 2) mecanismos que permiten exportar funciones, esto significa que un módulo de Apache puede exportar una o más funcionalidades para el uso de otro módulo. Esto permite desarrollar una funcionalidad en un módulo que puede ser usada por otro, u otros módulos que se invoquen más adelante. Esto tiene como desventaja que se genera una dependencia entre el módulo que exporta las funciones y el que las consume.
Los módulos tienen que ser cargados en el orden correcto y si el módulo que exporta la funcionalidad no es cargado antes que el que la consume se produce un

error.
Para solucionar este problema de dependencias Apache provee otro mecanismo denominado *optional functions* [14], estas son funciones exportadas por un módulo y luego consumidas por otro, pero sin generar una dependencia entre ambos. Básicamente se exportan las funciones de manera similar a la forma anterior, pero indicando que son funciones opcionales. Luego en el módulo que utiliza las funciones se consulta si está disponible la función antes de utilizarla, de forma que si no se encuentra el módulo que implementa las funciones no se produce ningún error, sino que simplemente no se utiliza ninguna función que fue implementada por éste. Este último mecanismo de *optional functions* es el que permite extender el funcionamiento de ModSecurity sin tener que modificar su código fuente, si se desea crear una variable, operador o función de transformación se debe crear un módulo Apache que implemente una función opcional determinada (que está ya definida y utilizada por ModSecurity de antemano) para luego, tener a disposición en el lenguaje de reglas la variable, operador o función de transformación desarrollada. En [13] se puede ver un ejemplo de uso de este mecanismo.

3.5. Flexibilidad de WACE

Con el objetivo que WACE posibilite la comunicación con más de un modelo de aprendizaje automático y permita a futuro agregar nuevos modelos, así como permitir cambiar el algoritmo que determina si se debe bloquear o no una transacción, se decidió tener componentes que implementen estas funcionalidades de forma independiente, pudiendo agregarlas o quitarlas fácilmente. Estos componentes se llaman *plugins*.
Se definen dos tipos de plugins:

- **plugins de modelo**: los plugins de este tipo son los encargados de comunicarse con los diferentes modelos de aprendizaje automático y retornar el valor de probabilidad de que una transacción sea un ataque.
- **plugin de decisión**: este plugin implementa el algoritmo de decisión que determina si una transacción es maliciosa o no, esto lo hace en base a los resultados de los plugins de modelo y de ModSecurity.

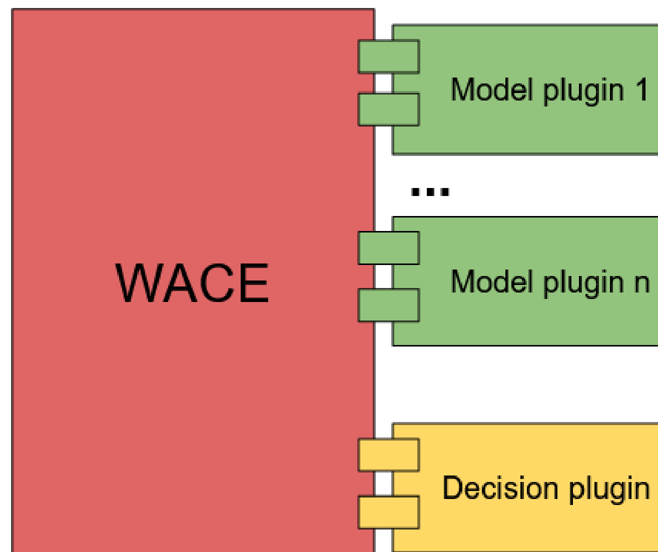

Figura 4: Plugins y WACE

3.6. Decisión

En esta sección se describen los distintos aspectos a considerar a la hora de tomar una decisión de bloquear o no una transacción.
Se presentan los resultados obtenidos por parte de ModSecurity y los modelos de aprendizaje automático, describiendo de qué manera estos resultados pueden ser combinados para llegar a un resultado final de decisión.
Por último, se introduce el concepto de balance el cual permite configurar el peso que tiene ModSecurity y los modelos en la decisión final.

3.6.1. Resultados de ModSecurity y modelos de aprendizaje automático

Es importante tener claro qué resultados se obtienen desde ModSecurity y los modelos de aprendizaje automático luego de analizar una transacción.

Resultados de ModSecurity

Teniendo en cuenta los dos modos de operación de ModSecurity[1] con el CRS, modo de detección tradicional y modo de detección de anomalías, este último es el que brinda un insumo más interesante para la toma de la decisión, ya que podemos obtener el valor del puntaje de anomalía acumulado e inclusive se pueden configurar variables específicas para un tipo de ataque. Por ejemplo, puede existir una variable que acumule el puntaje de anomalía solamente de las reglas que detectan SQL injection y análogamente pueden configurarse otras variables para otras clases de ataques. Haciendo uso de estos puntajes de anomalía específicos a determinada clase de ataque se puede llegar a saber el tipo de ataque asociado a la transacción que se está procesando por ModSecurity.
En el primer modo ModSecurity solo brindaría la información de si se debe bloquear o no la transacción, lo cual lo hace mucho menos interesante para tomar una decisión de bloqueo respecto al segundo.

Resultados de los modelos de aprendizaje automático

Los modelos de aprendizaje automático que se van a utilizar para este trabajo precisan de toda la request para brindar un resultado. El resultado va a ser un valor entre 0 y 1, indicando a mayor valor, una mayor probabilidad de que exista un ataque.

3.6.2. Variables para la decisión

Como ya se ha mencionado, ModSecurity no se va a utilizar por sí solo sino que se va a utilizar en conjunto con el Core Rule Set [1]. El Core Rule Set además de contener un conjunto de reglas de detección, define diversas variables que luego se utilizan dentro de dichas reglas. Algunas de estas variables son de particular interés ya que brindan información para la toma de la decisión. A continuación se enumeran las mismas:

- *inbound_anomaly_score_threshold:* Es la variable que almacena el valor del umbral de entrada. Este valor es el que se utiliza cuando ModSecurity funciona en modo de detección de anomalías y es el que se debe superar por el puntaje de anomalía para que una request sea determinada como maliciosa. Por defecto el CRS establece este umbral en 5.

- *outbound_anomaly_score_threshold:* Análogamente a la variable anterior, ésta variable se utiliza para establecer el umbral de salida, o sea el valor que debe superar el puntaje de anomalía acumulado por las reglas que analizan la respuesta para que se determine una transacción de respuesta como maliciosa. Por defecto el CRS establece este umbral en 4.

- *critical_anomaly_score*, *error_anomaly_score*, *warning_anomaly_score* y *notice_anomaly_score:* Este conjunto de variables se utilizan más bien como constantes ya que se incluyen en las reglas para sumar al acumulado del puntaje de anomalía en caso de que la regla tenga una evaluación positiva. Por ejemplo, una regla que busca algún indicador en la transacción que por su naturaleza, si se encuentra es considerado algo crítico, entonces dentro de dicha regla se suma al puntaje de anomalía el valor de la variable *critical_anomaly_score.*

Por defecto el CRS establece estas variables con los valores 5, 4, 3 y 2 respectivamente.

- *inbound_anomaly_score:* Es la variable que se utiliza para acumular el puntaje de anomalía de entrada. Esto quiere decir el puntaje obtenido al analizar una *request.*

- *outbound_anomaly_score:* Es la variable que se utiliza para acumular el puntaje de anomalía de salida. Es decir el puntaje acumulado al analizar una *response.*

- También hay un conjunto de variables que son usadas para acumular el puntaje de anomalía, pero discriminado por tipo de ataque. Estas variables son:

    - *sql_injection_score*
    - *xss_score*
    - *rfi_score*
    - *lfi_score*
    - *rce_score*
    - *php_injection_score*
    - *http_violation_score*
    - *session_fixation_score*

  Como se puede deducir por el nombre de las variables cada una corresponde a un tipo de ataque distinto, el uso de estas variables puede permitir saber de alguna manera el tipo de ataque que se detecta en la transacción. Por ejemplo si una request se evalúa como maliciosa y a su vez la variable *sql_injection_score* tiene un valor de 10 y las otras variables 0, se puede determinar que la transacción se detectó como maliciosa, o sea superó el umbral establecido por la variable inbound_anomaly_score_threshold (por defecto 5) porque ModSecurity consideró que contiene un ataque de *SQL injection.*

Por otro lado, considerando los modelos de aprendizaje automático, está establecido que WACE debe funcionar con más de uno, que cada modelo va a tener

asociado un valor de umbral y que como resultado del análisis de cada transacción cada modelo retorna un valor de probabilidad.
Por cada modelo se va a permitir configurar su umbral de manera de poder saber si el valor de probabilidad retornado por el modelo implica una transacción maliciosa o no.
La posible existencia de más de un modelo trae como consecuencia que se va a contar con, eventualmente, varios resultados de probabilidad por cada transacción. Esto da lugar a varias formas de considerar estos resultados y de combinarlos.
A continuación se enumeran las formas identificadas de tratar los resultados de los modelos.

3.6.3. Fórmulas de decisión

Antes de describir las diferentes formas de tratar con los resultados de probabilidad, se describen los siguientes elementos de notación:

- Se consideran n modelos $m_i \in M = \{m_0, m_1, ..., m_n\}$.
- Los umbrales de cada modelo se denotan como $u_i \in U = \{u_0, u_1, ..., u_n\}$, respectivamente.
- La probabilidad resultado asociada a cada uno de los modelos es $p_i \in P = \{p_0, p_1, ..., p_n\}$ respectivamente.
- Los valores de peso asociados a cada uno de los modelos, denominado $w_i \in W = \{w_0, w_1, ..., w_n\}$ respectivamente. Se establece que $wi \in W$ y $0 <= wi <= 1$.
- Se define r como el valor resultante luego de combinar los resultados de todos los modelos.

1- Resultado binario

Un enfoque, basado en un algoritmo de votación [19], es considerar el resultado de los modelos de aprendizaje automático como un resultado binario. Para lograr esto se calcula el resultado binario b_i, el cual es 1 si la probabilidad p_i retornada por el modelo es mayor a su umbral u_i y 0 en caso contrario. Esto quiere decir

que cuando se considera una transacción como maliciosa el resultado binario es 1 y 0 cuando no lo es. Luego de obtenido todos los resultados binarios de todos los modelos, se aplica el algoritmo de votación utilizando la siguiente formula:

$$r = \begin{cases} 1 & si \ \sum_{i=0}^{n} b_i > n/2 \\ 0 & si \ \sum_{i=0}^{n} b_i \leq n/2 \end{cases} \tag{1}$$

2- Resultado binario ponderado

Otra opción basada en la anterior consiste en la suma de los resultados binarios de clasificación de cada modelo, pero multiplicando cada uno de estos valores por un peso w_i, esto permite brindar mayor o menor importancia al resultado de cada modelo. La ecuación queda de la siguiente manera:

$$r = \begin{cases} 1 & si \ \sum_{i=0}^{n} b_i \times w_i > n/2 \\ 0 & si \ \sum_{i=0}^{n} b_i \times w_i \leq n/2 \end{cases} \tag{2}$$

3.6.4. Integración de resultados con ModSecurity

En la sección anterior se especifican formas de como combinar los resultados de los modelos de aprendizaje automático, pero para tomar una decisión final sobre la transacción esto no es suficiente, ya que también se debe tener en cuenta el resultado de ModSecurity.
Una manera de combinar estos dos resultados, considerando el método descripto en la ecuación 1, es sumar el r a un valor binario denominado $resultado_modsec \in [0, 1]$ el cual toma el valor 1 si el puntaje de anomalía calculado por ModSecurity supera su umbral y 0 en caso contrario.
Entonces, en base a estos dos resultados una fórmula para determinar si una transacción es maliciosa o no es:

$$resultado_final = \begin{cases} 1 & si \ (resultado_modsec + r) = 2 \\ 0 & si \ (resultado_modsec + r) < 2 \end{cases} \tag{3}$$

Considerando como maliciosa la transacción si el resultado_final es 1.

3.6.5. Flexibilidad en los criterios de decisión

Además de las opciones planteadas anteriormente, existen otras diversas formas de combinar el resultado de los diferentes modelos y ModSecurity. Inclusive hay todo un campo de estudio sobre los sistemas con múltiples clasificadores [16] en el cual se proponen diversas formas de combinar los resultados. Por esto y como se considera interesante tener la posibilidad de poder modificar estos criterios de decisión, este aspecto de WACE es extensible permitiendo la implementación y utilización a futuro de nuevos criterios de decisión.

3.6.6. Balance

Resulta interesante considerar una forma por la cual se pueda balancear la detección entre ModSecurity y modelos de aprendizaje automático. Con esto se busca tener la posibilidad de brindarle más importancia a los resultados de ModSecurity por frente al de los modelos y viceversa. Esto brinda a WACE una mayor flexibilidad y permite una personalización más específica para determinado ambiente de producción. Tener la posibilidad de realizar una configuración de este tipo permite balancear entre obtener una mayor capacidad de detección, ponderando más los resultados de ModSecurity, o tener una menor cantidad de falsos positivos, dando más importancia a los modelos de aprendizaje automático. Esto permite encontrar el balance adecuado entre ambas cosas que mejor se adapte a la aplicación que se pretende proteger.
Para representar el balance en el sistema se pueden definir pesos para el resultado de cada modelo y de ModSecurity, de manera de que ajustando estos pesos se pueda balancear la importancia en la decisión.

3.7. Procesamiento de transacciones

El hecho de tener que llevar a cabo una clasificación en tiempo real de las transacciones hace que sea de gran importancia el paralelismo de nuestra aplicación. En un ambiente de producción, un servidor web atiende muchos pedidos de diferentes clientes a la misma vez. De la misma manera, WACE debe ser capaz de procesar dichos pedidos.
Para esto WACE puede ejecutarse como un proceso *demonio*[18] el cual genera

un hilo de ejecución por cada transacción a procesar. La utilización de hilos de ejecución permite que múltiples subprocesos se ejecuten dentro del mismo programa en un espacio de direcciones de memoria compartida. Esto trae como ventaja que el programa ya estaría almacenado en memoria por lo tanto todos los threads accederían rápidamente a los datos que necesiten.

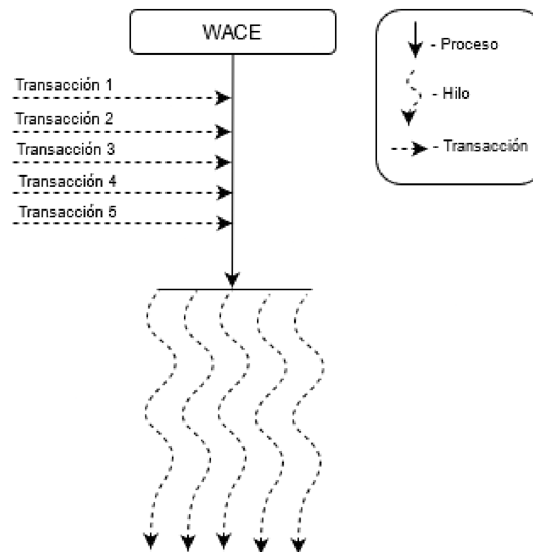

Figura 5: Formas de ejecución de WACE

En la figura 5 se visualiza el funcionamiento de WACE utilizando un solo proceso y abriendo múltiples hilos para atender las transacciones paralelamente.

4. Diseño

En esta sección se abordan los distintos aspectos que determinan como se va a construir WACE. En primer lugar se describe la arquitectura a utilizar, así como los componentes de la misma. Además, se detallan los diferentes mecanismos y estructuras de datos utilizados por los componentes del sistema para la comunicación entre los mismos.

4.1. Arquitectura de Microkernel

La arquitectura de microkernel [15] o también llamada arquitectura basada en plugins consiste en dos grandes elementos, el primero es un componente denominado *core* que se encarga del procesamiento principal de la aplicación y el segundo son un conjunto de componentes denominados *plugins*, que implementan cada uno, determinadas funcionalidades extendiendo el funcionamiento del core. Idealmente estos plugins están desacoplados entre sí y la comunicación solo se produce entre el core del sistema y los diversos plugins. El componente core es el encargado de las funcionalidades centrales y los plugins son componentes independientes que implementan alguna funcionalidad específica. El core necesita conocer qué plugins están disponibles y de qué manera puede comunicarse y acceder a ellos.

4.1.1. Ventajas y desventajas

A continuación se analizan los aspectos positivos y negativos de este patrón de arquitectura en base a determinados parámetros:

- **Agilidad:** La agilidad se refiere a la capacidad de responder rápidamente a cambios. Esta es una ventaja que posee este patrón ya que los cambios se pueden implementar rápidamente a través de nuevos plugins, evitando afectar todo el sistema.

- **Facilidad de despliegue:** Esto refiere a la facilidad de incorporar los cambios al sistema, esto es una ventaja ya que simplemente se implementan plugins nuevos sin necesidad de afectar al core ni los otros plugins dentro del sistema.

- **Pruebas:** Se refiere a la capacidad de testear cada plugin. Esto es una ventaja ya que cada plugin se puede testear independientemente del resto del sistema.
- **Performance:** En general la performance de este tipo de arquitecturas es alta, ya que se puede personalizar y optimizar las aplicaciones para que solo incluyan las características que se necesitan.
- **Escalabilidad:** Dependiendo de cómo se implementen los plugins se puede generar algo de escalabilidad.
- **Facilidad de desarrollo:** La arquitectura del microkernel requiere un diseño cuidadoso y una buena especificación de contratos a nivel de funciones, lo que la hace bastante compleja de implementar.

4.2. Arquitectura del sistema

La arquitectura de Microkernel es una buena candidata para el diseño de WACE, principalmente por la facilidad que brinda esta arquitectura para extender la funcionalidad, por ejemplo agregando nuevos modelos de aprendizaje automático y criterios de decisión, simplemente implementando nuevos plugins.
Concibiendo desde el diseño la mencionada facilidad de extensión, WACE termina obteniendo un potencial mucho más grande. La flexibilidad que brinda esta arquitectura permite desarrollar a futuro diferentes elementos que mejoren la eficiencia en la detección de transacciones maliciosas.

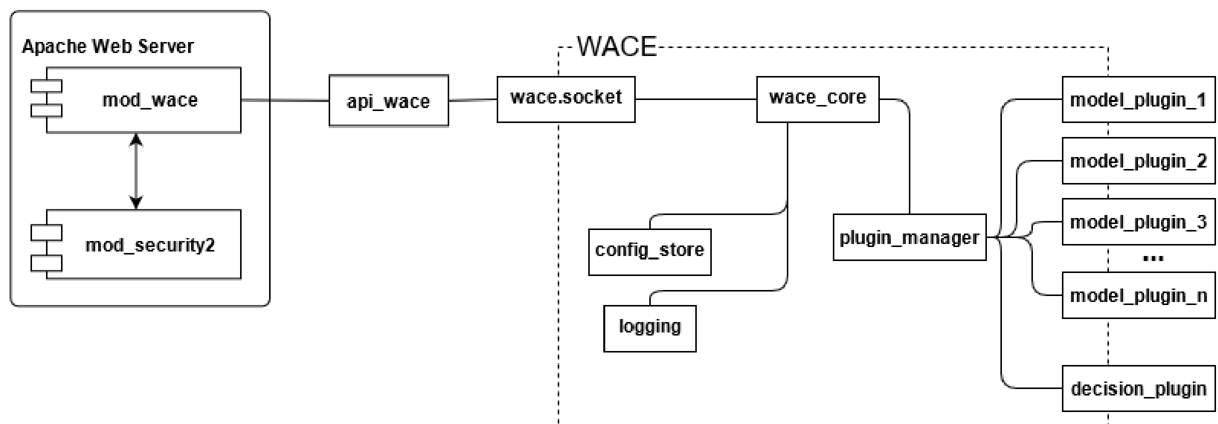

Figura 6: Arquitectura de WACE

En la figura 6 se ilustran los diferentes componentes de la arquitectura y la comunicación entre estos. A continuación se describen brevemente:

- **wace_core:** Este componente es el *core* de esta arquitectura de Microkernel y es el encargado de realizar las principales funcionalidades centrales del sistema, incluyendo la implementación de todos los mecanismos necesarios para la interacción con los diversos plugins y la api (*api_wace*).

- **api_wace:** Es el componente que se encargará de hacer de intermediario entre el módulo de Apache y el *wace_core*. Su responsabilidad principal es que las transacciones recibidas en el módulo lleguen hacia el core.
  Contar con esta API permite tener una arquitectura más flexible ya que posibilita la comunicación con WACE desde cualquier componente que haga

uso de la API. En otras palabras, WACE no esta limitado a funcionar solamente con un módulo de Apache. Esto quiere decir que se podría utilizar sin problemas WACE con ModSecurity 3 desarrollando una *api_wace* que tenga la capacidad de hacer de intermediaria entre estos dos.

- **mod_wace:** Este módulo implementa un operador de ModSecurity nuevo utilizando el método de extensión de ModSecurity mediante módulos de Apache (descripto en la sección 3.4.1). Su objetivo es obtener la transacción que está siendo analizada por ModSecurity, para luego enviarla hacia el *wace_core* mediante el uso de la *api_wace*. Adicionalmente una vez que la transacción es enviada a WACE, el módulo debe obtener el resultado del análisis.

- **config_store:** Es el componente encargado de manejar toda la información de configuración obtenida a partir del archivo de configuración de WACE.

- **logging:** Todos los mensajes de logs generados a partir de todos los componentes de WACE son manejados por esta parte del sistema, los mensajes son escritos en el archivo de logs indicado en la configuración.

- **plugin_manager:** Es el componente dentro del core del sistema que se encarga de coordinar la comunicación con los diferentes plugins. Todas las funcionalidades relacionadas con el descubrimiento, inicialización y registro de funciones de plugins, son implementadas en esta parte del sistema.

- **plugins:** Son los componentes independientes entre sí que tienen diferentes objetivos e implementan determinados aspectos de WACE.
  Se identifican los siguientes tipos de plugins:

  - **model_plugin:** El objetivo de este plugin es intermediar entre el *wace_core* y los modelos de aprendizaje automático. Para agregar al análisis de las transacciones un nuevo modelo de aprendizaje automático se debe simplemente registrar el plugin del modelo con el *wace_core*. Por cada modelo que se quiera integrar al procesamiento de las transacciones va a existir un plugin asociado.

  - **decision_plugin:** Al igual que los modelos, el criterio de decisión (Ver sección 3.6) se implementa como un plugin, solamente puede haber un

único plugin de decisión a la vez.

- **mod_security2:** Es el módulo Apache de ModSecurity versión 2.

4.2.1. Descubrimiento de plugins

En este tipo de arquitectura es importante contar con una etapa de descubrimiento en la cual el *core* del sistema, descubre nuevos plugins. En este caso los plugins se van a encontrar referenciados en la configuración con su nombre y su ruta completa, de esta forma, al leer la configuración, el *core* va a conocer esta información y va a poder cargar en el sistema uno a uno cada plugin.

4.2.2. Comunicación entre el core y los plugins

Luego de descubiertos, los plugins deben comunicarse con el *core*. Dicha comunicación se realiza mediante el uso de librerías dinámicas, básicamente los plugins se construyen como librerías que implementan un conjunto de funciones definidas, que luego son invocadas por el core.
Dichas funciones son las siguientes:

- Una *función de inicialización*, la cual tiene como objetivo el registro del plugin con el core del sistema.

- Una *función de análisis de la transacción* que básicamente se encarga de analizar los datos de la transacción por parte del modelo de aprendizaje automático y retornar un resultado. Esta función solo es necesaria en los denominados plugins de modelo (*model_plugin*).

- Por último en caso de que se trate de un plugin de decisión (*decision_plugin*) es necesaria una función que reciba los resultados de los modelos y de ModSecurity como parámetro y retorne un resultado de decisión indicando si se debe bloquear o no la transacción.

4.2.3. Comunicación entre el core y mod_wace

Como se puede apreciar en la figura 6 la API *api_wace* es la encargada de implementar el mecanismo de comunicación entre WACE y el módulo de Apache

mod_wace. Dicha API permite que las funcionalidades de WACE puedan ser invocadas desde el módulo de Apache. Su objetivo es funcionar de intermediario entre ambos sistemas y para ello implementa un método de comunicación con cada uno de estos.

La comunicación entre la *api_wace* y *mod_wace* se implementa mediante el uso de librerías dinámicas. Las funcionalidades de WACE van a poder ser utilizadas por *mod_wace* mediante la invocación de las funciones expuestas por la liberia dinámica *api_wace*.

Además de comunicarse con el módulo de Apache, la *api_wace*, debe a su vez comunicarse con el *wace_core*. Dicha comunicación se realiza mediante el uso de un método de comunicación interproceso o IPC por sus siglas en ingles. Existen diversos métodos de IPC [20] pero particularmente en el prototipo a implementar se utiliza el mecanismo de *File Sockets* o más comúnmente conocido como *UNIX Domain Socket*, el cual permite el intercambio de datos bidireccional entre procesos que se ejecutan en el mismo sistema operativo.

4.2.4. Protocolo de comunicación

Como se mencionó anteriormente, la comunicación entre la *api_wace* y el *wace_core* se realiza mediante el uso de un *File Socket*, esto genera la necesidad de definir un protocolo de comunicación para permitir que WACE reciba la información necesaria y a su vez retorne los resultados luego de realizar el análisis de los datos proporcionados. Para esto se establece un protocolo simple en el cual se utilizan los dos primeros caracteres de cada mensaje para identificar su contenido, seguido de estos caracteres la información del mensaje propiamente dicha, hasta llegar al carácter de fin de línea.

Identificador	**Descripción**
A0	Identificación del cliente y request.
Q0	*Request Line.*
Q1	*Request Method.*
Q2	*Request URI.*
Q3	*Query String.*
Q4	*Request Protcol.*
Q5	*Hostname.*
Q6	*Request Body.*
Q7	Número de encabezados.
Q8	*Request header.*
M0	*Inbound Anomaly Score.*
M1	*Inbound Anomaly Score Threshold.*
EE	Fin del mensaje.

Cuadro 1: Protocolo de comunicación

Como se puede observar en la tabla 4.2.4 con los mensajes mostrados se puede realizar un análisis completo de las *requests*. El mensaje A0 permite enviar información del cliente y de la transacción como por ejemplo IP, puerto de origen, ID de la request. Luego los mensajes Q0 hasta el Q8 son utilizados para enviar las diferentes partes de la transacción. Los mensajes M0 y M1 forman parte de información obtenida a partir del análisis realizado por ModSecurity a la request, refieren al puntaje de anomalía que obtuvo la transacción enviada y el umbral de puntaje de anomalía configurado respectivamente. Por último, el mensaje EE es

utilizado para indicar a WACE que toda la información fue enviada y que se espera por el resultado del análisis.
Los mensajes pueden llegar en cualquier orden a excepción del EE el cual debe llegar al final, ya que luego de este mensaje no se aceptan más datos y se comienza con el análisis de la transacción. Igualmente para evitar problemas, en la implementación de este prototipo se utiliza un filesocket del tipo *SOCK_SEQPACKET* [33]. Este tipo de socket asegura que los mensajes van a llegar en el orden que fueron mandados.
Para dejar más claro el concepto, a continuación se muestra un ejemplo.

Si la request que llega al servidor es:

```
GET https://www.fing.edu.uy/ HTTP/1.1
Host: www.fing.edu.uy
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64
Accept: text/html
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
```

ModSecurity, luego de analizar esta request no encuentra ningún elemento para bloquearla entonces el *Inbound Anomaly Score* es 0.

Utilizando el protocolo, los mensajes que la *api_wace* debe enviar hacia WACE son:

```
A0192.168.1.2:4444 - id:123Axc23xDse
Q0GET https://www.fing.edu.uy/ HTTP/1.1
Q1GET
Q2/
Q3GET https://www.fing.edu.uy/
Q4GET
Q5www.fing.edu.uy
Q75
Q8Host: www.fing.edu.uy
```

```
Q8User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64
Q8Accept: text/html
Q8Accept-Language: en-US,en;q=0.5
Q8Accept-Encoding: gzip, deflate, br
M00
M14
EE
```

Finalmente, luego de recibidos estos mensajes por WACE, la *api_wace* recibirá como respuesta 1 si WACE considera que hay que bloquear la transacción o 0 en caso contrario, en caso de que halla sucedido un error se retorna un valor mayor a 1.

El uso de un protocolo además de ser necesario para establecer la comunicación a través del FileSocket, brinda una ventaja adicional respecto a la flexibilidad en el uso de WACE. Al tener un protocolo bien establecido cualquier programa desarrollado en cualquier tipo de lenguaje que desee hacer uso de WACE puede hacerlo simplemente apegándose al protocolo de comunicación. Esto amplía el rango de posibilidades de uso de WACE.

4.2.5. Diagrama de secuencia

Para comprender de mejor manera el flujo de información entre los diversos componentes del sistema, en la figura 7 se presenta un diagrama de secuencia que representa desde el momento que llega una transacción al servidor web, hasta que este último recibe la clasificación de si es un ataque o no.

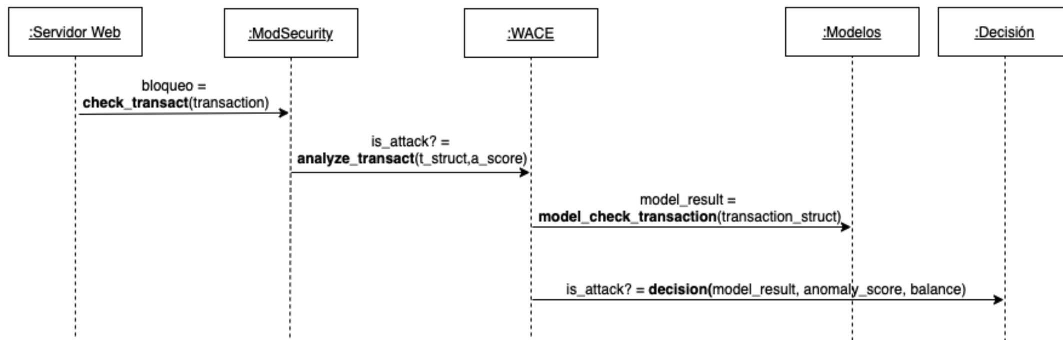

Figura 7: Diagrama de secuencia

Para un mayor entendimiento del diagrama presentado anteriormente, a continuación, se describen las funciones y sus valores retornados:

- **check_transact:** Esta función tiene como objetivo enviar la transacción recibida por el servidor web a ModSecurity, para luego ser analizada y obtener como resultado si se debe o no bloquear la transacción.

- **analyze_transaction:** Desde ModSecurity se invoca la regla que ejecuta el operador implementado para comunicarse con *wace* y se envían los diferentes elementos de la transacción (encabezado, cuerpo, parámetros, etc.) en un estructurado denominado *t_struct* junto con el puntaje de anomalía calculado por ModSecurity en la variable *a_score*. Luego de analizada por parte de WACE y ModSecurity la información se retorna 1 en la variable is_attack? si se encontró un ataque en la transacción y 0 en caso contrario.

- **model_check_transaction:** Una vez obtenidos los datos de la transacción, WACE envía estos hacia los modelos de aprendizaje automático. En retorno se recibe una estructura con los resultados del análisis por parte de dichos modelos.

- **decision:** Luego de obtenidos los resultados de los modelos, estos, junto con el puntaje de anomalía obtenido por ModSecurity, actúan de insumo para que el algoritmo de decisión determine si hay o no un ataque en la transacción analizada. En caso de que sí lo halla, la función retorna 1 en la variable *is_attack?*, en caso contrario 0.

4.2.6. Configuración

Muchos aspectos relacionados con WACE se pretende que sean configurables. Esto se implementa mediante el uso de un archivo de configuración en el cual, mediante ciertas directivas definidas, el usuario puede configurar diversos parámetros de WACE.
El archivo de configuración tiene los siguientes parámetros:

```
LogPath <path_log>

LogLevel <log_level>

ModelPluginPath <plugin_name> <plugin_path>

ModelPluginWeight <plugin_name> <plugin_weigth>

ModelPluginThreshold <plugin_name> <plugin_threshold>

DecisionPluginPath <plugin_name> <plugin_path>

SocketPath <plugin_path>

ModSecurityWeight <modsec_weight>

Balance <balance>

MaxQueuedConnections <max_connections>
```

En la siguiente tabla se describen los parámetros:

Nombre	Parametros	Descripción
LogPath	path_log	Especifica la ruta en donde se ubica el archivo de log.
LogLevel	log_level	Especifica el nivel de login (0-No log, 1-Minimo, 2-Medio, 3-Debug)
ModelPluginPath	plugin_name	Nombre del plugin
	plugin_path	Especifica la ruta en donde se ubica el model plugin deseado
ModelPluginWeight	plugin_name	Nombre del plugin
	plugin_weigth	Especifica el peso deseado para determinado plugin
DecisionPluginPath	plugin_name	Nombre del plugin
	plugin_path	Especifica la ruta al plugin de decisión
SocketPath	socket_path	Especifica la ruta en donde se ubica el socket
ModSecurityWeight	modsec_weight	Especifica el peso que se le da a ModSecurity a la hora de la decisión
Balance	balance	Especifica el balance entre ModSecurity y WACE. Este tiene que ser un numero del 0 al 10, si el valor es alto, se le da mayor consideración a los modelos, por lo contrario, si el valor es bajo se le da mayor importancia a Modsecurity.
MaxQueuedConnections	max_connections	Especifica la cantidad máxima de conexines que acepta el socket.
ModelPluginThreshold	plugin_name	Nombre del plugin
	threshold	Umbral del plugin

Cuadro 3: Directivas de configuración.

4.2.7. Estructuras de datos

A continuación se describen las diferentes estructuras de datos necesarias para interactuar entre los distintos componentes del sistema.

Request

Esta estructura representa los datos de la request que son enviados a los model_plugin para ser analizados por el modelo de aprendizaje automático correspondiente.

Nombre	**Tipo de dato**	**Descripción**
req_line	string	Request line.
req_method	string	Request method.
req_uri	string	Request URI.
query_string	string	Query string.
req_protocol	string	Request. protocol
hostname	string	Hostname.
req_body	string	Request body.
req_headers	list<string>	Lista de encabezados de la request.
header_count	integer	Cantidad de encabezados.
anomaly_score	float	Anomaly score generado por ModSecurity.
modsec_threshold	float	Umbral de ModSecurity.
client_data	string	Información de la transacción.

Cuadro 4: Estructura de datos - Request

Clasificación

Este dato es el que genera el model_plugin luego de realizar el análisis de la transacción.

Nombre	**Tipo de dato**	**Descripción**
probability	float	Probabilidad de ataque.

Cuadro 5: Estructura de datos - Clasificación

Decisión

Estos datos se utilizan en el decision_plugin para determinar el resultado final por parte de WACE (1 si se detectó una transacción maliciosa o 0 en caso contrario).

Nombre	Tipo de dato	Descripción
anomaly_score	integer	Anomaly score generado por ModSecurity.
threshold_modsecurity	int	Umbral de ModSecurity
cant_plugins	integer	Cantidad de resultados de model_plugins a analizar.
weight_models	<string,float>	Asocia el nombre de un modelo con su peso.
threshold_models	<string,float>	Umbral de los modelos
weight_modsecurity	float	Peso de ModSecurity.
wace_balance	float	Balance correspondiente a WACE.
modsec_balance	float	Balance correspondiente a ModSecurity.
results	<string,float>	Lista de resultados de plugins.

Cuadro 6: Estructura de datos - Decisión

4.2.8. Logging

Se definen tres niveles de logging,

- *Nivel 1:* Este primer nivel implica un registro de mensajes mínimo en los logs. Se produce un registro en los logs cuando se inicia y finaliza WACE, cuando se detecta una transacción maliciosa y cuando se produce un error o advertencia.

- *Nivel 2:* En el nivel dos, el registro en los logs es más detallado, registrando todo lo del nivel anterior y además el resultado del análisis de todo el resto de transacciones sean o no catalogadas como maliciosas.

- *Nivel 3:* En el último nivel además de registrar todo lo anterior, se agrega a los logs información más específica sobre el funcionamiento de la aplicación, pensado más para el *debugging* de la aplicación

Tanto el nivel a utilizar en el registro de los logs, como la ruta al archivo de logs es configurable.

5. Implementación

Teniendo en cuenta las decisiones de diseño expuestas anteriormente, en esta sección se profundiza en los aspectos relacionados a la implementación de WACE.

5.1. Lenguaje de programación

A la hora de la elección del lenguaje de programación hay que considerar los diferentes elementos del sistema que se visualizan en la figura 6. Básicamente para obtener un prototipo funcional se deben implementar los componentes:

- ***mod_wace***.
- ***api_wace***.
- ***wace_core***.
- al menos un ***model_plugin*** y un ***decision_plugin***.

Al ser ***mod_wace*** un módulo de Apache se opta por implementarlo en el lenguaje de programación más apropiado para este tipo de componentes que es C. En consecuencia, debido que la ***api_wace*** se incluye como una librería dinámica dentro del módulo ***mod_wace***, esta API puede también implementarse en C o en C++, en este último caso haciendo uso de ciertas directivas que permitan la invocación de las funciones de la API desde un programa en C.
Para facilitar la integración con el módulo, se decidió utilizar C para el desarrollo de la API.
En lo que respecta al ***wace_core*** el lenguaje a utilizar no está restringido, pero se optó por utilizar C++ para implementar todo en un lenguaje similar y además debido a su buena reputación para la implementación de sistemas que requieran de un buen desempeño y su facilidad para el manejo de strings se considera como la opción más apropiada. Como en el caso de la API, los plugins, ***model_plugin*** y ***decision_plugin*** deben implementarse en el mismo lenguaje que el ***wace_core***.

5.2. Componentes

En la siguiente imagen se pueden observar los principales componentes desarrollados:

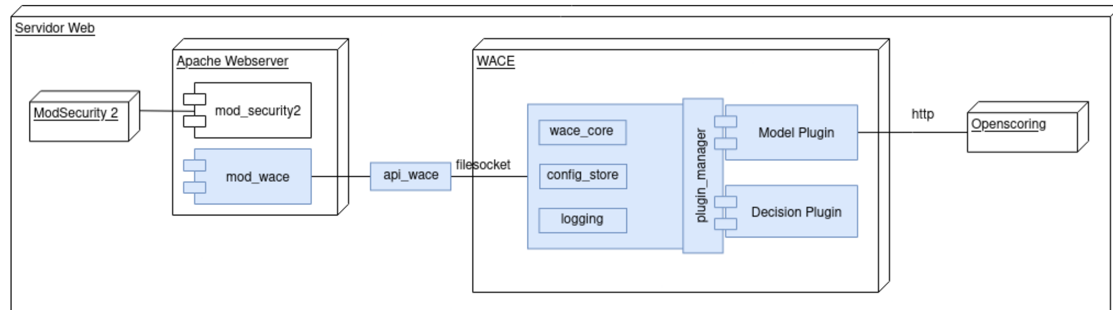

Figura 8: Diagrama de despliegue

A continuación se detallan las funcionalidades de cada componente implementado.

5.2.1. Core del sistema

Todas las funcionalidades del *core* del sistema se dividen en los cuatro componentes descriptos a continuación:

wace_core

La coordinación de todas las partes del sistema descriptas anteriormente se realiza en este componente principal. Además, las nuevas solicitudes son recibidas aquí implementando un mecanismo de multithreading donde cada nueva solicitud es atendida por un nuevo hilo en el proceso.
El *FileSocket* que recibe la información de las transacciones desde la *api_wace* es implementado en este componente.

config_store

Es el encargado de leer las directivas que se definen en el archivo de configuración. El archivo es analizado, se verifica que su sintaxis sea la correcta y luego se cargan en memoria todos los parámetros leídos para que luego puedan ser utiliza-

dos por el componente del sistema que lo requiera. En la tabla de la sección 4.2.6 se muestran las directivas de configuraciones posibles y para qué se utilizan.

plugin_manager

La administración del mecanismo de plugins se realiza aquí. Como ya se ha mencionado los plugins pasan por diferentes etapas, dentro del *plugin_manager* se proveen las funcionalidades necesarias para realizar estas etapas. A continuación se describen brevemente:

- **Descubrimiento:** Los plugins son descubiertos al leer el archivo de configuración de WACE. A partir de este, se obtiene el nombre y la ruta completa donde se encuentra el archivo *.so* correspondiente a cada plugin.
- **Carga:** Al ser los plugins librerías dinámicas, estos son cargados a partir de la ruta del archivo .so obtenida en la etapa anterior, haciendo uso de la funciones *dlsym* y *dlopen* de la librería *dlfn.h*.
- **Registro:** Luego de cargado el plugin se ejecuta su función de inicialización (Ver sección 5.2.2) la cual se encarga del registro en el sistema de las funciones implementadas por el plugin. El registro consiste en almacenar en una estructura, un puntero a la función que implementa cada plugin.
- **Ejecución de las funciones registradas:** Una vez que se registran las funciones, pueden ser invocadas por el *core* recorriendo la estructura que almacena dichas funciones y ejecutándolas como si fueran funciones del propio sistema.

logging

En este módulo se implementan los diferentes niveles de logging y la forma en la cual se van a escribir los mensajes en el archivo de log. Como ya se mencionó anteriormente, se cuentan con tres niveles de logging y dentro de cada nivel hay tres tipos de mensaje diferentes, estos son:

- **INFO:** En esta categoría se incluyen todos los mensajes que informan de alguna acción que se está tomando dentro del sistema. Un mensaje de este estilo no representa ningún problema en el sistema

- **WARNING:** Se registran mensajes que informan de advertencias del sistema. Las advertencias son eventos que indican de un problema o posible problema pero que no implica que no se pueda seguir ejecutando el sistema.

- **ERROR:** Con esta etiqueta se registran mensajes que representan errores del sistema. Los errores son eventos usualmente graves que impiden la ejecución normal del sistema.

5.2.2. Plugins

Para la implementación de plugins es necesario respetar determinados lineamientos:

- Se deben implementar en C++ como librerías dinámicas.

- Es necesario que contengan la implementación de una función de inicialización, la cual permite que los plugins se registren al sistema (ver apéndice 9.2 para mas detalles).

- Dentro de la función de inicialización, es necesaria la implementación de una función que ejecute el análisis de la transacción o la decisión, dependiendo de si se trata de un plugin de modelo o de decisión:

  - En caso de que sea un plugin de tipo modelo, la función que implementa la comunicación con el modelo debe tener el siguiente formato:

    ```
    double checkTransaction(Request req)
    ```

    La función toma como parámetro un elemento de tipo *Request*, este tipo es definido por WACE como un estructurado que contiene toda la información de una request, el plugin puede acceder directamente a los elementos de dicha estructura. La estructura se define de la siguiente manera:

    ```
    struct request_t{
        string req_line;
    ```

```
        string req_method;
        string req_uri;
        string query_string;
        string req_protocol;
        string hostname;
        string req_body;
        string * req_headers;
        int header_count;
        int anomaly_score;
        int modsec_threshold;
        string client_data;
    };
```

Luego para retornar el resultado de la request simplemente se debe retornar un número entre 0 y 1 que representa la probabilidad de que la transacción sea maliciosa según el análisis realizado por el modelo.

- En caso que se trate de un plugin de tipo decisión, la función que implementa la decisión debe tener la siguiente definición:

```
        int decision(DecisionInfo di){
```

La función toma como parámetro un elemento de tipo *DecisionInfo*, esta estructura almacena toda la información disponible para la toma de la decisión, incluyendo la información obtenida luego del análisis por parte de todos los modelos de aprendizaje automático integrados en WACE. Dentro del plugin de decisión los datos se pueden obtener accediendo directamente a la estructura.
El tipo de dato *DecisionInfo* contiene los siguientes elementos:

```
    struct decisioninfo_t{
        int anomaly_score;
        int threshold_modsecurity;
        int cant_plugins;
```

```
        map<string,float> weight_models;
        map<string,float> threshold_models;
        float weight_modsecurity;
        float wace_balance;
        float modsec_balance;
        map<string,float> results;
    };
```

Como se mencionó anteriormente existen dos tipos de plugins, los *model_plugin* y los *decision_plugin*. En el primero se encuentra la lógica para comunicarse con un determinado modelo de aprendizaje automático y en el segundo es donde se determina si la transacción es maliciosa o no en base a los resultados de los *model_plugin* y *ModSecurity*. A continuación se detallan como se construyeron los plugins utilizados en el prototipo implementado:

model_plugin:

Se desarrolla un plugin de este tipo para introducir en el análisis de la transacción un modelo de aprendizaje automático implementado en PMML que fue previamente entrenado y entregado por los tutores.
Para poder tener el modelo a disposición y que pueda recibir la información de las transacciones, además del plugin, se debe hacer uso de otra herramienta que permite *servir* el modelo. La herramienta referida es *Openscoring*[25], esta permite levantar un servidor REST, al cual, mediante transacciones HTTP, se le puede cargar el modelo implementado en PMML y luego consultarlo sobre el contenido de una transacción. Por esto el plugin internamente además de implementar las funciones necesarias para comunicarse con WACE, debe implementar el envío de un mensaje PUT y POST y las funciones necesarias que procesen las transacciones para que el modelo las interprete. Para esto se utilizó una librería que permite realizar consultas HTTP y otra que facilite el trabajo con archivos de tipo JSON.

decision_plugin:

En este plugin se implementa un algoritmo de decisión, el cual va a determinar si una transacción es maliciosa o no. La decisión se implementa mediante un algoritmo de votación con peso entre el resultado de los *model_plugin* y *ModSecurity* (ver apéndice 9.3 para más detalles).

5.2.3. mod_wace

Como se mencionó en el capítulo de diseño, *mod_wace* es el módulo de Apache que se encarga de la comunicación con WACE. En este módulo es donde se define el operador que se va a utilizar en las reglas. Para implementar esto último se utilizó el mecanismo de funciones opcionales de apache. Este proceso consiste en dos pasos:

1. Primero se le pide a Apache que busque la función de registro *modsec_register_operator*, que previamente habrá sido exportada por ModSecurity.

```
fn = APR_RETRIEVE_OPTIONAL_FN(modsec_register_operator);
```

2. En este paso se define el nombre del operador y dos funciones: *wace_init* y *wace_exec*, la primera se encarga de inicializar el operador, por ejemplo reservar memoria para uso interno del operador y la segunda función se encarga de procesar las funciones que defina el operador. En este caso *wace_exec* se encarga de enviar los datos de la request a WACE y recibir el resultado de este.

```
fn("wace", (void *)wace_init, (void *)wace_exec);
```

Una vez definido el operador se pueden definir reglas, por ejemplo:

```
SecRule TX:ANOMALY_SCORE "@wace" "id:949111, deny"
```

5.2.4. api_wace

La comunicación entre el módulo de Apache y WACE es posible gracias a este componente. La denominada *api_wace* es una librería dinámica que implementa una función con los elementos de la request y algunos datos de ModSecurity como parámetro. Al ser invocada esta función por parte del módulo de Apache, estos datos son enviados a WACE a través de su FileSocket respetando el protocolo de comunicación previamente definido (ver sección 4.2.4).

6. Pruebas del sistema

Para la realización de las pruebas se montó un ambiente de pruebas que consistió en una máquina virtual con Fedora Linux versión 33, 11GB de RAM y 1 CPU. En dicha máquina se instaló WACE, Modsecurity en su versión 2.9.3 y Apache 2.4.46 (ver sección 9.1).
Se realizaron dos tipos de pruebas, las primeras con el objetivo de analizar si el uso de WACE mejora la efectividad en la detección de transacciones maliciosas y las segundas para medir el desempeño de WACE en los que respecta al tiempo, principalmente para analizar cuanto tiempo se agrega al procesamiento de la transacción al hacer uso de WACE.

6.1. Datos de prueba

Se utilizó un conjunto de datos que contiene 25254 transacciones, estos datos fueron generados para un desafío propuesto en el 2007 en la 18th European Conference on Machine Learning (ECML) y en la 11th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)[34].
A partir de estos datos se construye un archivo de tipo CSV con todas las transacciones, este archivo contiene una columna para cada parte de la transacción (*method, path, http_version, content-length, content-language, content-encoding, content-loaction, content-md5, content-type, expires, connection, accept, date, mime-version, te, user-agent, warning y body*) y dos columnas más, la denominada *id* que contiene un identificador para cada transacción y la denominada *class* que indica la clasificación que tiene la transacción. Las transacciones pueden estar clasificadas en las clases: *SqlInjection[22], XSS[27], XPathInjection[28], LdapInjection[29], SSI[30], PathTransversal[31] y OsCommanding[32]* las cuales se corresponden con el ataque que indica su nombre, y la clase *Valid* que se utiliza para las transacciones que se consideran válidas y no se clasifican como maliciosa.
Dentro del conjunto de datos se encuentran transacciones que utilizan el método *POST, GET y PUT* y las versiones de HTTP utilizadas son, *HTTP/1.0 y HTTP/1.1.*

6.2. Pruebas de desempeño

A partir de los datos descriptos en la sección anterior se confeccionó un script en *Python* que procesa el conjunto de datos y realiza consultas al servidor web. Como resultados de su ejecución, el script genera un archivo CSV con tres columnas *ID_REQ, HTTP_CODE, CLASS*,

- *ID_REQ:* El número identificador de la request.
- HTTP_CODE: El código de respuesta HTTP generado por el servidor web. Este es 403 si la transacción fue bloqueada, y en caso contrario 200.
- *CLASS:* Indica si la transacción es efectivamente un ataque o no. Si la request no es maliciosa el valor en esta columna es *valid* y en caso que la transacción sea un ataque, el valor de esta columna es el nombre del ataque (ej. XSS). Esta última columna se obtiene a partir del conjunto de datos de prueba previamente clasificados.

De esta forma, a partir de estos datos se puede determinar cuántos falsos positivos, falsos negativos, verdaderos negativos y verdaderos positivos se generan con la utilización de WACE junto con ModSecurity, en comparación con solamente ModSecurity.
Antes de mostrar los resultados se consideran las siguientes definiciones:

- **Falso positivo (fp):** La transacción no es maliciosa y se detectó un ataque
- **Falso negativo (fn):** La transacción es maliciosa y no se detectó un ataque
- **Verdadero negativo (tn):** La transacción no es maliciosa y no se detectó un ataque.
- **Verdadero positivo (tp):** La transacción es maliciosa y se detectó un ataque.
- **Accuracy:** Este valor determina que tan bueno es el modelo clasificando correctamente a las transacciones. Refiere a la fracción de transacciones que fueron clasificadas de manera correcta.

- **Precisión:** Este indicador determina que proporción de las transacciones detectadas como maliciosas son realmente maliciosas.
- **Recall:** Determina que proporción de transacciones maliciosas son detectadas como tal.

Se realizaron varias pruebas considerando los distintos niveles de paranoia definidos en el CRS (desde el nivel 1 al 4). Para cada nivel de paranoia se realizaron dos pruebas, una utilizando solo Modsecurity y otra considerando en igual proporción los resultados del análisis de ModSecurity y WACE, es decir solo se bloquea si la mayoría de los modelos y ModSecurity detectan una amenaza.

6.2.1. Resultados utilizando paranoia level 4

Resulta interesante medir el desempeño de WACE en conjunto con ModSecurity utilizando el Core Rule Set en el nivel más alto de paranoia (paranoia level 4), este nivel es el que considera la mayor cantidad de reglas y permite llegar a detectar la mayor cantidad de ataques, incluidos aquellos que pueden ser poco comunes. El hecho de utilizar este nivel de paranoia trae como principal desventaja la generación de muchos falsos positivos.
A continuación se muestra la cantidad de falsos positivos, falsos negativos, verdaderos positivos y verdaderos negativos resultantes de la prueba:

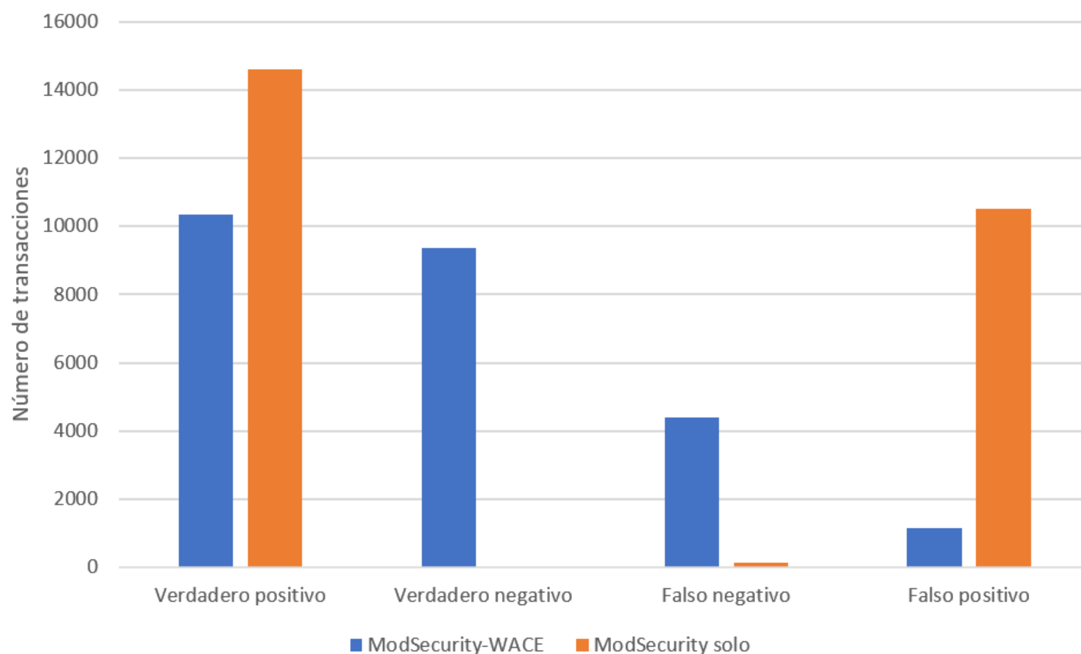

Figura 9: Resultados utilizando el nivel de paranoia 4

A continuación se calculan los valores de *accuracy, precision y recall*:

- **Accuracy:** En el caso de ModSecurity solo un 58 % de transacciones fueron clasificadas correctamente. Para el caso de ModSecurity con WACE un 78 % fueron clasificadas correctamente.

- **Precisión:** En el caso de ModSecurity, de las transacciones clasificadas como maliciosas el 58 % de ellas realmente lo son y en el caso de ModSecurity con WACE el 90 % de las transacciones clasificadas como maliciosas contienen realmente un ataque.
- **Recall:** En el caso de ModSecurity se detecta correctamente el 99 % de los ataques y en el caso de ModSecurity con WACE se detecta correctamente el 70 % de los ataques. Al analizar por si solo este indicador podría pensarse que ModSecurity por si solo es mejor, pero este porcentaje se da porque ModSecurity sin WACE clasifica prácticamente todas las transacciones como ataque.

Como se puede visualizar en la figura 9 el uso de ModSecurity por si solo con éste nivel de paranoia trae como resultado que la mayor parte de las transacciones sean detectadas como maliciosas (aunque realmente no lo sean), por esto se aprecia un valor muy alto de falsos positivos y también de verdaderos positivos. Inclusive se puede notar que prácticamente ninguna transacción no maliciosa fue clasificada como tal. Por lo tanto, el uso de solamente ModSecurity en este nivel de paranoia y con los datos de prueba utilizados no es tan bueno dada la cantidad de falsos positivos que se generan, ya que se bloquearía gran parte de las transacciones.
Por otro lado, visualizando los resultados utilizando ModSecurity junto con WACE, se pueden apreciar unos resultados mucho más alentadores presentando una reducción drástica de los falsos positivos, haciendo que el escenario de uso de ModSecurity junto con WACE sea mejor. Esto se ve reflejado en la diferencia de casi que el doble de precisión, implicando que cuando se bloquea una transacción utilizando WACE junto con ModSecurity es muy probable que realmente sea un ataque.
Igualmente, el uso de WACE con el modelo de aprendizaje automático utilizado no es perfecto y hay un 30 % de transacciones maliciosas que no fueron detectadas como tal.
En conclusión, se considera que el uso de WACE con ModSecurity brinda una mejora sustancial respecto a la utilización de este último por sí solo. Además, es importante tener en cuenta que los resultados de WACE van a depender de que tan buenos sean los modelos de aprendizaje automático utilizados y de la forma en la que se combinan los diferentes resultados de los mismos, esto quiere decir

que si se mejoran los modelos o el criterio de decisión, los resultados posiblemente se mejorarían aún más.

6.2.2. Resultados utilizando paranoia level 3, 2 y 1:

Así como se realizaron pruebas utilizando el paranoia level 4, resulta interesante analizar los resultados que se generan con el resto de niveles de paranoia (3, 2 y 1). Como mencionamos en la sección del estado del arte a medida que se aumenta el nivel de paranoia se van agregando cada vez más reglas y caracteres especiales a las mismas, esto tiene como ventaja que se aumenta el rango de detección de ataques, pero como principal desventaja el elevado número de falsos positivos que se generan a medida que se aumenta el nivel.
En este caso se presentan tres gráficas por cada nivel de forma conjunta:

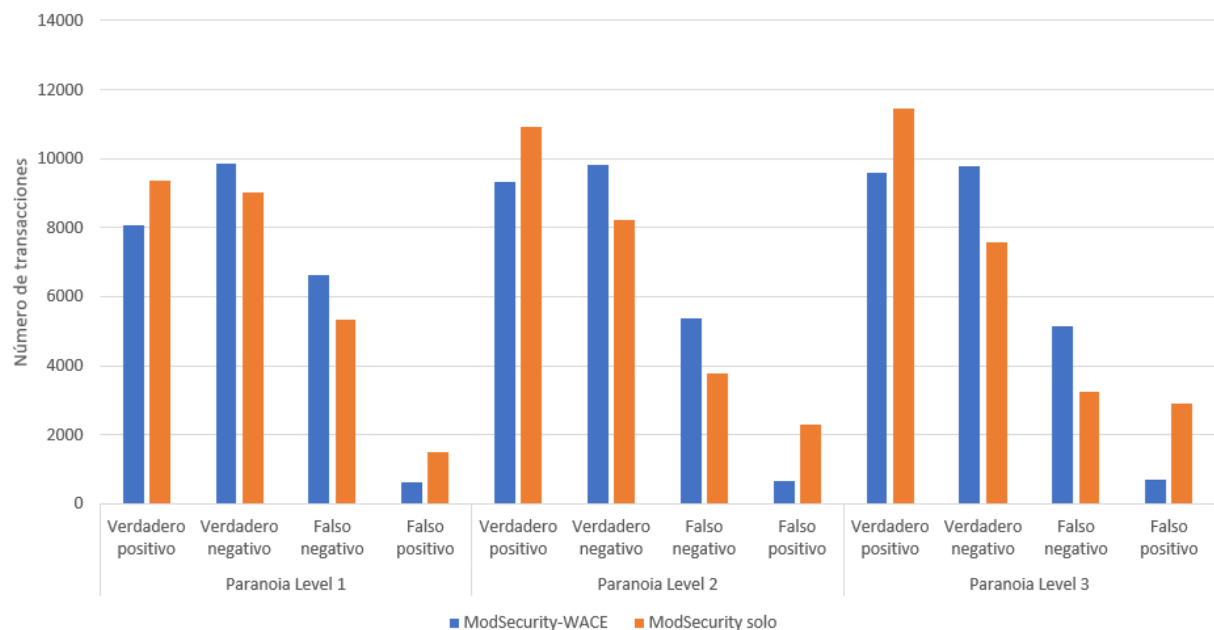

Figura 10: Resultados con Paranoia Level 1, 2 y 3

En la imagen 10 es posible observar los resultados con los niveles de paranoia 1, 2 y 3. En estos casos el uso de WACE no produce mejorías tan marcadas y prácticamente se mantienen resultados similares en los tres casos, lo más destacable es que al utilizar WACE se sigue viendo, como en el nivel 4 de paranoia, una reducción

importante de los falsos positivos. Puede que en estos casos el uso de WACE no tenga tanto impacto debido a que ModSecurity por si solo no genera tantos falsos positivos, pero a medida que se aumenta el nivel de paranoia, aumenta el beneficio de utilizar WACE.

A continuación se presenta una tabla la cual muestra los resultados obtenidos para el *accuracy, precisión y recall*:

		Acurracy	**Precision**	**Recall**
ModSecurity	paranoia level 1	71 %	92 %	54 %
	paranoia level 2	75 %	93 %	63 %
	paranoia level 3	75 %	79 %	77 %
ModSecurity con WACE	paranoia level 1	72 %	86 %	63 %
	paranoia level 2	75 %	82 %	74 %
	paranoia level 3	76 %	93 %	65 %

Cuadro 7: Desempeño ModSecurity y WACE

Como se puede apreciar en la tabla 7, el acurracy se mantiene similar en todos los casos dado que la relación de verdaderos positivos y verdaderos negativos entre el total de las transacciones no varía demasiado. Por otro lado, se puede observar que al utilizar ModSecurity con WACE y un nivel alto de paranoia (3 o 4) se gana una mayor precisión en comparación a solo utilizar ModSecurity. Por último, en el caso de solo ModSecurity es razonable pensar que al aumentar el nivel de paranoia se incremente la capacidad de detectar ataques (dentro de los que son ataques efectivamente), ya que se evalúan más reglas del CRS, mientras que ModSecurity con WACE se mantiene más estable.

6.2.3. Conclusiones de las pruebas de desempeño

El desarrollo de estas pruebas mostró ciertos resultados interesantes. En primer lugar, WACE introduce una mejora en la reducción del nivel de falsos positivos en todos los casos planteados, en algunos casos de forma más significativa (utilizando paranoia level 4) que en otros, pero siempre presentando un mejor nivel que si se utilizara ModSecurity por sí solo.
Otro aspecto que mejora WACE es la capacidad de clasificar correctamente, ya que

el valor de accuracy siempre es mayor utilizando WACE. Para las otras medidas como la precisión y recall se puede observar que para los casos de paranoia level 3, 2 y 1 con WACE se obtiene una leve mejora, mientras que utilizando el paranoia level 4, se presenta una amplia diferencia en el recall, pero esto es debido a que ModSecurity clasificó prácticamente todas las transacciones como ataque. En resumen, se obtuvieron resultados positivos que mejoran el resultado de Modsecurity por si solo. Estos resultados pueden mejorar aún más agregando nuevos plugins con modelos de aprendizaje automático mejores e implementando otros plugins de decisión.

6.3. Tiempos de procesamiento

En esta sección se analizan los tiempos de procesamiento de las transacciones para determinar el tiempo adicional que se genera con el uso de WACE.
Se midieron los tiempos en base a los siguientes escenarios:

- **WACE-ModSecurity**: WACE funcionando en conjunto con ModSecurity.
- **ModSecurity**: ModSecurity funcionando sin WACE.
- **WACE-ModSecurity (sin plugin de modelo)**: WACE funcionando en conjunto con ModSecurity pero sin el plugin de modelo, esto permite medir el tiempo que agrega WACE sin tener en cuenta el tiempo de análisis del modelo de aprendizaje automático.

El tiempo de procesamiento total obtenido en el procesamiento de 25254 transacciones de prueba en los escenarios anteriores fue de 23,14 minutos, 3,06 minutos y 5,13 minutos respectivamente.
Para visualizar mejor estos resultados a continuación se muestra una gráfica con el tiempo promedio por transacción en los tres casos:

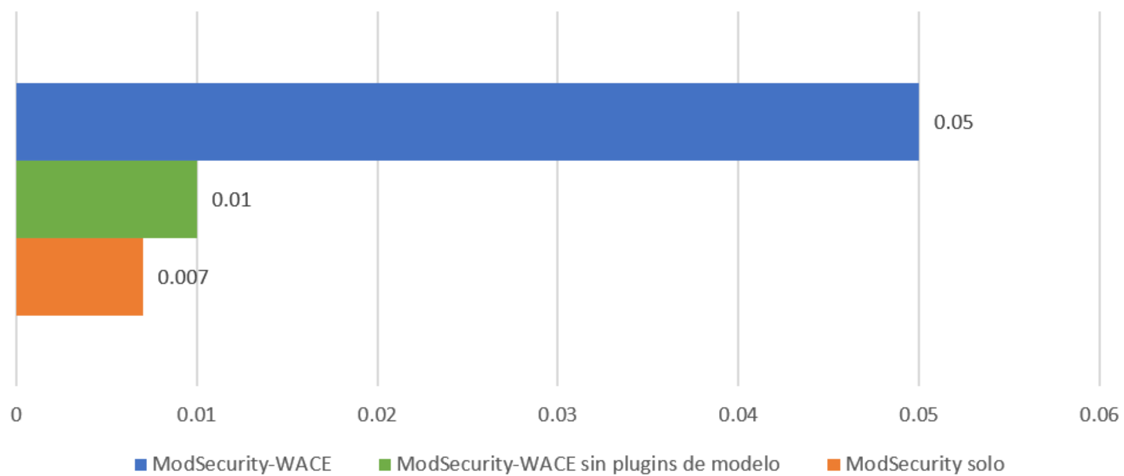

Figura 11: Tiempo de procesamiento promedio por transacción (segundos)

Como se puede observar en la gráfica, el uso de WACE introduce una demora de 4 centésimas de segundo más en promedio por transacción, esto se puede considerar como una demora relevante, pero visualizando el tiempo de WACE con ModSecurity sin el modelo de aprendizaje automático se puede ver que la mayoría del tiempo de procesamiento de la transacción se lo lleva el análisis por parte del modelo. Esto es positivo ya que introduce un punto de mejora del desempeño del sistema, utilizando una implementación más rápida del modelo se podría obtener un tiempo mejor. Por ejemplo, si se implementara en C++ el modelo y no se utilizara Openscoring en el medio, posiblemente se tendría un tiempo de procesamiento mejor.
La diferencia de media centésima de segundo entre ModSecurity por si solo y WACE sin modelos es prácticamente imperceptible, por esto se considera que la introducción de WACE al procesamiento de transacciones no produce una demora demasiado importante, inclusive con el módelo utilizado y Openscoring, cuatro centésimas de segundo para muchas aplicaciones web no afectaría su uso normal.

7. Trabajo a Futuro

Este proyecto tiene varios aspectos que podrían potenciar el uso de WACE en un trabajo futuro. A continuación se enumeran algunos de estos:

- **Análisis de response:** El prototipo implementado se limitó al análisis de las resquests, se podría extender el funcionamiento para que WACE y sus modelos analicen también las respuestas del servidor. Esto implicaría extender el protocolo de comunicación de WACE para recibir las diferentes partes de la response, se debería modificar el módulo de Apache para que obtenga los datos de la response a partir de ModSecurity y luego la API debería implementar una nueva función que permita al módulo enviar dichos datos. Esto puede parecer como mucho trabajo, pero es análogo a lo ya implementado para las requests.

- **Modelos expertos:** Como se menciona brevemente en este documento, existe la posibilidad de hacer que WACE seleccione, o brinde más peso, a los modelos de aprendizaje automático que sean más apropiados para estudiar el tipo de ataque que se presume pueda contener la transacción. Dado que WACE analiza la transacción luego que ModSecurity lo hace y este último mediante el CRS acumula el *anomaly_score* discriminado por tipo de ataque, se podría saber mediante estas variables específicas de qué tipo de ataque se trata y de esa forma indicarle a WACE que se concentre en el uso de los modelos que mejor funcionan para dicho tipo de ataque.

- **Pruebas con otros modelos:** Dado que gran parte de la demora de análisis de la transacción fue dada por el uso de OpenScoring y el procesamiento efectuado a la transacción para que el modelo pueda interpretarla, se podría implementar otro modelo de aprendizaje automático que utilice otra herramienta y así obtener mejores resultados.

- **Protocolo de comunicación:** El protocolo de comunicación diseñado es suficiente para la solución implementada, pero en caso que se deseen ampliar las funcionalidades de la herramienta podría existir la necesidad de que deba ser mejorado, por ejemplo, implementando mecanismos para manejar la corrupción o perdida de datos en la comunicación, estableciendo formas de

recuperación ante una pérdida de la conexión, u otros aspectos que mejoren la robustez del protocolo.

- **Criterios de decisión:** Existen muchos trabajos sobre sistemas con múltiples clasificadores[17], por lo tanto una posible mejora al sistema, basándose en algunos de estos trabajos, seria implementar otros algoritmos de decisión que permitan combinar de mejor manera los resultados de ModSecurity y los diferentes modelos de aprendizaje automático.

8. Conclusiones

El principal objetivo que aborda este trabajo es diseñar e implementar un prototipo que permita experimentar con la integración de modelos de aprendizaje automático y el WAF ModSecurity.
En primer lugar, se determinó el esquema a utilizar para el desarrollo del prototipo, dicha elección se considera acertada ya que permitió desarrollar una solución elegante y simple de implementar.
En cuanto al diseño, WACE se destaca por su arquitectura de plugins, la cual permitió que sea un sistema sumamente flexible, ya que mediante plugins se permite cambiar el algoritmo de decisión, así como también añadir o quitar modelos de aprendizaje automático al sistema.
Además, la arquitectura utilizada permite el desarrollo de nuevos plugins que mejoren el funcionamiento de WACE, ya sea mejorando el algoritmo de decisión con nuevos plugins de decisión o implementando nuevos plugins de modelo que se comuniquen con nuevos modelos de aprendizaje automático.
Otro aspecto importante que aporta a la flexibilidad de WACE es la posibilidad de configurar distintos parámetros que ayudan a la toma de decisión, como por ejemplo el peso por modelo.
Con respecto a la implementación, fue positivo el uso del mecanismo de funciones auxiliares de Apache debido a que facilitó la integración con ModSecurity mediante la implementación de un operador, lo cual redujo el problema de integración al desarrollo de un módulo de Apache. Este mecanismo fue realizado para integrar a WACE con ModSecurity 2, al momento de la realización del prototipo se encontró que ModSecurity 3 no estaba lo suficientemente maduro para ser utilizado. Debido al diseño concebido, esto no es un problema, ya que a futuro puede ser integrado ModSecurity 3 con WACE sin necesidad de realizar ninguna modificación sobre WACE.
En linea con lo anterior, se destaca la definición de un protocolo de comunicación con WACE, el cual no solo permite utilizar a WACE con ModSecurity, sino que también con cualquier sistema que se comunique utilizando el protocolo definido.
Por último, luego de realizar varias pruebas del uso de WACE en diferentes escenarios se concluye que se logró reducir la cantidad de falsos positivos generados por ModSecurity. Un punto a mejorar es el tiempo extra que genera WACE con

esta implementación, de todos modos dada la arquitectura de WACE esto puede ser mejorado.
A nivel general, entre las dificultades y desafíos que se encontraron en el transcurso del proyecto, se destaca la carencia de modelos de aprendizaje automáticos disponibles para realizar más pruebas. El modelo que se disponía necesitaba mucho preprocesamiento de las transacciones, lo que llevó a aumentar un poco el tiempo de procesamiento de cada transacción.
Resumiendo, se concluye que el trabajo realizado cumplió el objetivo, se logró implementar una herramienta que integra a ModSecurity modelos de aprendizaje automático y se obtuvo una mejora de los resultados de ModSecurity en un porcentaje no despreciable, en particular en lo que respecta a la reducción de falsos positivos con niveles de paranoia altos.

Referencias

[1] Outeda F., Cuttica E. - ***WACE: Un integrador de clasificadores de ataques web*** - Estado del arte. Proyecto de grado, Instituto de Computación, Facultad de Ingeniería, Universidad de la República.

[2] ***Apache 2.0 License*** `https://www.apache.org/licenses/LICENSE-2.0.html` - Último acceso: Junio 2021.

[3] Zhang M., Xu B. & Lu S. (2015) - ***An Intelligent Framework to Detect Network Intrusion***, National Key Laboratory of Science and Technology on Information System Security Beijing, China & Information Engineering University Zhengzhou, Henan Province, China.

[4] Folini C. (2016) - ***Handling false positives with the owasp modsecurity core rule set.*** - `https://www.netnea.com/cms/apache-tutorial-8_handling-false-positives-modsecurity-core-rule-set/` - Último acceso: Junio 2021.

[5] Betarte G., Martínez R. & Pardo A. (2018) - ***Web Application Attacks Detection Using Machine Learning Techniques.***, 17th IEEE International Conference on Machine Learning and Applications (ICMLA), 2018, pp. 1065-1072.

[6] Montes N., Betarte G., Martínez R., Pardo A. (2021) - ***Web Application Attacks Detection using Deep Learning***, CIARP 25.

[7] D. Appelt, C. D. Nguyen, A. Panichella and L. C. Briand,- ***A Machine-Learning-Driven Evolutionary Approach for Testing Web Application Firewalls***, in IEEE Transactions on Reliability, vol. 67, no. 3, pp. 733-757, Sept. 2018, doi: 10.1109/TR.2018.2805763.

[8] ***Flask*** - `https://palletsprojects.com/p/flask/` - Último acceso: Junio 2021.

[9] ***Flask-RESTFul API*** - `https://flask-restful.readthedocs.io/en/latest/` - Último acceso: Junio 2021.

[10] ***TensorFlow Serving*** - `https://www.tensorflow.org/tfx/guide/serving` - Último acceso: Junio 2021.

[11] Ingold S. - ***Alternativas e implementación de prueba de concepto para servir modelos de Machine Learning a ModSecurity***

[12] ***Modelo de aprendizaje automático en PMML*** - `https://gitlab.fing.edu.uy/gsi/waf-wace/-/blob/master/Modelo%20PMML/Modelo.pmml` - Último acceso: Junio 2021.

[13] ***Módulos de apache con ModSecurity*** - `https://github.com/SpiderLabs/ModSecurity/blob/v2/master/ext/README` - Último acceso: Junio 2021.

[14] Kew N. (2007) - ***The Apache Modules Book***, Prentince Hall.
ISBN-13: 978-0132409674 ISBN-10: 0132409674

[15] Richards M. - ***Software Architecture Patterns***, O'REILLY - `https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch03.html` - Último acceso: Junio 2021.

[16] Dietterich T.G. (2000) - ***Ensemble Methods in Machine Learning. In: Multiple Classifier Systems***, MCS 2000. Lecture Notes in Computer Science, vol 1857. Springer, Berlin, Heidelberg. `https://doi.org/10.1007/3-540-45014-9_1`

[17] ***MCS - International Workshop on Multiple Classifier Systems*** - `https://link.springer.com/conference/mcs` - Último acceso: Junio 2021.

[18] ***deamon Linux manual page*** - `https://man7.org/linux/man-pages/man7/daemon.7.html` - Último acceso: Junio 2021.

[19] S.B. Kostsiantis (2007) - ***Supervised Machine Learning: A Review Of Classification Techniques***, Pag. 17 Sec. 7.3, IOS Press (2007).

[20] Shapley Gray J. (2003) - ***Interprocess Communications in Linux: The Nooks & Crannies***, Prentice Hall.

[21] Ristic I. (2010) - ***ModSecurity Handbook***, Pag. 66 Capt. 5, Feisty Duck. ISBN-13: 978-1907117077 ISBN-10: 1907117075

[22] ***SQL Injection*** - `https://owasp.org/www-community/attacks/SQL_Injection` - Último acceso: Junio 2021.

[23] ***ModSecurity*** - `https://github.com/SpiderLabs/ModSecurity` - Último acceso: Junio 2021.

[24] ***ModSecurity versión 2*** - `https://github.com/SpiderLabs/ModSecurity/tree/v2/master` - Último acceso: Junio 2021.

[25] ***Openscoring*** - `https://github.com/openscoring/openscoring` - Último acceso: Junio 2021.

[26] ***Core Rule Set*** - `https://owasp.org/www-project-modsecurity-core-rule-set/` - Último acceso: Junio 2021.

[27] KristenS - ***Cross Site Scripting (XSS)*** - `https://owasp.org/www-community/attacks/xss/` - Último acceso: Junio 2021.

[28] ***XPATH Injection*** - `https://owasp.org/www-community/attacks/XPATH_Injection` - Último acceso: Junio 2021.

[29] ***LDAP Injection*** - `https://owasp.org/www-community/attacks/LDAP_Injection`

[30] Weilin Zhong, Nsrav - ***Server Side Includes Injection (SSI)*** - `https://owasp.org/www-community/attacks/Server-Side_Includes_(SSI)_Injection` - Último acceso: Junio 2021.

[31] KristenS - ***Path trasversal*** - `https://owasp.org/www-community/attacks/Path_Traversal` - Último acceso: Junio 2021.

[32] Weilin Zhong, Rezos - ***Code Injection*** `https://owasp.org/www-community/attacks/Code_Injection` - Último acceso: Junio 2021.

[33] ***Unix socket*** - `https://www.man7.org/linux/man-pages/man7/unix.7.html` - Último acceso: Junio 2021.

[34] ***Analyzing Web Traffic ECML/PKDD 2007 Discovery Challenge*** `https://www.lirmm.fr/pkdd2007-challenge/index.html` - Último acceso: Junio 2021.

[35] ***Anomaly Scoring Mode*** `https://coreruleset.org/docs/anomaly.html` - Último acceso: Junio 2021.

9. Apéndices

9.1. Instalación y despliegue

9.1.1. Instalación de WACE

Para instalar WACE, simplemente se debe ejecutar el script install.sh que se encuentra dentro del directorio *wace_core* mediante el comando sudo *sh ./wace_core/install.sh* Este script se encarga de compilar la aplicación y poner los archivos en los lugares correctos. En el siguiente listado puede ver los archivos que componen la herramienta.

- **/usr/bin/wace**: El archivo binario de la aplicación.
- **/etc/wace/wace.conf**: El archivo de configuración de WACE
- **/etc/systemd/system/wace.service**: El archivo de configuración del servicio *wace*.
- **/etc/wace/plugins/*.so**: El directorio que contienen todos los plugins de modelo y decisión.
- **/var/lib/wace/wace.socket**: El filesocket para la comunicación con WACE.
- **/var/log/wace/wace_core.log**: El archivo de logs de la herramienta.

9.1.2. Elementos adicionales a WACE

Para que el sistema de análisis de transacciones funcione se requieren de otros elementos además de WACE.

ModSecurity
WACE esta desarrollado para funcionar con ModSecurity versión 2, por ello es necesario contar con una instalación de ModSecurity en dicha versión. Además de ModSecurity, se puede utilizar el Core Rule Set en su ultima versión configurándolo para funcionar en el modo de detección de anomalías y realizando una modificación en uno de sus archivos de reglas para agregar a WACE al procesamiento.

Modulo de Apache

Para conectar a WACE con ModSecurity se implementa un operador llamado *wace* el cual permite que la transacción que procesa ModSecurity pueda ser accedida por WACE. Para compilar e instalar el módulo Apache que implementa este operador se debe ejecutar el comando make dentro del directorio apache_module. Es importante contar con la librería lib_apiwace.so instalada en el sistema, si ésta no se encuentra se puede ejecutar el comando make en el directorio lib_apiwace y se compilará e instalará la librería.

Configuración del CRS

Para que WACE reciba la información de la transacción se debe agregar entre las reglas al operador *wace*. El CRS funcionando en modo de detección de anomalías toma la decisión de bloqueo o no de la transacción en el archivo REQUEST-949-BLOCKING-EVALUATION particularmente la regla que analiza si se supero el *anomaly threshold* es la 949110 esta debe ser reemplazada por una regla que utilice de operador a wace, tome como variable al ANOMALY_SCORE y reciba como argumento del operador al anomaly_score_threshold. Por ejemplo la regla podría ser algo como:

```
SecRule TX:ANOMALY\_SCORE "@wace %{tx.inbound_anomaly_score_threshold}"\
    "id:959110,\
    phase:2,\
    deny,\
    t:none,\
    msg:'Transaction blocked by WACE (Total Anomaly Score: %{TX.
        ANOMALY_SCORE})',\
    severity:'CRITICAL'"
```

Con esta regla y todos los elementos mencionados anteriormente instalados, WACE esta listo para ser utilizado.

9.2. Función de inicialización

La función de inicialización es necesario para que los plugins se registren en el sistema, la misma debe tener el siguiente formato:

```
int init_<nombre_del_plugin>(PluginManager pm) {
    registerModelPluginFunction(pm, checkTransaction);
    return 0;
}
```

El nombre de la función debe ser *init_* seguido por el nombre que tenga el plugin. La función debe retornar un entero el cual debe ser 0 en caso que la función se ejecute exitosamente, o cualquier otro valor en caso de error. Como parámetro se recibe una instancia de *PluginManager* y dentro de la función se llama a la función implementada por WACE *registerModelPluginFunction* la cual recibe como primer parámetro la instancia del *PluginManager* y como segundo parámetro el nombre de la función que implementa el plugin para analizar la transacción, en el ejemplo, *checkTransaction*. En este ejemplo se muestra la función de un plugin de modelo, en el caso que se trate de un plugin de decisión, la función que debe ser invocada dentro de la función de inicialización es *resgisterDecisionPluginFunction* y toma los mismos parámetros con la diferencia que el segundo parámetro indica el nombre de la función que implementa la decisión.

9.3. Plugin de decisión

A continuación se da un poco más de detalle sobre el algoritmo utilizado en el plugin de decisión.
En primer lugar, se cuentan cuantos plugins detectaron un ataque en la transacción, para esto se suma uno cuando la probabilidad de que la misma haya sido un ataque es mayor o igual al umbral definido para el plugin. Si la mayoría de los plugins detectaron un ataque, entonces la transacción es considerada maliciosa.

```
//attack count
for(pair<string,float> plugin_res : di->results){
    if (plugin_res.second >= di->threshold_models[plugin_res.first]){ //
        compare with the config plugin thresh
```

```
        cant_attacks++;
    }
}
//if the majority of the models detects an attack
if (cant_attacks > di->cant_plugins/2){
    models_clasification = 1;
}
```

En segundo lugar, se determina si ModSecurity detectó un ataque en la misma transacción, para esto se compara el valor del puntaje de anomalía con el umbral de ModSecurity. Si el puntaje de anomalía es mayor al umbral, la transacción es considerada maliciosa.

```
//si el anomaly score supera el umbral de modsec, este ultimo retorna 1
if (di->anomaly_score > di->threshold_modsecurity){
    modsecurity_clasification = 1;
}
```

Por último, tanto el *model_clasification* como el *modsecurity_clasification* se multiplica cada uno por un peso, el peso es un valor entre 0 y 1 configurable por el usuario. El valor de los pesos se utiliza para establecer un balance entre ModSecurity y los modelos de aprendizaje automático, es decir ofrecer la posibilidad de dar más importancia a la hora de la decisión a uno u al otro.