

Universidad  
Rey Juan Carlos

Escuela Técnica Superior  
de Ingeniería Informática

Grado en Ingeniería de la Ciberseguridad  
Curso 2023-2024

Trabajo Fin de Grado

**AUTOMATIZACIÓN DE LA GESTIÓN DE LA  
SEGURIDAD *CLOUD NATIVE* A TRAVÉS DE  
GENERADORES NLP**

Autor: Carlos Eloy Sánchez Moreno  
Tutor: Alfonso de Jesús Pérez Martínez  
Cotutor: David de la Hoz Alías





# Resumen

Uno de los problemas recurrentes de asegurar las aplicaciones basadas en microservicios utilizando un Sistema de Detección de Intrusos (IDS) es mantener sus reglas actualizadas, especialmente si aparecen nuevas técnicas y sistemas de ataque de forma constante en los sistemas de las organizaciones. Gracias a los desarrollos de los algoritmos de Aprendizaje Automático es posible aplicar modelos de procesamiento del lenguaje natural (NLP) para automatizar los sistemas de seguridad programática modernos, como Falco.

En esta investigación, discutiremos la aplicación de modelos NLP en aplicaciones basadas en microservicios respaldadas por un IDS para automatizar el proceso de creación de reglas, e investigar su factibilidad en un entorno de producción. Para el desarrollo de esta investigación, crearemos un generador de reglas para Falco, un sistema de detección de eventos apoyado por la Cloud Native Computing Foundation (CNCF) y su comunidad, usando la descripción de las propias reglas para entrenar estos modelos, que creará reglas basadas en descripciones de técnicas de ataque MITRE.

## Palabras clave:

- Python
- Cloud Native
- Ciberseguridad
- Aprendizaje automático
- NLP
- Falco



# Índice de contenidos

<b>Índice de cuadros</b>	<b>VIII</b>
<b>Índice de figuras</b>	<b>1</b>
<b>1. Introducción</b>	<b>3</b>
1.1. Estructura del trabajo . . . . .	3
1.2. Contexto . . . . .	4
1.3. Objetivos y alcance . . . . .	6
<b>2. Desarrollo del Trabajo</b>	<b>8</b>
2.1. Diseño de la Aplicación . . . . .	9
2.2. Fase 1: Recopilación de reglas . . . . .	12
2.3. Fase 2: Afinamiento . . . . .	13
2.4. Fase 3: Experimentación . . . . .	14
<b>3. Análisis de los resultados</b>	<b>18</b>
3.1. Prueba Manual . . . . .	18
3.2. Pruebas Automáticas . . . . .	21
3.3. Consumo de recursos . . . . .	22
<b>4. Conclusiones</b>	<b>23</b>
4.1. Limitaciones y Riesgos de Seguridad y privacidad . . . . .	24
4.2. Trabajos Futuros . . . . .	26
<b>Bibliografía</b>	<b>26</b>
<b>Apéndices</b>	<b>28</b>
<b>A. Apéndice 1: Prompt de generación de reglas</b>	<b>31</b>
<b>B. Apéndice 2: Fuentes de origen de las reglas recopiladas</b>	<b>34</b>
<b>C. Apéndice 3: Glosario</b>	<b>35</b>



# Índice de cuadros

2.1. Descripción de las columnas del Dataset . . . . .	12
--	----





# Índice de figuras

1.1. Cronograma de evolución de los NLP . . . . .	6
2.1. Cadena de proceso de la aplicación . . . . .	11
2.2. Esquemas de afinamiento del modelo . . . . .	14
3.1. Tasa de éxito de las distintas versiones . . . . .	21
3.2. Consumo de memoria de vídeo por entrenamiento . . . . .	22
3.3. Evolución del consumo de la memoria de vídeo . . . . .	22



# 1

## Introducción

### 1.1. Estructura del trabajo

Este trabajo se divide en cuatro capítulos principales y tres apéndices. En el primer capítulo repasaremos la historia de los últimos avances sobre el campo de los NLP y hemos explicado por qué es importante investigar sus aplicaciones para avanzar hacia la automatización de los procesos de seguridad y cuáles son los objetivos y alcance de este trabajo

En el segundo capítulo, explicaremos cómo hemos desarrollado el trabajo y los experimentos que hemos realizado con los generadores de reglas que hemos creado para comprobar su efectividad, en el tercer capítulo ofreceremos los resultados obtenidos y los analizaremos. Finalmente, en el último capítulo y explicaremos las limitaciones detectadas en este trabajo, planteamos las cuestiones de seguridad en relación a este trabajo y qué avances se pueden realizar para mejorarlo.

Además, este documento dispone de tres apéndices: en el primero ofrecemos las instrucciones del generador con LLM, cuya ejecución se realiza *online*, en el segundo ofrecemos las fuentes de los repertorios de reglas recopiladas, y en el último ofrecemos un glosario de términos que son necesarios de entender para comprender este trabajo.

Por último, este trabajo viene acompañado con un documento aparte con el código de la aplicación ejecutada. Así, este trabajo queda huérfano (incompleto) si no se dispone de este documento.

## 1.2. Contexto

El procesamiento del lenguaje natural, o NLP, es el campo que investiga la interpretación de mensajes comprensibles para humanos con el fin de generar un resultado deseado. El objetivo de un NLP puede ser alguna de estas tareas[1]:

- Generación de texto en lenguaje natural (text-to-text)
  
- Predicción de palabras (mask prediction)
  
- Clasificación de texto
  
- Extracción de respuestas y realización de resúmenes
  
- Traducción de texto
  
- Cálculo de la entropía

Es posible introducir en NLP otros campos si se considera otro tipo de entradas, como por ejemplo visión artificial (reconocimiento visual de texto o descripción de una imagen), o reconocimiento de audio (transcripción oral), pero el tratamiento habitual suele ser realizado con texto.

Si tomamos el texto como entrada, el procesamiento del lenguaje se realiza mediante la *tokenización*, que es el proceso de asignar identificadores numéricos arbitrarios (o tokens) a cada palabra. Así, dependiendo de la tarea realizada, se usa un modelo que a su vez utiliza un algoritmo que procesará el conjunto de tokens como un vector para generar una salida, que puede ser satisfactoria o no. Usando algoritmos de aprendizaje automático, se puede aprovechar la salida para calcular un error, y tomando el error como referencia a minimizar, se ajustan los pesos del modelo para reducir dicho error, obteniendo al final un resultado que se acerque a lo deseado.

### 1.2.1. Revisión histórica de los NLP

Antes de continuar, realizaremos una breve revisión histórica sobre este campo. El procesamiento del lenguaje natural comienza en los años 30 con el desarrollo de la primera máquina traductora de texto, creada por Georges Artsrouni, mediante el mapeado de dos lenguas distintas[2], y posteriormente mejorado gracias al uso de estructuras sintácticas (Noam Chomsky, 1957).

Si consideramos todo proceso de criptoanálisis y descifrado de texto como un NLP, el desarrollo de este campo podría remontarse a la Antigua Roma (Cifrado César)[3]. En los años 60 y 70 se hacen los desarrollos más significativos, con la invención del concepto de 'tokens' (Roger Schank, 1969) y de la Red de Transición Aumentada, ATN (William Woods, 1970), que permite representar los distintos componentes de una oración como estados de un grafo. Entre los años 80 y 90 se empieza a trabajar en el desarrollo de sistemas OCR con Aprendizaje Automático para el reconocimiento de caracteres manuscritos: en 1983[4] se propone el uso de umbrales, y en 1992[5] se propone por primera vez el uso de redes neuronales para esta tarea, demostrando lo crucial que son los algoritmos de aprendizaje máquina para el procesamiento del lenguaje natural.

Con la demostración de la efectividad del modelo de transformadores (Attention is All you Need. Google, 2017)[6], junto a los desarrollos en el procesamiento de grandes cantidades de información y las mejoras del hardware, se ha posibilitado la implementación de aplicaciones NLP más complejas y con un mayor número de parámetros, alcanzando por primera vez los mil millones y medio con el modelo GPT-2 (OpenAI, 2019).

Hoy en día, la aplicación de los NLP está en auge, existe modelos preentrenados multimodales (tanto propietarios como de código abierto) y permite la automatización de aquellos procesos en donde se requiera interacción humana (aquellos en donde haya tareas de dialogar, comprender o traducir información comprensible por un humano), lo que generaría ahorros significativos en el campo del DevOps y el mantenimiento de la ciberseguridad.

Así en general, según un sondeo de Deloitte, el 82% de las compañías encuestadas han afirmado observar retornos en sus inversiones en IA, y aquellas compañías del sector de la tecnología, telecomunicaciones y entretenimiento observaron un ROI de hasta el 20% [7]

Los últimos modelos más importantes hasta la fecha son ChatGPT-4o (OpenAI, 2024), Meta Llama 3 (Meta, 2024) y Mixtral (Mistral AI, 2024). Todos estos son modelos LLM ya preentrenados, y se espera en el futuro que las nuevas versiones sean más efectivas que las actuales.

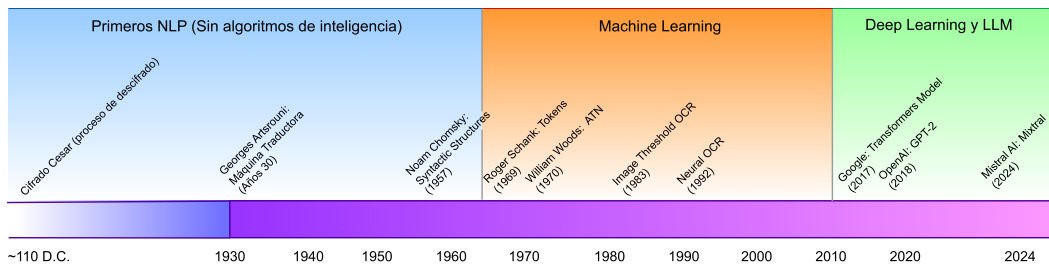


Figura 1.1: Cronograma de evolución de los NLP

### 1.3. Objetivos y alcance

El objetivo principal de esta investigación es analizar si es factible utilizar algoritmos de IA para automatizar el proceso de gestión de generación de reglas de detección de amenazas en entornos en la Nube. Esta tarea requiere de intervención humana puesto que hay que desarrollar la automatización comprendiendo la técnica o táctica detectadas.

Así, hemos decidido investigar la automatización de generación de reglas para Falco, que es un sistema IDS que detecta comportamientos inusuales dentro de máquinas en la nube. Hemos elegido este componente porque es de código abierto, cuenta con el apoyo de la CNCF y es fundamental para la seguridad en entornos de nube.

Para esto, esperamos que las reglas generadas por la aplicación generador de reglas cumplan, como pauta general, con los siguientes requisitos:

1. Las reglas generadas deben ajustarse a la sintaxis de reglas establecida por Falco.
2. El vocabulario usado en las reglas generadas por la aplicación deben existir en el vocabulario nativo empleado por Falco
3. Las reglas generadas deben ajustarse a la descripción de entrada ofrecida por las técnicas y tácticas recopiladas

4. Las reglas generadas sean eficaces a la hora de detectar aquellas anomalías y actividades descritos por dichas técnicas y tácticas
5. La cantidad de recursos computacionales consumidos por la aplicación permita poder ejecutar en entornos de producción con tarjetas gráficas estándar

Sin embargo, no se espera que las reglas generadas por el generador sean eficientes en tamaño o eficaces para detectar ataques, dado que la descripción de la regla no tiene por qué ser precisa, por lo que no se considera ninguna métrica que mida la eficiencia o eficacia de la regla.

En cuanto al alcance, nuestra investigación cubrirá el proceso de creación de un generador de un generador de reglas, primero a través del afinamiento de modelos ya existentes y luego mediante el uso del último LLM más potente disponible hasta la fecha.



# 2

## Desarrollo del Trabajo

El proceso de generación de reglas es un proceso cíclico, y consiste en los siguientes pasos:

1. **Aprendizaje.** Primero, usando el repertorio de reglas como punto de partida, se entiende la estructura de las reglas Falco y se aprenden los patrones que hacen que estas reglas sean eficaces.
2. **Recopilación de información.** Se informa de las últimas amenazas detectadas, enfocándose en aquellas que conciernen a la organización, y a través del uso de procesos de inteligencia se extraen las técnicas y tácticas utilizadas.
3. **Descripción de la forma de detección.** A partir de la información recopilada, se describen aquellas anomalías que permitirían detectar estas técnicas y tácticas.
4. **Traducción a reglas.** A partir de la descripción sobre cómo detectar dichas técnicas y tácticas, se traduce esta descripción a reglas Falco, que utilizan su propia sintaxis.
5. **Evaluación de las reglas.** Finalmente, se evalúa la eficacia de estas reglas a través de ciertas métricas, de forma que las reglas buenas son utilizadas para alimentar el repertorio de reglas, retroalimentando el proceso.

De los pasos descritos, nos interesa especialmente el proceso de aprendizaje, la traducción de las descripciones a reglas y la evaluación de los resultados. Así, en este capítulo explicaremos como hemos desarrollado la aplicación para crear el generador de reglas Falco, y explicaremos también los experimentos que hemos realizado para analizar las reglas obtenidas por este generador.

### 2.1. Diseño de la Aplicación

Para crear el generador de reglas usando modelos de lenguaje, no necesitamos diseñar el modelo desde cero: basta con afinar modelos de lenguaje ya existentes mediante un entrenador, que utiliza un dataset para ajustar los pesos de los parámetros de cada modelo.

Así, para investigar cuál es la mejor forma de crear el generador de reglas, planteamos 3 distintas versiones del generador de reglas, que se diferencian por el tipo de entrenamiento usado y el dataset empleado, más una adicional en donde se usará un LLM preentrenado:

#### 2.1.1. Versión 1

Esta versión está formada por varios modelos Seq2Seq y emplea un entrenamiento supervisado. El dataset está disgregado en distintos campos específicos. La evaluación de la precisión se realiza utilizando la métrica ROUGE.

Ventajas: La principal ventaja es la mayor exactitud en la salida, ya que se indica explícitamente qué es lo que debe generar el modelo.

Desventajas: La complejidad del dataset es mayor, ya que está dividido en múltiples campos, y la versión depende de otros modelos para otros campos, lo que puede dificultar la integración y el mantenimiento.

#### 2.1.2. Versión 2

La segunda versión está compuesta por varios modelos Decoder y se entrena de manera no supervisada. El dataset utilizado consiste en la concatenación de una descripción y otro atributo, separados por dos puntos. La métrica utilizada para la evaluación es la perplejidad.

Ventajas: Entre las ventajas se encuentran una comprensión del lenguaje más sólida, un dataset más simple al tener una sola columna, y un entrenamiento más flexible.

Desventajas: A pesar de sus ventajas, esta versión usa múltiples modelos, lo que puede aumentar la complejidad. Además, tiene una mayor probabilidad de generar alucinaciones y contenido irrelevante.

### 2.1.3. Versión 3

En la tercera versión, se utiliza un único modelo Decoder, compartiendo el mismo entrenador y métrica que la versión 2. El dataset está formado exclusivamente por las reglas Falco en crudo.

Ventajas: Esta versión tiene la ventaja de que el modelo genera todos los campos, además de compartir las ventajas de la versión 2, como la comprensión sólida del lenguaje y un dataset más simple.

Desventajas: La desventaja principal radica en que la entrada de entrenamiento y el dataset son demasiado grandes en contenido, lo que puede dificultar la especialización y comprensión de los campos específicos por parte del modelo.

### 2.1.4. Generador LLM

El Generador LLM es un modelo al que se le suministra una serie de instrucciones para generar una regla Falco. El prompt está predefinido y la comunicación con el modelo se realiza de forma **online** mediante una API.

Ventajas: No necesita afinamiento y no consume recursos locales, ya que se depende de una API externa.

Desventajas: Su desventaja principal es que depende de una conexión a Internet y de la calidad del servicio proporcionado. Además, el control sobre el modelo es limitado, lo que puede afectar la personalización y la precisión en tareas específicas.

### 2.1.5. Modelos adicionales

Para generar el nombre de las reglas, todas las versiones dependen de un generador de nombres, que no es más que un resumidor de texto, mientras que las versiones 1 y 2 del generador de reglas tienen que apoyarse en otros modelos para generar el campo 'output' y clasificar la prioridad de las reglas generadas en función de la descripción.

El modelo clasificador por prioridad tiene su propio entrenador, utiliza la métrica de exactitud de predicción, y ofrece como salida una etiqueta de las 7 pre-definidas EMERGENCY, ALERT, CRITICAL, ERROR, WARNING, NOTICE, INFO, DEBUG

Más adelante, en la fase de experimentación, explicamos que hemos entrenado un modelo extractor de semántica. Este modelo traduce la condición de una regla a una descripción, y utiliza el mismo entrenador que el de la versión 1.

### 2.1.6. Estructuración

La creación de las versiones 1, 2 y 3 se lleva a cabo en 3 fases:

1. **Fase 1:** Obtención de información: recopilamos las reglas para crear datasets de entrenamiento.
2. **Fase 2:** Afinamiento: los modelos de lenguaje vírgenes son reentrenados para ser capaces de generar reglas Falco
3. **Fase 3:** Evaluación: los generadores de reglas son evaluados con distintas pruebas y métricas para determinar cuál es el mejor modelo generador

El siguiente diagrama explica cómo se relacionan las diferentes fases:

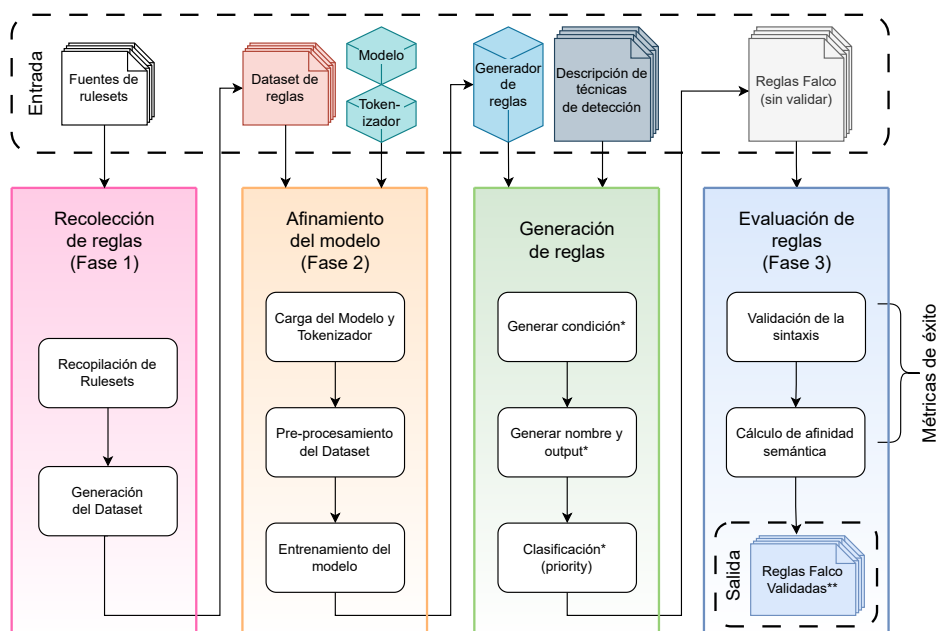


Figura 2.1: Cadena de proceso de la aplicación

## 2.2. Fase 1: Recopilación de reglas

Para seguir este trabajo, necesitamos entender qué es una regla Falco, cómo se estructura el repertorio de reglas y qué es lo que vamos a usar para entrenar las distintas versiones.

### 2.2.1. Estructura de una regla

Una regla Falco es un objeto nombrado estructurado en YAML, que debe disponer de los siguientes 5 atributos obligatorios[8]:

Variable	Valor
Nombre	Texto Arbitrario. Identifica la regla
Descripción	Explicación de alto nivel sobre esta regla. Texto arbitrario
Condición	Lógica por el que se activa esta regla. Sigue una sintaxis propia
Prioridad	Determina la importancia de esta regla. Sus valores pueden ser: {EMERGENCY, ALERT, CRITICAL, ERROR, WARNING, NOTICE, INFORMATIONAL, DEBUG}
Salida	Texto de notificación que emitirá Falco cuando la regla sea activada. Sigue una sintaxis propia

Cuadro 2.1: Descripción de las columnas del Dataset

Estas reglas están recogidas dentro de un repertorio de reglas, o *ruleset*, un documento YAML que contiene los siguientes tipos de objeto:

- **Reglas:** elemento nombrado con una condición definida que indica la acción que debe tomar Falco cuando la condición ocurra. Por defecto, Falco informa de la ocurrencia.
- **Macros:** fragmentos de reglas que pueden ser reutilizados en cualquier regla definido en el repertorio y que son llamados con una palabra clave
- **Listas:** conjunto de objetos utilizados por cualquier regla o macro para referenciar a dichos objetos con una sola palabra clave

Para simplificar el trabajo, solo hemos extraído las reglas de cada repertorio, y de cada regla solo recogeremos los cinco atributos obligatorios. Para simplificar el vocabulario, es posible expandir las reglas sustituyendo los identificadores de listas y macros por el contenido de dichas listas y macros, con el coste de incrementar el tamaño de la información de entrenamiento.

En cuanto a los repertorios de reglas, hemos conseguido extraer 282 reglas para el entrenamiento, extraídos de los diferentes rulesets que hemos recopilado de

6 fuentes abiertas, todas ellas procedentes de Github. Las fuentes utilizadas están descritas en el segundo apéndice.

### 2.2.2. Diseño de los datasets

Para cada versión del generador de reglas y otros modelos creados hemos diseñado 3 datasets con contenidos y estructuración distintas:

Estos datasets a su vez están divididas en subconjuntos (subsets), que son usados para distintos propósitos:

- **Subset de entrenamiento:** son las muestras que usará el entrenador para afinar el modelo. Corresponde a un 80 % del dataset original
- **Subset de validación:** contiene las muestras que usará el entrenador para validar el entrenamiento. Corresponde a un 10 % del dataset
- **Subset de pruebas:** Son las muestras que se emplea para probar la salida del modelo ya afinado. Corresponde al último 10 % del dataset

Tanto el subset de entrenamiento como el de validación son usados durante el entrenamiento para entrenar y validar que el entrenamiento es adecuado, mientras que el subset de pruebas es usado después del entrenamiento para poder realizar pruebas de calidad posteriores, como veremos en la próxima sección.

## 2.3. Fase 2: Afinamiento

Para afinar los modelos de cada versión mediante entrenamiento, necesitamos implementar tres entrenamientos distintos:

- **Entrenador Supervisado (V1, extractor semántica):** Se usa el modelo `google-t5/t5-small`, un modelo encoder-decoder, para tareas de extracción semántica. La métrica utilizada es ROUGE, que mide la calidad del resumen generado comparando con textos de referencia.
- **Entrenador No Supervisado (V2, V3):** Se emplea el modelo GPT2, un modelo decoder, para entrenamiento no supervisado. La métrica utilizada es la perplejidad, que evalúa la capacidad del modelo para predecir una muestra.
- **Entrenador para el Clasificador de Prioridad:** Es un entrenador no supervisado que usa el modelo `distilbert-base-uncased`, un modelo decoder, para la clasificación de prioridad. La métrica utilizada es accuracy, que mide la precisión del clasificador.

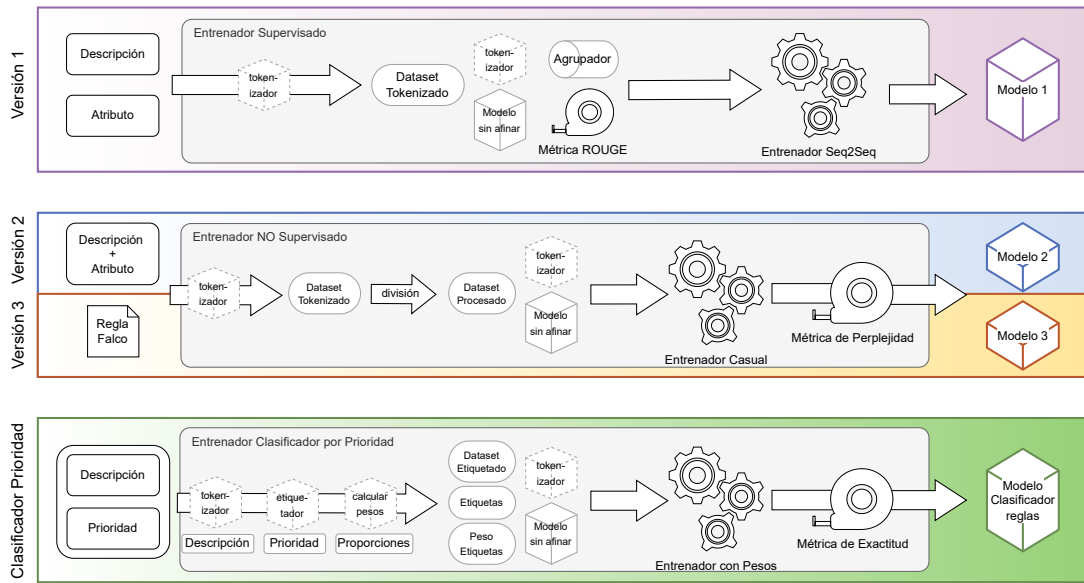


Figura 2.2: Esquemas de afinamiento del modelo

## 2.4. Fase 3: Experimentación

Para probar la eficacia de los modelos que hemos entrenado, hemos diseñado dos pruebas:

- Prueba manual:** dado una descripción de prueba que no se corresponde a la descripción de ninguna regla, las distintas versiones de generador de reglas generarán una regla a partir de la descripción y se imprimirá en pantalla. Un evaluador humano juzgará si la regla es válida y es afín a la descripción
- Pruebas Automáticas:** usando una muestra de prueba (Dataset de prueba), cada versión de generador generará una regla a partir de la descripción de una regla existente dentro de dicha muestra, y se evaluará la sintaxis, la afinidad semántica de la condición de la regla generada, y la similitud de dicha condición con la de la regla original.

A continuación explicaremos cómo hemos calculado las métricas empleadas en las pruebas automáticas.

### 2.4.1. Diseño de las Pruebas Automáticas

Para estimar tanto la afinidad semántica (la similitud de la traducción de la regla generada y la descripción de entrada) como la similitud entre la condición generada y la condición original, emplearemos el cálculo de la similitud del coseno.

Para la afinidad semántica, necesitamos interpretar las reglas generadas para luego compararlo con la descripción, y para este propósito hemos creado el modelo extractor de semántica con el entrenador supervisado. Sin embargo, este modelo puede generar inexactitudes que puede afectar el resultado de este cálculo.

En cuanto a la evaluación la sintaxis de la condición regla generada, hemos programado una función que evalúa si esta condición cumple con las siguientes reglas[8]:

1. Cada elemento de la condición está separado por un carácter especial o un espacio
2. Paréntesis emparejados (cada uno abierto tiene su cierre)
3. Valor seguido por operador
4. Los conjuntos de items se encuentran entre paréntesis (cada elemento está separado por coma)
5. Los elementos globales son declarados con un 'glob' seguido de un glob entre comillas dobles
6. Los operadores binarios lógicos no tienen por qué tener un valor a la derecha
7. Operadores unitarios sucedidos por un campo entre paréntesis, sin espacios
8. Se pueden usar los siguientes operadores binarios: =, !=, <=, >=, >
9. Los binarios escritos solo pueden ser los siguientes: `contains`, `icontains`, `startswith`, `endswith`, `glob`, `in`, `intersects`, `pmatch`, `exists`, `bcontains`, `bstartswith`
10. Una condición puede usar los siguientes operadores unarios: `val()`, `tolower()`, `toupper()`



### 2.4.2. Requisitos Técnicos para las Pruebas

Para realizar las pruebas, hemos utilizado una instancia "GPU T4" de Google Colab, que ofrece gratuitamente durante 3 horas una tarjeta gráfica para entrenar modelos de lenguaje. El código ha sido ejecutado localmente en esta instancia.

La razón por el que usamos una GPU en vez de una CPU para este propósito es porque están especializadas en realizar operaciones con matrices, permitiendo paralelizar cálculos complejos durante la fase del entrenamiento y liberar la CPU para otros procesos cruciales para el sistema.

Una instancia "GPU T4" de Google Colab cuenta con las siguientes especificaciones:

- **Tarjeta Gráfica:** NVIDIA Tesla T4
- **Almacenamiento:** 78.19 GB
- **Memoria RAM:** 12.67 GB
- **Memoria de Vídeo (VRAM):** 11.92 GB

Además, hemos usado el siguiente software para ejecutar el código:

- **Motor de ejecución:** Python 3.10.12
- **Tecnología de procesamiento:** CUDA 12.2
- **Motor de ejecución de modelos:** PyTorch 2.3.0
- **Plataforma de IA:** HuggingFace

Para los generadores de reglas, hemos utilizado varios modelos para entrenar las distintas versiones del generador de reglas. Los tres primeros modelos son de libre disposición y los más pequeños en términos de parámetros. El modelo LLM utilizado es el modelo LLM más potente disponible hasta la fecha:

- **Entrenamiento supervisado (V1, extractor semántica):** google-t5
- **Entrenamiento no supervisado (V2, V3):** GPT2
- **Clasificador de Prioridad:** distilbert-base-uncased
- **Generador LLM:** GPT-4o (propietario)

Es importante señalar que los modelos LLM disponibles online, sean de código abierto o no, deben ser desplegados en equipos capaces de ejecutarlos, lo que puede resultar costoso debido a su tamaño. Por esta razón, muchos de estos modelos se pueden acceder mediante una API de pago.

En el caso de GPT-4o, el costo de generación de texto es de 0,005\$ por cada 1000 tokens de entrada más 0,015\$ por cada 1000 tokens de salida generadas[9]. El *prompt* para la generación de reglas que hemos utilizado, disponible en el segundo apéndice de este documento, tiene un tamaño de 733 tokens según el tokenizador de GPT-4[10].

# 3

## Análisis de los resultados

Ahora, procederemos a mostrar y analizar los resultados de las pruebas realizadas con nuestra aplicación.

### 3.1. Prueba Manual

Para la prueba manual, hemos utilizado la descripción de un método de detección para la técnica ATT&CK T1098.002:

*Monitor for unusual Exchange and Office 365 email account permissions changes that may indicate excessively broad permissions (including memberships in privileged groups) being granted to compromised accounts.*

Al finalizar la ejecución de las pruebas, hemos obtenido las siguientes salidas de cada una de las distintas versiones del generador de reglas:

## Versión 1

En la salida de la versión 1 del generador vemos que ha generado la regla parafraseando el contenido de la descripción de prueba, no siendo capaz de generar una regla Falco propiamente como tal. Vemos también que el campo output es idéntico al de la condición generada

```
1 condition: sss (including membership in privileged groups) to
  compromised accounts.
2 output: sss (including membership in privileged groups) to
  compromised accounts.
```

## Versión 2

En esta versión vemos que ha entendido en cierta medida la sintaxis de Falco y es capaz de utilizar palabras clave de la propia sintaxis de Falco.

```
1 condition: pnum(typeer (exacceptd[/ notc, " (sh,sh,sh,sh,sh,sh,
  sh,sh-sh-sh- (us,att,att,att,att-sh-sh- (us,att,
2 output: ((evt.typeen (ac= host) and evt.dir=< andred.name) and
  (proc.name and ind.name) and (contain a.id!= host) and (proc
  .c
```

## Versión 3

En cuanto a la tercera versión de generador de reglas, este no ha conseguido comprender la sintaxis de una regla Falco ni las palabras clave, resultando en una regla Falco incompleta y no válida.

Creemos que la razón de este resultado es porque el generador ha intentado entender la regla entera en vez de intentar comprender cada campo, mezclando todos los campos como si fuera un único texto

```
1 condition: 'rule\n
2 output:\n- hosterv ruleig,\n host canerp, rule rule canicef
  /-\n- container\n-ame.ameid, \%, icef /- in, \% below,\n
```

## Generador LLM

Observamos que el generador LLM ha comprendido correctamente la sintaxis de una regla Falco y ha sido capaz de generar una a partir del `prompt` que le hemos proporcionado a dicho LLM, pudiendo incluir el atributo opcional `'tags'`. Esto se debe porque en el único ejemplo de salida ofrecida viene con este campo.

```
1 - rule: Unusual Exchange/Office 365 Email Account Permissions
    Change
2   desc: Monitor for unusual Exchange and Office 365 email
        account permissions changes that may indicate excessively
        broad permissions, including memberships in privileged
        groups, being granted to compromised accounts.
3   condition: >
4     evt.type="execve"
5     and evt.dir=<direction>
6     and evt.pathname="/path/to/permission/change/command"
7     and (evt.arg.parameters contains "Add-MailboxPermission" or
        evt.arg.parameters contains "Set-MailboxPermission" or
        evt.arg.parameters contains "Add-RoleGroupMember")
8     and (evt.uid != <expected_user_id> and evt.gid != <
        expected_group_id>)
9   output: An unusual email account permissions change was
        detected (user=%user.name user_uid=%user.uid command=%proc
        .cmdline)
10  priority: warning
11  tags: [security, exchange, office365, permissions,
        email_account, privileged_group]
```

Sin embargo, observamos que el generador LLM ha dejado marcas de valores para reemplazar (*placeholders*), como `<expected_user_id>`, o esta pseudo-dirección `/path/to/permission/change/command`, lo cual nos indica que al generador LLM le falta el contexto. Aun así, En cuanto el coste de la generación, este ha sido de aproximadamente 0.001\$

## 3.2. Pruebas Automáticas

La Versión 1 presenta una alta tasa de éxito en sintaxis (más del 80%) y afinidad semántica (cerca del 70%), pero su rendimiento en similitud es bajo. Esto sugiere que, aunque las reglas generadas son sintácticamente correctas y semánticamente relevantes, no son muy similares a las descripciones de entrada.

La Versión 2 tiene un éxito moderado en sintaxis (aproximadamente 50%) y similitud (alrededor del 20%), pero su afinidad semántica es muy baja, lo que indica que, aunque las reglas generadas son parcialmente correctas y algo similares a las descripciones de entrada, no logran capturar bien el significado subyacente de las descripciones.

La Versión 3 muestra un rendimiento prácticamente nulo en todas las métricas, lo que indica que no es efectiva para generar reglas válidas en ningún aspecto.

El generador LLM destaca significativamente con una tasa de éxito del 100% en sintaxis, una alta tasa de éxito en similitud (aproximadamente 60%), y una tasa baja en afinidad semántica (cerca del 10%). Esto indica que, aunque las reglas generadas son completamente correctas en términos de sintaxis y bastante similares a las descripciones de entrada, aún tienen margen de mejora en términos de capturar la semántica completa de las descripciones.

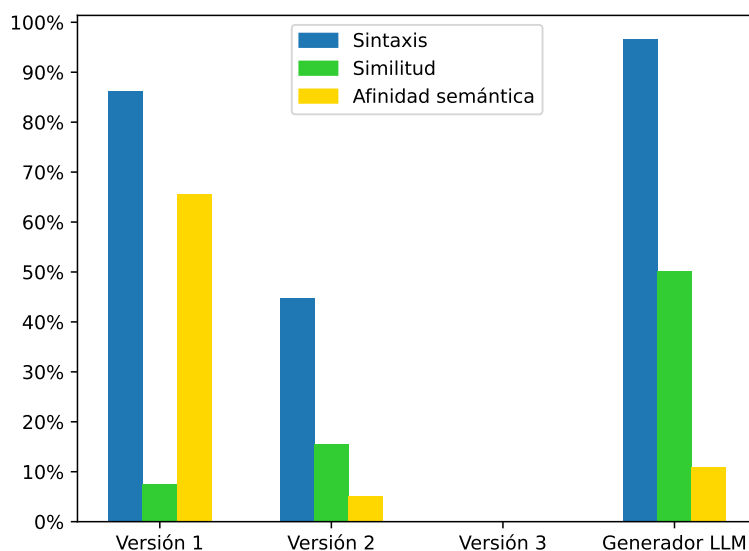


Figura 3.1: Tasa de éxito de las distintas versiones

### 3.3. Consumo de recursos

A lo largo de la ejecución, la aplicación ha generado 14,23 GB de información, de los cuales 17,36 MB es el peso de todos los datasets generados y 445.45 KB corresponde al peso de todos los repertorios de reglas que hemos recopilado.

En cuanto al consumo de memoria de vídeo (VRAM), la media de consumo de la memoria de vídeo ha sido de 5,96 GB, con un pico de consumo de VRAM de 7,59 GB durante el entrenamiento del modelo extractor de semántica. Esto se debe al tamaño de la salida de referencia, que ha sido las descripciones de cada una de las reglas.

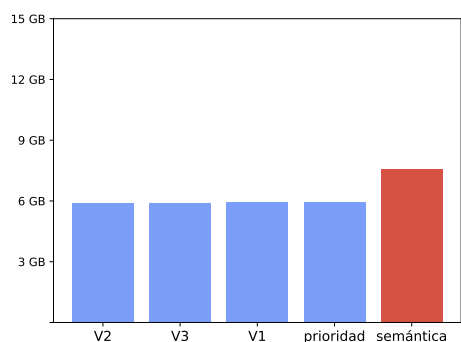


Figura 3.2: Consumo de memoria de vídeo por entrenamiento

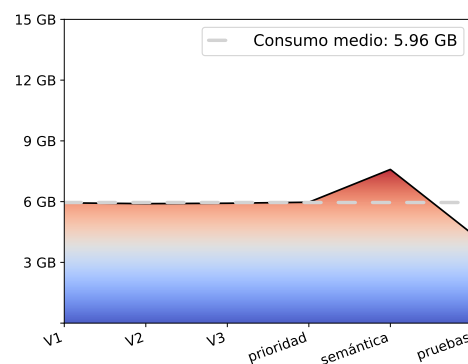


Figura 3.3: Evolución del consumo de la memoria de vídeo

En cuanto al coste de generación de reglas con el modelo LLM, una regla Falco generada por el modelo GPT-4o ha costado 0,006\$ de media.

# 4

## Conclusiones

Como hemos explicado en el capítulo de objetivos, el objeto de este trabajo es investigar la viabilidad de utilizar algoritmos NLP para automatizar el proceso de creación de reglas en Falco, y como se ha podido observar a lo largo de la lectura, creemos que los resultados son prometedores:

En las pruebas realizadas hemos observado que las reglas generadas cumplían con la sintaxis de Falco y se ajustaban adecuadamente a las descripciones proporcionadas, especialmente la versión 1 y el generador LLM. Y en cuanto a la reglas generadas por las distintas versiones de generador de reglas durante las pruebas manuales, el generador LLM ha sido capaz de generar una regla completa con atributos opcionales, y mostró una alta tasa de éxito en términos de sintaxis y afinidad semántica.

Además, en la prueba manual se ha usado una descripción de una forma de detección de una técnica de MITRE ATT&CK real y el generador LLM ha sido capaz de generar una regla que se ajuste a esto, demostrando que un generador de reglas puede crear reglas basadas en descripciones de técnicas MITRE, pero también nos demuestra que no sería capaz de generar alertas si fueran descripciones de amenazas propias de MITRE, porque en la misma regla generada por el generador LLM se ha empleado *placeholders* porque le faltaba el contexto de esta regla.

Sin embargo, lo más desalentador es el consumo de recursos: aunque el consumo promedio de memoria de video es exhaustivo, con un consumo pico de hasta 7.59 GB, el afinamiento ha sido realizado con las versiones de menos parámetros de los modelos de lenguaje que hemos usado.



Teniendo en cuenta los costes de generación de reglas observados y los resultados obtenidos, concluimos que la mejor versión del generador de reglas es el generador LLM, pero existe ciertos problemas de nuestra aplicación que deben ser abordados, como veremos en el siguiente apartado.

### **4.1. Limitaciones y Riesgos de Seguridad y privacidad**

#### **1. Problemas de Escalabilidad**

La escalabilidad de la aplicación es limitada debido al uso intensivo de GPU para entrenar los modelos, lo que requiere hardware especializado y es costoso. Además, la aplicación puede tener dificultades para generar múltiples reglas si las descripciones de los ataques son ambiguas, lo que podría reducir la efectividad en la detección de amenazas. Mejorar la calidad de las descripciones de entrada y desarrollar modelos más robustos podría mitigar estos problemas.

#### **2. Problemas de ambigüedad**

Como hemos explicado anteriormente, el generador de reglas que usa el modelo LLM no es capaz de generar reglas en base a las descripciones de amenazas realizadas por MITRE debido a que son ambiguas. Sería posible suministrar un contexto sobre el entorno donde se debe aplicar la regla para que esta sea más completa.

Sin embargo, los generadores de reglas generan una única regla para una descripción dada, por lo que si cierta amenaza solo es identificable si la amenaza utiliza una combinación de técnicas específica, o la descripción de la amenaza es ambiguo porque su naturaleza es muy compleja, lo cual requiere en ambos casos realizar una interpretación y crear más de una regla, ninguno de los generadores de reglas que hemos creado sería capaz de cubrirlo.

#### **3. Problemas de Seguridad de la Aplicación**

La aplicación enfrenta riesgos significativos como la posibilidad de que las muestras del dataset estén contaminadas, o que la comunicación entre la aplicación y HuggingFace o el proveedor del modelo LLM pueda ser interceptada. Así, Un adversario podría introducir datos maliciosos en el dataset, comprometiendo la calidad de las reglas generadas, o interceptar las comunicaciones, accediendo a

información sensible. La implementación de técnicas de validación de datos, cifrado de comunicaciones y autenticación fuerte son medidas cruciales para mitigar estos riesgos.

#### **4. Problemas de dependencia a Internet**

La dependencia de Internet para acceder a modelos alojados en plataformas como HuggingFace y OpenAI o proveedor de otro modelo LLM que se haya usado en la aplicación es un riesgo considerable. Problemas de conectividad o caídas en el servicio pueden interrumpir la generación de reglas, y la latencia en la red puede afectar la velocidad de respuesta de la aplicación. Desarrollar modelos que puedan ejecutarse localmente y establecer mecanismos de caché podrían reducir esta dependencia y mejorar la disponibilidad del servicio.

## 4.2. Trabajos Futuros

Para mejorar la aplicación, hemos identificado los siguientes puntos de mejora:

### 1. Retroalimentación del Dataset

Incrementar el número de muestras del dataset retroalimentándolo con las reglas generadas más buenas permite crear un ciclo de mejora constante, lo que podría ayudar mejorar la precisión y la calidad de las nuevas reglas generadas, siempre y cuando se usen las métricas correctas

### 2. Optimizar el almacenamiento

Aunque el número de muestras usados en esta investigación es muy pequeña (282 muestras) y se ha podido almacenar las muestras en un único dataset, es posible que el número de muestras disponibles crezca, sea porque hay nuevas aportaciones o porque estamos retroalimentando el dataset. En este caso es necesario considerar usar sistemas de almacenamiento, como bases de datos documentales.

### 3. Creación de un modelo propio

Como hemos observado antes, las 3 versiones del generador de reglas dependen de otros modelos para, o bien completar las reglas o bien evaluar la afinidad semántica. Un modelo unificado con su propio tokenizador podría ayudar a optimizar el consumo de recursos, además de reducir la dependencia de internet si se implementa con un framework, como TensorFlow o PyTorch.

### 4. Modelo descriptor de reglas

Los generadores de reglas solo generan una regla a partir de una descripción. Sin embargo, es posible que la descripción sea ambigua, por ser una técnica, o se necesite más de una regla para adecuarlo a la descripción. Para ello, sería recomendable investigar la aplicación de modelos de lenguaje para crear distintas descripciones de reglas que ayuden a generar las reglas más adecuadas

# Bibliografía

- [1] H. Face, “The hugging face course, 2022,” <https://huggingface.co/course>, 2022, [Online, accessed 09-06-2024].
- [2] P. Johri, S. K. Khatri, A. Al-Taani, M. Sabharwal, S. Suvanov, and A. Chauhan, *Natural Language Processing: History, Evolution, Application, and Future Work*. ResearchGate, 01 2021, pp. 365–375.
- [3] Suetonius, *De Vita Caesarum, Divus Iulius*. Fordham University, 110 D.C., [Through Internet Sourcebooks Project, Accessed 12-06-2024]. [Online]. Available: <https://sourcebooks.fordham.edu/ancient/suetonius-julius.asp>
- [4] J. M. White and G. D. Rohrer, “Image thresholding for optical character recognition and other applications requiring character image extraction,” *IBM Journal of Research and Development*, vol. 27, no. 4, pp. 400–411, 1983.
- [5] M. Sabourin and A. Mitiche, “Optical character recognition by a neural network,” *Neural Networks*, vol. 5, no. 5, pp. 843–852, 1992. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0893608005801443>
- [6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” 2023.
- [7] Deloitte, “State of ai in the enterprise, 2nd edition,” [https://www2.deloitte.com/content/dam/insights/us/articles/4780\\_State-of-AI-in-the-enterprise/DI\\_State-of-AI-in-the-enterprise-2nd-ed.pdf](https://www2.deloitte.com/content/dam/insights/us/articles/4780_State-of-AI-in-the-enterprise/DI_State-of-AI-in-the-enterprise-2nd-ed.pdf), 2018, [Accessed 10-06-2024].
- [8] F. Community, “Falco documentation,” Aug 2023, [Accessed 12-06-2024]. [Online]. Available: <https://falco.org/docs/>
- [9] OpenAI, “Openai: Pricing,” Aug 2023, [Accessed 12-07-2024]. [Online]. Available: <https://openai.com/api/pricing/>
- [10] [GPT-4o tokenizer is not available up to date].

# Apéndice







## Apéndice 1: Prompt de generación de reglas

*El siguiente prompt ha sido utilizado en GPT-4o:*

```
1 [SYSTEM]
2 You are a security engineer, and your task is to automate the
  generation of Falco rules to detect and prevent various
  attack techniques. You will receive a description of an
  attack technique, and you need to produce a corresponding
  Falco rule in YAML format. Guidelines for generating the
  Falco rule:
3 - Rule Definition:
4     - Define a rule name that briefly describes the
      detection objective.
5     - Provide a description (desc) of what the rule is
      detecting and why it's important.
6     - Mandatory fields are "Rule", "Description", "
      Condition", "Output" and "Priority", in that order.
      "Rule" field is the header of the Rule object, and
      contains the name of the rule, while the rest of
      fields are sub-fields of the Rule object
7 - Condition:
8     - Use appropriate conditions to detect the described
      attack technique.
9     - Include relevant process names, arguments, and
      context (such as user IDs or working directories).
10    - Ensure the conditions are specific enough to avoid
```



---

```

    false positives but general enough to catch the
    described attack.
11 - Output:
12     - Craft an output message that clearly states what was
        detected, including relevant details such as the
        command line and user name. Use up to 50 words
13 - Priority and Tags:
14     - Set an appropriate priority level (info,
        warning, critical). Include relevant tags
        for categorization (e.g., [security,
        intrusion]).
15
16 It is expected the user will submit only a description for a
    specific rule. Give only the Falco rule in YAML format. Do
    not comment it nor use Markdown. Once you finished, in a new
    line write [END]
17
18 [USER]
19 Please generate a Falco Rule for the following description:
20 A shell was used as the entrypoint/exec point into a container
    with an attached terminal. Parent process may have
    legitimately already exited and be null (read
    container_entrypoint macro). Common when using "kubectl exec
    " in Kubernetes. Correlate with k8saudit exec logs if
    possible to find user or serviceaccount token used (fuzzy
    correlation by namespace and pod name). Rather than
    considering it a standalone rule, it may be best used as
    generic auditing rule while examining other triggered rules
    in this container/tty.
21
22 [ASSISTANT]
23 - rule: Terminal shell in container
24   desc: >
25     A shell was used as the entrypoint/exec point into a
        container with an attached terminal. Parent process may
        have
26     legitimately already exited and be null (read
        container_entrypoint macro). Common when using "kubectl
        exec" in Kubernetes.
27     Correlate with k8saudit exec logs if possible to find user
        or serviceaccount token used (fuzzy correlation by
        namespace and pod name).
28     Rather than considering it a standalone rule, it may be
        best used as generic auditing rule while examining other
        triggered
29     rules in this container/tty.
30   condition: >

```

```
31     spawned_process
32     and container
33     and shell_procs
34     and proc.tty != 0
35     and container_entrypoint
36     and not (never_true)
37 output: A shell was spawned in a container with an attached
      terminal (evt_type=%evt.type user=%user.name user_uid=%
      user.uid user_loginuid=%user.loginuid process=%proc.name
      proc_exepath=%proc.exepath parent=%proc.pname command=%
      proc.cmdline terminal=%proc.tty exe_flags=%evt.arg.flags %
      container.info)
38 priority: NOTICE
39 tags: [maturity_stable, container, shell, mitre_execution,
      T1059]
40
41 [USER]
42 Please generate a Falco Rule for the following description:
43 <new rule>
44 (...)
```

# B

## Apéndice 2: Fuentes de origen de las reglas recopiladas

- Fuente 1: Repositorio oficial de Falco:  
<https://github.com/falcosecurity/rules>
- Fuente 2: Repositorio cloud-native-security-hub (falcosecurity-retire):  
<https://github.com/falcosecurity-retire/cloud-native-security-hub>
- Fuente 3: Repositorio falco\_extended\_rules (CloudDefenseAI):  
[https://github.com/CloudDefenseAI/falco\\_extended\\_rules](https://github.com/CloudDefenseAI/falco_extended_rules)
- Fuente 4: Repositorio Mitre-Attack-Falco-AWS (n1g3ld0ug1a5):  
<https://github.com/n1g3ld0ug1a5/Mitre-Attack-Falco-AWS>
- Fuente 5: Repositorio Sysdigfalcorules (hidd3ncod3s):  
<https://github.com/hidd3ncod3s/sysdigfalcorules>
- Fuente 6: Repositorio falco-rules-test (fsdaniel):  
<https://github.com/fsdaniel/falco-rules-test>



## Apéndice 3: Glosario

- **Cloud Native:** Conjunto de tecnologías que implementa la infraestructura que posibilita la externalización y virtualización de los recursos de forma flexible y ágil.
- **CNCF:** Cloud Native Computing Foundation. Es la principal entidad encargada de apoyar las iniciativas para el entorno de nube
- **NLP:** Procesamiento del Lenguaje Natural. Disciplina por el que se busca procesar el lenguaje natural por parte de un agente artificial mediante algoritmos y representaciones
- **LM (y LLM):** Modelo de Lenguaje. Representación probabilística de un lenguaje natural sobre un espacio N-dimensional. Es un modelo grande (LLM) si el vector dispone de muchos parámetros. Estos modelos suelen ser usados de forma *online*
- **Token:** también conocido como parámetro, es la representación numérica de una letra, morfema o palabra en un modelo
- **Vector de Tokens:** Vector: conjunto de coordenadas de un punto dentro de un espacio vectorial
- **Espacio Vectorial:** conjunto N-dimensional de todos los posibles vectores que puede representar el modelo
- **Dataset:** Conjunto de datos para ser procesado.
- **Modelo Seq2Seq:** También llamado codificador/decodificador, es un tipo de modelo de inteligencia artificial diseñado para transformar una secuencia de datos a otra secuencia de datos (en nuestro caso, texto). Estos modelos son ideales para tareas donde la entrada y la salida son secuencias de texto, como la traducción automática, el resumen de texto y la generación de descripciones
- **Modelo Decodificador:** también conocidos como modelos autoregresivos, son modelos de lenguaje que predicen el próximo token siguiendo un modelo matemático. Estos modelos no necesitan de apoyarse en ninguna salida de referencia.

- 
- **Métrica ROUGE:** *Recall-Oriented Understudy for Gisting Evaluation*. Métrica que se utiliza para evaluar tareas de resumen de texto, donde se comparan las frases generadas automáticamente con referencias humanas propias de la métrica.
  - **Métrica de Perplejidad:** mide la capacidad de un modelo de lenguaje para predecir una secuencia de palabras. Se calcula como la exponencial de la entropía cruzada media entre la distribución de probabilidad verdadera y la predicha por el modelo. Una menor perplejidad indica un modelo mejor, ya que significa que el modelo asigna mayores probabilidades a las palabras correctas. Se utiliza principalmente en modelos de lenguaje y generación de texto para evaluar su rendimiento y capacidad de generalización.
  - **Similitud del coseno:** fórmula que mide la cercanía angular entre dos vectores de tokens. Se calcula como  $(\vec{u} \cdot \vec{v}) / (|\vec{u}| \cdot |\vec{v}|)$