Propuesta de un sistema para identificar y clasificar ciberataques a través de honeypots SSH

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Pablo José Rocamora Zamora
Tutor/es:
Gregorio Martínez Pérez
Manuel Gil Pérez

19 de Septiembre de 2019

Facultad
Informática
Universidad
Murcia

Propuesta de un sistema para identificar y clasificar ciberataques a través de honeypots SSH

Autor
Pablo José Rocamora Zamora

Tutor/es
Gregorio Martínez Pérez
Departamento de Ingeniería de la Información y las Comunicaciones (DIIC)
Manuel Gil Pérez
Departamento de Ingeniería de la Información y las Comunicaciones (DIIC)

Grado en Ingeniería Informática

Murcia, 19 de Septiembre de 2019

A mi familia y profesores por ayudarme durante toda mi etapa académica.

Las organizaciones gastan millones de dólares en firewalls y dispositivos de seguridad, pero tiran el dinero porque ninguna de estas medidas cubre el eslabón más débil de la cadena de seguridad: la gente que usa y administra los ordenadores.

Kevin Mitnick.

Declaración firmada sobre originalidad del trabajo

D./Dña. **Pablo José Rocamora Zamora**, con DNI **11111111A**, estudiante de la titulación de **Grado en Ingeniería Informática** de la Universidad de Murcia y autor del TFG titulado "**Propuesta de un sistema para identificar y clasificar ciberataques a través de honeypots SSH**".

De acuerdo con el Reglamento por el que se regulan los Trabajos Fin de Grado y de Fin de Máster en la Universidad de Murcia (aprobado C. de Gob. 30-04-2015, modificado 22-04-2016 y 28-09-2018), así como la normativa interna para la oferta, asignación, elaboración y defensa de los Trabajos Fin de Grado y Fin de Máster de las titulaciones impartidas en la Facultad de Informática de la Universidad de Murcia (aprobada en Junta de Facultad 27-11-2015)

DECLARO:

Que el Trabajo Fin de Grado presentado para su evaluación es original y de elaboración personal. Todas las fuentes utilizadas han sido debidamente citadas. Así mismo, declara que no incumple ningún contrato de confidencialidad, ni viola ningún derecho de propiedad intelectual e industrial

Murcia, a 19 de Septiembre de 2019

Fdo.: Pablo José Rocamora Zamora
Autor del TFG

Índice general

Índice de figuras

Índice de tablas

Resumen

Los sistemas de seguridad clásicos que se usan actualmente están basados en firmas (antivirus y firewall), pero tienen el problema de que, generalmente, no son demasiado eficientes contra nuevas amenazas o amenazas conocidas pero que utilizan nuevos patrones de ataque. Por esta razón, actualmente, ya no es suficiente con actuar de forma reactiva ante tales amenazas, sino que es necesario también ser capaces de actuar de forma proactiva, con el objetivo de anticiparse a los posibles ataques y evitar que estos se lleguen a producir. Para resolver este problema surgen los honeypots, una herramienta que permite la recolección de información sobre las acciones que realizan posibles atacantes; en cierta manera, en lugar de obtener información los atacantes sobre tu sistema, obtienes tú información sobre los atacantes, permitiendo que seas capaz de analizar sus objetivos y comportamientos.

Dada la ingente cantidad de ataques recibidos en los honeypots y la gran cantidad de información que éstos registran, no es posible realizar un análisis manual de estos datos. Asimismo, los honeypots únicamente registran la información, pero no son capaces de hacer ningún tipo de análisis de los datos registrados. Por ambos motivos surge la necesidad de disponer de un sistema capaz de clasificar el comportamiento de los atacantes en base a las acciones realizadas, con el objetivo de identificar los distintos tipos de ataque y el nivel de amenaza que éstos representan.

El Trabajo Fin de Grado que se presenta tiene como finalidad la generación de un sistema capaz de clasificar el comportamiento de los atacantes en base a las acciones realizadas (por ejemplo, ejecución de comandos) durante la duración de sus sesiones en un entorno protegido y emulado como el que proporciona la honeypot SSH Cowrie. Este proyecto ha sido diseñado para resolver el reto propuesto por el Instituto Nacional de Ciberseguridad (INCIBE) en el Track de Transferencia de las Jornadas Nacionales de Investigación en Ciberseguridad (JNIC) llamado *IN1. Clasificación de ataques a honeypots SSH*. Para realizar este reto se va a partir de un conjunto de datasets públicos con logs de Cowrie ya generados en formato textual.

Para realizar esta tarea, lo primero que se ha realizado ha sido un estudio sobre qué es un honeypot. Una vez que se han tenido conocimientos suficientes sobre los honeypots de forma genérica, se ha procedido a investigar sobre el honeypot SSH Cowrie. Este es un honeypot de media interacción y el más actualizado en la actualidad, ya que proporciona mayores capacidades de registro de las acciones ejecutadas por los atacantes sin poner en riesgo el sistema de producción subyacente, algo que sí hacen las honeypots de alta interacción. Después, se ha procedido a instalar y configurar Cowrie, usando Docker y la imagen oficial de Cowrie para instalarla como contenedor docker.

Los principales cambios que se realizaron en la configuración fueron para intentar hacer que el honeypot sea lo menos identificable posible.

Una vez que se empezaron a obtener logs, se procedió a realizar un análisis de estos para ver cuál era su formato, estructura, la información que registran y cuáles de estos son más relevantes. Con toda esta información sobre cómo es la estructura de logs en Cowrie, se ha investigado sobre la existencia de alguna herramienta capaz de extraer toda la información de los logs, pero no existe ninguna que sea capaz de extraer toda la información. Por este motivo, uno de los objetivos planteados en este Trabajo Fin de Grado ha sido el desarrollo de un parser capaz de generar otro fichero con las sesiones generadas en un formato estructurado como es JSON.

Para almacenar los datos se ha seleccionado la base de datos Elasticsearch (JSON), dado que es capaz de operar con una gran cantidad de datos en tiempo real. Además, cuenta con Kibana, una herramienta para la visualización de datos a través de tablas y gráficas usado habitualmente en proyectos de similares requisitos.

Una vez subidos los datos en Elasticsearch, el objetivo principal para el uso de Kibana es la obtención y visualización de datos de este. Una vez que se tienen acceso a los datos se pueden crear todas las tablas y diagramas que se consideren más representativos, como puede ser el de credenciales usadas, comandos ejecutados, ficheros descargados o mapa de calor según la procedencia del atacante. Con los datos almacenados a partir de los logs registrados por Cowrie, el siguiente objetivo era llevar a cabo un análisis manual de los mismos, de los que se han obtenido las siguientes conclusiones:

- Se han registrado 16.967.189 conexiones, en las cuales hay 66.209 direcciones IP diferentes.
- En un análisis inicial del mapa de calor geográfico de ataques, los tres países que mayor cantidad de ataques han realizado han sido China, EE. UU. y Rusia.
- Las credenciales usadas han sido *root/admin* en un 83% de las ocasiones.
- Los comandos utilizados son en gran medida para la obtención de información del sistema.
- Los clientes SSH más usados son *OpenSSH* y *Go*. Además, se han detectado intentos de buffer overflow aprovechándose de que se puede enviar un string con el cliente SSH que se quiera.
- Se ha detectado que un 10,9% de las sesiones son escaneos de puertos.
- El 99% de las sesiones tienen un nivel de peligrosidad bajo, en las cuales no se han llegado a ejecutar ningún comando.
- La mayoría de ficheros descargados y analizados tienen como finalidad realizar un ataque DDoS, entre las que se ha detectado la botnet Mirai.

Finalmente, este Trabajo Fin de Grado también propone la realización de una serie de experimentos con distintos algoritmos de Machine Learning con el objetivo de clasificar sesiones e identificar posibles tipos de ataques. En un primer experimento se plantea el uso de un algoritmo no supervisado de clustering para intentar agrupar sesiones y ver si existen relaciones entre ellas. Después de su ejecución, y no obtener resultados positivos a los esperados, dentro del mismo experimento se plantea el uso de algoritmos

supervisados. Estos requieren que los datos estén etiquetados y para etiquetarlos se propone como objetivo realizar un sistema experto basado en reglas, con el objetivo de etiquetar todas las sesiones según la severidad de los comandos ejecutados, siendo esta la feature utilizada. Después de esta segunda prueba de ejecución también se obtuvieron malas precisiones con estos algoritmos.

Viendo que las features utilizadas en las pruebas anteriores no habían sido correctas, en un segundo experimento se plantea modificar el sistema experto de reglas con el objetivo de mejorar la precisión. En esta ocasión, se plantea seguir usando como feature los comandos ejecutados, pero en lugar de etiquetarlos por nivel de severidad se propone modificar su etiqueta. A continuación, se muestran algunos de esos grupos: lectura en disco, obtención de información del SO y descargas en internet. Aunque después de su puesta en ejecución se consiguen mejores resultados, aunque todavía no sustanciables en precisión, se propone como siguiente objetivo cambiar el dataset por uno que lleve más tiempo capturando datos y, por ende, que contenga más datos sobre acciones ejecutadas por los atacantes. Como resultado de este segundo experimento, utilizando un nuevo dataset público, se han llegado a alcanzar mejores resultados en precisión después de su ejecución con distintos algoritmos de Machine Learning.

Finalmente, se plantea abordar un último experimento con el objetivo de analizar si añadiendo otras features, que podrían ser relevantes, se obtienen mejores resultados en las métricas de evaluación. Estas nuevas features seleccionadas para su evaluación son el tiempo total de sesión, el análisis de ficheros descargados, el país atacante y el cliente SSH. Como resultado de la ejecución de este último experimento se ha obtenido una precisión del 99,82%, con lo que se consigue una mejora muy leve pero cercana al 100%.

La conclusión que se ha obtenido es que, aunque es posible realizar clasificaciones de los ataques, requieren un estudio previo de estos para poder etiquetarlos, siendo esta tarea bastante compleja. El uso de honeypots nos recuerda la importancia de abordar la seguridad desde todas las perspectivas disponibles. Recopilar información usando honeypots no es un reto como tal, pero el conseguir discernir entre las intenciones de los atacantes sí que lo es. Usando honeypots y distintas técnicas de Machine Learning, como las aplicadas en este proyecto, se podría obtener una radiografía en tiempo real de los ataques realizados, permitiendo adaptarnos a estos.

Extended Abstract

The classic security systems that are currently based on signatures (antivirus programs and firewalls), but have the problem that, usually, they are not too efficient against new threats or known threats but they use new attack patterns. For this reason, these days it is no longer enough to act reactively before such threats, but it is also necessary to act proactively in order to anticipate possible attacks and prevent them from reaching produce. To solve this problem, honeypots arise, which allow the collection of information on the actions that possible attackers can carried out. In a certain way, instead of obtaining information about the attackers regarding our system, we can get information about the attackers, to analyze their objectives and behaviors.

Given the huge number of attacks received in honeypots and the large amount of information they record, it is not possible to perform a manual analysis of this data. Also, honeypots only record the information, but are not able to do any type of analysis of the recorded data. For both reasons, there is a need to have a system capable of classifying the behavior of attackers based on the actions carried out in order to identify the different types of attack and the level of threat they represent.

The presented Master's thesis is aimed at generating a system capable of classifying the behavior of attackers based on the actions performed (for example, execution of commands) during the duration of their session in a protected and emulated environment such as the one provided by the honeypot SSH Cowrie. This project has been designed to solve the challenge proposed by the National Cybersecurity Institute (INCIBE) in the Transfer Track of the National Cybersecurity Research Conference (JNIC) called *IN1. Classification of attacks on SSH honeypots*. To carry out this challenge, it has been based on a set of public datasets with Cowrie logs already generated in textual format.

The first thing that has been done to conduct the planned tasks is a study on what a honeypot is, how it works, the different types that exist, what are its capabilities, what is it used for and the use made of the data that the honeypot is generating. Once we have had sufficient knowledge about honeypot generically, we have proceeded to investigate the honeypot SSH Cowrie. Once all this analysis has been carried out, Cowrie has been installed and configured. To this end, Docker and the official image of Cowrie have been used to install it as a docker container, although it had to be modified since the logs were shown on the screen instead of being saved in file. Finally, they have been deployed in two Raspberry Pi 3 Model B in different locations capturing data for about 5 months. To configure it, a study was carried out of all the features it has, in addition to the plugins which are adding extra functionalities. The main changes made were

trying to make the honeypot as unidentifiable as possible by changing the hostname, increasing the maximum session duration significantly so that human attackers could perform more actions, etc.

Once they began to obtain logs, they proceeded to perform an analysis of these to see which was their format, structure, the information they record and which of are more relevant. In this analysis, a quite important problem was identified: only the line on which a new connection is created has a session identifier; the rest uses a temporary identifier composed of a number and the IP address that initiates the connection, both separated by a comma. This makes the process of rebuilding sessions very complex, since it is necessary to perform an analysis of all the lines to find out which session they belong to. In case the JSON format that Cowrie generates is used, this problem does not exist, since it labels each line with the ID of the session to which this information belongs. It was also detected that, when a file is downloaded in the log, the download itself is recorded without any information about which session is the one that initiated the download. To find out, we need to look at previous lines which session has executed a command *wget*.

With all this information about how the log structure in Cowrie is, we have investigated the existence of some tools capable of extracting all the information from the logs, but there is none capable of extracting all the information; for this reason, it has proceeded to develop a parser capable of reading all the files in chronological order, obtaining all the sessions of each of the files and, subsequently, generating another file with the sessions generated in JSON format. This parser, in addition to converting the logs to JSON format, incorporates extra information that Cowrie does not provide. This information is: it adds the geographic information of the IP that carries out the attack; it adds the SSH reputation of the IP; it performs an analysis with VirusTotal of the downloaded files and adds it; it uses an expert system based on rules to label the sessions according to their severity, which is obtained by analyzing the executed commands; it labels the session in case a brute force attack has been received; and finally, it tags the connection if it has been a port scan. All this extra information added by the parser will help in the manual and the automatic classification stages (with Machine Learning techniques) of data.

To obtain the SSH reputation and analyze the files downloaded in VirusTotal, a REST API has been designed with a local database that caches the requests. This is because the use of external APIs puts significant latency and usually limit the number of requests per minute. Due to this, we managed to ask for that new information limiting the use of the external API and, in case the limit is exceeded, there is a daemon that is responsible for continuing to ask until we getting answer.

Once we have all the sessions in JSON format, we have proceeded to search for the most indicated database to store this information. After an analysis of the advantages and disadvantages of the relational (SQL) and non-relational (NoSQL) databases, it has been concluded that the most optimal database is the non-relational database, since it is a flexible approach to changes that allows horizontal scalability and is optimized

to large data queries. These requirements are important for the project and relational databases. Among them, it has chosen to use Elasticsearch. This is because it has Kibana for data visualization. Thanks to the fact that a non-relational database is used, no conversion of the data has to be performed, since they store the data in JSON format. Using Elasticsearch and Kibana is quite widespread nowadays, since the union of both makes it a very powerful tool for data storage and visualization through different types of graphics with real-time data refresh. It should be mentioned that the ELK stack also supplies Logstash, a tool for data processing and insertion into Elasticsearch that cannot be used in our project, since there is no relationship between the data, as mentioned earlier. It can be said that the developed parser has the function of supplying Logstash. Once Elasticsearch and Kibana have been selected, both tools have been studied. The data has been inserted into Elasticsearch using as an index the name of the honeypot to which they belong, having that index an associated mapping that indicates the type of each of the data that is added. When the data is entered in Elasticsearch, a series of updates are carried out. These modifications consist of looking for those files that the attackers have downloaded, and trying to download and analyze them against VirusTotal to check if they are identified as malware and, if so, to see what kind of malware they belong to. These steps are necessary, since only logs are available; we do not have access to downloads made by attackers.

In Kibana, we need to configure the index pattern that will be used to obtain the Elasticsearch data. When this pattern is defined, a series of filters are created according to the types of data to be searched, in order to do the search for data more efficient when representing them in the different graphs. With these created filters, we proceed to create all the tables and diagrams that are considered most representative, such as credentials used, executed commands, downloaded files or heat map, depending on the origin of the attacker. From this manual analysis, we have obtained the following conclusions:

- 16,967,189 connections have been registered, in which there are 66,209 different IP addresses; this indicates that there are many connections from the same addresses.
- In an initial analysis of the geographic heat map of attacks, it was detected that 75% of the connections were made from Ireland and the United Kingdom, which is contradictory, since they are countries that do not usually appear in the lists of countries that carry out the most of the attacks. This is possibly due to the use of these countries as a proxy to carry out the attacks from there. So, a new map has been made, but this time putting as a filter that each IP is unique, making the heat map to change radically. On this occasion, the three countries that have carried out the most attacks have been China, EE. UU. and Russia, being this data much more credible.
- The credentials used have been *root/admin* the 83% of the time. These are used in a large number of devices and, making a slightly larger analysis, it has been detected that there are hardly any brute force attacks, so, in general, most con-

nections are directed to devices with default credentials.

- The commands used are largely for obtaining system information. This may be due to Cowrie's limitation regarding the number of commands available.
- The most used SSH clients are *OpenSSH* and *Go*; these are programmed in C and Go respectively, so it is expected that the scripts that automate the attacks are programmed in those languages or, in the case of *OpenSSH*, that make use of the operating system's own library. In addition, buffer overflow attempts have been detected, taking advantage of the fact that the desired SSH client can be sent. Most of these attempts have the payload to send an email to an address or make a connection against an IP. This has been supposed to notify the attacker that the SSH server is vulnerable.
- It has been detected that 10.9% of the sessions are port scans, although this is not entirely true, since Cowrie only detects the application level scans, which are those that get to ask for the SSH version from honeypot SSH Cowrie. It is not able to detect the transport level scans, since it is a medium-interaction honeypot.
- 99% of sessions have a severity level of 4. This denotes that they have not been able to execute any command and there are only 21,753 connections that have a severity level of 1, which is the maximum, this meaning that the software has been downloaded from the internet or a program has been compiled.
- Most files downloaded and analyzed are intended to perform a DDoS attack. In addition, the Mirai botnet has also been detected. It has been detected that some attackers use false names in the files and change the extension for what is supposed to be fooling possible firewalls or IDSs.

Finally, a series of experiments have been carried out with different Machine Learning algorithms in order to classify sessions and identify possible types of attacks.

In an initial experiment, an unsupervised clustering algorithm was used to try to group sessions and check if there were relationships between them, but this was unsuccessful. For this reason, we proceeded to use supervised algorithms (Decision Tree, Random Forest and SVM). These require the data to be tagged and to label them by an expert system based on rules, with the aim of tagging all sessions according to the severity of the commands executed. 80% of the data were used for the training and 20% for the evaluation processes. Bad precisions were also obtained with these algorithms.

In a second experiment, given that the features we used had not been correct, the expert system was modified to try to improve accuracy. On this occasion, although the executed commands were still used as a feature, instead of labeling them by their severity level, the label was modified according to the following groups: disk reading, disk writing, obtaining OS information, connections and Internet downloads, compilation or installation of programs, execution of programs and eliminating processes in the OS. It was improved from 49% to the 76% but this is not enough, so it was tried to change the dataset for one that had been capturing data for a longer time and, therefore, they had more data. With this new dataset, an accuracy of 98% was reached.

Finally, we proceeded to a final experiment adding new features with the aim of checking if they were relevant. These new features were the total session time, the analysis of downloaded files, the attacker's country and the SSH client. With this new experiment we reached an accuracy of 99,82%, so we get a very slight improvement, but close to 100%.

The conclusions that have been obtained are that, although it is possible to make classifications of the attacks, they require a previous thorough study of these to label them, being this a quite complex task. The use of honeypots reminds us the importance of addressing security from all available perspectives. Collecting information using honeypots is not a challenge itself, but being able to discern between the intentions of the attackers. Using honeypots and different Machine Learning techniques such as those applied in this project, a real-time radiography of the attacks made could be obtained, allowing us to adapt to them.

1. Introducción

Los sistemas de seguridad clásicos que se usan actualmente están basados en firmas (antivirus y firewall), pero tienen el problema de que, generalmente, no son demasiado eficientes contra nuevas amenazas o amenazas conocidas pero que utilizan nuevos patrones de ataque. Por esta razón, actualmente, ya no es suficiente con actuar de forma reactiva ante tales amenazas, sino que es necesario también ser capaces de actuar de forma proactiva, con el objetivo de anticiparse a los posibles ataques y evitar que estos se lleguen a producir.

Para resolver este problema surgen los *honeypots*, como bien se establece en [1]. Un honeypot es un software diseñado con el propósito de atraer a posibles atacantes, simulando que es un sistema real y vulnerable a posibles ataques [2]. Esta herramienta permite la recolección de información sobre las acciones que realizan posibles atacantes; en cierta manera, en lugar de obtener información los atacantes sobre tu sistema, obtienes tú información sobre los atacantes, permitiendo que seas capaz de analizar sus objetivos y comportamientos [3].

Dado que gran cantidad de los dispositivos expuestos a internet están basados en UNIX, es una buena opción usar el honeypot Cowrie [4, 5]. Éste es un popular honeypot SSH de media interacción. Una de las mejores maneras de identificar a los honeypots es por el nivel de interacción que ofrece al atacante dentro del honeypot, siendo los de media interacción una mezcla entre los de baja, que solo simulan una parte de un servicio, y los de alta, que suelen ser sistemas reales que ofrecen un control total sobre el host donde está instalado y que, por consiguiente, suponen un riesgo muy alto para el resto del sistema. Cowrie está diseñado para registrar ataques de fuerza bruta y ejecución de comandos (*uname*, *cat*, *wget* y *tar*).

El proyecto que se presenta en esta memoria de Trabajo Fin de Grado se basa en la generación de un sistema capaz de clasificar el comportamiento de un atacante en base a las acciones realizadas (por ejemplo, ejecución de comandos) durante la duración de su sesión en un entorno protegido y emulado como el que proporciona la honeypot SSH Cowrie. Este proyecto ha sido diseñado para resolver el reto propuesto en el Track de Transferencia de las Jornadas Nacionales de Investigación en Ciberseguridad (JNIC), edición 2018, llamado *IN1. Clasificación de ataques a honeypots SSH* [6], cuyo retador era el Instituto Nacional de Ciberseguridad de España (INCIBE).

1.1. Motivación

Dada la ingente cantidad de ataques recibidos en los honeypots y la gran cantidad de información que éstos registran, no es posible realizar un análisis manual de estos datos. Asimismo, el honeypot únicamente registra la información, pero no es capaz de hacer ningún tipo de análisis de los datos registrados. Por ambos motivos surge la necesidad de disponer de un sistema capaz de clasificar el comportamiento de los atacantes en base a las acciones realizadas, con el objetivo de identificar los distintos tipos de ataque y el nivel de amenaza que éstos representan.

Actualmente hay una gran cantidad de dispositivos con acceso a internet y que son fácilmente accesibles mediante meta-buscadores como Shodan y Censys. Podemos ver en la Figura 1.1 que Shodan tiene registrados en su base de datos 18 millones de dispositivos que tienen operativos el protocolo SSH para un acceso remoto.

Por esta razón, es necesario obtener toda la información posible sobre los atacantes: metodología, objetivos y sistemas de interés, ya que con esta información nos podremos adelantar a sus ataques y securizar los dispositivos. Este incremento de dispositivos se debe al crecimiento del Internet of Things (IoT). Habitualmente, estos dispositivos se basan en UNIX y tienen el puerto SSH habilitado para la gestión remota. Estos tipos de ataques ya han sucedido. A finales del año 2016, la botnet Mirai infectó a millones de dispositivos IoT usando el protocolo Telnet y realizó un ataque DDoS que dejó sin servicio a varias empresas importantes a nivel mundial durante varias horas [7].

Actualmente se ha detectado una variante de Mirai llamada Sora que usa el protocolo SSH para su propagación [8].

Figura 1.1: Dispositivos SSH registrados en Shodan

Aunque en 2002 se puso en producción el primer honeypot y han seguido desarrollándose y actualizándose, actualmente no está muy extendido su uso para analizar datos e identificar nuevos patrones de ataque, pero esta tendencia está cambiando y cada vez se están realizando más investigaciones al respecto.

1.2. Objetivos

El principal objetivo de este trabajo es diseñar e implementar un sistema capaz de identificar y clasificar ciberataques, para lo cual se utilizan logs obtenidos del honeypot SSH Cowrie. Estos logs serán almacenados en la base de datos NoSQL (Elasticsearch) y se clasifican los ataques usando técnicas de Machine Learning. Para llevar a cabo este proyecto, es necesario que se cumpla una serie de objetivos. Son los siguientes:

1. Estudio inicial de las tecnologías involucradas y del estado del arte relacionado.
2. Instalar y configurar el honeypot SSH Cowrie; una vez realizado esto, exponerlo a internet con la intención de que reciba ataques y se obtengan datos de estos.
3. Estudio de los logs generados por Cowrie para analizar cuáles son las características más relevantes de estos.
4. Creación de una herramienta capaz de convertir esos datos no estructurados en datos estructurados (formato JSON) para que puedan ser almacenados en una base de datos.
5. Investigar cuál es la base de datos más adecuada para los requisitos de este proyecto e insertar los datos en esta.
6. Clasificación manual de los datos almacenados en la base de datos para obtener los datos más relevantes y mostrarlos con distintas tablas y gráficas para un primer análisis de resultados.
7. Selección de las features más discriminantes para aplicar en Machine Learning.
8. Selección y uso de técnicas Machine Learning para la clasificación de los datos usando el dataset generado y el de scriptzteam [9].
9. Análisis de los resultados obtenidos mediante técnicas de Machine Learning.

1.3. Estructura del documento

Este documento se encuentra definido en cuatro partes claramente diferenciadas, además de esta introducción. Estas son las siguientes:

- En el capítulo 2 (Background y estado del arte) presentamos las principales tecnologías que intervienen en este Trabajo Fin de Grado y las investigaciones con los resultados obtenidos de estas.
- En el capítulo 3 (Análisis de objetivos y metodología) presentamos cuáles son los objetivos de este Trabajo Fin de Grado y las herramientas que se han utilizado para llevarlo a cabo.
- En el capítulo 4 (Diseño y resolución del trabajo realizado) presentamos cuál ha sido el escenario que se ha diseñado, el software que se ha desarrollado, la clasificación manual que se ha realizado con los datos y, por último, las pruebas realizadas con Machine Learning a esos datos para intentar clasificarlos.
- En el capítulo 5 (Conclusiones y vías futuras) presentamos cuáles han sido las conclusiones obtenidas después de realizar este Trabajo Fin de Grado y posibles vías futuras para su continuación.

2. Background y estado del arte

Uno de los requisitos del reto en el que se ha participado en el Track de Transferencia de las JNIC es realizar un análisis de los logs generados por el honeypot SSH Cowrie. Por este motivo se va a realizar un análisis de cómo funcionan, los tipos que existen y los resultados obtenidos con sus datos.

El análisis de logs de honeypots para entender cuáles son los objetivos de los atacantes es algo muy valioso, pero todavía no está totalmente extendida su investigación. A nivel nacional, se han encontrado muy pocos artículos científicos que hablen de este tema, aunque sí se ha detectado una creciente subida en cuando al número de TFGs en universidades sobre este tema. A nivel internacional, por el contrario, sí que se han realizado mayor cantidad de investigaciones usando honeypots.

2.1. Honeypots

Se puede definir un honeypot como un sistema cuya finalidad es atraer a posibles atacantes simulando que es un objetivo con cierto valor [10]. Se pueden usar como un Sistema de Detección de Intrusiones (IDS) para detectar y alertar de posibles ataques o como una herramienta de la red que consigue distraer al atacante en un sistema que no tiene ningún valor para él, pero estas no son las tareas para las que ha sido diseñado. Su principal tarea es ser usado para obtener información del atacante, como puede ser cuáles son las herramientas utilizadas o su objetivo como vectores de ataque.

Según Joshi and Sardana [11], el principal valor de un honeypot es que sea comprometido para que los investigadores puedan analizar los ataques realizados e intentar adelantarse a futuros ataques. Por este motivo nunca puede tener información relevante (como pueden ser contraseñas) o, si las tiene, han de ser falsas.

2.1.1. Clasificación de los honeypots

Según Melese and Avadhani [12] podemos clasificar los honeypots de dos formas: según su nivel de interacción y según su propósito. A continuación, se explican ambas.

Clasificación según su nivel de interacción

En la clasificación según su nivel de interacción [13, 14] se pueden encontrar tres tipos bien diferenciados: baja interacción, media interacción y alta interacción. Antiguamente

se decía que había únicamente dos tipos, y a los de media interacción se los conocía como híbridos, pero actualmente ese término ya no se usa.

- **Honeypot de baja interacción** (LIH): Este tipo de honeypot suele simular únicamente servicios, por lo que la actividad del atacante es muy limitada y, por ende, la información que puedes obtener de él también lo es. Por ejemplo, pueden simular un servicio FTP para simular el proceso de logueo pero no permitir ningún otro tipo de interacción. Como ventaja, está la simplicidad de la configuración y el bajo consumo de recursos. Como desventaja, está que son fácilmente detectables por sus limitaciones.
- **Honeypot de media interacción** (MIH): Este tipo de honeypot es una mezcla entre los de baja interacción y los de alta interacción. Tienen menos limitaciones que los de baja interacción, pero siguen basándose en simular el servicio o sistema operativo. Por ejemplo, un servicio SSH que permite el logueo y la ejecución de una serie de comandos. Como ventaja, tenemos que es capaz de recoger mayor cantidad de datos que los de baja interacción con una configuración relativamente sencilla y que no supone el compromiso del resto de hosts del sistema de producción, al ejecutarse habitualmente en un entorno virtual. Como desventaja, tenemos que siguen siendo sistemas simulados y están limitados.
- **Honeypot de alta interacción** (HIH): Este tipo de honeypot es el más complejo que existe. No limita al atacante en ningún aspecto al usar sistemas operativos y servicios reales. Es el que permite al atacante un mayor nivel de interacción y, por ende, el más peligroso. Como ventaja, nos encontramos conque podemos recolectar una gran cantidad de información. Como desventaja, encontramos la complejidad de configuración, que es muy elevada y, ante una mala configuración, un atacante puede usar ese honeypot como herramienta de ataque interno. También suele tener un coste mayor porque requiere mayor hardware al tener que usar sistemas operativos reales.

Clasificación según su propósito

En la clasificación según su propósito, se dividen los honeypots en dos tipos: los de investigación y los de producción; el primero de ellos es el más usado actualmente, aunque esto puede cambiar en un futuro cercano.

- **Los honeypots de investigación** son usados principalmente en entornos académicos, aunque también los usan entornos militares y empresas dedicadas al sector de la seguridad informática. La finalidad de estos honeypots es recopilar la mayor cantidad de información y lo más detallada posible, así como patrones de ataques y amenazas. Después, los investigadores usan esta información para analizar la situación actual de los ataques e intentar adelantarse a próximos tipos de ataques. Según Casanovas and Tapia [15], un problema de éstos es que son desarrollados durante una tesis u otro proyecto académico y, una vez terminado el proyecto, es descontinuado, haciendo que haya gran cantidad de honeypots

obsoletos.

- **Los honeypots de producción** son usados por empresas como mecanismo de detección de intrusos, aunque también buscan que un posible atacante se distraiga con éste. Lo más importante es que desde el honeypot no se puedan realizar ataques contra otros dispositivos de la red. Suelen estar disponibles los 365 días de año, algo que no sucede con los de investigación. Cada vez tienen mayor importancia gracias a lo útil que es la información que capturan.

2.1.2. Ventajas y desventajas

Las principales ventajas que tienen los honeypots son las siguientes:

- Son sistemas fáciles de configurar y desplegar con el fin de obtener información valiosa.
- Sirve para detectar ataques tanto dentro como fuera de la organización.
- No reciben falsos positivos, ya que interpretan cada nueva conexión como un ataque, dado que nadie debería de acceder a ellos.
- En general, los recursos que hay que asignar son mínimos, tanto el hardware sobre el que se despliega como el consumo de red.

Las principales desventajas que tienen los honeypots son las siguientes:

- Es un elemento pasivo, por lo que, si no reciben ataques, no tienen ninguna utilidad.
- No añaden seguridad a la red de forma activa.
- Es un elemento de riesgo para la red, ya que pueden utilizarlo como pivoting para atacar.

2.1.3. Tipos de honeypots

Existen multitud de honeypots que se suelen clasificar según el servicio que simulan; algunos de los más conocidos son los siguientes:

- **Glastopf**: Este es un honeypot de baja interacción [16] usado para aplicaciones web. Es capaz de simular vulnerabilidades y recopilar información sobre los ataques recibidos. Se centra únicamente en simular tipos de ataque en vez de aplicaciones web, haciéndolo más versátil.
- **Artemisa**: Este es un honeypot VoIP de baja interacción [16] y usa el protocolo SIP. Funciona a modo de cliente que se conecta a una centralita real. Cuando recibe una llamada, registra la información, la analiza y clasifica el ataque recibido.
- **Conpot**: Este es un honeypot de baja interacción [17] diseñado para simular ser un sistema de control industrial, como pueden ser Industrial Control System (ICS) y Supervisory Control And Data Acquisition (SCADA). Es fácilmente modificable y ampliable.

- **Kippo**: Este es un honeypot SSH de media interacción [1], diseñado para registrar ataques de fuerza bruta y la interacción de comandos a través de SSH. Proporciona un sistema de archivos falso con capacidades de crear y eliminar ficheros. Su desarrollo está descontinuado.
- **Cowrie**: Este es un honeypot SSH de media interacción [1] basado en Kippo. Ha continuado el proyecto extendiendo el software implementado con nuevos comandos y nueva funcionalidad implementada a través de plugins, como es la salida de logs en formato JSON o enviarla a una base de datos directamente.
- **ThingPOT**: Este es un honeypot de media interacción [18] que implementa los protocolos XMPP y HTTP usados en IoT. Imita un dispositivo IoT para recibir ataques y registrarlos. Fue creado después de la aparición del bot Mirai que usó dispositivos IoT para realizar un ataque DDoS.

2.1.4. Datos obtenidos de los honeypots SSH

Sadasivam et al. [19] indican que la IP del atacante no tiene por qué ser fiable por sí misma; es necesaria más información, ya que la dirección IP del atacante puede ser su dirección IP, la IP de una víctima del atacante o un servidor proxy.

Según Sadasivam et al. los mismos ataques con IPs dentro de un mismo rango de direcciones tienen una baja probabilidad de recibirse, lo que permite una detección precisa de estos ataques. Por el contrario, los ataques distribuidos con diferentes rangos de direcciones son más comunes, y para detectarlos se usan las siguientes features:

- Mismo Sistema Operativo (OS) utilizado por el atacante.
- Misma librería SSH.
- Mismo lenguaje de programación.
- Mismo algoritmo de cifrado.
- Mismos pares de usuario/contraseña utilizados.
- Peticiones realizadas en una ventana de tiempo cercana.

En los datasets obtenidos por Sadasivam et al. el 65% de los ataques provenían del continente asiático, tenido China el 61%, seguido de EE. UU. con el 19%. El cliente SSH más usado ha sido *SSH-2.0-PUTTY*. Se han considerado ataques de fuerza bruta todas aquellas sesiones que han realizado más de 3 intentos de login en una ventana de tiempo de 10 minutos.

ElevenPaths ha obtenido que el protocolo que más tráfico recibe es SSH [20]. La mayoría de esos ataques han sido usando un diccionario de credenciales.

Según ElevenPaths, los clientes SSH más utilizados son *libssh2* (10267 ocasiones), *libssh* (10264), *PuTTY* (4458) y *JSCH* (106). A partir del cliente SSH se puede obtener una relación con el lenguaje de programación utilizado en el script o en la botnet que ha realizado el ataque; por ejemplo, el cliente SSH *Paramiko* usa Python, *JSCH* usa Java y *libssh* usa C. También existen otros clientes SSH más atípicos, como *MEDUSA*, que está asociado a una herramienta de pentesting (en este caso el escáner de fuerza

bruta llamado Medusa).

ElevenPaths al igual que Sadasivam et al. [19] han comprobado que las direcciones IP que mayor cantidad de ataques tienen provienen de China y EE. UU. respectivamente. Han realizado un análisis de las credenciales utilizadas para ver si se puede obtener información de estas. La mayoría de los ataques han ido con las credenciales *root* y *admin* que están presentes en los sistemas UNIX, después han usado otras como *user*, *ubnt* y *pi* usadas en routers, dispositivos Ubiquiti y la Raspberry Pi, respectivamente.

Por último, en ElevenPaths han detectado que no se realizan ataques de fuerza bruta, sino que intentan buscar sistemas mal configurados con las credenciales por defecto.

Otro análisis de los ataques ha sido realizado por Kheirkhah et al. [21], han realizado un análisis de los ataques, obteniendo la siguiente información:

- El sistema operativo más usado por los atacantes es Linux con el 82%. Se infiere que uno de los motivos es porque permite mayor libertad para interceptar conexiones de red.
- Europa es el continente con mayor cantidad de logins satisfactorios (50,6%); después van América del Norte (15,4%) y Asia (19,1%).
- América del Norte es el continente con mayor cantidad de logins incorrectos (34,7%).

Kheirkhah et al. dividen los logs generados en tres categorías:

- Solamente inicio de sesión, que puede tener como objetivo probar las credenciales o volver más tarde.
- Ejecución de comandos básicos para obtener información del sistema, como pueden ser los comandos: *w*, *uname*, *uptime* o *cpuinfo*.
- Descarga y ejecución de malware, siendo este grupo el más peligroso.

Los comandos que más se han ejecutado en su honeypot según Kheirkhah et al. son los siguientes: *w* (13,61%), *uname -a* (9,47%) y *wget* (5,68%).

Kheirkhah et al. indican que una de las mayores desventajas de los honeypots es la inadaptabilidad del sistema, ya que se necesitan aprender nuevos comportamientos a partir de las interacciones de los atacantes, haciendo que siempre vayan un paso por detrás.

Por otro lado, Dowling et al. [22] identifican los tipos de ataque que han registrado en el honeypot SSH Kippo. Estos son los siguientes:

- Ataques de diccionario.
- Ataques de fuerza bruta.
- Scripts de reconocimiento.
- Botnets.
- Ataques lanzados.
- Ataques individuales.

De todas las sesiones registradas por Dowling et al. el 94% fueron ataques por diccionario. Cada ataque por fuerza bruta trae consigo un ataque de diccionario. Los ataques individuales usan IPs aleatorias, mientras que los ataques lanzados usan IPs

recurrentes. Además, los ataques individuales tienen errores en la escritura y pausas entre comandos.

Finalmente, para Dowling et al. los comandos más ejecutados por el grupo de las botnets que han analizado son los siguientes: *iptables stop*, *killall -9*, *wget*, *chmod 777* y *rm -rf*.

Las investigaciones analizadas tienen un inconveniente, y es que se centra únicamente en un pequeño conjunto de datos, ya que suponen que son los más relevantes. Kheirkhah et al. [21] es el que realiza el análisis más completo, llegando hasta a analizar el malware descargado por los atacantes, pero tiene el inconveniente de que únicamente recoge información durante 7 semanas.

El objetivo que se abordará en este Trabajo Fin de Grado, será ser capaces de generar logs usando el honeypot SSH Cowrie (requisito del reto *IN1. Clasificación de ataques a honeypots SSH* propuesto por INCIBE, almacenándolos en una base de datos que nos permitirá su posterior análisis, con el objetivo secundario de ser capaces de continuar el análisis en tiempo real mientras se reciben nuevos logs procedentes de otros honeypots. Con este se conseguirá un análisis de los ataques en tiempo real.

2.2. Técnicas de Machine Learning

Uno de los requisitos del reto en el que se participa es realizar un sistema capaz de clasificar los ataques recibidos, para esto se van a usar distintas técnicas de Machine Learning con el objetivo de clasificar los distintos tipos de ataques recibidos.

El Machine Learning (ML) [23], [24], en español Aprendizaje Automático, es un subcampo de las ciencias de la computación y una rama de la Inteligencia Artificial. Su objetivo es construir programas que mejoren automáticamente en base a una experiencia previa que se va adquiriendo conforme pasa el tiempo. Estos programas acaban siendo capaces de aprender gracias a los datos y ser capaces de reconocer patrones y tomar decisiones sin intervención humana.

En general, la mayoría de algoritmos de Machine Learning reutilizan una serie de algoritmos ya diseñados; estos son los siguientes:

- Algoritmos de Regresión: buscan modelar la relación que existe entre distintas variables. Usan una medida de error que se buscará reducir durante un proceso iterativo con el fin de conseguir unas predicciones más optimas.
- Algoritmos basados en Instancia: buscan crear un modelo de una base de datos en la que se añaden nuevos datos, realizando comparaciones con los datos ya existentes para encontrar la mejor pareja existente y hacer predicciones.
- Algoritmos de Árbol de Decisión: buscan tomar decisiones de forma muy rápida usando los valores reales que tienen los atributos de los datos.
- Algoritmos de Clustering: buscan agrupar datos existentes (usando puntos centrales) en los que no se conocen sus características comunes o se necesitan saber.

- Algoritmos de Reducción de Dimensión: buscan reducir o comprimir de forma no supervisada el número de dimensiones que tienen los datos para así simplificar su visualización.

A continuación, se van a presentar los dos tipos de aprendizaje automático que existen en Machine Learning, que son aprendizaje supervisado y no supervisado. De estos se va a inferir la evaluación del modelo diseñado en base a una fase de entrenamiento con los datos proporcionados.

2.2.1. Aprendizaje supervisado

El aprendizaje supervisado está compuesto por un conjunto de algoritmos que trabajan con datos etiquetados. Esta etiqueta permite clasificar el tipo de dato que es. Un ejemplo habitual es el uso de correo entrante, en el que los correos pueden ir etiquetados como spam o no.

Los principales algoritmos que crean este bloque son los siguientes:

Decision Trees (DT): Este es uno de los algoritmos más utilizados en Machine Learning para clasificación y regresión. Se basa en el algoritmo de regresión explicado anteriormente y su objetivo es crear un modelo que pueda predecir el valor de una variable usando reglas de decisión simples inferidas de los datos [25].

Ventajas:

- Son simples de entender e interpretar. También pueden ser visualizados.
- Requieren poca preparación de los datos que se usan una vez que se encuentran etiquetados. Otras técnicas requieren la normalización de datos, siendo este un proceso más costoso.

Inconvenientes:

- Los hijos pueden crear árboles muy complejos, para los que se puede necesitar una poda.
- El problema de aprender un Decision Tree óptimo es NP-completo.

Random Forest: Este es un algoritmo usado para clasificación, aunque también puede llegarse a usar para problemas de regresión. Cada árbol se crea a partir de muestras extraídas del conjunto de entrenamiento. Intenta resolver el problema de que el algoritmo "memorice" (también conocido como overfiting[1]) las soluciones en lugar de aprender [26].

Ventajas:

- Tiene un buen funcionamiento sin ajuste de hiperparámetros.
- Usando múltiples árboles se reduce en gran cantidad el overfiting.

[1] "En aprendizaje automático, el sobreajuste (overfitting) es el efecto de sobreentrenar un algoritmo de aprendizaje con unos ciertos datos para los que se conoce el resultado deseado". Fuente: Wikipedia

Inconvenientes:

- Tiene un coste de tiempo mucho mayor que crear un Decision Tree.
- No funciona bien con datasets pequeños.

k-Nearest-Neighbor (k-NN): Este se basa en los algoritmos basados en instancia. Busca realizar una clasificación basándose en la distancia con los vecinos más cercanos, usando como referencia la distancia euclidiana estándar. Se suele aplicar para detectar anomalías [27].

Ventajas:

- Es sencillo de aprender e implementar.
- No necesitas conocer k (cantidad de puntos vecinos) previamente.

Inconvenientes:

- Requiere todo el dataset para el entrenamiento de cada punto.
- Consume mucha memoria y CPU.

Support Vector Machines (SVM): Es un algoritmo usado para la clasificación, regresión y detección de valores atípicos. Con un conjunto de ejemplo de entrenamiento se pueden etiquetar las clases y entrenar una SVM capaz de crear un modelo que predice la nueva clase de la muestra [28].

Ventajas:

- Funciona en espacios con muchas dimensiones.
- Es versátil, ya que permite especificar diferentes funciones de kernel.

Inconvenientes:

- No proporciona directamente la probabilidad, sino que es necesario calcularla con un costoso proceso de validación.
- El número de características es mayor que el de muestras.

2.2.2. Aprendizaje no supervisado

El aprendizaje no supervisado está compuesto por un conjunto de algoritmos que trabajan con datos sin etiquetar, siendo el algoritmo el que intente etiquetarlos. Sus principales ventajas son:

- No necesita un agente externo que etiquete los datos.
- Suele requerir menor tiempo de entrenamiento que el aprendizaje supervisado.
- La arquitectura es más simple y habitualmente solo tienen una capa.

k-means: Este se basa en los algoritmos de clustering. Se usa cuando se dispone de una gran cantidad de datos no etiquetados y se necesita saber las posibles relaciones que hay entre ellos, si es que existen. Se usa principalmente para descubrir grupos ocultos [29].

Ventajas:

- Su implementación es sencilla.
- Es muy eficiente dentro de los algoritmos de clustering.

Inconvenientes:

- El resultado varía dependiendo de las semillas utilizadas al inicio.
- Necesitas conocer k (cantidad de grupos) previamente.

Principal Component Analysis (PCA): Es un algoritmo de clustering. Funciona extrayendo las características más relevantes y eliminando las de menor valor. Al eliminar variables conseguimos una reducción de dimensiones [30].

Ventajas:

- Al reducir dimensiones es capaz de mantener la información útil de todas las variables iniciales.

Inconvenientes:

- Cada componente principal es una combinación lineal de todas las variables originales.
- Es muy influenciado por los valores atípicos de los datos.

t-Distributed Stochastic Neighbor Embedding (t-SNE): Es una alternativa a PCA con un enfoque distinto. Convierte las afinidades de los puntos de datos en probabilidades, haciendo que sea sensible a la estructura local [31].

Ventajas:

- Muestra la estructura de muchas escalas en un único mapa.
- Reduce la tendencia a acumular puntos en el centro.

Inconvenientes:

- Es computacionalmente mucho más costo que PCA.
- El algoritmo es estocástico, haciendo que los reinicios con diferentes semillas den datos diferentes.

Uniform Manifold Approximation and Projection (UMAP): Es una nueva técnica que se basa en la reducción de dimensiones. Se puede usar de forma similar a t-SNE, pero también es capaz de realizar reducciones en dimensiones no lineales [32].

Ventajas:

- Es mucho más rápido que t-SNE.
- A diferencia de t-SNE, que solo se usa para la visualización, UMAP tiene capacidad de reducción de dimensiones.

Inconvenientes:

- Es una técnica bastante reciente, por lo que se puede considerar todavía inmadura.
- Tiene pocas librerías y no son totalmente estables.

2.2.3. Técnicas Machine Learning aplicadas a logs de honeypots

A continuación, se van a explicar algunos resultados obtenidos en investigaciones relacionadas con aplicar técnicas de Machine Learning a logs generados por honeypots.

Sadasivam et al. [33] hacen una diferenciación de los ataques recibidos en dos grupos, que son los siguientes:

- **Ataques severos**, son aquellos que han ejecutado uno o más comandos, ya que ponen en riesgo al sistema.
- **Ataques no severos**, son aquellos que no han ejecutado ningún comando. Estos pueden ser por logins fallidos (ataques de fuerza bruta), logins correctos (comprobación de credenciales) o un escaneo de puertos.

Las features que Sadasivam et al. consideran relevantes para diferenciar los ataques son:

- **Ventana de tiempo** entre paquetes de llegada, siendo aproximadamente de 10 minutos.
- Tamaño del **payload** del paquete.

Los algoritmos de Machine Learning que Sadasivam et al. han usado en la investigación, y que ofrecen mejores resultados para el estudio, han sido:

- **J48**, algoritmo usado para generar un Decision Trees y usados para la clasificación.
- **PART**, usado como clasificador basado en reglas.
- Ambos comparten las siguientes precisiones: accuracy 0.999, recall 0.928, precision 1.00, F2-Score 0.942.

Por otro lado, Sharafaldin et al. [34] han analizado todo el tráfico TCP entrante en sus honeypots con el objetivo de obtener un conjunto de features que le permitan detectar distintas categorías de ataques. Esta investigación la han realizado para el Instituto de Ciberseguridad de Canadá. Los autores de este trabajo han han desplegado un escenario de grandes prestaciones en el que realizan los ataques. Los atacantes realizaban los ataques con IPs públicas y diversos sistemas operativos y para las víctimas montaron infraestructuras de seguridad con routers, firewalls, servidores y estaciones de trabajo.

Sharafaldin et al. realizaron los diversos ataques según el día de la semana con el objetivo de tener así etiquetados los datos por el día.

- Lunes: tráfico benigno.
- Martes: tráfico SSH, SFTP y BForce.
- Miércoles: tráfico generado por ataques DoS.
- Jueves: ataques web e infiltración.

- Viernes: tráfico generado por ataques de botnets y tráfico generado por ataques DDoS.

Con estas pruebas, Sharafaldin et al. generaron 5 grupos de ataques y, usando los algoritmos de Machine Learning RF, k-NN e ID3, obtuvieron los resultados satisfactorios que se muestran en la Tabla 2.1.

Algoritmo	**Precision**	**Recall**	**F1-Score**	**Tiempo**
ID3	0,98	0,98	0,98	235 seg
RF	0,98	0,97	0,97	74 seg
k-NN	0,96	0,96	0,96	1908 seg

Tabla 2.1: Medidas de rendimiento (Fuente: Sharafaldin et al.)

Otro de los trabajos a destacar es el propuesto por Pauna and Bica [35], donde proponen una honeypot auto-adaptativa a comportamientos de los atacantes. Usa un honeypot de media interacción basado en Python que emula un servidor SSH. Usa una variante de aprendizaje por refuerzo para ser auto-adaptativa. Las mejoras que trae son las siguientes:

- Mayor escalabilidad.
- Mayor localización.
- Mayores capacidades de aprendizaje.

Finalmente, Dowling et al. [36] intenta evitar que los atacantes puedan detectar que se encuentran en un honeypot. Las mejoras que esto trae son las siguientes:

- Mayor cantidad de ejecución de comandos.
- Usa aprendizaje por refuerzo.
- Hace recomendaciones de cambio en el desarrollo del honeypot para permitir mayor interacción con el atacante.

Las investigaciones realizadas han obtenido buenos resultados, pero tienen una serie de inconvenientes. El primero es que algunas han partido de datos previamente etiquetados, esto es un inconveniente porque el etiquetado se ha realizado usando ataques ya conocidos, y no valdría para nuevos tipos de ataques. Otro inconveniente es que realizan unos grupos demasiado amplios a la hora de clasificar, por ejemplo, ataques severos y no severos.

El objetivo que se abordará en este Trabajo Fin de Grado será ser capaces de realizar una clasificación más específica sin un etiquetado previo de los distintos tipos de ataques.

2.3. Bases de datos para almacenar logs de honeypots

Un requisito del reto en el que se participa, es ser capaz de almacenar la información con el objetivo de poder consultarla posteriormente en las siguientes fases del análisis. Esto se conseguirá utilizando una base de datos.

Para almacenar los logs generados por honeypots disponemos de dos tipos de bases de datos (DB): las bases de datos relacionales (SQL) y las bases de datos no relacionales (NoSQL). No hay un tipo de base de datos mejor que otro, sino que cada uno tiene sus características, las cuales vamos a ver a continuación [37]:

Bases de datos relacionales: Estas bases de datos cumplen el modelo relacional, que significa que los datos están relacionados entre sí mediante identificadores. Usan como lenguaje Structured Query Language (SQL).

Ventajas:

- Atomicidad en las operaciones: garantiza que una operación no se complete si no se realiza al 100%.
- Madurez, ya que lleva en uso desde los años 80.
- Estándar bien definido para cualquier operación.

Inconvenientes:

- Escalabilidad al crecer: requieren muchos recursos para mantener el rendimiento de una única máquina.
- Cambios de estructura de los datos: requieren una modificación de las relaciones, lo que puede ser muy costoso.

Bases de datos no relacionales: Estas bases de datos no cumplen el modelo relacional, lo que significa que los datos no están relacionados entre sí. No usa SQL como lenguaje y se suelen almacenar en formato JSON.

Ventajas:

- Flexible frente a cambios, ya que la información va en formato JSON y se puede modificar éste sin afectar a la configuración de la base de datos.
- Escalabilidad horizontal: permite crecer en cantidad de máquinas y no es una única máquina muy potente.
- Optimización de las consultas para grandes cantidades de datos.

Inconvenientes:

- Atomicidad en los nodos-réplica puede no ser consistente; este problema puede solucionarse parcialmente con la consistencia eventual[2].

[2]La consistencia eventual es una técnica para mejorar el rendimiento que permite propagar los cambios realizados al resto de nodos "con el tiempo", permitiendo que en un momento dado una consulta arroje un valor desactualizado, conocido como lectura sucia.

- Falta de estandarización, ya que es una tecnología reciente y cada fabricante la desarrolla según sus necesidades.

Las bases de datos relacionales más usadas en la actualidad son MySQL y Oracle, seguidas por SQL Server y PostgreSQL. La base de datos no relacionales que están teniendo más éxito en la actualidad son MongoDB, seguida por Redis, Elasticsearch y Cassandra.

Si analizamos los requisitos de este proyecto respecto a las ventajas y desventajas de los distintos tipos de bases de datos mencionados en la sección 2.3, se pueden identificar los siguientes requisitos:

- Se necesitará un sistema flexible frente al cambio, ya que Cowrie es un sistema de desarrollo y puede modificar los JSON o crear nuevos (como ha pasado durante el desarrollo de este proyecto).
- Escalabilidad horizontal, ya que es interesante poder desplegar una red de honeypots (honeynet) que vaya enviando los logs a distintos clústers.
- Optimización de las consultas para grandes cantidades de datos, ya que las únicas operaciones que se realizan son inserción y consultas de datos.
- La atomicidad no es importante, puesto que solo se harán búsquedas y la consistencia eventual es suficiente.

Con esta serie de requisitos se aprecia que la BD más adecuada para nuestro escenario es del tipo NoSQL, ya que sus ventajas resultan beneficiosas mientras que sus inconvenientes no nos perjudican. Una vez decidido el tipo de BD que se va a usar (NoSQL), hay que elegir cuál se utilizará dentro del mismo.

Finalmente se ha decido usar Elasticsearch [38]. Este es un gestor de datos y motor de búsqueda basado en Apache Lucene desarrollado en Java, diseñado para operar con una gran cantidad de datos en tiempo real. Los principales motivos que ha llevado a su uso son que Cowrie, en un futuro, lo soportará de forma oficial a través de un plugin que permite enviar los datos directamente. Además, dentro del ecosistema de Elasticsearch se encuentra Kibana [39], una herramienta para la visualización de datos a través de tablas y gráficas usado habitualmente con las que poder realizar un análisis visual de los datos. Ambos forman parte de la suite ELK.

2.4. Conclusiones del estado del arte

Como se puede ver en la Tabla 2.2, el análisis de los datos generados por honeypots está realizándose, aunque normalmente ese análisis no suele utilizar todos los datos, sino que solo se centran en algunos de ellos sin obtener una visión global de todos. El análisis del malware descargado no es algo muy habitual que se haga, aunque esto puede ser debido a que algunos honeypots no permiten las descargas. Por otro lado, almacenar la información en una base de datos no lo ha hecho ninguno o, si lo ha hecho, no lo ha indicado. Esto se puede deber a que hagan análisis estático sobre los datos. Finalmente, aplicar técnicas de Machine Learning se ha realizado obteniendo en

algunos casos el éxito esperado, pero en otros no, en los que se han realizado, no se han acompañado de análisis manuales de esos datos.

	Credenciales	**Comandos**	**geoinfo**	**malware**	**ML**	**BD**
[33]	No	No	No	No	Sí	No
[19]	No	No	Sí	No	No	No
[34]	No	No	No	No	Sí	No
[20]	Sí	No	Sí	No	No	No
[21]	Sí	Sí	Sí	Sí	No	No
[35]	No	No	No	No	Sí	No
[36]	No	No	No	No	Sí	No
[22]	No	Sí	No	No	No	No

Tabla 2.2: Comparación de los diferentes artículos investigados.

El objetivo de este Trabajo Fin de Grado va a ser realizar todas las tareas mostradas y analizadas a lo largo de esta sección, buscando obtener la mayor cantidad de información de los atacantes en forma de análisis manual, información que haya sido generada por logs registrados por el honeypot SSH Cowrie, y posteriormente, usar Machine Learning para intentar detectar información relevante que pueda haberse pasado por alto, almacenando toda la información en una base de datos adecuada para los requisitos.

3. Análisis de objetivos y metodología

En esta sección se explican cuáles son los objetivos del trabajo, así como las herramientas que se han utilizado para llevarlo a cabo.

El principal objetivo de este trabajo es diseñar e implementar un sistema capaz de identificar y clasificar ciberataques, para lo que se usan logs obtenidos del honeypot SSH Cowrie. Estos logs serán almacenados en la base de datos Elasticsearch (NoSQL) y se clasifican los ataques usando técnicas de Machine Learning. Este objetivo se ha dividido en los siguientes subobjetivos:

1. Instalación y configuración del honeypot SSH Cowrie, junto con su exposición a internet con la consiguiente generación de logs.
2. Estudio inicial de los logs en crudo.
3. Creación de un Parser para la conversión de los logs en crudo a un formato estructurado, como es JSON.
4. Instalación y configuración de la base de datos NoSQL Elasticsearch para el almacenamiento de los logs en JSON como salida del Parser anterior.
5. Clasificación manual de los datos almacenados en la base de datos usando Kibana para la generación de gráficas.
6. Selección y uso de técnicas Machine Learning para la clasificación de los datos.
7. Análisis y discusión de los resultados obtenidos mediante técnicas basadas en Machine Learning.

El orden de la siguiente explicación, en base a los subobjetivos anteriores, define los distintos pasos que se han ido realizando como parte de una metodología incremental, los cuales se han ido refinando a lo largo de su desarrollo según los resultados intermedios obtenidos en cada uno de ellos. Se va a proceder a explicar más detalladamente cada uno de estos subobjetivos.

1. Instalación y configuración del honeypot SSH Cowrie

Se va a proceder a instalar y configurar el honeypot SSH Cowrie, para lo que se usarán dos Raspberry Pi 3 Model B ubicadas en lugares diferentes, teniendo ambas direcciones IP diferentes y usando el puerto 22. Para su instalación se va a usar Docker, permitiendo un despliegue independiente del sistema operativo. La configuración se hará buscando que sea lo menos identificable posible como un honeypot.

2. Estudio inicial de logs

Se va a realizar un estudio de los logs para entender su formato, la información que contienen y cuáles de éstos son relevantes para el proyecto.

3. Creación de un Parser

Se va a desarrollar un programa capaz de parsear los logs en crudo a un formato estructurado como es JSON. Además de parsear los logs se le añadirá información que no es capaz de obtener Cowrie usando un sistema experto basado en reglas.

4. Instalación y configuración de una base de datos NoSQL

Se va a instalar y configurar una base de datos capaz de almacenar toda la información suministrada. Se va a usar una de tipo NoSQL ya que se tiene que guardar JSON y se busca que sea capaz de recibir grandes cantidades de datos de una honeynet.

Se ha usado la suite de Elasticsearch como gestor de datos y Kibana como herramienta para la visualización de los datos. Elasticsearch es capaz de almacenar todos los datos y realizar eficientes búsquedas en éstos mientras que Kibana es capaz de acceder este gestor de datos y a través de diversos diagramas mostrarlos, pudiendo actualizar los diagramas en tiempo real en caso de que lleguen nuevos datos.

5. Clasificación manual de los datos

Se va a crear una serie de tablas y diagramas en Kibana para mostrar aquellos datos que puedan resultar interesantes, como son: credenciales, clientes SSH, comandos ejecutados o ficheros descargados.

6. Clasificación de los datos con técnicas de Machine Learning

Se va a realizar una clasificación usando una serie de algoritmos de Machine Learning. Algunos de los algoritmos supervisados que se van a utilizar son: k-Nearest-Neighbor, Support Vector Machines y Random Forest. Algunos de los algoritmos no supervisados que se van a utilizar son: K-Means clustering, t-Distributed Stochastic Neighbor Embedding y Uniform Manifold Approximation and Projection.

7. Análisis y discusión de los resultados obtenidos en Machine Learning

Se va a realizar un estudio de los resultados obtenidos para comprobar si los algoritmos usados han sido efectivos o, por el contrario, no se ha logrado ningún avance mediante estas técnicas.

4. Diseño y resolución del trabajo realizado

En esta sección se expone el escenario diseñado, la configuración que se ha realizado a Cowrie para la generación de logs, el desarrollo del Parser y de una API REST a la que consultar la reputación SSH de una IP y la amenaza que representan las descargas de los atacantes, la configuración de la suite ELK (Elasticsearch y Kibana) para almacenar y visualizar la información, la clasificación de los logs y, finalmente, las pruebas utilizando técnicas de Machine Learning que se han realizado así como el análisis y discusión de sus resultados.

4.1. Escenario diseñado

En esta sección se explica el escenario que se ha diseñado para este trabajo. Aquí se explicará una idea general de este y, más adelante, en cada una de las secciones, se explicará detalladamente cada uno de los componentes que lo integran.

El escenario diseñado, que se puede ver en la Figura 4.1, cuenta con 6 componentes que son los siguientes:

- Cowrie: Es el honeypot SSH, y su función es la de generar logs en base a las interacciones realizadas por los atacantes (por ejemplo, ejecución de comandos en el terminal) para posteriormente ser analizados. En la figura se puede ver que está conectado con Elasticsearch, aunque se permita su conexión, en la actualidad no existe tal conexión ya que no funciona el plugin para ello, la idea es que en un futuro cuando el plugin de conexión con Elasticsearch funcione poder enviarle los datos directamente, evitando así tener que guardarlos localmente en ficheros de texto para luego enviarlos.
- Elasticsearch: Es el gestor de datos y motor de búsqueda que usaremos para almacenar los datos. Pueden ser uno o varios nodos, aunque en este caso únicamente será uno, ya que es suficiente así.
- API REST: Es una API programada en Python con una base de datos local que cachea las peticiones que le llegan, te permite tanto obtener la reputación SSH de una IP como obtener de VirusTotal cuántos antivirus detectan un fichero o URL como maliciosa.
- Parser: Este programa tiene como función principal la de parsear los logs que

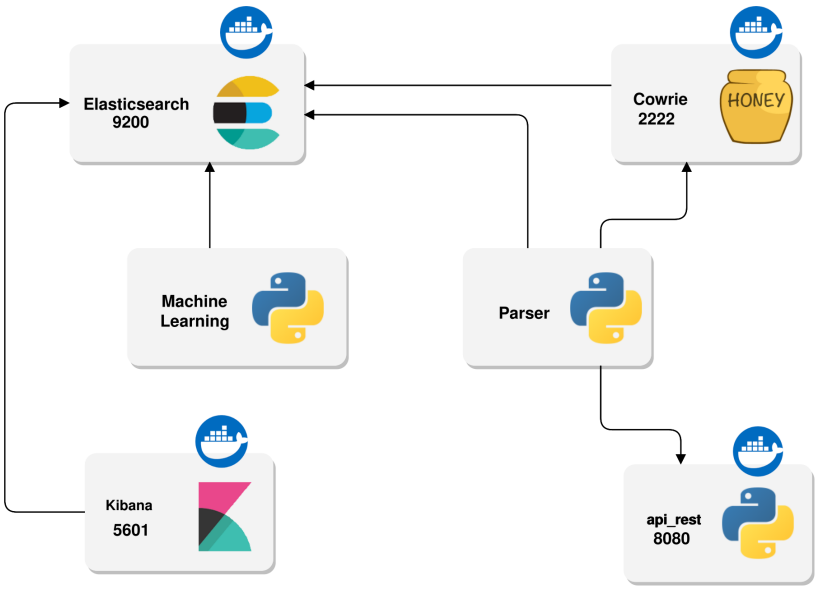


Figura 4.1: Escenario final diseñado

genera Cowrie, comunicarse con la API REST para obtener la información que necesite y, finalmente, enviar los JSON generados a Elasticsearch para su posterior procesado.

- Machine Learning: Este programa implementa los algoritmos de Machine Learning que se han usado, se comunica con Elasticsearch para obtener los datos y genera las gráficas con los resultados obtenidos para su posterior análisis.
- Kibana: Es la plataforma de visualización de datos que se conecta con Elasticsearch, permite crear gráficas y tablas a través de búsquedas muy optimizadas, y permite la actualización de las gráficas en tiempo real conforme llegan nuevos datos.

El escenario se ha desarrollado usando *Docker* para permitir un rápido despliegue en cualquier sistema operativo. Para desplegar este escenario se usa *docker-compose*. Podemos ver el código que se ha usado para el despliegue en el "Anexo I. Escenario Docker".

4.2. Instalación y configuración de Cowrie

En esta sección se explica en detalle cómo son los logs que genera el honeypot SSH Cowrie y la configuración que se ha realizado con la intención de hacerlo menos identificable por los atacantes como un honeypot.

El proyecto de Cowrie al estar activo, va publicando actualizaciones cada cierto tiempo, a día de hoy (*09/07/2019*) que estoy escribiendo este documento, Cowrie va por la

versión 1.6.0, esta ha arreglado varios de los problemas que se han detectado durante la realización de este proyecto y también cuenta con plugins que añaden funcionalidad extra, pero que en este caso al participar en un reto propuesto por INCIBE y contar con logs ya generados de hace tiempo no se puede usar la última versión, sino que se tiene que configurar de forma similar a como se configuró inicialmente para tener unos logs similares.

En el inicio de este desarrollo se trabajó con de Cowrie 1.5.1, ya que era la versión con la que se han generado los logs, pero no tendría que haber ningún problema con futuras versiones mientras mantenga el mismo formato de salida de los logs en crudo (en formato textual). La última versión testeada es la 1.6.0 y funciona todo correctamente.

4.2.1. Instalación de Cowrie

Para ejecutar Cowrie vamos a usar *Docker*. Existe una imagen oficial de Cowrie en Docker Hub, pero ésta tiene configurada en el *Dockerfile* la salida de logs en crudo como salida estándar del sistema (*stdout*), por ese motivo se va a crear un *Dockerfile* usando como plantilla el oficial, pero estableciendo que la salida de logs en crudo no vaya por *stdout*. Podemos ver esta nueva imagen de Cowrie en [40].

Con esta imagen ya se puede crear un contenedor de Docker con el que se desplegará Cowrie en dos dispositivos, para aumentar la cantidad de logs que se obtengan. El contenedor se lanzará con el siguiente comando:

```
1 docker pull procamora/cowrie
```

Cowrie será desplegado en dos dispositivos diferentes: una Raspberry Pi 3 Model B, que estará recogiendo logs durante unos 4 meses de forma ininterrumpida, y un ordenador de sobremesa que no siempre estará encendido.

4.2.2. Estudio de los logs en crudo

En un estudio inicial de los logs se han visto las principales líneas que se necesitan parsear. Los eventos que se tienen que parsear son los siguientes:

- Inicio de una sesión.
- User-Agent del cliente SSH.
- Dirección IP del atacante.
- Algoritmos de intercambio de claves y algoritmos de cifrado de datos.
- Lista de pares usuario/contraseña con el resultado del login *(succeeded/failed)*.
- Lista de comandos utilizados con el resultado de la ejecución del comando.
- Lista de descargas realizadas con su hash y ruta de descarga.
- Fin de la sesión con la duración total.

Pueden ser interesantes para Machine Learning algunas features como el tiempo de sesión, ya que puede haber similitudes entre los ataques según este. También los comandos ejecutados, algoritmos de cifrado y el User-Agent del cliente SSH.

También pueden ser interesantes para la generación de gráficas, y para observar la información común que tienen los atacantes que hemos registrado el User-Agent del cliente SSH, la dirección IP con su información geográfica, la lista de pares de usuario/contraseña, la lista de comandos y los ficheros descargados. Además, se puede realizar un estudio de los ficheros descargados para ver cuál es el propósito de estos (la eliminación de defensas de la máquina como firewall o antivirus, la propagación del malware, el robo de información, etc.).

Podemos ver a continuación una versión simplificada, para su mejor visionado, de los logs que genera Cowrie. En el "Anexo II. Ejemplo de log de Cowrie", se puede ver un ejemplo completo de la salida que genera Cowrie.

Código 4.1: Log simplificado generado por Cowrie

```
1 [CowrieSSHFactory] New connection: 172.22.0.1:59128 (172.22.0.2:2222) [session: ea75ae90eab4]
2 [HoneyPotSSHTransport,1,172.22.0.1] Remote SSH version: b'SSH-2.0-OpenSSH_8.0'
3 [HoneyPotSSHTransport,1,172.22.0.1] login attempt [b'root'/b'test1'] succeeded
4 [HoneyPotSSHTransport,1,172.22.0.1] CMD: wget https://procamora.github.io/code/optimiza_pdf.sh
5 [HoneyPotSSHTransport,1,172.22.0.1] Command found: wget https://procamora.github.io/code/↩
      ↪ optimiza_pdf.sh
6 [cowrie.commands.wget.HTTPProgressDownloader#info] Starting factory <HTTPProgressDownloader: b'↩
      ↪ https://procamora.github.io/code/optimiza_pdf.sh'>
7 [HTTPPageDownloader] Downloaded URL (b'https://procamora.github.io/code/optimiza_pdf.sh') with ↩
      ↪ SHA-256 e0806dff0dd6 to var/lib/cowrie/downloads/e0806dff0dd6
8 [HoneyPotSSHTransport,1,172.22.0.1] Connection lost after 48 seconds
```

Este log de ejemplo corresponde a una conexión que ejecuta el comando *wget* para descargarse un fichero y, posteriormente, cierra la sesión después de 48 segundos. En este ejemplo se puede ver tanto el User-Agent como las credenciales de login utilizadas.

Analizando un poco más en detalle los logs que genera Cowrie detectamos algunos problemas, como puede ser que únicamente está etiquetada la línea de *New connection* con el identificador (ID) de sesión, pero para el resto de líneas usa una combinación de un número seguido de la IP de la conexión. Esto hace más complejo relacionar un ID de sesión con el resto de información de esa sesión. También está el problema de que cuando se realiza una descarga toda la información de esa descarga aparece en una línea sin ningún identificador de a quién corresponde esa descarga. Estos problemas se solucionan cuando la salida se configura en formato JSON. En ese caso, en cada línea se encuentra el *ID* de la sesión a la que corresponde esa información.

Durante este estudio inicial, se habló con el desarrollador principal de Cowrie, Michel Oosterhof, para ver si existía un plan para el futuro de cambiar el honeypot de media interacción a alta para ser capaces de capturar el tráfico de las sesiones y además poder ejecutar comandos en un sistema operativo real no simulado, pero dijo que ahora mismo su objetivo era arreglar todos los bugs que tiene Cowrie e implementar nuevas funcionalidades, pero no llegar hasta ese nivel de complejidad.

4.2.3. Configuración de Cowrie

Debido a que hay que trabajar con unos logs ya generados, y sin posibilidad de modificarlos, se va a proceder a revisar cuál es la configuración que han usado en el reto IN1 propuesto por INCIBE, para usar esa misma configuración, pero creando nuevos logs.

Esta configuración es prácticamente la que viene por defecto y no es adecuada, ya que se puede detectar fácilmente que es un honeypot por los siguientes motivos:

- Tiene el hostname por defecto, que es *srv04*.
- La sesión tiene un timeout de 180 segundos, tiempo que puede ser insuficiente si el atacante es un humano.
- Permite la conexión con distintos usuarios y contraseñas; si usas un diccionario de contraseñas podrás acceder con todas las combinaciones de *root* excepto *root/root*, lo que hace que sea sospechoso.

Debido a estos problemas, se investigará cuál es la mejor configuración posible. Inicialmente se usará esta configuración, pero más adelante se cambiará por la que se considere más correcta con el fin de obtener datos de mejor calidad.

Los logs que se generen con estos dos honeypots se usarán en la etapa inicial del desarrollo del Parser, ya que inicialmente no se tiene acceso a los logs del reto IN1 de INCIBE por motivos de confidencialidad. Después ya se usarán esos logs en los que hay mayor cantidad de datos por haber estado funcionando durante meses.

La configuración que se va a modificar es la siguiente:

Código 4.2: Configuración de Cowrie modificada

```
1 hostname = busybox
2 fake_addr = 10.15.1.168
3 arch = linux-arm-lsb
4 kernel_version = 2.6.11-4-amd86
5 kernel_build_string = #1 SMP Debian 2.6.11-4+deb7u1
6 hardware_platform = x86
7 operating_system = GNU/Linux
8 version = SSH-2.0-OpenSSH_4.2
9
10 interactive_timeout = 1800
11
12 [output_jsonlog]
13 enabled = true
14 logfile = ${honeypot:log_path}/cowrie.json
```

En las líneas 1-8 establecemos información de la máquina que puede consultar el atacante como puede ser el sistema operativo, hostname, IP, versión del kernel y arquitectura. En la línea 10 aumentamos el tiempo total de sesión de 180 a 1800 segundos, tiempo más que suficiente ya que posteriormente se ha calculado el tiempo medio de todas las sesiones y era de 97 segundos, aunque había un 27% de ellas que habían llegado a los 180 segundos por lo que el honeypot les habría desconectado con un timeout. Finalmente, se habilita el plugin que genera los logs en formato JSON ya que, aunque no se usarán en este proyecto, para futuras configuraciones debería de ser la salida por defecto.

Una vez realizados todos estos pasos se procede a desplegar los contenedores de Cowrie en diversos equipos para la generación de logs y su posterior análisis.

4.3. Desarrollo del Parser

En esta sección se explica todo el proceso que se ha llevado a cabo para desarrollar el Parser con su diseño y programación, la API REST también con su diseño y programación y, por último, la suite ELK (Elasticsearch y Kibana) y las configuraciones realizadas para su correcto funcionamiento.

4.3.1. Parser

En esta subsección se explica todo el desarrollo que se ha seguido a la hora de programar el Parser, junto con su funcionamiento y una serie de diagramas de clases y secuencia.

Para poder trabajar con los logs de Cowrie es necesario tratarlos antes y tenerlos en un formato estructurado, en este caso será JSON. Cowrie permite generar los logs con este formato, pero debido a que se ha usado la configuración por defecto no ha sido posible, ya que este plugin no viene habilitado por defecto. Por este motivo ha sido necesario desarrollar un programa para parsear los logs, convirtiéndose éste en una de las temáticas principales de este trabajo.

Inicialmente, se ha investigado si ya existe algún programa capaz de parsear los logs en crudo de Cowrie a un formato JSON o incluso usar el propio Cowrie para que genere el JSON, pero no ha sido posible, ya que no existe ningún programa que haga un parseo de todos los datos; todos los que se han analizado solo obtienen información parcial como puede ser obtener todas las IPs atacantes y los pares de usuario/contraseña, pero en todos se pierde una gran cantidad de información que podría ser bastante relevante. Por esto se ha procedido a desarrollar un Parser que realice principalmente esta función, aunque posteriormente se le añada una mayor funcionalidad, como puede ser la obtención de la información geográfica, reputación SSH de una IP o un sistema basado en reglas para etiquetar las sesiones.

Además, se considera que este Parser puede ser de gran utilidad, ya que en internet se ha encontrado gran cantidad de logs en crudo de Cowrie con los que podremos aumentar la cantidad de información que vamos a analizar. Un ejemplo de alguno de los repositorios de logs que se han revisado y analizado serían los siguientes: el dataset scriptzteam presentado en [9], que es un repositorio automatizado al que se comenzaron a subir los logs en 2017 y tiene 5 repositorios con grandes cantidades de ellos; otro sería el presentado en [41], perteneciente a un administrador de sistemas que subió logs durante el año 2018; y el dataset de Rapid7 [42], que contiene gran cantidad de logs generados por la honeynet de Rapid7's Heisenberg Cloud.

Cabe mencionar que esos logs no son exactamente los mismos que genera actualmente Cowrie, por lo que se ha requerido el desarrollo de un pequeño script que deje los logs como están actualmente. Este script también se explicará más adelante.

Descripción

Para el desarrollo del Parser, el cual puede encontrarse en un repositorio de GitHub [43], se ha usado Python, ya que es un lenguaje muy usado en la actualidad y tiene librerías para cubrir todas las necesidades del proyecto (Elasticsearch y Machine Learning). Se usará la versión 3.7, ya que es la última que hay publicada a fecha de hoy. Cuenta con numerosas mejoras de rendimiento respecto a versiones anteriores [44], algo que resulta crítico en este caso, ya que el tiempo de ejecución es muy importante por la gran cantidad de logs que se van a analizar. Alguna de las características más relevantes que se han usado de las últimas versiones de este lenguaje son las siguientes:

- Tipado estático básico (Type hints), disponible desde Python 3.5. Está definido en el PEP 484 [45]. Se va a usar para definir el tipo estático de los argumentos que recibe una función y su retorno. Esta propiedad hace que Python sea un lenguaje con tipado estático.
- Tipado estático avanzado, disponible desde Python 4.0 pero se pude usar gracias a la librería *___future___*. Está definido en el PEP 563 [46]. Se va a usar para resolver un problema que surge con el tipado estático básico mencionado en el punto anterior, debido a que una clase no puede indicar que retorna un objeto de su misma clase, ya que para el compilador aún no ha sido definida y no la conoce, devolviendo un error.
- f-Strings, disponible desde Python 3.6. Está definido en el PEP 498 [47]. Se van a usar para la creación de string complejos, haciendo que sea más simple y clara su implementación.
- Clases de Datos (Data Classes), disponible desde Python 3.7. Está definido en el PEP 557 [48]. Permite que la representación de una clase esté en su definición.

La principal funcionalidad del Parser consiste en parsear los logs en crudo que genera Cowrie, obteniendo un fichero con todas las conexiones que se han realizado en formato JSON. Es capaz de leer todos los ficheros de un directorio ordenándolos por orden cronológico e ir analizándolos individualmente. Este es un requisito necesario ya que no se sabe la cantidad de logs que se van a analizar, y puede que no haya suficiente memoria RAM para almacenarlos todos en memoria. Las características más relevantes del Parser son:

- Parseo de logs en crudo.
- Recomponer sesiones partidas en ficheros diferentes.
- Obtención de la reputación SSH y geolocalización de las IP registradas (continente, país, coordenadas).
- Etiquetado de las conexiones usando un sistema experto basado en reglas, que etiqueta según la peligrosidad de la conexión, si es un escaneo o si es un ataque de fuerza bruta.
- Salida de la información en formato JSON compatible con el que genera Cowrie por defecto.

- Script para enviar los datos a Elasticsearch con el mapping[1] necesario para cada tipo de datos.
- Análisis de los ficheros descargados usando la API pública de VirusTotal.
- Script para convertir logs antiguos de Cowrie en logs actuales y poder parsearlos.

El proceso del Parser se puede dividir en tres etapas muy diferenciadas, pero antes de explicarlas, se comentarán los ficheros intermedios que se generan para una mejor comprensión, siendo estos los siguientes:

- **cowrie.completed.json**: fichero principal auxiliar que contiene todas las sesiones que se han iniciado y cerrado en ese fichero.
- **cowrie.session.json**: fichero auxiliar con sesiones iniciadas, pero sobre las que no se encontró el cierre de la sesión, por lo que estará almacenado en otro fichero.
- **cowrie.nosession.json**: fichero auxiliar que tiene sesiones que fueron establecidas (iniciadas) en ficheros anteriores al actual, pero que no sabemos en cuál específicamente.

Una vez explicados los ficheros intermedios, se pueden explicar las etapas, que son las siguientes:

Primera etapa: Esta es la etapa más importante del programa; lee en orden cronológico cada fichero del directorio y lo procesa. Este procesamiento consiste en leer todas las líneas y reconstruir todas las sesiones que hay en él. A medida que procesa un fichero, va añadiendo las sesiones encontradas en los tres ficheros mencionados anteriormente. A la hora de generar el JSON de las sesiones, le añade información que se obtiene externamente, como pueden ser los datos geográficos y reputación SSH de la IP o si es un ataque de fuerza bruta por el número de intentos de login realizados. La clase encargada de realizar esta tarea es *Parser*, que explicaremos en la sección 4.3.1 y se basa en expresiones regulares para parsear cada una de las líneas y saber si son o no necesarias; en caso de serlo, obtiene la información correspondiente de esa línea.

Segunda etapa: En esta, tenemos como objetivo buscar aquellas sesiones incompletas. Esto se debe a que se han realizado en el transcurso de dos días diferentes; ya que Cowrie genera un log diferente cada día natural y puede darse el caso de que una conexión esté en dos días diferentes por iniciarse antes de las 00:00 y terminar después.

En esta etapa buscamos para cada conexión del fichero de sesiones iniciadas (fichero *cowrie.session.json*) si existe alguna como el mismo ID de sesión en el fichero de sesiones terminadas (fichero *cowrie.nosession.json*), y en caso afirmativo se añaden al fichero de sesiones completas. Este proceso añade un poco de latencia al programa, pero conseguimos que prácticamente todas las sesiones incompletas dejen de estarlo. La clase encargada de realizar esta tarea es *CompleteSession*, que explicaremos más adelante junto con el diagrama de clases.

Tercera etapa: Esta es la última etapa del Parser cuyo único objetivo es convertir

[1]El mapping es el proceso de definir cómo se almacena e indexa un documento y los campos que contiene.

los JSON generados en la fase previa a un JSON compatible con el que genera Cowrie con el plugin activado. La clase encargada de realizar esta tarea es *Compatible*, la cual también explicaremos más adelante junto con el diagrama de clases.

Cuando finalice esta tercera fase, tendremos dos ficheros que son con los que se trabajará y contienen toda la información útil que se ha podido obtener de los logs en crudo; estos ficheros son:

- **cowrie.compatible.json**: este es el fichero principal que contiene todas las sesiones que se han iniciado y cerrado en ese fichero; se encuentra en un formato JSON compatible con el que genera Cowrie. Para crear este fichero, se utiliza el fichero *cowrie.completed.json*.
- **cowrie.nosession2.json**: este es un fichero que contiene aquellas sesiones que solo se han podido recuperar parcialmente, para reducir al mínimo la pérdida de información, de cuyas sesiones se obtiene solo los datos geográficos de la IP, que se obtienen siempre. Las conexiones que entran en este fichero son principalmente escaneos de puertos en los que Cowrie solo detecta el inicio de sesión, pero nada más. Para crear este fichero, se utiliza el fichero *cowrie.nosession.json*.

El Parser se ha desarrollado en cinco fases incrementales, cada una de ellas basándose en la anterior para realizar mejoras o añadir funcionalidad extra. Estas fases han sido:

Primera fase

En esta primera fase se seleccionó el conjunto de campos más relevantes que se ha detectado; estos fueron la hora, IP, cliente SSH, credenciales usadas y comandos ejecutados.

Como base de datos se usó una relacional, que fue SQLite, aunque posteriormente se descartó el modelo relacional y se usó una no relacional. Por ende, la salida que genera el programa es un fichero de texto con las instrucciones *INSERT INTO*, los cuales se añadirán a la base de datos.

Cabe mencionar que esta primera fase del desarrollo, aunque era funcional, tenía un rendimiento bastante escaso, que en una futura fase se detectaría y solventaría. Para parsear un conjunto de logs de 1 GB tardará alrededor de 6,2 horas.

Segunda fase

En esta fase se añadió el resto de campos que guarda Cowrie para tener unos datos completos respecto a los logs en crudo.

En esta fase de descarta el modelo relacional y se buscan alternativas basadas en NoSQL que se usen actualmente. Se descubre que Cowrie tiene implementado un plugin no funcional para usar Elasticsearch y que en la actualidad se usa en multitud de soluciones y plataformas, por lo que finalmente se decide usar la suite ELK (Elastic-

search y Kibana). Por ende, la salida que genera el programa ahora es un fichero de texto con un JSON que contiene toda la información de una sesión.

Se mejora bastante la eficiencia del tiempo de ejecución para el parseo de 1 GB, el cual pasa de 6,2 horas a 2 horas y media.

Tercera fase

En esta fase se programa desde cero toda la funcionalidad de parseo, de forma que se busca únicamente la eficiencia. Se usan estructuras de datos como son los diccionarios en Python con un orden de complejidad[2] de *O(1)* para la obtención de elementos. Con estas mejoras, el tiempo de ejecución de 1Gb de logs pasa de 2 horas y media a 88 segundos. Ésta se puede considerar como la primera versión del programa.

Cuarta fase

Es esta fase se ha añadido funcionalidad extra no proporcionada por Cowrie por defecto pero que se considera útil, como pueden ser las siguientes mejoras:

- Comprobar si una sesión ha sido un escaneo de puertos: son aquellas en las que no se tiene intento de login. Por la implementación de Cowrie, no es capaz de detectar un escaneo de puertos a nivel de transporte, sino que necesita llegar a nivel de aplicación y abrir una conexión TCP.
- Comprobar si una sesión se ha obtenido a través de un ataque de fuerza bruta: esto lo comprobamos viendo si hay más de dos intentos de login diferentes.
- Obtención de la reputación SSH e información geográfica de IP perteneciente a la sesión: la reputación se obtiene de una fuente abierta. En la sección 4.3.2 se explicará con mayor detalle. Para la obtención de la información geográfica se usará una base de datos pública, la cual se puede descargar y también trabajar con ella localmente. Aunque el proyecto esté descontinuado desde el 2 de enero del 2019, se llama GeoLite2 [49]. Esta misma base de datos es la que usará la suite ELK por defecto.
- Obtención del nivel de amenaza de una sesión: para este nivel se ha desarrollado un pequeño sistema experto basado en reglas que usa los comandos ejecutados para asignar una etiqueta de peligrosidad de la conexión del 1 al 4, siendo el nivel 3 una serie de comandos que solo muestran información del sistema sin llegar a modificarlo, el nivel 1 los comandos que descargan ficheros de internet y el nivel 2 el resto de comandos disponibles. El nivel 4 es especial, ya que solo entran en esté aquellas conexiones que no han ejecutado ningún comando. El nivel más crítico es el 1.

Además, se han creado dos scripts que han resultado ser necesarios. El primero de ellos es un script para leer los ficheros generados por el Parser y enviarlos a Elas-

[2]Podemos ver los tiempos de acceso de las diferentes estructuras de datos en: `https://wiki.python.org/moin/TimeComplexity`.

ticsearch, creando primero un mapping de los datos que se envían y, posteriormente, enviando el fichero en bloques de 10000 líneas, algo totalmente necesario por cuestiones de eficiencia en el tiempo de ejecución, ya que enviarlos de uno en uno ralentizaría el proceso. Después, se ha creado un script para convertir logs antiguos de Cowrie en logs iguales que los actuales. En estos, las líneas cambian el formato de marca de tiempo que hay que modificar y puede tener logs de tráfico Telnet que no analizamos, por lo que se pueden eliminar para reducir el tamaño del fichero y el posterior tiempo de ejecución del parseo.

Quinta fase

Es esta fase se ha buscado que el JSON generado por el Parser sea lo más parecido posible al que genera Cowrie de forma nativa, para que en un futuro se puedan añadir a la misma base de datos nuevos logs, sin posibilidad de crear ningún tipo de incompatibilidad. Se le han añadido al JSON algunos atributos que no tienen los originales, los cuales han sido comentados en la fase anterior.

También se ha modificado el script para enviar los datos a Elasticsearch, ya que solo se disponen de logs para analizar porque los ficheros descargados por los atacantes no están disponibles. Esto se debe a que pueden no permitirse las descargas en los honeypots o que solo se hayan publicado los logs. Por estos motivos no tenemos los ficheros descargados disponibles para poder ser analizados y etiquetados, de forma que se desarrolló la API REST que se explica en la sección 4.3.2 al cual se le mandan las URLs de los ficheros descargados para que la plataforma de VirusTotal las analice y se suban a Elasticsearch ya etiquetadas.

Diagramas de clases y secuencia

A continuación, se muestran los diagramas de clases del Parser, los cuales se han divido en dos para una mejor visualización de los mismos. Primero se va a explicar el módulo *Table* que únicamente tiene como objetivo almacenar la información relativa a una sesión y su forma de representación, que en este caso será en formato JSON; después se explicará el diagrama completo del Parser.

En la Figura 4.2 podemos ver el módulo *Table* y todas las clases que heredan de él. Cada una de esas clases hijas corresponden a un evento que ha generado Cowrie. La clase Table es abstracta y serializable. Todas las clases que heredan de ésta implementan una función para generar un JSON con los datos de esa clase y otro para cargar en memoria la clase a través de un JSON. Estas clases son:

- La clase *Client* contiene información del cliente SSH del atacante, como pueden ser los algoritmos de cifrado utilizados o la versión del cliente SSH.
- La clase *Auth* contiene la información de un intento de login, que es un usuario y una contraseña.
- La clase *Downloads* contiene la información de descarga de un fichero. Esta información es la URL desde donde se ha obtenido, el shasum y el resultado de

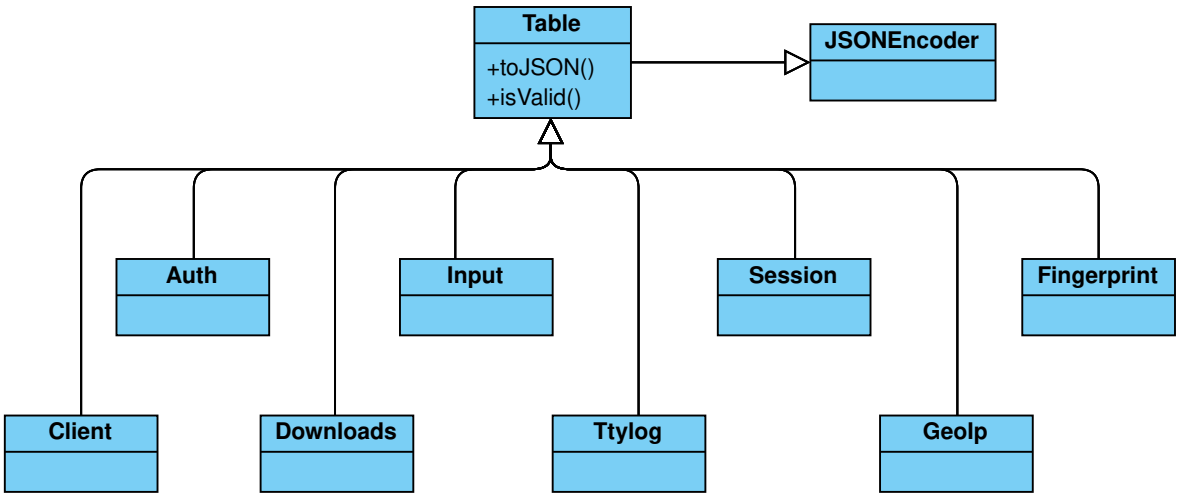


Figura 4.2: Diagrama de clases del módulo Tables

analizarlo por varios motores de antivirus diferentes.
- La clase *Input* contiene la información de la ejecución de un comando, como puede ser el comando ejecutado y si se ha podido ejecutar con éxito.
- La clase *Ttylog* contiene la información de dónde se guardan los comandos ejecutados en la sesión. Actualmente no se ha encontrado de utilidad esta información, pero aun así se almacena por si en un futuro resulta importante.
- La clase *Session* contiene la información básica de la sesión, como puede ser el inicio y fin de ésta y la IP.
- La clase *GeoIp* contiene la información geográfica de la IP, como puede ser las coordenadas, país y ciudad de origen.
- La clase *Fingerprint* contiene la información del fingerprint del atacante. Esta información no ha resultado interesante para el proyecto, pero aun así se almacena por si en un futuro resulta importante.

En la Figura 4.3 podemos ver el diagrama de clases del Parser. Éste muestra las relaciones que hay entre cada una de las clases.

- La clase *Run* es la clase que ejecuta el programa. Se encarga de llamar a las tres clases que ejecutan las tres etapas del programa mencionadas anteriormente, que son *Parser*, *CompleteSession* y *Compatible*.
- La clase *Parser* es la que implementa la mayor funcionalidad. Es la encargada de leer todos los ficheros y pasar cada una de las líneas a la clase *NewConnection* para que éste la procese.
- La clase *ConnectionAux* contiene información inicial que se obtiene al leer una nueva conexión, como la hora de inicio, el ID y la IP.
- La clase *NewConnection* contiene toda la información de una sesión; además, es la encargada de procesar cada una de las líneas para comprobar si en esa línea hay información necesaria y, en caso afirmativo, obtenerla y guardarla.

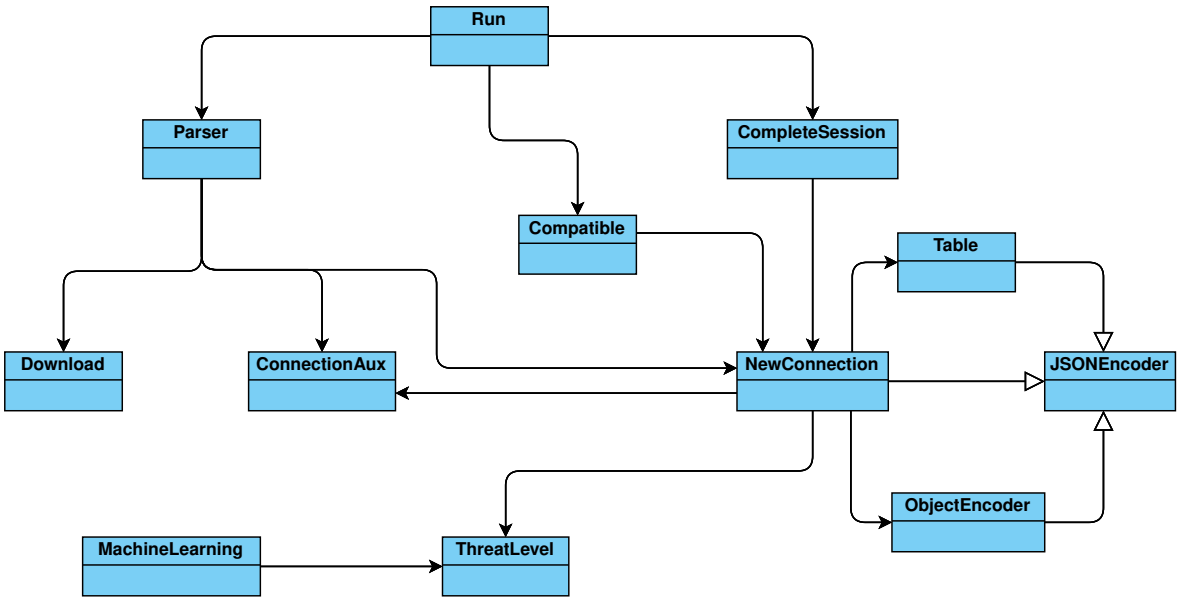


Figura 4.3: Diagrama de clases del Parser

- La clase *Download* contiene la información de una descarga de un fichero. Es una clase usada auxiliarmente por el Parser, ya que en el log no te indica a quién pertenece esa descarga y es necesario buscarlo manualmente por los comandos ejecutados en cada sesión.
- La clase *ThreatLevel* es la clase encargada de implementar un sistema experto basado en reglas que identifica el nivel de peligrosidad de la sesión en base a los comandos ejecutados.
- La clase *CompleteSession* es la encargada de buscar las sesiones incompletas entre los distintos ficheros para completar aquellas que se puedan.
- La clase *Compatible* es la encargada de convertir el JSON generado por el Parser en uno con la misma estructura que el que genera Cowrie.
- La clase *MachineLearning* contiene los algoritmos que se van a implementar de Machine Learning. En la sección 4.5 se explicarán más detalladamente.

Por último, se va a explicar un diagrama de secuencia en el que se muestra de forma simplificada cómo sería el proceso de parseo de un fichero. Este se puede ver en la Figura 4.4.

1. La clase *Run* inicia el proceso pidiendo al Parser que procese un fichero. Para procesarlo, la clase *Parser* se ejecutará en tres partes (pasos 2-3, 4 y 5-9).
2. Se lee el fichero que se está analizando línea a línea, buscando aquellas que correspondan a una nueva sesión.
3. Si la línea corresponde a una nueva sesión, se guarda en un diccionario donde la clave de éste es el número de línea donde se encuentra en el fichero, y el valor asociado a la clave es una instancia de la clase *ConnectionAux* con la información que suministra esa sesión (fecha de inicio, IP, ID de la sesión).

4. Esta función es la encargada de enlazar el ID de sesión con el string *número,ip* ya que, como se mencionó anteriormente, las líneas no usan el ID de la sesión sino ese string. Para enlazar esta información se parte del diccionario creado anteriormente, y para cada sesión que hay en este, se busca el string a partir de la línea en la que se encuentra. Normalmente suele estar en la siguiente línea, pero al permitir conexiones concurrentes puede darse el caso de que no.
5. Finalmente recorremos por última vez el fichero línea a línea.
6. Para cada línea comprobamos si el string *id,ip* existe en el diccionario; si existe, significa que es una sesión que se ha iniciado en el fichero. Después se intenta añadir la información de esa línea a los datos de la sesión.
7. Este es un ejemplo de una línea que tiene información del cliente SSH donde, a través de expresiones regulares, se obtiene toda esa información y se carga dentro de la clase *TableClients*.
8. Si el string *id,ip* no existe significa que esa sesión se ha iniciado en otro fichero, por lo que se crea una nueva sesión y se añade al diccionario para que se pueda añadir información de esa sesión perteneciente a otras líneas.
9. Si la línea corresponde a un comando de descarga se crea una clase de tipo *Download* y se añade a una lista interna de descargas para luego buscar cuál ha sido la sesión que ha creado esa descarga, ya que esta línea no tiene identificador de la sesión que inicio la descarga.
10. Por último, se guardan todas las sesiones en formato JSON en los ficheros correspondientes según sean sesiones completas (fichero *cowrie.completed.json*), iniciadas pero no finalizadas (fichero *cowrie.session.json*) y no iniciadas pero sí finalizadas (fichero *cowrie.nosession.json*).

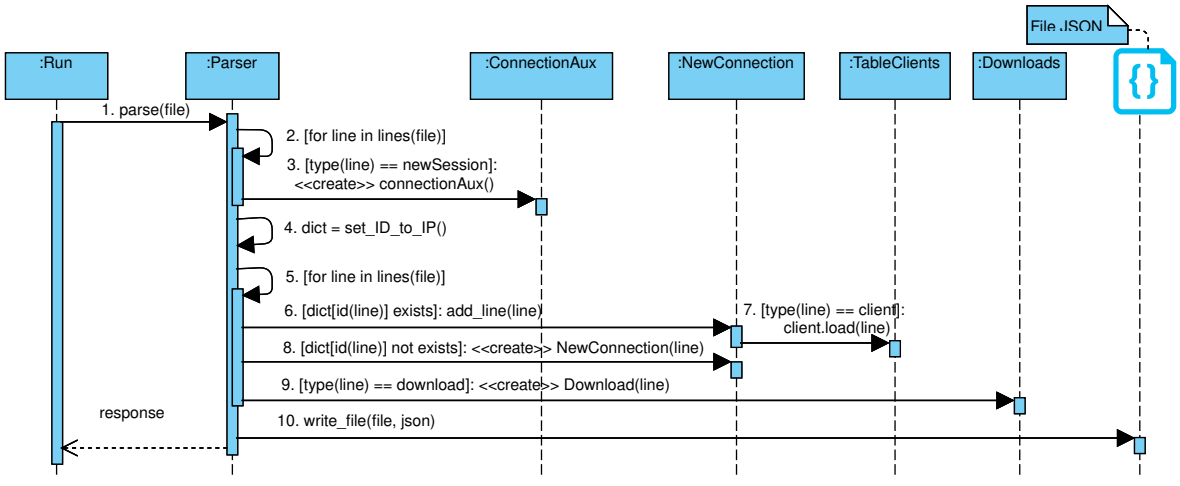


Figura 4.4: Diagrama de secuencia del parseo de un fichero

Funcionamiento y pruebas de rendimiento

Se puede ver un ejemplo del comando ejecutado con su salida abreviada a continuación:

Código 4.3: Ejecución del Parser

```
INFO - run - Start Parser
INFO - parser - Analizando: github_scriptzteam/cowrie.log.2015-10-04
INFO - parser - Analizando: github_scriptzteam/cowrie.log.2015-10-05
...
INFO - parser - Analizando: github_scriptzteam/cowrie.log.2016-02-14
INFO - parser - Time total: 636.202100645 seg
```

Una mejora secundaria del Parser es la reducción del tamaño de los logs una vez analizados: de un repositorio de 1,8 GBs se ha pasado a 476 MBs, por lo que se obtiene una reducción de tamaño de aproximadamente el 73% respecto al fichero original. Esto es algo interesante, ya que si recibimos una gran cantidad de logs de distintos honeypots obtendríamos una reducción del tamaño de los logs bastante significativa.

Para la cantidad de 1,8 GBs de logs en crudo, se ha tardado en parsear un total de 11,5 minutos. En total se han analizado para este proyecto 82 GBs de logs entre los distintos repositorios disponibles.

4.3.2. API REST

En esta subsección se explica el desarrollo de la API REST junto con sus características y un diagrama de secuencia de una petición de análisis de un hash.

Esta es una API programada en Python con una base de datos local que cachea las peticiones que le llegan, te permite tanto obtener la reputación SSH de una IP como obtener de VirusTotal cuántos antivirus detectan un fichero o URL como maliciosa. Para realizar peticiones a APIs externas se va a desarrollar una API REST que sea la encargada de realizar estas acciones. Esto se debe a que la mayoría de éstas tienen un límite de peticiones por minuto, una limitación bastante importante para el proyecto, ya que aquellas peticiones que exceden las permitidas son rechazadas y hay que volver a enviarlas más tarde.

Por ese motivo se ha creado una base de datos local que guarde los elementos consultados con el fin de que, en futuras peticiones, primero compruebe si está almacenado localmente lo que se busca y así reducir el posible número de peticiones a APIs externas. Como se permiten conexiones concurrentes, es necesario aplicar algún mecanismo que permita el acceso concurrente a la base de datos (SQLite) de lectura, pero solo permita una escritura simultánea. En este caso se ha resuelto usando semáforos. Esta API va a realizar cuatro funciones, que son las siguientes:

- Obtener el número de motores de antivirus que detectan como malware los ficheros que se envían; en este caso serán los que se descargan los atacantes que se han conectado al honeypot.
- Descargar ficheros online para su posterior análisis.
- Obtener el hash asociado a un fichero que se obtiene de una URL.

- Obtener la reputación SSH de todas las direcciones IPs que se han conectado.

Para obtener el número de motores de antivirus vamos a usar la API pública de `virustotal.com`. El principal problema de ésta es que solo permite 4 conexiones por minuto natural, por lo que ralentiza bastante la obtención de los datos ya que, como se trabaja con una gran cantidad de datos, se hacen muchas peticiones seguidas. Inicialmente se usó la API pública, pero finalmente se consiguió una API académica que permite 1000 peticiones por minuto natural. Se puede ver cómo obtener está en el "Anexo III. API académica de VirusTotal".

Para comprobar la reputación SSH de una dirección IP, se han buscado fuentes abiertas de información, pero las que hay son de pago o no permiten conexiones a través de APIs públicas, por lo que se ha usado finalmente una página que se dedica a monitorizar ataques a través de honeypots SSH (`https://threatwar.com/ip`) y que, además, es la única que se ha encontrado de libre acceso. El principal inconveniente de esta web es que no es una API, por lo que hay que hacer *web scraping* para obtener los datos, y si una IP no se encuentra registrada en la página, lo indicará con un timeout. Es por esto por lo que se ha configurado un timeout bajo (500 ms), para evitar un aumento excesivamente alto del tiempo de ejecución con este procesamiento.

Esta API se ha implementado sin llamadas bloqueantes. Cuando se recibe una solicitud, si no puede procesarla inmediatamente, usa los códigos de estado HTTP en la respuesta informando del estado.

Para el desarrollo de esta API REST se ha usado Python 3.7, ya que es el lenguaje de programación que se utilizará durante todo el proyecto. Los métodos implementados en esta API se explican a continuación, aunque también podemos ver en la Tabla 4.1 un resumen de estos.

1. El recurso *analizeHash* es un método que recibe un hash como argumento. Primero comprueba si lo tiene cacheado localmente y, si es así, lo envía; en caso contrario, realiza una petición a la API de VirusTotal y reenvía el JSON que recibe.
2. El recurso *analizeUrl* es un método similar al anterior, con la diferencia de que recibe la información a través de un JSON. Una vez recibido, comprueba si está almacenado localmente; en caso de que no lo esté, realiza un análisis de la URL, algo muy útil cuando no se tienen los hashes.
3. El recurso *downloadUrl* es un método que permite descargar ficheros. Primero comprueba si ya está descargado localmente y, en caso de no estarlo, crea un demonio para que lo descargue. Después responde indicando que se está procesando su solicitud. Una vez descargado el fichero se calcula su hash y se manda a VirusTotal.
4. El recurso *getHash* es un método para obtener el hash de una URL. Este método únicamente usa la base de datos local.
5. El recurso *getReputationIp* es un método para obtener la reputación SSH de una IP. Envía la petición a la web threatwar y, si está, devuelve el número de ataques que ha realizado.

HTTP	Recurso	Retorno	Descripción
GET	/analizeHash	Ok PARTIAL_CONTENT	Método para analizar un hash
POST	/analizeUrl	Ok PARTIAL_CONTENT	Método para analizar una URL
POST	/downloadUrl	Ok PARTIAL_CONTENT	Método para descargar el fichero asociado a una URL
POST	/getHash	Ok NOT_FOUND	Método para obtener el hash asociado a una URL
GET	/getReputationIp	Ok NOT_FOUND	Método para obtener la reputación SSH de una IP

Tabla 4.1: Recursos implementados para la API REST

Además de esta funcionalidad ya explicada, cuenta con una cola de peticiones pendientes de procesar: un demonio intenta procesarlas para ver si hay alguna cada 20 segundos. Si se supera el límite de peticiones por minuto, aquellas que no se puedan enviar se quedan en esa lista a la espera de enviarse cuando sea posible. Con esto se consigue que en una primera petición se llene esa cola y, pasado un tiempo, en una segunda petición está toda la información disponible localmente.

Analizar un hash

El proceso de análisis de un hash se puede ver en la Figura 4.5. Se inicia cuando un agente externo manda una petición GET a la API REST (paso 1) pidiendo un análisis de un hash. El siguiente paso es comprobar si se tiene almacenado localmente ese resultado (paso 2) en la base de datos. Si no se tiene almacenado, se manda una petición de análisis a la API pública de VirusTotal (paso 4). Cuando llega la respuesta, se añade esa información en la base de datos local con un *mutex* (paso 5) ya que existe concurrencia. Después se responde al usuario con la información solicitada (paso 6). Por el contrario, si ya se tiene la información en la base de datos, se obtiene (paso 7) y se responde al usuario con esta (paso 8).

4.3.3. Elasticsearch

En esta subsección se explica todo el proceso que se ha llevado a cabo para la instalación y configuración de la base de datos Elasticsearch, así como los procesos de inserción y actualización de datos que se han llevado a cabo usando diagramas de secuencia para una mayor claridad.

Cabe mencionar que dentro de la suite ELK existe una tercera tecnología. Esta es Logstash que permite analizar datos no estructurados e insertarlos en Elasticsearch, pero en este caso no podemos usarlo, ya que no es capaz de relacionar los datos entre

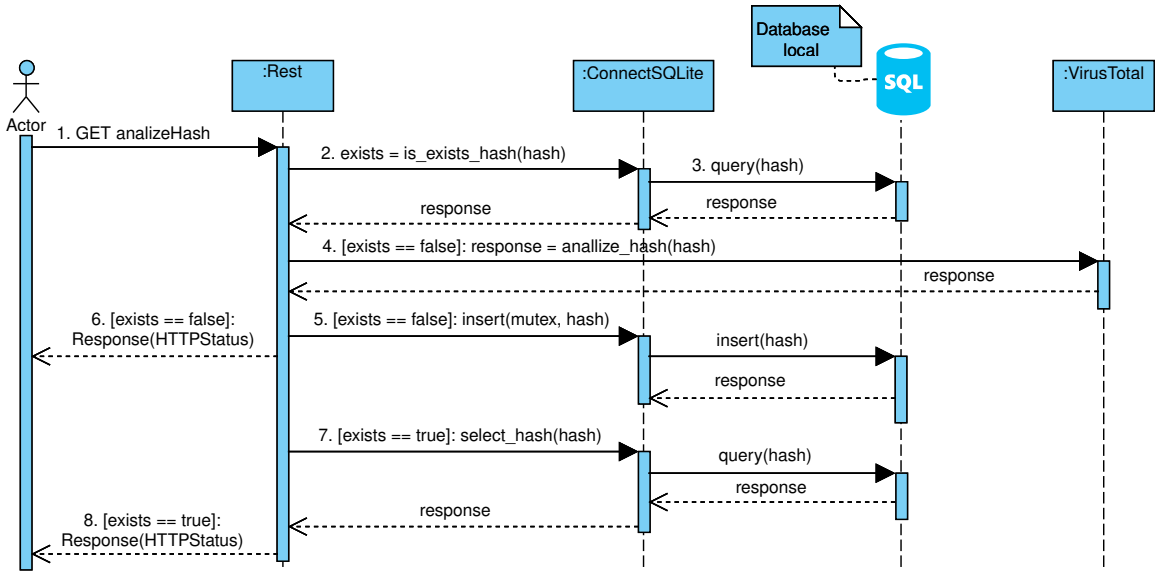


Figura 4.5: Diagrama de secuencia de una petición a la API REST para analizar un hash

sí debido al problema mencionado anteriormente (sección 4.2.2) en el que Cowrie solo se identifica con el ID de la sesión la primera línea. Por este motivo ha sido necesario desarrollar el Parser. En caso de contar con logs en formato JSON, se podría usar esta tecnología.

La configuración de Elasticsearch que se ha usado es básica, se ha levantado en modo *single-node*, esto es debido a que solo usaremos un nodo. Después se ha levantado con docker junto con Kibana, estando así ya lista para funcionar. Se ha usado la versión 6.6.0 aunque mitad de este proyecto se publicó oficialmente la versión 7.0.0, pero se descartó su uso ya que, aunque traía novedades interesantes, no planteaba ninguna mejora para este proyecto, pero hacía necesario modificar la configuración previa de Kibana ya que esta no era compatible con la nueva versión. Además, cabe mencionar que, aunque se haya levantado Elasticsearch en un único nodo, si hubiese una gran cantidad de honeypots mandando logs se podría desplegar una red de nodos de Elasticsearch con una mínima configuración previa.

Se ha creado un índice por cada honeypot con el formato *cowrie-honeypot*, siendo honeypot el nombre de este, ya que así se quedan los logs más ordenados. Cada honeypot tiene un mapping asociado que indica el tipo al que pertenece cada atributo. Podemos ver un ejemplo a continuación, donde se han definido 3 objetos *eventid* que es de tipo text, *ip* que es de tipo ip y *location* que es de tipo geo_point. Este último representa las coordenadas (latitud y longitud) donde se encuentra la dirección IP.

```
{
 "mappings": {
  "object": {
    "dynamic": "true",
    "properties": {
```

```
6       "eventid": {"type": "text", "fields": {"keyword": {"type": "keyword","ignore_above": ↩
            ↪ 256}}},
7       "ip": {"type": "ip"},
8       "location": {"type": "geo_point"},
9   }}}}
```

Además de los tipos mencionados anteriormente, se han usado otros diferentes como pueden ser: *date* que permite formatear un tipo de fecha y *bool* para mostrar valores de verdadero o falso.

En las primeras fases de desarrollo del Parser, se tenía un único JSON complejo que contenía toda la información relacionada con la sesión, pero esto no era una idea óptima, ya que Elasticsearch no está optimizado para este tipo de búsquedas y hacía más compleja la obtención de información de objetos dentro de listas del JSON. Posteriormente, cuando se pasó a generar los JSON con el formato de Cowrie, cada evento pasaba a ser un JSON independiente, todos ellos referenciados con el ID de la sesión. De esta forma, Elasticsearch sí que era capaz de hacer búsquedas complejas de una forma eficiente y sencilla.

En la sección 4.3.1 se comentó que se había desarrollado un script para enviar los datos a Elasticsearch; podemos ver el proceso de inserción de datos en la Figura 4.6 y el proceso de actualización de las descargas en la Figura 4.7. Ambos diagramas de secuencia se han simplificado para una mejor compresión, los cuales se presentan a continuación.

Inserción de datos

El proceso de inserción de datos se puede ver en la Figura 4.6. Consta de dos partes: en la primera (paso 1) se crea un índice con un mapping para los datos que se van a enviar, el nombre del índice es el nombre que tiene el honeypot. Una vez creado el índice, se procede a leer el fichero de logs (*cowrie.compatible.json*) en bloques de 10000 líneas (paso 2), después se recorrer todas las líneas (paso 3) realizando una serie de comprobaciones, que son las siguientes. Comprobar si existe algún evento del tipo *eventid = cowrie.session.file_download* que corresponde a descargas realizadas por atacantes; si existe y no tiene actualizado el valor *dangerous*, que corresponde con el número de motores de antivirus que consideran ese hash una amenaza, se manda una petición a la API REST para que te diga su valor (paso 4). Este te lo envía si lo tiene y, si no, manda una petición a VirusTotal. Después se comprueba si existe algún evento del tipo *eventid = cowrie.extend* y, en caso afirmativo, si la IP de la sesión no tiene asignada reputación SSH, que puede deberse a un timeout durante la primera petición ya que la API externa a veces falla. Si no tiene asignada reputación, se manda una petición para obtenerla (paso 5). Finalmente se envía ese bloque de líneas (paso 6) y se procede a realizar los pasos anteriores con el siguiente bloque de líneas.

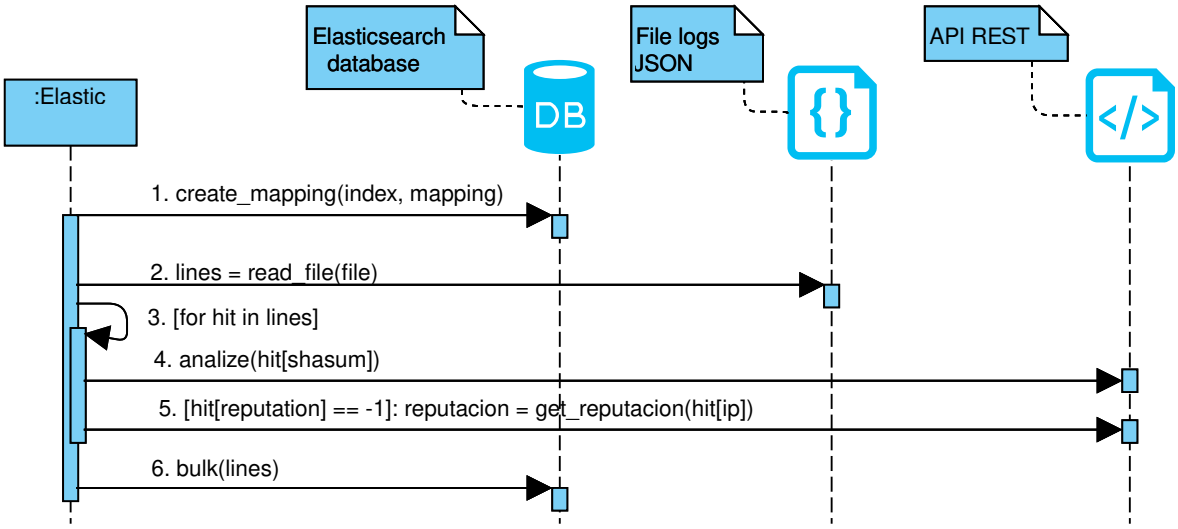


Figura 4.6: Diagrama de secuencia de la inserción de datos en Elasticsearch

Actualización de datos

El proceso de actualización de descargas se puede ver en la Figura 4.7. Éste es algo más complejo y lento, ya que puede llegar a descargarse los ficheros para analizarlos, pero a cambio obtenemos una información completa. Es posible que no exista el *eventid = cowrie.session.file_download* de un fichero, puesto que se puede usar como política de seguridad que se bloquee la conexión una vez que se solicita iniciar la descarga, haciendo que no se inicie la descarga y, por consiguiente, ese evento.

Consta de dos pasos bien diferenciados, que son los siguientes:

El primer paso consiste en buscar (paso 1) todos los eventos del tipo *eventid = cowrie.input* que han ejecutado un binario como *wget* o *curl* y que no tienen asociado un *eventid = cowrie.file_download*. Después se recorren todas esas sesiones (paso 2) y se pregunta por el hash asociado a esa URL (paso 3), ya que VirusTotal tiene esa información, algo bastante útil, ya que se ha comprobado que gran cantidad de las URLs de las que se disponen están offline, y si no fuera por esto, no se podría obtener ninguna información. Con estos datos se crea un JSON del tipo mencionado anteriormente (paso 4) y se envía a Elasticsearch (paso 5) para su registro.

El segundo paso consiste en buscar todos los eventos del tipo *eventid = cowrie.file_-download* que tengan el valor *dangerous* sin valor (paso 6), y recorrerlos uno a uno (paso 7) para pedir a la API REST que nos dé el valor analizando directamente la URL (paso 8). Este método obtiene siempre resultado, ya que en el paso anterior se mandó la URL a VirusTotal para que lo analizara y ya estará cacheada la respuesta localmente, por lo que obtendremos un valor de 0 a 70, siendo ese número la cantidad de motores de antivirus que lo detectan como malware. Finalmente, se construye un JSON con la información que hay que actualizar (paso 9) y se manda la actualización

a Elasticsearch (paso 10).

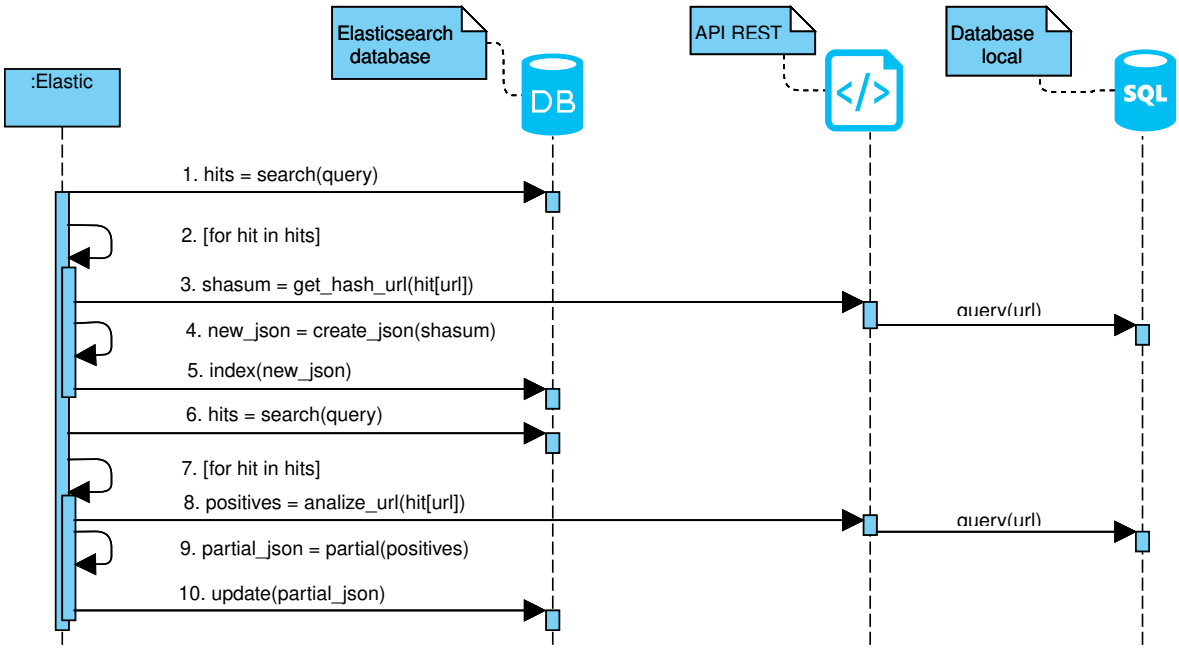


Figura 4.7: Diagrama de secuencia de la actualización de descargas en Elasticsearch

Las búsquedas en Elasticsearch se realizan a través de un JSON y son muy potentes. A continuación, se muestra cómo sería la búsqueda de la ejecución de comandos cuyo binario es curl o wget. Además, se busca en bloques de 1000; esto se debe a que no es eficiente hacer búsquedas en bloques mayores, por lo que si se quieren encontrar todos hay que realizar un *scroll*, que permite realizar búsquedas con desplazamientos.

Código 4.4: Busqueda Elasticsearch de comandos wget y curl

```
{
  "size": 1000,
  "query": {
    "bool": {
      "should": [
        { "term": {"binary": "wget" }},
        { "term": {"binary": "curl" }}
      ]
}}}
```

Finalmente, queda comentar que se ha seguido un orden muy específico a la hora de hacer las peticiones a la API REST para reducir al mínimo las posibles peticiones a API externas por las limitaciones que esto conlleva, tanto por limitación de peticiones por minuto en VirusTotal, como por tiempo de petición en threatwar.

Para explicar ese orden, se va a usar un pequeño autómata, que se puede ver en la

Figura 4.8.

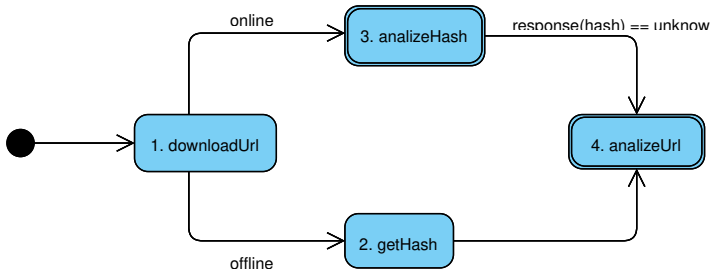


Figura 4.8: Autómata actualizaciones

1. Esta es la fase inicial. Lo primero que se hace es intentar descargar el fichero para poder calcular su hash y tenerlo disponible para un futuro análisis del mismo. Dependiendo de si la URL está offline u online, saltaremos al paso 2 ó 3 respectivamente.
2. Esta fase pregunta a VirusTotal si conoce el hash asociado a la URL, ya que es necesaria para almacenarla en el JSON. De este paso se salta al 4, que es donde se preguntará por el análisis de esa URL. Es necesario preguntar previamente por el hash, ya que en este análisis no se muestra el hash del fichero.
3. Si la URL está online y se tiene el hash, se hace un análisis del hash, y en caso de que no haya información se hará un análisis de la URL, aunque por lo que se ha visto no suele ser necesario ya que casi todos los ficheros utilizados son conocidos y están ya almacenados.
4. Esta es la última fase, en la que se realiza un análisis de la URL asociada al fichero. En los logs analizados se han encontrado y analizado 455 URLs, y de todas ellas solo 23 no se han encontrado en la base de datos de VirusTotal.

Siguiendo este orden, se consigue obtener toda la información asociada a una URL (hash y nivel de amenaza) reduciendo al mínimo las peticiones a VirusTotal.

4.3.4. Kibana

En esta subsección se explica la configuración que se ha realizado con Kibana para buscar, filtrar y generar gráficas y tablas con los datos de Elasticsearch con las que poder obtener visualmente unos primeros resultados sobre la procedencia de los ataques.

Una vez que se tienen todos los datos en Elasticsearch se tiene que configurar Kibana para obtener esos datos y mostrarlos. Para eso hay que seguir cuatro pasos que se explican a continuación.

Paso 1: Management

Lo primero que se necesita es indicar cuál es el patrón de índice que se va a usar; se comentó anteriormente que todos tienen un índice con el formato *cowrie-honeypot*, por

lo que se usará el índice *cowrie-** para que sea válido para todos. Este patrón de índice usa el mapping que se asignó en el índice de Elasticsearch para saber los tipos que tienen cada uno de los datos.

A la hora de crear este patrón de índice hay que indicar cuál es nombre del campo que contiene la fecha de inicio del evento, algo sumamente importante para que Kibana sea capaz de mostrar correctamente los datos por fecha, permitiendo después hacer filtros dependiendo de la antigüedad de los datos.

Paso 2: Discover

Una vez que ya se ha creado el índice, se podrían crear las tablas y gráficas con los datos, pero esas búsquedas no serían eficientes ya que se buscaría en todos los elementos almacenados. Para hacerlo más eficiente, se crean filtros que solo obtengan los elementos que se deseen buscar. Esto se realiza en el apartado *Discover*, podemos ver a continuación un ejemplo de un filtro que solo busca aquellos eventos que tengan información del cliente SSH y que la versión se haya obtenido correctamente, ya que hay algunos que no suministran esa información:

Código 4.5: Filtro para busquedas de clientes SSH

```
eventid.keyword : "cowrie.client.version" and NOT version.keyword : "unknown"
```

Para diseñar estos filtros hay que usar el lenguaje de Kibana que permite tanto escribirlo directamente como hacerlo de forma gráfica.

Paso 3: Visualize

Después de crear los filtros ya se pueden crear las tablas, gráficas, etc. Para esto hay que seleccionar el tipo de visualización que se desea y el filtro que se usará para obtener los datos. Existe gran cantidad de tipos de visualización, pero los más prácticos son tablas, diagramas circulares y mapas geográficos.

Kibana es una herramienta muy potente, ya que te permite crear complejos filtros internos con los que obtener los datos que necesitas, por ejemplo, te permite crear filtros que solo cuenten una única vez un término. Un ejemplo práctico sería el mapa de la Figura 4.11, que posteriormente se explicará en detalle. Aquí solo se permite que cada IP cuente uno.

La mayoría de los datos se van a visualizar a través de tablas, ya que es la forma más óptima de representación. Se van a crear tablas y gráficas para los siguientes elementos:

- Tabla de usuarios y contraseñas
- Tabla de clientes SSH
- Tabla de direcciones IP
- Mapa geográfico según la IP
- Tabla de comandos ejecutados
- Gráfica de Threat Level
- Gráfica de ataques por fuerza bruta
- Gráfica de escaneos de puertos
- Tabla de ficheros descargados

Paso 4: Dashboard

Por último, una vez que ya se han creado todos los elementos gráficos que se necesitan, solo queda diseñar los distintos escritorios que son necesarios añadiéndoles los elementos visuales requeridos. Como ejemplo, que se explica en la próxima subsección, se puede ver en la Figura 4.9 cómo sería el resultado final de un escritorio.

Si en el patrón de índice se indicó cuál es la marca de tiempo en estos escritorios, es posible hacer filtrados rápidos por tiempo, siendo posible coger los datos de los últimos 7 días o de los últimos 30 días. También es posible realizar filtros dentro la barra de filtros de Kibana para obtener ciertos datos.

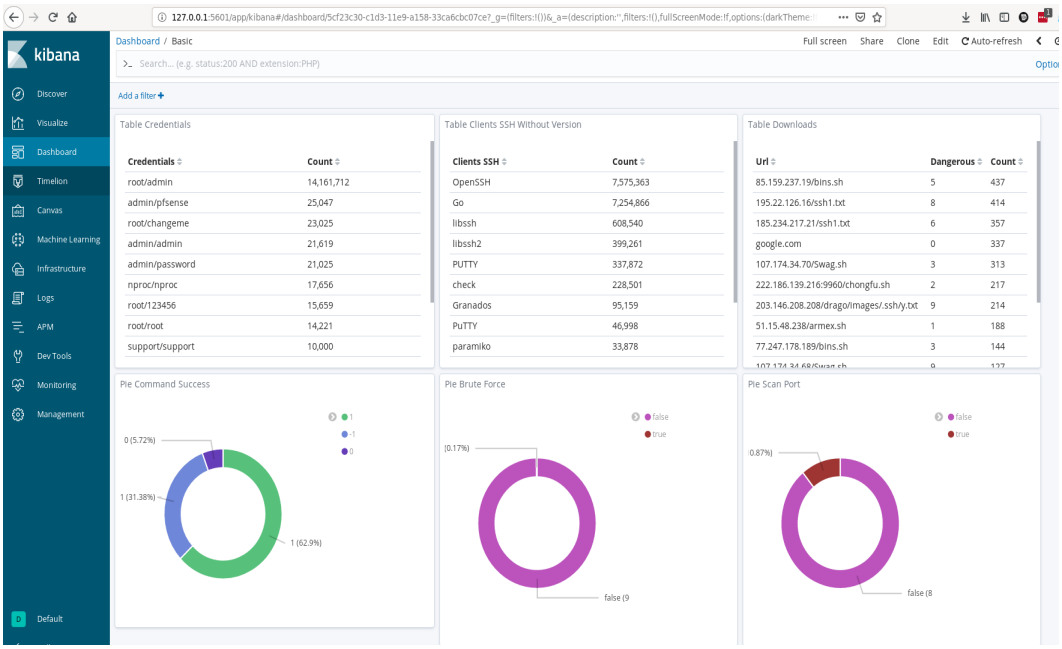


Figura 4.9: Escritorio de Kibana con distintas tablas y diagramas

Kibana también cuenta con un módulo para Machine Learning, pero este solo se puede conseguir instalando la extensión X-Pack, siendo está de pago. Además de que, por ende, no puedes modificar los algoritmos ya implementados reduciendo así su utilidad para este proyecto.

4.4. Clasificación de logs generados por Cowrie

En esta sección se va a mostrar toda la información que se ha obtenido de los logs de honeypots SSH Cowrie, para la representación de los datos se usarán las gráficas y tablas obtenidas de Kibana. Además, se van a explicar las conclusiones que se han obtenido de estos datos.

En total se han usado logs de 6 honeypots, de los cuales se han conseguido 82 GBs de logs que contienen 16.967.189 conexiones, en las cuales hay 66.209 direcciones IP únicas. Los logs usados han sido de los repositorios públicos mencionados en la sección 4.3.1 y de los logs propios, ya que no se pueden publicar los datos de los logs referentes al reto realizado.

En la Figura 4.10 se puede ver la distribución geográfica de los ataques. Se puede apreciar claramente cómo la gran mayoría de los ataques han sido a través de Irlanda y Reino Unido. Este mapa no tiene mucho sentido por ese motivo ya que [50] y [20] indican que los países que más ataques realizan son China, Rusia y EE. UU. Debido a esto, se ha realizado la Tabla 4.2 con las IPs, el país y cantidad de conexiones asociada a cada una, obteniendo la siguiente información.

Figura 4.10: Mapa de calor de ataques

Dirección IP	Frecuencia	País
5.188.86.174	1.128.71	Irlanda
5.188.86.211	1.009.42	Irlanda
5.188.87.53	681.125	Irlanda
5.188.87.55	671.261	Irlanda
5.188.87.51	654.598	Irlanda

Tabla 4.2: Tabla de frecuencia de conexiones según la IP

De esta tabla se puede ver que las cinco direcciones IP que más conexiones han realizado proceden de Irlanda, lo cual hace que en el mapa de calor no se pueda apreciar

correctamente. Esto puede que se deba a que haya proxies públicos o privados que usan los atacantes para que su IP origen este dentro de Europa, ya que es habitual bloquear ciertos rangos de IP de países como China o Rusia desde donde suelen proceder gran cantidad de ataques. Por ese motivo puede ser una buena idea hacer que cada dirección IP solo cuente una vez en la gráfica. Además, viendo la cantidad de conexiones realizadas desde Irlanda y Reino Unido, puede ser una buena medida de seguridad el bloquear conexiones de estos países si no son necesarias conexiones desde ellos, ya que entre los dos representan el 75% de los ataques recibidos.

En la Figura 4.11 se puede ver la distribución geográfica de los ataques, pero en esta ocasión con una agrupación de peticiones por dirección IP única, ya que así los posibles proxies que se usen solo cuentan uno, en vez de sumar uno por cada conexión. Analizando este nuevo mapa, ya se ve algo más coherente, puesto que los países que más ataques realizan son China, EE. UU. y Rusia.

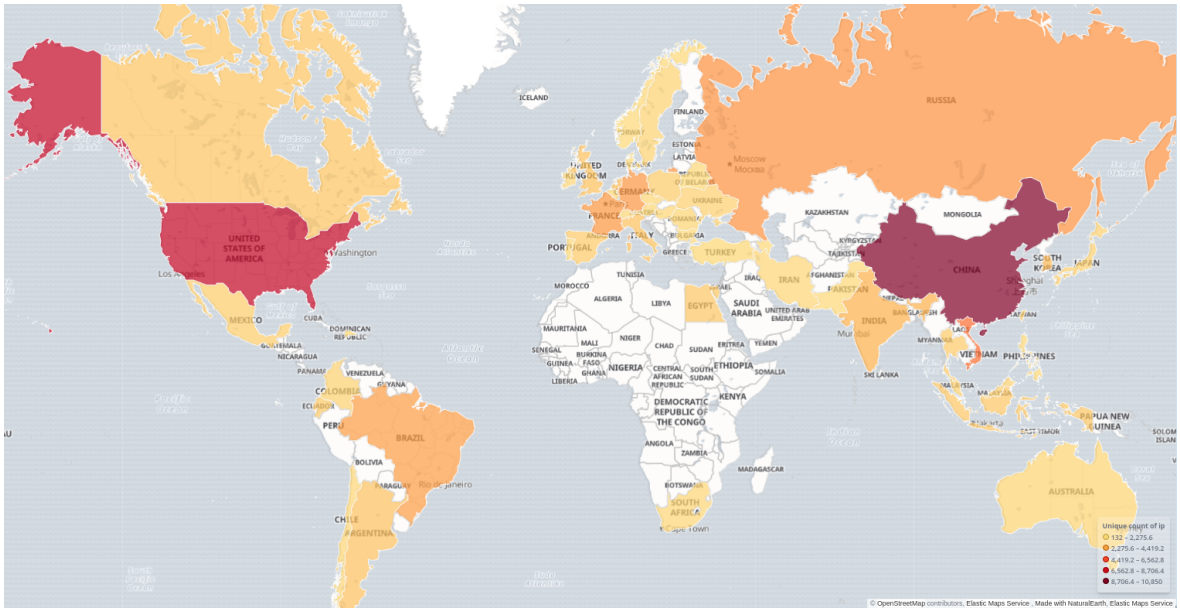


Figura 4.11: Mapa de calor de ataques con IP única

En este nuevo mapa podemos ver que China, EE. UU. y Rusia tienen 10.850, 8.748 y 3.539 respectivamente. Esta cantidad de direcciones IP es mucho menor que la del anterior mapa, pero hace que éste sea mucho más realista. El problema es que se desconoce el país de origen de la gran cantidad de ataques realizados a través de Irlanda y Reino Unido, pero seguramente se trate de bots realizando ataques automatizados.

Sería interesante la creación de una base de datos de reputación SSH de direcciones IP, ya que es algo habitual tenerlas para otros servicios como puede ser el correo electrónico y así mejorar los filtros anti-spam, pero no existe tal base de datos para un servicio tan crítico como es SSH.

En la Tabla 4.3(a) se pueden ver los nombres de usuario más usados, siendo *root* el que

se ha usado con mayor frecuencia; después, *admin*, y el resto apenas son anecdóticos.

En la Tabla 4.3(b) se pueden ver las contraseñas más usadas, siendo *admin* la que se ha usado con mayor frecuencia; el resto se ha usado en un menor número de ocasiones.

Usernames	Frecuencia
root	14.338.888
admin	169.990
user	24.753
nproc	17.656
test	17.200
adm	15.023
support	14.627
oracle	10.141
1234	9.363

(a) Tabla de usuarios

Passwords	Frecuencia
admin	14.191.472
123456	81.049
password	48.266
1234	32.010
123	26.420
changeme	25.825
pfsense	24.523
12345	23.275
nproc	17.656

(b) Tabla de contraseñas

Tabla 4.3: Tabla de usuarios y contraseñas utilizados

Aunque de las tablas anteriores de usuarios y contraseñas (Tabla 4.3) se puede ver claramente cuáles son los más usados individualmente, en la Tabla 4.4(a) se pueden ver cuáles han sido las combinaciones más usadas. En primer lugar, está *root/admin* que es usada por multitud de fabricantes como credenciales por defecto, como pueden ser Apple[3], Intel y D-Link. En segundo lugar, aunque con menor cantidad, está *admin/pfsense* que es usada en firewalls. En tercer lugar, está *admin/changeme* que es usada por Sun Microsystems y Dell EMC. Las siguientes credenciales de la tabla son las típicas usadas principalmente por routers.

Como se puede ver, no se han realizado demasiados ataques a dispositivos muy específicos, como en el caso de Ubiquiti (*ubnt/ubnt*) y Raspberry Pi (*pi/raspberry*) que han recibido 4.911 y 4.548 respectivamente, sino que se han usado las principales contraseñas de los diccionarios más habituales. Se puede concluir que las credenciales *root/admin* son las más atacadas por bots, por lo que es indispensable cambiarlas en caso de que esa sea la credencial de cualquier dispositivo, siendo una buena opción que los fabricantes no usaran esas credenciales en sus dispositivos por defecto.

En la Tabla 4.4(b) se pueden ver los comandos más ejecutados. En general, si los analizamos todos, la gran mayoría son usados para obtener información del sistema. Esto se debe a que Cowrie, al virtualizar los comandos, cuenta con una cantidad muy limitada de estos. Los comandos que salen en esta tabla y que no son para obtener información del sistema han sido analizados uno a uno y son de scripts que han utilizado bots que se han conectado repetidamente, haciendo que suba mucho su número de

[3]Se ha usado la siguiente URL: `https://github.com/danielmiessler/SecLists/blob/master/Passwords/Default-Credentials/default-passwords.csv` como fuente para buscar cuáles son las credenciales usadas por defecto según el fabricante

Credenciales	Frecuencia
root/admin	14.161.712
admin/pfsense	25.047
root/changeme	23.025
admin/admin	21.619
admin/password	21.025
nproc/nproc	17.656
root/123456	15.659
root/root	14.221
support/support	10.000

(a) Tabla de credenciales

Comandos	Frecuencia
cat /proc/cpuinfo	22.687
grep name	19.906
wc -l	15.837
/gisdfoewrsfdf	14.753
uname -a	11.379
cd	7.819
cat //.nippon	6.813
rm -f //.nippon	6.700
free -m	6.521

(b) Tabla de comandos ejecutados

Tabla 4.4: Tablas de credenciales usadas y comandos ejecutados

ejecuciones en la tabla.

Algunos otros comandos que no salen en la tabla por ocurrir en menor medida han sido para modificar variables de entorno, añadir claves SSH, cambiar la contraseña y limpieza de ficheros e historial.

En la Tabla 4.5(a) se pueden ver los principales clientes SSH que han usado los atacantes. Esto se considera de especial interés, ya que con este dato se puede suponer cuál es el lenguaje de programación usado por el bot para el ataque. Esta tabla se ha usado como feature para Machine Learning ya que tiene la versión y puede ser discriminante. En la Tabla 4.5(b) los podemos ver, pero sin la versión, mejorando la agrupación de estos. Se puede ver que *OpenSSH* y *Go* son los clientes más usados por los atacantes. Si se desgrana uno a uno estos clientes obtendríamos la siguiente información:

- **OpenSSH**: Programado en C. Posiblemente sea el más usado, ya que se encuentra en la mayoría de distribuciones Linux instalado por defecto. Puede ser usada por código en C/C++ o en scripts. `https://www.openssh.com/`.
- **Go**: Programado en Go. Éste es un lenguaje que lleva un tiempo creciendo en popularidad. Se han encontrado diversas botnets que usan este lenguaje y al poder usarse como una librería facilita mucho la creación de scripts para realizar ataques. `https://godoc.org/golang.org/x/crypto/ssh`.
- **libssh**: Programado en C. Este cliente también viene instalado en distintas distribuciones Linux y derivados. Es una implementación libre del protocolo SSH que puede ser usada en este lenguaje o en scripts. `https://www.libssh.org/`.
- **PuTTY**: Programado en C y disponible para Windows es un cliente con entorno gráfico, por lo que la automatización de script está muy limitada, aunque cuenta con la utilidad *plink* que funciona únicamente por consola y se pueden crear scripts. `https://www.putty.org/`.
- **check_ssh**: No se ha encontrado información sobre este cliente SSH.

- **Granados**: Programado en C# (.NET). Es un lenguaje con bastante popularidad por lo que es lógico que se use la librería SSH de este lenguaje en script con los que atacar. `http://granados.sourceforge.net/`.
- **paramiko**: Programado en Python. También es un lenguaje bastante popular muy usado por pentesters. Esta librería permite una gran configuración de la conexión SSH. `http://www.paramiko.org/`.

Podemos ver que, aunque eliminemos la versión del cliente, el resultado no varía prácticamente en los primeros. Esto se debe a que son ataques provenientes de los mismos bots.

Clientes SSH	Frecuencia
SSH-2.0-OpenSSH_7.3	7.517.861
SSH-2.0-Go	7.254.738
SSH-2.0-libssh-0.6.3	474.818
SSH-2.0-PUTTY	337.872
SSH-2.0-libssh2_1.7.0	255.016
SSH-2.0-check_ssh_2.2.1	228.501
SSH-2.0-Granados-1.0	95.159
SSH-2.0-libssh-0.2	59.435
SSH-2.0-libssh2_1.4.3	52.652

(a) Tabla de clientes SSH

Clientes SSH	Frecuencia
OpenSSH	7.575.363
Go	7.254.866
libssh	608.540
libssh2	399.261
PUTTY	337.872
check_ssh	228.501
Granados	95.159
PuTTY	46.998
paramiko	33.878

(b) Tabla de clientes SSH sin versión

Tabla 4.5: Tablas con los clientes SSH usados

Además de estos clientes SSH, se han encontrado otros un poco más sospechosos, ya que se han considerado intentos de buffer overflow. Esto es mucho más peligroso, ya que intentan aprovecharse de que el cliente SSH puede mandar su User-Agent como un string al servidor e intentan acceder a zonas de memoria no permitidas para conseguir explotar una posible vulnerabilidad. Podemos ver algunos de los que se han usado a continuación:

```
\x00\x00\x00\x00\x00\x00\x00\x00\x03\x00\x00\x00'MAIL FROM:<AnabelleDarstt@aaifgg.com>
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00-tcpCONNECT ↩
    ↪ 211.231.108.47:25 HTTP/1.0\x00
\x16\x03\x01\x00\x9a\x01\x00\x00\x96\x03\x03\xd5\xfd>@\xd5t\xea'\xd0\x03
\xc8\x00\x00D<\x7f\x00\x00\xe0 *E<\x7f\x00\x00\x98vx\xfd~\xb8c\xdf\x90\x0e\x9f><\x7f\x00\x00>B\↩
    ↪ xb6D3\xa1h\xcc\xcd\xf8|'\x04e\xd4\x07\xd1\x0b\xbcB\x8f\xb1\xc1&\x0c\xfcq\xf4\xee}\x00\x91↩
    ↪ ;*\x7f'\x18\x95\xdb\x07\xaa}\xc1,\xe6\xa3Y\x9cSSH-2.0-OpenSSH_7.3
\x00\x00\x8d>\x00\x00\x00'250 Ok, message accepted for delivery
```

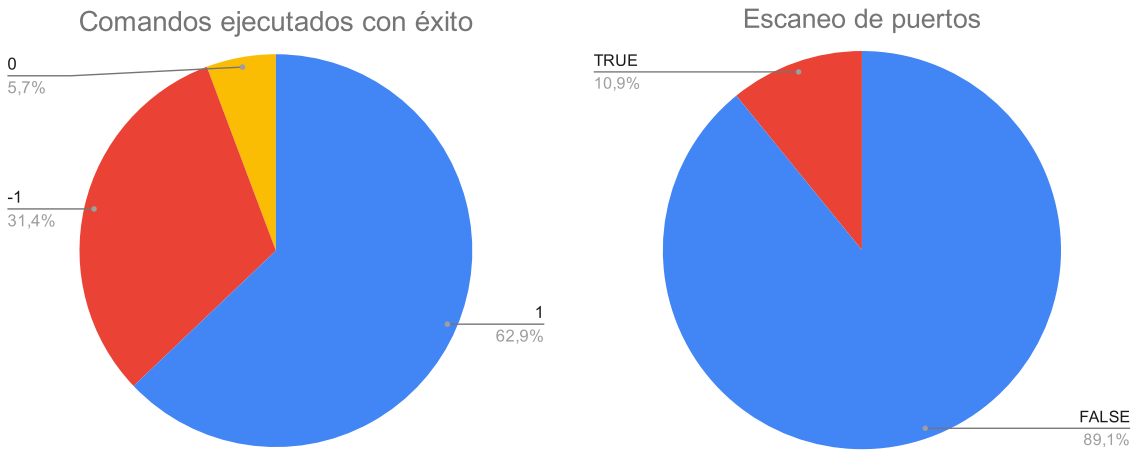
En general, la gran mayoría de estos intentos de buffer overflow tienen únicamente como objetivo informar al atacante de si ha funcionado o no. Esto se realiza de diversas formas, siendo la más habitual intentar mandar un correo electrónico o realizar una conexión TCP a un servidor.

En la Figura 4.12(a) se pueden ver los comandos que se han podido ejecutar con

éxito (1), que son la mayoría con un 62,9%. Sabemos que hay un 5,7% que no ha podido ejecutarse (0) y se ha buscado el porqué. Esto se debe a que la mayoría son scripts que han intentado ejecutar programas después de descargarlos, pero estos no se han podido ejecutar por no haberse descargado o que usan binarios que no existen en Cowrie. Respecto al 31,4% que no se ha podido comprobar (-1), la gran mayoría se debe a la complejidad de parsear un comando, ya que se usan expresiones regulares y hay que asegurarse de que todo está correctamente escapado en el código. Después está el problema de los comandos encadenados que usan expresiones regulares, dificultando mucho el poder separarlos correctamente para después buscarlos uno a uno si han podido ejecutarse correctamente.

Las siguientes tres gráficas se han inferido con los datos que se han obtenido. No son datos que estén en los logs, pero se consideran que son bastante relevantes y, además, algunos se ha usado como feature para Machine Learning.

En la Figura 4.12(b) se pueden ver que ha habido un 10,9% de las sesiones que han sido un escaneo al puerto 22. Este dato no es completamente real, ya que Cowrie solo registra aquellos escaneos que abran una conexión SSH con la intención de obtener la versión del servidor SSH de Cowrie (nivel aplicación); un escaneo a nivel de transporte no es detectado y por lo tanto no se registra.



(a) Gráfica de comandos ejecutados **(b)** Gráfica de escaneo de puertos

Figura 4.12: Gráficas de escaneos de puertos y comandos ejecutados con éxito

En la Tabla 4.6(a) se puede ver el etiquetado que se ha hecho a todas las conexiones usando un sistema experto basado en reglas. Este etiquetado se hace en función de los comandos que se han ejecutado:

1. Este nivel es el más crítico y lo obtienen aquellas conexiones que realizan alguna descarga y obtienen ficheros desconocidos para el honeypot. Podemos ver que es

el segundo más usado.
2. Este nivel lo obtienen aquellas conexiones que realizan modificaciones del sistema, ya sea modificando ficheros, eliminado procesos o borrando. Es el menos usado, ya que los scripts si hacen estas acciones es porque se han descargado ficheros antes.
3. Este nivel lo obtiene aquellas conexiones que solo ejecutan comandos para mostrar información del sistema; en general, las conexiones de este tipo suelen ser manuales y la duración de ésta, mayor.
4. Este nivel es el más leve, lo obtienen aquellas conexiones que no ejecutan ningún comando. Como puede verse, son la mayoría.

Aunque este etiquetado ayude a hacerse una idea de los ataques recibidos, no funciona bien a la hora de hacer clasificaciones usando clustering en Machine Learning, por lo que se ha tenido que modificar posteriormente según otros criterios como pueden ser: escritura en disco, lectura de disco, conexión a internet, ejecución de programas y matar procesos.

En la Tabla 4.6(b) se pueden ver lo que se considera ataques de fuerza bruta, aunque en su mayoría no lo son, ya que se permiten todas las credenciales excepto *root/root*. Para considerarlo ataque por fuerza bruta tiene que haber puesto más de 2 credenciales. En general, los bots analizados no suelen realizar estos ataques y saltan a la siguiente IP si fallan en el intento de login.

Threat Level	Frecuencia
4	16.923.388
1	21.753
3	20.508
2	1.540

(a) Tabla de Threat Level

Ataques	Frecuencia
false	16.939.020
true	28.169

(b) Tabla de ataques de fuerza bruta

Tabla 4.6: Tablas con los clientes SSH usados

En la Tabla 4.7 se pueden ver los ficheros que se han descargado los atacantes y su peligrosidad; en total se han realizado 4.989 descargas. Como puede verse, todos los ficheros que se han descargado están almacenados en VirusTotal y son considerados peligrosos por algún antivirus. El único que se salva de ser considerado peligroso es *google.com*, que es usado por los atacantes para comprobar si hay conexión a internet. Aunque no se ha llegado a hacer, si se elimina la IP y se agrupara solo por nombre del fichero descargado, se harían agrupaciones mucho mejores, ya que hay una gran cantidad de URLs que se usan para descargar el mismo fichero.

Todas las URLs actualmente de la tabla excepto google están offline, por lo que se entiende que fueron levantadas durante los ataques pero que al pasar cierto tiempo han cambiado de IP, algo que ha dificultado el análisis.

Se ha detectado que algunos atacantes intentan camuflar la extensión del fichero

Ficheros descargados	Peligrosidad	Frecuencia
85.159.237.19/bins.sh	5	437
195.22.126.16/ssh1.txt	8	414
185.234.217.21/ssh1.txt	6	357
google.com	0	337
107.174.34.70/Swag.sh	3	313
222.186.139.216:9960/chongfu.sh	2	217
203.146.208.208/drago/images/.ssh/y.txt	9	214
51.15.48.238/armex.sh	1	188
77.247.178.189/bins.sh	3	144
107.174.34.68/Swag.sh	9	127

Tabla 4.7: Tabla de ficheros descargados

que se descargan para evitar posibles firewalls o IDSs; por ejemplo, un atacante se descargó el fichero `idip.do.am/cache.pdf` que a simple vista parece un fichero *PDF* con un nombre inofensivo, pero internamente es un ShellBot escrito en Perl para realizar DDoS usando IRC. Podemos ver el tipo de fichero a continuación:

```
[procamora@4770K TFG]$ file cache.pdf
cache.pdf: a /usr/bin/perl script executable (binary data)
```

También se ha hecho un análisis manual de algunos ficheros descargados para entender cuál es el objetivo de los atacantes. El primer problema es que al ser logs antiguos la mayoría de URLs están offline en estos momentos, como ya se ha comentado previamente, pero buscando por el nombre del script en bases de datos de malware, como es el caso de virusign[4] se han podido encontrar algunos de estos ficheros y descargar para su posterior análisis. En general, la gran mayoría de scripts analizados tienen como objetivo usarse para realizar un DDoS, un grupo pequeño son para descargarse otros scripts que no han sido analizados y ejecutarlos después, y un número muy reducido de ellos busca replicarse en otros dispositivos, algo que con Cowrie no ha sido posible ya que estos bots suelen usar los comandos *curl* y *scp* para esta tarea, pero ninguno de los dos está implementado en Cowrie actualmente. Podemos ver unos ejemplos a continuación de los ficheros descargados:

- Los ficheros *ssh1.txt*, *y.txt* y *cache.pdf* son scripts programados en Perl cuya finalidad es realizar un DDoS usando un canal IRC.
- Los ficheros *bins.sh*, *Swag.sh*, *donkyballs.sh* y *sunlessesbins.sh* son scripts programados en Bash cuya finalidad es descargarse una serie de binarios como apache2, tftp o telnetd y ejecutarlos. El antivirus los detecta como pertenecientes a la botnet Mirai que ataca a dispositivos IoT.
- Los ficheros *chongfu.sh*, *hammer.py* y *viteza.py* son scripts programados en Bash

[4]La descarga de malware en esta base de datos está protegida por contraseña, siendo esta *infected/infected*, y la contraseña del fichero 7z que lo contiene es *infected*

y Python cuya finalidad es realizar un DDoS, donde viteza.py, además, es capaz de realizar un test de velocidad para obtener el ancho de banda disponible del dispositivo infectado.
- El fichero *NET10086* es un binario ELF de 32 bits clasificado como troyano que afecta a Linux.

4.5. Experimentación con técnicas de Machine Learning

En esta sección se va a explicar el análisis de los datos realizados con Machine Learning y los resultados obtenidos de este. Se han realizado tres experimentos diferentes usando dos datasets.

4.5.1. Primer experimento

Los algoritmos de cifrado de clave y MAC son especificados a Cowrie en la configuración, esto imposibilita que el atacante pueda elegir uno que no sea seguro, ya que tiene que estar en una lista predefinida, limitando mucho la variedad de ésta y haciendo que estos algoritmos no puedan ser features discriminantes entre sesiones, por eso se han tenido que descartar como feature, aunque [19] lo considere relevante. Como feature, se ha considerado más relevante utilizar los comandos ejecutados.

Como primer experimento se va a usar el dataset generado localmente, éste cuenta con 1.561.311 sesiones, y se busca comprobar si hay alguna relación entre las sesiones. Se usó el algoritmo de clustering (k-means), ya que el dataset no estaba etiquetado, con el objetivo de ver si se podían separar en k grupos. Para ello, se tuvieron que realizar las siguientes acciones:
- Resolver el problema de tener un número elevado de dimensiones (por las features), que hacían que no se pudiese representar gráficamente. Esto se resuelve usando el algoritmo de reducción de dimensiones UMAP.
- Obtener el k óptimo, usando el *método del codo*, que dio entre 6 y 7 clusters.
- Verificar que el número de clusters es correcto; se ha utilizado la técnica de tuned hyperparameter.

Una vez aplicados todos estos pasos, podemos ver el resultado obtenido en la Figura 4.13, donde se puede ver que no ha sido posible particionar en grupos bien diferenciados. Por este motivo se va a realizar un etiquetado usando un sistema experto basado en reglas. Éste etiqueta las sesiones según los comandos usados, para lo que se usará el estudio realizado por Kheirkhah et al. [21]. En la sección 4.3.1 se explica cuáles son las reglas utilizadas para este etiquetado.

Los algoritmos de clasificación usados han sido: k-NN, Decision Tree (DT), Random Forest (RF) y SVM, todos se basan en aprendizaje supervisado, por lo que usará el etiquetado que se aplicó previamente al dataset. Después se dividió éste en dos datasets: uno de entrenamiento compuesto por el 80% de los datos (17.080 vectores) y uno de evaluación compuesto por el 20% de los datos (4.270 vectores).

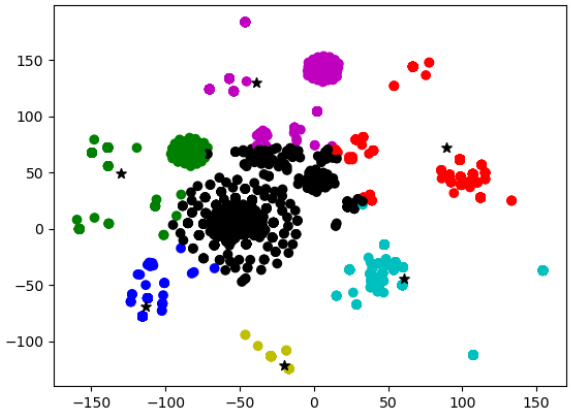


Figura 4.13: Resultado del clustering aplicado a las sesiones

Al igual que con el clustering fue necesario aplicar la técnica de tuned hyperparameter para el refinado de la configuración óptima de cada algoritmo. Los valores obtenidos se pueden ver en la Tabla 4.8.

Algoritmo	Precision	Recall	F1-Score	Accuracy
k-NN	**0,4991**	**0,4662**	**0,4709**	**0,4698**
Decision Tree	0,3287	0,3298	0,3276	0,3312
Random Forest	0,3285	0,3299	0,3275	0,3309
SVM	0,3286	0,3298	0,3276	0,3311

Tabla 4.8: Resultados del experimento 1 con dataset propio

Tras obtener estos resultados tan bajos, siendo todos menores del 50% se usó Random Forest para comprobar la relevancia de cada una de las features, obteniendo que el 60% de ellas no tenía relevancia para determinar la clase del vector.

4.5.2. Segundo experimento

Como segundo experimento se usó el algoritmo de generación de reglas A-priori, que tiene como objetivo encontrar un conjunto de reglas que cumplan una serie de características y pesos indicados, ambos indicados como parámetros. Se aplicó este algoritmo a los comandos para ver cuál es el objetivo de los atacantes, pudiendo clasificar éste de la siguiente forma:

- Lectura en disco.
- Escritura en disco.
- Obtención de información del SO.
- Conexiones y descargas en internet.
- Compilación o instalación de programas.
- Ejecución de programas.
- Eliminar procesos del SO.

Además de los comandos, se añadieron como features las credenciales usadas y el cliente SSH, debido a que Sadasivam et al. [19] consideran estos como una característica capaz de relacionar diferentes ataques.

Para el nuevo etiquetado del dataset se modificó el sistema experto basado en reglas usado anteriormente. Finalmente, en la Tabla 4.9 se pueden ver los resultados obtenidos.

Algoritmo	Precision	Recall	F1-Score	Accuracy
k-NN	0,7674	0,8103	0,7526	0,7515
Decision Tree	0,8297	**0,8392**	**0,8205**	**0,8609**
Random Forest	0,8297	**0,8392**	**0,8205**	**0,8609**
SVM	**0,8298**	0,8391	0,8204	0,8607

Tabla 4.9: Resultados del experimento 2 con dataset propio

Estos nuevos resultados mejoran considerablemente respecto al anterior experimento, pero siguen sin ser aceptables. Se ha considerado que, al usar un dataset propio que lleva apenas unos pocos meses capturando datos, puede ser el motivo de que obtengamos malos resultados, ya que tiene una escasa variedad. Se vio en la sección 4.4 que el 75% de las sesiones provenían de Irlanda y Reino Unido. Por este motivo se ha probado a realizar el mismo experimento, pero en esta ocasión con el dataset de scriptzteam [9] que lleva mucho más tiempo capturando datos. Con este dataset se han mejorado considerablemente los resultados, llegando a una precisión de 98,74%. Usando el algoritmo Decision Tree, podemos ver la muestra completa en la Tabla 4.10, con lo que podemos concluir que los datos del honeypot que se ha montado son insuficientes por el momento.

Algoritmo	Precision	Recall	F1-Score	Accuracy
k-NN	0,9859	0,9573	0,9708	0,9765
Decision Tree	**0,9874**	**0,9574**	**0,9714**	**0,9767**
Random Forest	0,9859	0,9573	0,9708	0,9765
SVM	**0,9874**	0,9573	**0,9714**	0,9865

Tabla 4.10: Resultados del experimento 2 con dataset scriptzteam

4.5.3. Tercer experimento

Pese a que se obtuvieron muy buenos resultados con el anterior experimento, se ha conseguido contar con nueva información, que será usada como feature. Esta es la siguiente:

- Tiempo total de la sesión. Sadasivam et al. [19] considera esta feature relevante.
- Número de motores de antivirus que detectan una descarga como malware. Actualmente no se ha visto ningún artículo que hable de esta feature, solo [21]

hace un análisis estático del malware, pero puede que sea relevante a la hora de clasificar ataques.
- País al que pertenece la sesión. Al igual que la anterior, no se ha indicado en ningún artículo que se haya probado, pese a que siempre es mostrada en todas las clasificaciones manuales.
- Versión SSH del cliente usado. Sadasivam et al. [19] lo considera una feature relevante.

Usando estas nuevas features unidas a la del experimento anterior se han conseguido unos resultados mejores que pueden verse en la Tabla 4.11.

Algoritmo	**Precision**	**Recall**	**F1-Score**	**Accuracy**
k-NN	0,9666	0,9642	0,9652	0,9775
Decision Tree	0,9968	0,9968	0,9968	0,9979
Random Forest	**0,9982**	**0,9973**	**0,9977**	**0,9989**
SVM	0,9917	0,9883	0,9900	0,9948

Tabla 4.11: Resultados del experimento 3 con dataset scriptzteam

Como principales conclusiones de los experimentos presentados más arriba, se han conseguido precisiones muy buenas, siendo la precisión más baja del 96% y la mayor del 99%. Se ha podido ver que la diferencia entre los datasets es algo importante, siendo el dataset generado localmente de baja calidad, ya que ha estado poco tiempo capturando datos y hay una gran cantidad de datos repetidos, haciendo que las sesiones útiles sean inferiores de las esperadas. En cambio, el dataset de scriptzteam [9], aunque tiene una menor cantidad de datos, ha estado más tiempo capturando ataques y esto lo hace de mejor calidad.

5. Conclusiones y vías futuras

En este trabajo se ha presentado un sistema capaz de analizar los logs generados por el honeypot SSH Cowrie con el objetivo de detectar y clasificar los ataques recibidos.

Para cumplir las ideas propuestas en este trabajo, ha sido necesario realizar una serie de pasos. En primer lugar, se realizó una investigación del funcionamiento de los honeypots y el estado del arte relacionado. Enlazando con esa parte, se investigó sobre el estado del arte relacionado con aplicar técnicas de Machine Learning a datos obtenidos de honeypots. Entender estos algoritmos ha requerido un estudio detallado del funcionamiento de cada uno de ellos.

Una vez analizado el estado del arte asociado a ambos, se ha realizado y presentado un estudio de los logs generados por Cowrie, donde se ha visto que no existe ninguna herramienta capaz de convertirlos a un formato estructurado, por lo que se ha tenido que desarrollar una. Cuando los logs se han convertido al formato JSON, se ha realizado una búsqueda de las distintas bases de datos NoSQL capaz de almacenar esos datos, llegando a la conclusión de que Elasticsearch es la mejor opción ya que cuenta con Kibana como herramienta de visualización de los datos, muy usada actualmente.

Tras seleccionar esta opción se ha procedido a estudiar el funcionamiento de esta solución y realizar un despliegue del honeypot Cowrie junto con Elasticsearch y Kibana usando la aplicación Docker, previo estudio de esta. Después, se ha procedido a insertar toda la información en Elasticsearch y con Kibana donde generar todas las tablas y gráficas de datos representativos según el estudio del arte.

Una vez realizada toda la clasificación manual, se ha procedido a seleccionar y definir las features más relevantes según el estado del arte y ver qué precisión obteníamos a la hora de agrupar con distintos algoritmos de Machine Learning. En un primer experimento no se obtuvieron buenos resultados, pero en los posteriores sí. Esto se consiguió cambiando a un dataset más variado en el tiempo y con nuevas features, que se obtuvieron diseñando una API que analizaba los ficheros descargados por los atacantes usando VirusTotal.

Las principales conclusiones obtenidas del desarrollo de este trabajo son las siguientes:

- El protocolo SSH es el que más ataques recibe siendo además de los que cuenta con más dispositivos, esto se incrementa cada vez más gracias al uso de dispositivos IoT, que muchas veces no se cambian las credenciales por defecto, lo cual es peligroso como ya demostró la botnet Mirai, que realizó el mayor ataque DDoS de la historia, usando dispositivos IoT. Por este motivo se tendría que extremar la precaución y no dejar nunca contraseñas por defecto.

- Los datos obtenidos de un honeypot pueden ser muy valiosos, pero es necesaria una configuración correcta; en este caso, al generar estos datos en un formato no estructurado ha sido necesario desarrollar un programa capaz de estructurarlos, algo que ha supuesto la mitad de esfuerzo para la realización de este trabajo si se hubieran generado en formato JSON, cosa que es posible indicándolo en la configuración. Así, se habría podido pasar directamente al proceso de introducir los datos en una base de datos para su posterior análisis, dedicando mucho más tiempo a esta fase que es más importante.
- Con el análisis de los datos se ha visto que el 83% de las sesiones han usado las mismas credenciales y apenas se han obtenido datos valiosos de esas sesiones. Sería recomendable crear una mejor política de acceso al honeypot para que los datos que se obtengan sean más relevantes.
- Realizar un análisis en tiempo real de los ficheros descargados por atacantes puede ser muy útil ya que eres capaz de identificar cuáles son las intenciones de los atacantes en la actualidad.

En las vías futuras abiertas para mejorar el actual sistema, se pueden encontrar las siguientes posibles mejoras:

- Usar un honeypot de alta interacción (HIH) que no esté limitado por los comandos programados. De esta forma también seríamos capaces de capturar flujos de datos.
- En caso de no contar con un *HIH* y tener que usar Cowrie, sería muy interesante implementarlo a modo de proxy para que cada nueva sesión creara un contenedor con Docker y se conectase a él. Así permitiría mayor funcionalidad en vez de usar un entorno virtual como el que usa Cowrie. Además, permitiría capturar las llamadas realizadas al sistema, lo que puede valer como feature en Machine Learning.
- Cowrie es fácilmente detectable como honeypot independientemente de cómo se configure; sería una buena idea contactar con el desarrollador para que haga unas modificaciones y que sea más difícil detectarlo. En el "Anexo IV. Detectar honeypot Cowrie" se explica cómo detectarlo.
- Usar herramientas como *fail2ban* puede ser interesante para bloquear aquellas conexiones que están conectándose cada poco tiempo, ya que no dan información útil y generan gran cantidad de datos. Podemos ver una implementación para Cowrie en `https://github.com/RoastingMalware/cowrie-fail2ban`.
- Crear una honeynet distribuida por distintos países: al estar analizando los ataques recibidos desde distintos puntos, se pueden realizar análisis geográficos interesantes como, por ejemplo, ver si según el país origen del honeypot cambia el país origen del atacante.
- Mezclar distintos tipos de honeypots para realizar un análisis de varios protocolos, no centrándose únicamente en uno. Un ejemplo de esto, unido al punto anterior de crear una honeynet, sería: `https://github.com/cribdragg3r/ModernHoneyNet`.

Bibliografía

[1] Marcin Nawrocki, Matthias Wählisch, Thomas C Schmidt, Christian Keil, and Jochen Schönfelder. A survey on honeypot software and data analysis. *arXiv preprint arXiv:1608.06249*, 2016.

[2] Honeynet Project. Know Your Enemy: Honeynets, 2006. Disponible en: `http://old.honeynet.org/papers/honeynet/`.

[3] Jorge Ismael Campoverde Armijos and Pedro Hecht. Honeypot como herramienta de prevención de ciberataques, 2015. Disponible en: `http://bibliotecadigital.econ.uba.ar/download/tpos/1502-1212_CampoverdeArmijosJI.pdf`.

[4] Michel Oosterhof. Cowrie honeypot, 2015. Disponible en: `https://www.cowrie.org/posts/2015-07-05-cowrie`.

[5] Michel Oosterhof. Cowrie, 2018. Disponible en: `https://github.com/cowrie/cowrie`.

[6] JNIC. Jornadas Nacionales de Investigación en Ciberseguridad Edición 2018/19, 2018. Disponible en: `https://transferencia.jnic.es/edicion-2018`.

[7] INCIBE. IoT: riesgos del internet de los trastos, 2017. Disponible en: `https://www.incibe.es/protege-tu-empresa/blog/iot-riesgos-del-internet-los-trastos`.

[8] Bleeping Computer. Mirai utiliza Aboriginal Linux para infectar múltiples plataformas, 2018. Disponible en: `https://www.seguridad.unam.mx/mirai-utiliza-aboriginal-linux-para-infectar-multiples-plataformas`.

[9] scriptzteam. REALTiME SSH HoneyPot v1.1, 2017. Disponible en: `https://github.com/scriptzteam/REALTiME-SSH-HoneyPot_v1.1`.

[10] Symantec employee. What is a honeypot? How it can lure cyberattackers, 2019. Disponible en: `https://us.norton.com/internetsecurity-iot-what-is-a-honeypot.html`.

[11] RC Joshi and Anjali Sardana. *Honeypots: A new paradigm to information security*. CRC Press, 2011.

[12] Solomon Z Melese and PS Avadhani. Honeypot system for attacks on SSH protocol. *International Journal of Computer Network and Information Security*, 8(9): 19, 2016.

[13] Tomas Sochor and Matej Zuzcak. Study of internet threats and attack methods using honeypots and honeynets. In *International Conference on Computer Networks*, page 119–120. Springer, 2014.

[14] Georg Wicherski. Medium interaction honeypots. *German Honeynet Project*, 2006.

[15] Eduardo Esteban Casanovas and Carlos Ignacio Tapia. Honeynets como herramienta de prevención e investigación de ciberataques, 2013. Disponible en: `http://conaiisi.unsl.edu.ar/2013/138-535-1-DR.pdf`.

[16] Tomasz Grudziecki, Paweł Jacewicz, Łukasz Juszczyk, Piotr Kijewski, Paweł Pawliński, and CERT Polska. Proactive Detection of Security Incidents. In European Network and Information Security Agency (ENISA). *enisa*, pages 96–106, 2012.

[17] Lukas Rist, Johnny Vestergaard, Daniel Haslinger, Andrea Pasquale, and John Smith. CONPOT ICS/SCADA Honeypot, 2019. Disponible en: `http://conpot.org/`.

[18] Meng Wang, Javier Santillan, and Fernando Kuipers. ThingPot: an interactive Internet-of-Things honeypot. *arXiv preprint arXiv:1807.04114*, 2018.

[19] Gokul Kannan Sadasivam, Chittaranjan Hota, and Bhojan Anand. Honeynet Data Analysis and Distributed SSH Brute-Force Attacks. In *Towards Extensible and Adaptable Methods in Computing*, page 107–118. Springer, 2018.

[20] ElevenPaths. Honeypotting, un oído en Internet, 2018. Disponible en: `https://es.slideshare.net/elevenpaths/honeypotting-un-odo-en-internet`.

[21] Esmaeil Kheirkhah, SM Poustchi Amin, HA Jahanshahi Sistani, and Haridas Acharya. An experimental study of SSH attacks by using honeypot decoys. *Indian Journal of Science and Technology*, 6(12):5567–5578, 2013.

[22] Seamus Dowling, Michael Schukat, and Hugh Melvin. A ZigBee honeypot to assess IoT cyberattack behaviour. In *2017 28th Irish Signals and Systems Conference (ISSC)*, page 1–6. IEEE, 2017.

[23] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2009.

[24] Juan Ignacio Bagnato. Guía de Aprendizaje, 2019. Disponible en: `https://www.aprendemachinelearning.com/guia-de-aprendizaje/`.

[25] scikit learn. Official documentation. Decision Trees (DT), 2019. Disponible en: https://scikit-learn.org/stable/modules/tree.html.

[26] scikit learn. Official documentation. Forests of randomized trees, 2019. Disponible en: https://scikit-learn.org/stable/modules/ensemble.html#forest.

[27] scikit learn. Official documentation. Neighborhood Components Analysis (k-NN), 2019. Disponible en: https://scikit-learn.org/stable/modules/neighbors.html#neighborhood-components-analysis.

[28] scikit learn. Official documentation. Support vector machines (SVM), 2019. Disponible en: https://scikit-learn.org/stable/modules/svm.html.

[29] scikit learn. Official documentation. K-means, 2019. Disponible en: https://scikit-learn.org/stable/modules/clustering.html#k-means.

[30] scikit learn. Official documentation. Principal component analysis (PCA), 2019. Disponible en: https://scikit-learn.org/stable/modules/decomposition.html#pca.

[31] scikit learn. Official documentation. t-distributed Stochastic Neighbor Embedding (t-SNE), 2019. Disponible en: https://scikit-learn.org/stable/modules/manifold.html#t-sne.

[32] Leland McInnes. Uniform Manifold Approximation and Projection for Dimension Reduction (UMAP), 2019. Disponible en: https://umap-learn.readthedocs.io/.

[33] Gokul Kannan Sadasivam, Chittaranjan Hota, and Bhojan Anand. Detection of Severe SSH Attacks Using Honeypot Servers and Machine Learning Techniques. *Software Networking*, 2018(1):79–100, 2018.

[34] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization. In *ICISSP*, page 108–116. Springer, 2018.

[35] Adrian Pauna and Ion Bica. RASSH-Reinforced adaptive SSH honeypot. In *2014 10th International Conference on Communications (COMM)*, page 1–6. IEEE, 2014.

[36] Seamus Dowling, Michael Schukat, and Enda Barrett. Using Reinforcement Learning to Conceal Honeypot Functionality. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, page 341–355. Springer, 2018.

[37] Javier Terrero Prior. NOSQL vs SQL. Key differences and when to choose each, 2016. Disponible en: https://pandorafms.com/blog/nosql-vs-sql-key-differences/.

[38] Elastic B.V. The heart of the Elastic Stack, 2019. Disponible en: https://www.elastic.co/products/elasticsearch.

[39] Elastic B.V. Your window into the Elastic Stack, 2019. Disponible en: https://www.elastic.co/products/kibana.

[40] Pablo J. Rocamora. Docker Hub Cowrie, 2019. Disponible en: https://hub.docker.com/r/procamora/cowrie.

[41] Simonas Kareiva. Cowrie logs, 2018. Disponible en: https://github.com/kareiva/cowrie-logs.

[42] Rapid7 Labs. Rapid7 Heisenberg Cloud Honeypot Cowrie Logs, 2016. Disponible en: https://opendata.rapid7.com/heisenberg.cowrie/.

[43] Pablo J. Rocamora and José María Jorquera Valero. IN1-INCIBE, 2019. Disponible en: https://github.com/procamora/IN1-INCIBE.

[44] Geir Arne Hjelle. Cool New Features in Python 3.7, 2018. Disponible en: https://realpython.com/python37-new-features/.

[45] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. PEP 484 – Type Hints, 2014. Disponible en: https://www.python.org/dev/peps/pep-0484/.

[46] Łukasz Langa. PEP 563 – Postponed Evaluation of Annotations, 2017. Disponible en: https://www.python.org/dev/peps/pep-0563/.

[47] Eric V. Smith. PEP 498 – Literal String Interpolation, 2015. Disponible en: https://www.python.org/dev/peps/pep-0498/.

[48] Eric V. Smith. PEP 557 – Data Classes, 2017. Disponible en: https://www.python.org/dev/peps/pep-0557/.

[49] GeoLite2 Free Downloadable Databases, 2019. Disponible en: https://dev.maxmind.com/geoip/geoip2/geolite2/.

[50] Joshua Faust. Distributed Analysis of SSH Brute Force and Dictionary Based Attacks, 2018. Disponible en: https://repository.stcloudstate.edu/msia_etds/56/.

A. Anexo I. Escenario Docker

En este anexo se muestra el fichero de *docker-compose* que se ha utilizado para el proyecto, con el cual se levantan todos los servicios necesarios usando contenedores. Estos servicios son:

- **elasticsearch**: Se usa la imagen oficial de esta base de datos. Como configuración destacable tenemos que asignamos 1 GB a la máquina virtual de Java para que funcione mejor y que se usa el puerto 9200 y 9300.
- **kibana**: Se usa la imagen oficial de este servicio web. Como configuración destacable tenemos que se especifican en las variables de entorno donde se encuentra la base de datos a la que se va a conectar y el puerto que se usa, que es el 5601.
- **rest_malware**: Para usar este contenedor primero es necesario crear la imagen de él. Esto se hace usando el Dockerfile que se muestra en el código A.2. Como configuración destacable tenemos que usar el puerto 8080. Como este contenedor usa Flask como servidor web, para obtener un log de mejor calidad, se activa la salida por consola permitiendo que usen colores.
- **cowrie**: Se usa una imagen de Cowrie modificada para este proyecto. Como configuración destacable tenemos que la configuración de Cowrie se pasa montando los volúmenes virtuales y que se usa el puerto 2222.

Código A.1: docker-compose usado en el proyecto

```yaml
version: '2.2'

services:
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:6.6.0
    container_name: elasticsearch
    environment:
      ES_JAVA_OPTS: "-Xmx1024m -Xms1024m"
    volumes:
      - ./elasticsearch/custom_elasticsearch.yml:/usr/share/elasticsearch/config/elasticsearch.yml
    ports:
      - 9200:9200
      - 9300:9300
    networks:
      - elk

  kibana:
    image: docker.elastic.co/kibana/kibana:6.6.0
    container_name: kibana
    environment:
      - "server.name=kibana"
      - "elasticsearch.hosts=http://elasticsearch:9200"
    ports:
      - 5601:5601
    networks:
```

```
26        - elk
27      links:
28        - elasticsearch
29
30    rest_malware:
31      build:
32        context: malware/
33        dockerfile: Dockerfile
34      container_name: rest_malware
35      volumes:
36        - ./malware/:/root/
37      ports:
38        - 8080:8080
39      networks:
40        - elk
41      tty: true
42      environment:
43        - TERM=xterm-256color
44
45    cowrie:
46      image: procamora/cowrie:latest
47      container_name: cowrie
48      volumes:
49        - ./cowrie/etc/:/cowrie/cowrie-git/etc/
50        - ./cowrie/log/:/cowrie/cowrie-git/var/log/cowrie/
51      ports:
52        - 2222:2222
53      networks:
54        - elk
55
56  networks:
57    elk:
58      driver: bridge
```

Código A.2: Dockerfile usado para crear el contenedor rest_malware

```
1  FROM python:3.6-alpine
2
3  COPY requirements.txt /root
4  COPY rest.py /root
5  COPY connect_sqlite.py /root
6  COPY virustotal.py /root
7  COPY malware.db.sql /root
8
9  WORKDIR /root
10
11 RUN pip install -r requirements.txt
12
13 EXPOSE 8080
14
15 CMD ["python", "-u", "rest.py"]
```

B. Anexo II. Ejemplo de log de Cowrie

En este anexo se muestra un ejemplo de cómo sería el log que genera Cowrie para una conexión. A continuación, se realizará una explicación más pormenorizada:

- Línea 1: Se inicia la conexión, se puede ver la IP del atacante y el puerto usado (172.22.0.1:59128).
- Líneas 2-7: Se obtiene el cliente SSH del atacante y se muestran los algoritmos de cifrado y MAC que se usan.
- Líneas 8-19: Muestran el proceso de login, el cual finaliza en la línea 19 usando las credenciales *root/test1* que obtienen un login correcto.
- Líneas 23-34: Una vez permitido el login se obtiene una shell en la que se establecen una serie de variables que envía el atacante. Entre ellas tenemos el tamaño de la consola y el lenguaje de esta, siendo en este caso castellano.
- Líneas 35-36: El cliente pide la ejecución del comando *whoami*, qie es encontrado y ejecutado.
- Líneas 37-41: El cliente pide la ejecución del comando *wget*, que es encontrado y ejecutado. La ejecución de este comando hace que se inicie el proceso de descarga del fichero y, una vez finalizado, se obtiene el hash y el fichero es guardado en un directorio.
- Líneas: 46-54: Se cierra la conexión SSH.

Código B.1: Log en formato textual generado por Cowrie

```
1 2019-07-29T21:01:02.016977Z [cowrie.ssh.factory.CowrieSSHFactory] New connection: 172.22.0.1:59128 ↩
      ↪ (172.22.0.2:2222) [session: ea75ae90eab4]
2 2019-07-29T21:01:02.018429Z [HoneyPotSSHTransport,1,172.22.0.1] Remote SSH version: b'SSH-2.0-OpenSSH_8.0'
3 2019-07-29T21:01:02.020023Z [HoneyPotSSHTransport,1,172.22.0.1] SSH client hassh fingerprint: 2↩
      ↪ e0b1db6a436814a699e5c982f48226a
4 2019-07-29T21:01:02.022668Z [HoneyPotSSHTransport,1,172.22.0.1] kex alg, key alg: b'ecdh-sha2-nistp256' b'ssh-↩
      ↪ rsa'
5 2019-07-29T21:01:02.022780Z [HoneyPotSSHTransport,1,172.22.0.1] outgoing: b'aes256-ctr' b'hmac-sha2-256' b'none↩
      ↪ '
6 2019-07-29T21:01:02.022903Z [HoneyPotSSHTransport,1,172.22.0.1] incoming: b'aes256-ctr' b'hmac-sha2-256' b'none↩
      ↪ '
7 2019-07-29T21:01:03.830628Z [HoneyPotSSHTransport,1,172.22.0.1] NEW KEYS
8 2019-07-29T21:01:03.833181Z [HoneyPotSSHTransport,1,172.22.0.1] starting service b'ssh-userauth'
9 2019-07-29T21:01:03.838167Z [SSHService b'ssh-userauth' on HoneyPotSSHTransport,1,172.22.0.1] b'root' trying ↩
      ↪ auth b'none'
10 2019-07-29T21:01:03.842595Z [SSHService b'ssh-userauth' on HoneyPotSSHTransport,1,172.22.0.1] b'root' trying ↩
      ↪ auth b'publickey'
11 2019-07-29T21:01:03.846020Z [SSHService b'ssh-userauth' on HoneyPotSSHTransport,1,172.22.0.1] public key ↩
      ↪ attempt for user b'root' of type b'ssh-rsa' with fingerprint 88:08:8e:fc:14:be:40:56:8c:d1:a1:8c↩
      ↪ :92:51:57:5a
12 2019-07-29T21:01:03.851589Z [SSHService b'ssh-userauth' on HoneyPotSSHTransport,1,172.22.0.1] b'root' failed ↩
      ↪ auth b'publickey'
```

```
2019-07-29T21:01:03.852187Z [SSHService b'ssh-userauth' on HoneyPotSSHTransport,1,172.22.0.1] reason: ('Incorrect signature', None)
2019-07-29T21:01:03.854774Z [SSHService b'ssh-userauth' on HoneyPotSSHTransport,1,172.22.0.1] b'root' trying auth b'publickey'
2019-07-29T21:01:03.857591Z [SSHService b'ssh-userauth' on HoneyPotSSHTransport,1,172.22.0.1] public key attempt for user b'root' of type b'ssh-rsa' with fingerprint 34:f2:36:12:9c:16:42:c6:e5:49:ee:cb:e9:74:d9:8c
2019-07-29T21:01:03.864921Z [SSHService b'ssh-userauth' on HoneyPotSSHTransport,1,172.22.0.1] b'root' failed auth b'publickey'
2019-07-29T21:01:03.865524Z [SSHService b'ssh-userauth' on HoneyPotSSHTransport,1,172.22.0.1] reason: ('Incorrect signature', None)
2019-07-29T21:01:10.574034Z [SSHService b'ssh-userauth' on HoneyPotSSHTransport,1,172.22.0.1] b'root' trying auth b'password'
2019-07-29T21:01:10.576714Z [SSHService b'ssh-userauth' on HoneyPotSSHTransport,1,172.22.0.1] login attempt [b'root'/b'test1'] succeeded
2019-07-29T21:01:10.578162Z [SSHService b'ssh-userauth' on HoneyPotSSHTransport,1,172.22.0.1] Initialized emulated server as architecture: linux-x86-lsb
2019-07-29T21:01:10.579664Z [SSHService b'ssh-userauth' on HoneyPotSSHTransport,1,172.22.0.1] b'root' authenticated with b'password'
2019-07-29T21:01:10.579986Z [SSHService b'ssh-userauth' on HoneyPotSSHTransport,1,172.22.0.1] starting service b'ssh-connection'
2019-07-29T21:01:10.580796Z [SSHService b'ssh-connection' on HoneyPotSSHTransport,1,172.22.0.1] got channel b'session' request
2019-07-29T21:01:10.581137Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport,1,172.22.0.1] channel open
2019-07-29T21:01:11.158164Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport,1,172.22.0.1] pty request: b'xterm-256color' (27, 212, 0, 0)
2019-07-29T21:01:11.158371Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport,1,172.22.0.1] Terminal Size: 212 27
2019-07-29T21:01:11.159474Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport,1,172.22.0.1] request_env: b'LANGUAGE'=b'en_US:es'
2019-07-29T21:01:11.160455Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport,1,172.22.0.1] request_env: b'LC_MONETARY'=b'es_ES.UTF-8'
2019-07-29T21:01:11.161332Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport,1,172.22.0.1] request_env: b'LANG'=b'en_US.UTF-8'
2019-07-29T21:01:11.162189Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport,1,172.22.0.1] request_env: b'LC_MEASUREMENT'=b'es_ES.UTF-8'
2019-07-29T21:01:11.163055Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport,1,172.22.0.1] request_env: b'LC_TIME'=b'es_ES.UTF-8'
2019-07-29T21:01:11.163917Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport,1,172.22.0.1] request_env: b'LC_COLLATE'=b'es_ES.UTF-8'
2019-07-29T21:01:11.164780Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport,1,172.22.0.1] request_env: b'LC_NUMERIC'=b'es_ES.UTF-8'
2019-07-29T21:01:11.165675Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport,1,172.22.0.1] getting shell
2019-07-29T21:01:16.817860Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport,1,172.22.0.1] CMD: whoami
2019-07-29T21:01:16.820976Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport,1,172.22.0.1] Command found: whoami
2019-07-29T21:01:44.852720Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport,1,172.22.0.1] CMD: wget https://procamora.github.io/code/optimiza_pdf.sh
2019-07-29T21:01:44.855605Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport,1,172.22.0.1] Command found: wget https://procamora.github.io/code/optimiza_pdf.sh
2019-07-29T21:01:44.857056Z [cowrie.commands.wget.HTTPProgressDownloader#info] Starting factory <HTTPProgressDownloader: b'https://procamora.github.io/code/optimiza_pdf.sh'>
2019-07-29T21:01:45.067048Z [HTTPPageDownloader (TLSMemoryBIOProtocol),client] Downloaded URL (b'https://procamora.github.io/code/optimiza_pdf.sh') with SHA-256 e0806dff05f299106aacc618db24d5c553e92da27e77afb6ff9df60cee5e1dd6 to var/lib/cowrie/downloads/e0806dff05f299106aacc618db24d5c553e92da27e77afb6ff9df60cee5e1dd6
2019-07-29T21:01:45.072739Z [cowrie.commands.wget.HTTPProgressDownloader#info] Stopping factory <HTTPProgressDownloader: b'https://procamora.github.io/code/optimiza_pdf.sh'>
2019-07-29T21:01:48.543263Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport,1,172.22.0.1] CMD: cat optimiza_pdf.sh
2019-07-29T21:01:48.546237Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport,
```

```
      ↪ ,1,172.22.0.1] Command found: cat optimiza_pdf.sh
44 2019-07-29T21:01:50.628576Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport↩
      ↪ ,1,172.22.0.1] CMD: exit
45 2019-07-29T21:01:50.631884Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport↩
      ↪ ,1,172.22.0.1] Command found: exit
46 2019-07-29T21:01:50.632200Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport↩
      ↪ ,1,172.22.0.1] exitCode: 0
47 2019-07-29T21:01:50.632313Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport↩
      ↪ ,1,172.22.0.1] sending request b'exit-status'
48 2019-07-29T21:01:50.632933Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport↩
      ↪ ,1,172.22.0.1] Closing TTY Log: var/lib/cowrie/tty/23↩
      ↪ ead11d1caa3fa7475cb12822d4a63640c0770d4664ee9bcc09fa673e04a2aa after 39 seconds
49 2019-07-29T21:01:50.633974Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport↩
      ↪ ,1,172.22.0.1] sending close 0
50 2019-07-29T21:01:50.634628Z [SSHChannel session (0) on SSHService b'ssh-connection' on HoneyPotSSHTransport↩
      ↪ ,1,172.22.0.1] remote close
51 2019-07-29T21:01:50.634929Z [HoneyPotSSHTransport,1,172.22.0.1] Got remote error, code 11 reason: b'↩
      ↪ disconnected by user'
52 2019-07-29T21:01:50.635363Z [HoneyPotSSHTransport,1,172.22.0.1] avatar root logging out
53 2019-07-29T21:01:50.635578Z [HoneyPotSSHTransport,1,172.22.0.1] connection lost
54 2019-07-29T21:01:50.635689Z [HoneyPotSSHTransport,1,172.22.0.1] Connection lost after 48 seconds
```

C. Anexo III. API académica de VirusTotal

Los pasos que se han realizado (en inglés) para obtener la API académica de VirusTotal han sido los siguientes:

Paso 1. Registrarse en VirusTotal

Lo primero que necesitaremos es registrarnos con el correo académico. Esto se hará en la siguiente URL: `https://www.virustotal.com/gui/join-us`

Paso 2. Solicitad API académica

Solicitar en el siguiente formulario (link) una API académica, explicando en qué consiste el proyecto.

Paso 3. Responder al correo que te mandan

Pasado un tiempo, en mi caso menos de una hora, se ha recibido un correo en el que se indica que tienen dos formas para acceder a los datos:

- Acceso a una API académica que incluye una mayor cantidad de peticiones por minuto.
- Acceso a una carpeta de malware para utilizarla en la investigación académica.

Nos indican que para ser considerado para cualquiera de esas dos opciones tenemos que responder a ese ticket con la siguiente información:

1. La dirección de correo académica que se ha usado para registrarse en VirusTotal. (por ejemplo, *name@university.edu*).
2. Indicar el tipo de API al que se quiere acceder: API académica o repositorio de malware.
3. Describir el uso que se le va a dar a la API y el proyecto que se está realizando.
4. Rellenar este formulario: link.
5. Enviar un correo electrónico no académico (gmail) si hemos seleccionado el repositorio de malware (opcional).

Finalmente, nos piden que les mencionemos en nuestra publicación por el servicio que nos han prestado.

También nos dan la posibilidad de enviar nuestra investigación para que ellos la publiquen y así compartir nuestros hallazgos.

Paso 4. Obtener la API

Pasado cierto tiempo, en mi caso al día siguiente, recibiremos el correo que nos informa si nos conceden la API académica. En total, el proceso duró menos de 24 horas.

En mi caso me la concedieron y tiene una serie de limitaciones que son las siguientes:

- Duración de la API: 6 meses
- Peticiones por minuto: 1000
- Peticiones por día: 20000
- Peticiones por mes: 600000

D. Anexo IV. Detectar honeypot Cowrie

Para detectar si un equipo puede ser un honeypot existen diversas técnicas, pero una que se ha encontrado y que funciona muy bien para detectar honeypots de Cowrie consiste en modificaciones sobre el banner SSH que se envía. Se ha probado en diversos equipos, tanto honeypots de Cowrie como equipos reales, y su efectividad ha sido del 100%. Consta de cuatro pasos que son los siguientes:

- Paso 0: Comprueba si la cabecera del servidor SSH es una de las que hay disponibles en el fichero de configuración de Cowrie.
- Paso 1: Envía un banner con un protocolo incorrecto, como puede ser *SSH-1337*. Si lo acepta puede que sea un honeypot.
- Paso 2: Envía un banner con un protocolo corrupto, como puede ser *SSH-2.0-OpenSSH\n\n\n*. Si lo acepta puede que sea un honeypot.
- Paso 3: Envía un banner con 2 protocolos separados por *\n*. Si lo acepta puede que sea un honeypot.

Una vez enviados esos banners, si han dado positivos dos o más pasos (del 1 al 3), se considera un honeypot; si solo ha dado positivo uno puede que lo sea, pero no es seguro; y si no ha positivo ningún paso no se considera un honeypot.

Podemos ver el script en la siguiente URL: `https://github.com/blazeinfosec/detect-kippo-cowrie`, además se muestra a continuación:

Código D.1: Código para detectar un honeypot Cowrie

```
#!/usr/bin/python3
# detectKippoCowrie.py
#
# https://github.com/blazeinfosec/detect-kippo-cowrie
# Proof of concept to detect Kippo and Cowrie SSH honeypots
#
# by Julio Cesar Fort
# Copyright 2016-2018 Blaze Information Security

import socket
import sys

CRED = '\033[91m'
CEND = '\033[0m'

DEFAULT_BANNER = "SSH-2.0-OpenSSH_"
DEFAULT_KIPPOCOWRIE_BANNERS = ["SSH-2.0-OpenSSH_5.1p1 Debian-5", "SSH-1.99-OpenSSH_4.3", "SSH-1.99-OpenSSH_4.7",
                               "SSH-1.99-Sun_SSH_1.1", "SSH-2.0-OpenSSH_4.2p1 Debian-7ubuntu3.1",
                               "SSH-2.0-OpenSSH_4.3", "SSH-2.0-OpenSSH_4.6", "SSH-2.0-OpenSSH_5
```

```
                                    ↪ .1p1 Debian-5",
20                                  "SSH-2.0-OpenSSH_5.1p1 FreeBSD-20080901", "SSH-2.0-OpenSSH_5.3p1 ↩
                                    ↪ Debian-3ubuntu5",
21                                  "SSH-2.0-OpenSSH_5.3p1 Debian-3ubuntu6", "SSH-2.0-OpenSSH_5.3p1 ↩
                                    ↪ Debian-3ubuntu7",
22                                  "SSH-2.0-OpenSSH_5.5p1 Debian-6", "SSH-2.0-OpenSSH_5.5p1 Debian↩
                                    ↪ -6+squeeze1",
23                                  "SSH-2.0-OpenSSH_5.5p1 Debian-6+squeeze2",
24                                  "SSH-2.0-OpenSSH_5.8p2_hpn13v11 FreeBSD-20110503",
25                                  "SSH-2.0-OpenSSH_5.9p1 Debian-5ubuntu1", "SSH-2.0-OpenSSH_6.0p1 ↩
                                    ↪ Debian-4+deb7u2",
26                                  "SSH-2.0-OpenSSH_5.9", "SSH-2.0-OpenSSH_6.0p1 Debian-4+deb7u2"]
27
28 DEFAULT_PORT = 22
29 VERBOSE = True
30 ERROR = -1
31
32
33 def get_ssh_banner(banner_from_server):
34     """
35     This function receives the banner of the SSH server. It returns true if
36     the server advertises itself as OpenSSH.
37     """
38     banner = banner_from_server.decode('utf-8').strip()
39
40     if banner in DEFAULT_KIPPOCOWRIE_BANNERS:
41         print("[!] Heads up: the banner of this server is on Kippo/Cowrie's default list. May be promising...")
42
43     return DEFAULT_BANNER in banner
44
45
46 def connect_to_ssh(host, port):
47     try:
48         socket.setdefaulttimeout(5)
49         sockfd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
50         sockfd.connect((host, port))
51
52         banner = sockfd.recv(1024)
53
54         if get_ssh_banner(banner):
55             if VERBOSE:
56                 print("[+] %s:%d advertised itself as OpenSSH. Continuing..." % (host, port))
57             else:
58                 print("[!] %s:%d does not advertise itself as OpenSSH. Quitting..." % (host, port))
59                 return False
60
61     except Exception as err:
62         print("[!] Error connecting to %s port %d: %s" % (host, port, str(err)))
63         return False
64
65     return sockfd
66
67
68 def probe_bad_version(sockfd):
69     try:
70         sockfd.sendall('SSH-1337\n'.encode('utf-8'))
71     except Exception as err:
72         print("[!] Error sending probe #1: %s" % str(err))
73
74     response = sockfd.recv(1024)
75     sockfd.close()
76
77     if VERBOSE:
78         print(response)
```

```
79
80      if b"bad version" in response:
81          if VERBOSE:
82              print("[*] Got 'bad version' in response to probe #1. Might be a honeypot!\n")
83          return True
84      else:
85          return False
86
87
88  # this probe works against Cowrie, but also some misconfigured versions of OpenSSH 5.3
89  def probe_spacer_packet_corrupt(sockfd):
90      try:
91          sockfd.sendall("SSH-2.0-OpenSSH\n\n\n\n\n\n\n\n\n\n".encode('utf-8'))
92      except Exception as err:
93          print("[!] Error sending probe #2: %s" % str(err))
94
95      response = sockfd.recv(1024)
96      sockfd.close()
97
98      if b"corrupt" in response or b"mismatch" in response:
99          if VERBOSE:
100             print("[*] Got 'packet corrupt' or 'protocol mismatch' in response of probe #2. Might be a ↩
                    ↪ honeypot!\n")
101         return True
102     else:
103         return False
104
105
106 def probe_double_banner(sockfd):
107     try:
108         sockfd.sendall("SSH-2.0-OpenSSH_6.0p1 Debian-4+deb7u2\nSSH-2.0-OpenSSH_6.0p1 Debian↩
                ↪ -4+deb7u2\n".encode('utf-8'))
109     except Exception as err:
110         print("[!] Error sending probe #3: %s" % str(err))
111
112     response = sockfd.recv(1024)
113     sockfd.close()
114
115     if b"corrupt" in response or b"mismatch" in response:
116         if VERBOSE:
117             print("[*] Got 'packet corrupt' or 'protocol mismatch' in response of probe #3. Might be a ↩
                    ↪ honeypot!\n")
118         return True
119     else:
120         return False
121
122
123 def detect_kippo_cowrie(host, port):
124     score = 0
125
126     print("[+] Detecting Kippo/Cowrie technique #1 - bad version")
127     sockfd = connect_to_ssh(host, port)
128
129     if sockfd:
130         if probe_bad_version(sockfd):
131             score += 1
132     else:
133         print("Socket error in probe #1")
134         sys.exit(ERROR)
135
136     print("[+] Detecting Kippo/Cowrie technique #2 - spacer")
137     sockfd = connect_to_ssh(host, port)
138
139     if sockfd:
```

```
140             if probe_spacer_packet_corrupt(sockfd):
141                 score += 1
142         else:
143             print("Socket error in probe #2")
144             sys.exit(ERROR)
145 
146         print("[+] Detecting Kippo/Cowrie technique #3 - double banner")
147         sockfd = connect_to_ssh(host, port)
148 
149         if sockfd:
150             if probe_double_banner(sockfd):
151                 score += 1
152         else:
153             print("Socket error in probe #3")
154             sys.exit(ERROR)
155 
156         return score
157 
158 
159 def main():
160     if len(sys.argv) >= 1:
161         host = sys.argv[1]
162         port = int(sys.argv[2])
163     else:
164         print('Argument incorrect')
165         return
166 
167     score = detect_kippo_cowrie(host, port)
168 
169     print("\t\t\t[+] Detection score for %s on port %d: %d" % (host, port, score))
170 
171     if score >= 2:
172         print("\t\t\t[*] IT'S A TRAP! %s on port %d is definitely a Kippo/Cowrie honeypot [*]" % (host, ↩
                ↪ port))
173     elif score == 1:
174         print("\t\t\t[+] %s:%d may be a Kippo/Cowrie honeypot or a misconfigured OpenSSH" % (host, port↩
                ↪ ))
175     elif score == 0:
176         print("\t\t\t[!] %s on port %d is not a honeypot." % (host, port))
177 
178 
179 if __name__ == '__main__':
180     main()
```