Universidad Internacional de La Rioja (UNIR)

ESIT

Máster universitario en Seguridad Informática

IDS colaborativo basado en Blockchain

Trabajo Fin de Máster

presentado por: Mínguez Julián, Javier

Director/a: García Arroyo, José Luís

Ciudad: Sevilla

Fecha: 19/09/2019

Resumen

La ciberseguridad en las organizaciones es una clara necesidad en los tiempos actuales debido al creciente número de ciberataques en los últimos años. Este trabajo desarrolla una herramienta que proporciona un sistema de detección de intrusos colaborativo, basado en comunicaciones a través de una red Blockchain, presentando la implementación de los requisitos definidos y las pruebas de funcionalidad realizadas. Esta herramienta aporta a la comunidad una base para el estudio y el desarrollo de sistemas colaborativos de detección de intrusos sobre redes Blockchain, ayudando a crear soluciones Open Source de estos sistemas, ayudando ha empezar a cubrir el vacío del estado del arte que existe relacionado con este tipo de herramientas. Se concluye con una herramienta Open Source para su uso por organizaciones en entornos de desarrollo, pretendiendo, en definitiva, que cualquier organización pueda implementar un sistema CIDS a medida sin grandes costes económicos. Las principales funcionalidades del mismo han sido testadas con éxito. El código de la aplicación está disponible en https://github.com/JaviMJ/Guardians---CIDS-based-on-Blockchain.

Palabras Clave: ciberseguridad, sistema de detección de intrusos, colaborativo, Open Source, Blockchain.

Abstract

Cybersecurity in organizations and companies is a current priority due to the increase in the number of cyberattacks in the last few years. This essay develops a solution that provides a collaborative intrusion detection system (CIDS), based in communications through a Blockchain network. The implementation of the defined requirements and how the functionality tests which have been carried out are presented. It is concluded with a proposal of an Open Source tool in order to be used by organizations in development environments. The final aim is that any organization can implement their customized CIDS without any further costs. This tool contributes to the community with a functional prototype for research and develops collaborative intrusion detection systems with communication based on Blockchain. Finally, this proposal helps to create new open source solutions of CIDS, covering, this way, the knowledge gap in this particularly field of study. The code of the application is available in https://github.com/JaviMJ/Guardians---CIDS-based-on-Blockchain.

Keywords: CIDS, cybersecurity, Blockchain, Open Source, prototype.

Índice

Índice de ilustraciones

1. Introducción

En la actualidad las empresas privadas y los organismos públicos poseen infraestructuras de red con múltiples servicios expuestos a internet, lo que puede ser un problema si no se gestiona correctamente la seguridad de los mismos.

Si se remite al informe publicado por el CCN-CERT, Centro Criptográfico Nacional, sobre ciberamenazas y tendencias de 2018, este indica que a nivel nacional se registraron 26.500 ciberincidentes en 2017, lo que supone un 26,65% más que en 2016 [1].

Estos ciberincidentes se dirigen principalmente contra infraestructuras y servicios web. También van dirigidos a usuarios, dispositivos móviles, entre otros, pero en menor medida.

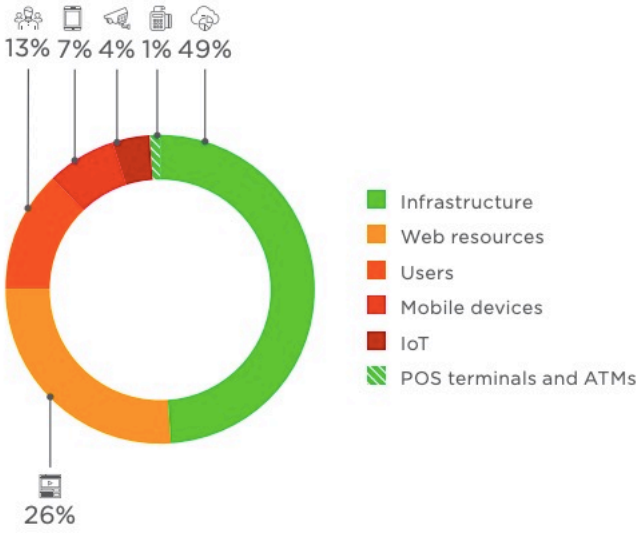

Ilustración 1 - Objetivos de ataque - https://www.ptsecurity.com/upload/corporate/ww-en/analytics/Cybersecurity-threatscape-2018-eng.pdf

La mayoría de estos ataques provienen normalmente del exterior, a través de los servicios publicados y la infraestructura conectados a internet. Estos ataques pueden causar graves daños a los organismos afectados tanto a nivel de reputación como a nivel económico, ya sea por multas o debido a la perdida de confianza de los clientes. Suelen estar motivados por distintas razones: acceso a información sensible, beneficios económicos, hacktivismo (los ataques llevan un fin político) o ciberguerra (llevada a cabo por los gobiernos).

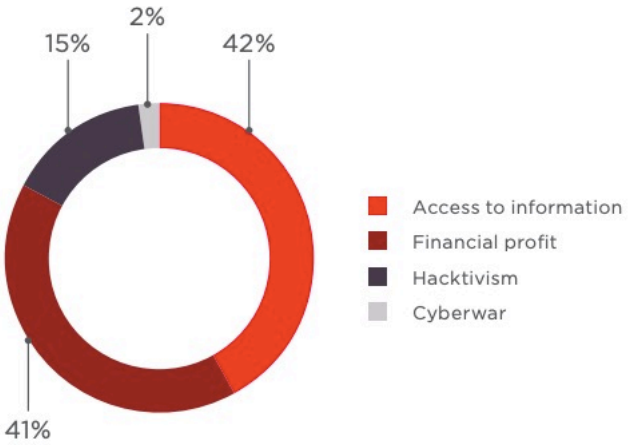

Ilustración 2 - Motivaciones de ataques - https://www.ptsecurity.com/upload/corporate/ww-en/analytics/Cybersecurity-threatscape-2018-eng.pdf

Para evitar ser afectados por este tipo de ataques, las organizaciones establecen sistemas de seguridad en sus infraestructuras. Estos sistemas poseen varios componentes como puede ser el firewall que analiza y gestiona los paquetes de red entrantes y salientes de las distintas redes de la organización, los proxies que gestionan los paquetes de red a nivel de aplicación, métodos de autentificación como pueden ser LDAP o Active Directory, detectores de intrusos (de estos en particular se hablará a continuación) entre otros.

Los IDS son sistemas de detección de intrusos que trabajan dentro de la red de la organización para detectar y, en algunos casos, bloquear ciberataques. Los hay de distintos tipos según su localización en la red. La implementación de una buena política de IDS es fundamental en una arquitectura de seguridad, ya que este recurso, si se actualiza constantemente, es capaz de mantener la infraestructura distante de diversos ataques [2].

Dichos IDS son muy útiles a la hora de detectar ataques, pero tienen ciertas limitaciones, como pueden ser la vista limitada del sistema por su localización en la misma o la dificultad de definir reglas para todos los patrones de ataques. Por ello surgen los CIDS (Collaborative Intrusion Detection System), que no son más que, distintos detectores de intrusos que comparten entre ellos información para evitar las limitaciones anteriormente mencionadas.

Los CIDS dan una buena solución a las limitaciones de los IDS, sin embargo, surgen nuevas limitaciones como pueden ser la seguridad en el envío de información o el consenso entre nodos. En esto es cuando la tecnología Blockchain nos puede ayudar a evitar estas y otras limitaciones de los CIDS, ya que por su definición es una red en la que la comunicación entre

nodos se hace de forma segura, se llega a un consenso de la información entre todos y se descentraliza la información evitando puntos críticos en nuestras infraestructuras.

Este trabajo trata de desarrollar una herramienta de código abierto CIDS basada en Blockchain, dando una base para desarrollar soluciones CIDS. La estructura de este trabajo será la siguientes:

1. **Introducción**
   En este punto se desarrollará la motivación del proyecto explicado a un nivel entendible por personas sin conocimientos del tema, planteamiento del problema y de cómo este trabajo piensa dar una posible solución a dicho problema y estructura de la memoria.

2. **Estado del Arte**
   En este apartado se hará referencia a la investigación de estudios académicos que ya existen sobre el tema, al estudio de los antecedentes y a las herramientas encontradas.

3. **Alcance y Objetivos**
   Se tratará de explicar qué va a abarcar el trabajo, cuáles son los objetivos y qué novedad van a aportar.

4. **Diseño del Sistema**
   Se describirán los requisitos a cumplir, se explicará el diseño del sistema, el diseño de alto nivel y el diseño de bajo nivel.

5. **Implementación**

   Se explicará la implementación del código, relacionada con los requisitos expuestos en el capítulo de diseño.

6. **Evaluación**
   Se reflexionará sobre la evaluación del trabajo realizado verificando el cumplimiento de los requisitos y de aportación novedosa a la comunidad académica.

7. **Conclusiones y trabajo futuro**

   Se proporcionarán las conclusiones finales del trabajo, indicando las aportaciones finales y las líneas de escalado del trabajo, incluyendo también futuras líneas de investigación y desarrollo.

2. Estado del arte

Antes de empezar con los antecedentes de este proyecto se definirá y mostrará un pequeño resumen sobre los IDS y de cómo la tecnología Blockchain puede ayudar a un funcionamiento más óptimo de estos detectores.

2.1 IDSs

Antes de que se exponga que es un sistema de detección de intrusos, se explicará qué es una intrusión en la red, ya que es un término que es necesario conocer. Una intrusión en la red es un ataque que proviene de fuera de la red local, normalmente de internet [3]. Por otro lado, un sistema de detección de intrusos o IDS, es un sistema que monitoriza el tráfico en busca de actividades sospechosas y alerta si dicha actividad es descubierta [4]. Básicamente los IDS realizan la misma función en una red que un sistema de alarmas de seguridad en un edificio o en una oficina. Al igual que un sistema de alarmas, el IDS monitoriza en busca de intrusiones e informa si las detecta, pero no hace nada para prevenir dicha intrusión. Existe un tipo de IDS que además de notificar mediante alerta, bloquea dicho tráfico. Estos son los sistemas de prevención de intrusos o IPS, que además de monitorizar la red al detectar un intruso bloquean su entrada. Esta información queda almacenada para poder ser investigada posteriormente [5]. Este tipo de IDS es muy agresivo y se debe tener cuidado a la hora de implantarlos en sistemas en productivo ya que, debido a falsos positivos, podrían bloquear tráfico permitido por error. La forma de integrar estos sistemas es configurar un IDS y optimizar sus reglas para evitar los falsos positivos, antes de implantar el IPS. Estos sistemas se configuran en redes con un nivel muy alto de seguridad, debido a la criticidad de estos.

Este trabajo se centra en los IDS, ya que si fuera necesario la implantación de IPS solo habrá que cambiar la configuración del IDS.

Los IDS usan los métodos de coincidencia de patrones detectando los ataques por sus firmas o las acciones que llevan a cabo. La efectividad de este método depende de la calidad de la base de datos de firmas, la cual siempre debe de estar al día. La detección de patrones funciona muy bien con ataques ya conocidos, pero no tienen nada que hacer contra nuevos ataques, ya que aun no ha sido introducida su firma en la base de datos haciendo imposible su detección por este método.

El otro método, de detección de intrusos, se basa en las anomalías estadísticas. Para usar este método es necesario establecer un baseline para determinar que comportamiento se

considera sospechoso, para así monitorizar las actividades que se salgan de los parámetros definidos [3]. Esto nos permite detectar ataques que aún no han sido registrados en la base de datos de firmas.

Dicho esto, existen varios tipos de IDS según su posicionamiento en la red y el método de identificación que usen:

- **NIDS (**Network IDS**):** son un tipo de detector de intrusos que se posicionan en puntos estratégicos de la red para inspeccionar todo el tráfico, tanto de entrada como de salida.

- **HIDS (**Host IDS**):** este tipo se instala en todos los sistemas de la red en cuestión. La ventaja de este IDS es que es capaz de detectar más anomalías de tráfico saliente de la red que la que se obtienen mediante sistemas NIDS. También es más efectivo a la hora de detectar paquetes provenientes de malware de algún host.

- **Signature-based IDS:** son un tipo de NIDS que compara este tráfico con una base de datos de firmas de amenazas conocidas, algo parecido a un antivirus.

- **Anomaly-based IDS:** es otro tipo de NIDS que también compara el tráfico, pero con un baseline del tráfico de la red en cuestión. Este tipo de IDS alerta de actividad potencialmente maliciosa.

Con esto concluimos que los IDS son sistemas que monitorizan el trafico, de una u otra forma, para detectar anomalías y tráfico malicioso para lanzar alertas con la información recopilada.

2.2 Blockchain

Blockchain es una tecnología que se define como un sistema distribuido de información que comparte y replica dicha información entre los participantes de la red peer-to-peer [6]. La estructura de datos de la red es construida desde una lista de bloques interconectados. Cada bloque contiene y se identifica por medio de su hash criptográfico y, además el bloque contiene el hash del bloque anterior a él. Gracias a esta propiedad se establece una relación criptográfica entre los distintos bloques. Cualquier participante de la red Blockchain puede consultar, no manipular, la lista de hashes de la correlación de bloques. Debido a esto, se

considera una red muy segura, ya que es difícil poder falsificar una transacción cuando cada participante en la red conoce la lista de transacciones aceptadas.

Actualmente se utilizan tres implementaciones de Blockchain según como esté gestionado el control de permisos:

- **Public:** son redes públicas en la que todos los participantes puedes leer y mantener el registro de bloques. Un ejemplo de estas redes pueden ser Bitcoin y Ethereum.

- **Consortium:** en este tipo de redes existe un consorcio de participantes responsables del mantenimiento de la cadena, como por ejemplo Hyperledger. Solo estos podrán escribir en registro de transacciones.

- **Private:** en las redes privadas existe una sola entidad que controla el sistema, no hay ningún consenso entre pares. Solo el nodo maestro podrá escribir en la cadena.

Wüst y Gervais (2017) proponen un esquema del proceso que ayuda a determinar si un aplicativo necesita usar Blockchain, y en ese caso que tipo de red Blockchain sería la adecuada.

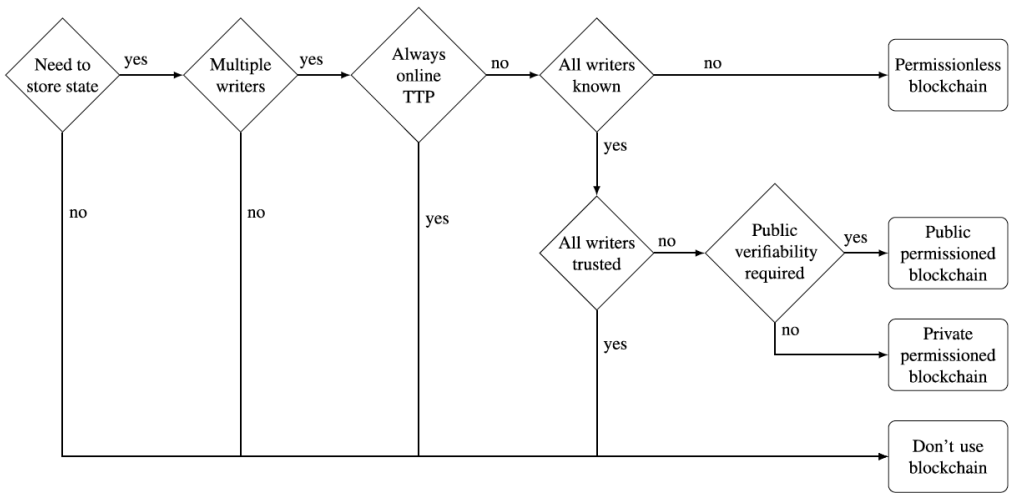

Ilustración 3 - Esquema de elección de tipo de blockchain - Wüst, K., & Gervais, A. (2017). Do you need a Blockchain? In IACR Cryptology ePrint Archive

El proceso de actualización del Blockchain se realiza mediante un protocolo, el cual logra consenso, dando garantías de que todos los participantes tengan visión y conformidad con el registro válido de transacciones, asegurando la integridad y la consistencia del mismo. Este

protocolo puedes variar dependiendo del tipo de implementación de Blockchain que se utilice y del modelo de amenaza.

Las redes públicas utilizan protocolos computacionalmente complejos (Proof-of-Work) o basados en la posesión de un recurso escaso del sistema (Proof-of-Stake). En cambio, las redes de consorcio y privadas utilizan algún tipo de algoritmo de tolerancia de fallo bizantino o benigno, como puede ser PBFT o SIEVE, para lidiar con nodos maliciosos [6]. El contenido exacto de los bloques que se usan en este protocolo puede variar entre las distintas implementaciones del mismo. Normalmente, además del payload (información relevante del aplicativo en si), suelen incluir una marca de tiempo y los valores de los hashes criptográficos de todos los bloques de la cadena.

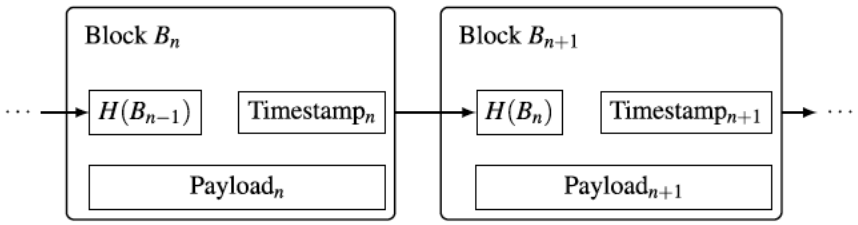

Ilustración 4 - Vista esquemática de una blockchain - Meng, W., Tischhauser, E. W., Wang, Q., Wang, Y., & Han, J. (2018). When intrusion detection

Para reducir el almacenamiento requerido de la Blockchain, las transacciones individuales pueden ser “hasheadas” basándose en el árbol de Merkle, así la raíz de este árbol es una representación compacta de todos los payloads involucrados.

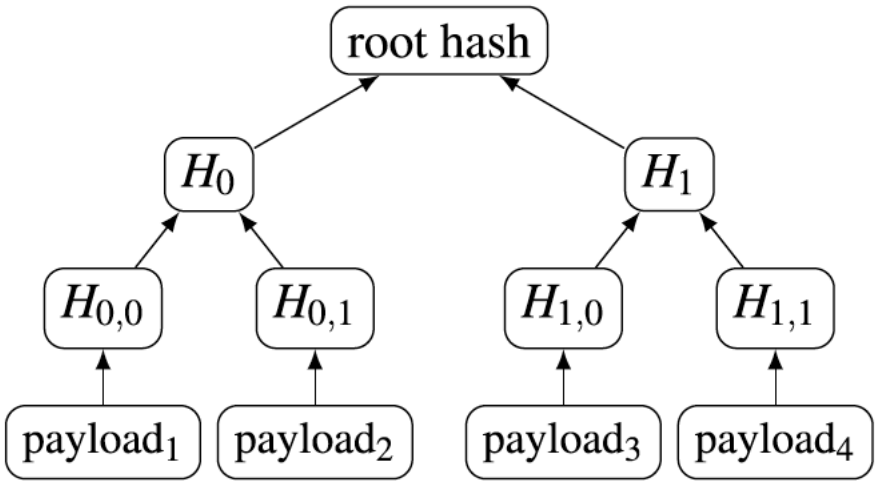

Ilustración 5 - Representación compacta de transsacciones blockchain - Meng, W., Tischhauser, E. W., Wang, Q., Wang, Y., & Han, J. (2018). When intrusion detection

2.3 CIDSs

Un sistema colaborativo de detección de intrusos o CIDS, se compone de un número N de nodos con sistemas IDS corriendo en ellos que se comunican entre si intercambiando información. Estos surgen con el fin de mejorar el rendimiento de un único IDS, que puede ser eludido por ataques avanzados o de denegación de servicios. La causa raíz de este problema es que normalmente los IDS no tienen una visión suficiente de los entornos que protege, mientras que los CIDS permiten a varios nodos IDS entender mejor el contexto de los entornos por el intercambio de información entre ellos [6].

Otras limitaciones que buscan solucionar los CIDS son las siguientes:

- La eficiencia y cobertura de estos sistemas que depende del ingenio para describir patrones de ataque o firmas maliciosas en esa localización específica.

- Utilización de “Loose rules”: mejor cobertura, sin embargo, pero mayor número de falsos positivos.

- Utilización de “Strict rules”: más precisión, pero mayor número de falsos negativos.

Por ello los CIDSs intentan remediar esto mediante múltiples detectores especializados en diferentes partes del sistema, que combinan, comparten y gestionan las distintas reglas y alarmas. Gracias a la combinación de esta información se mejora la eficiencia de la detección.

Una aproximación de un diseño de IDS es la siguiente.

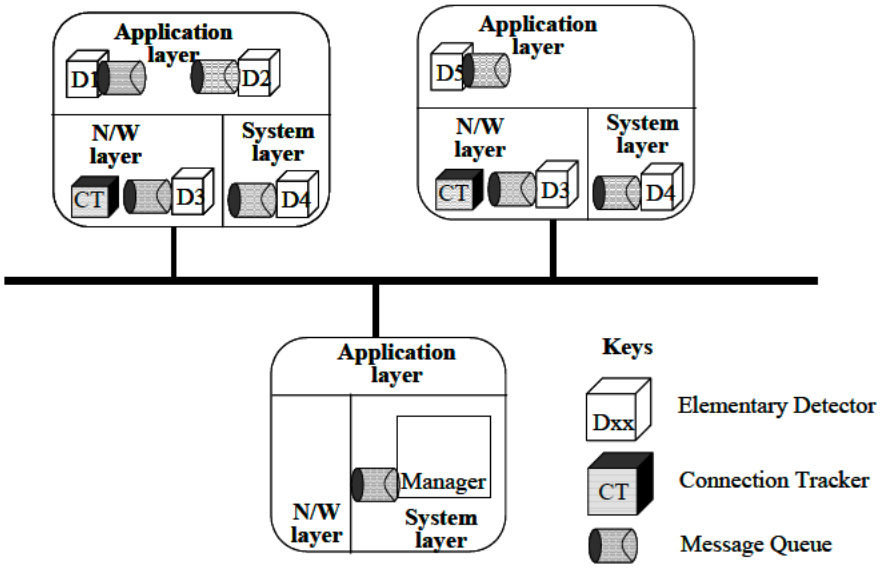

Ilustración 6 - Esquema CIDS - Wu, Y., Foo, B., & Mei, Y. (2003). Collaborative Intrusion Detection System : A Framework for Accurate and Efficient IDS Challends of current IDS

Normalmente estos sistemas se clasifican según tres categorías:

- Sistemas colaborativos jerarquizados, como EMERALD y DIDS.
- Sistemas colaborativos de subscripción, como COSSACK y DOMINO.
- Sistemas colaborativos peer-to-peer, como Netbait y PIER.

El problema que surge con este tipo de sistemas es la gestión de la confianza entre los mismos. Específicamente, la gestión de confianza en los CIDS se puede distinguir en función del objetivo general del mecanismo de confianza. En este contexto, la confianza computable será comúnmente aplicada para calificar el nivel de confianza entre los nodos de monitorización [7]. Con esto se quiere decir que, si un nodo es comprometido o empieza a enviar información falsa, su nivel de confianza descenderá pudiendo llevarse a cabo distintas acciones, como por ejemplo introducir ese nodo en una blacklist. Otro enfoque sería que el modelo de confianza se basara en la medición de la calidad de las aletas lanzadas en cada nodo o asignándoles puntuaciones especificas a cada alerta según parámetros de la misma.

La mayoría de los trabajos propuestos en esta área usa mecanismos de confianza computacionales basados en modelos matemáticos para medir el nivel de confianza de los

nodos de monitorización [6]. Básicamente cada nodo utiliza su propia experiencia respecto a otros nodos, y usando mecanismos de confianza computacionales puede afectar a la confianza en los otros nodos. Aún así, no se ha llegado a un método que aporte un grado alto de responsabilidad y consenso entre nodos de un CIDS.

2.4 Requisitos de un sistema CIDS

Se tiene que tener en cuenta que el principal problema de los sistemas CIDS es asegurar un mecanismo de confianza fiable. Para ello, se especifican una serie de requisitos que debe tener un sistema CIDS, los cuales son:

- **Responsabilidad:** los participantes deben de ser responsables de sus acciones.
- **Integridad:** la integridad de las alertas debe de ser muy importante para poder detectar ataques tanto en tiempo de ejecución como en un análisis posterior.
- **Resistencia:** el sistema no debe poseer SPoFs y la misma no debe depender de un numero pequeño de participantes.
- **Consenso:** el sistema debe de ser capaz de alcanzar un consenso en la calidad de la información de las alertas individuales y en la confiabilidad de los participantes.
- **Escalabilidad:** el sistema debe de ser escalable a un gran número de participantes/monitores y también manejar correctamente las salidas masivas de participantes en la red.
- **Mínimo coste:** los costes de las comunicaciones y de la computación deben ser lo más bajo posible.
- **Privacidad:** los participantes deben de poder reservarse sus derechos de privacidad y revelar solo las alertas que desee. Sin embargo, al mismo tiempo, deben de cumplirse los requerimientos de responsabilidad e integridad.

Para satisfacer estos requerimientos en dicho trabajo se considera que una buena solución para cubrir los mismos es usar una red Blockchain. En el siguiente apartado se tratan las razones por las cuales Blockchain es una tecnología beneficiosa para los sistemas CIDS.

2.5 CIDS basado en Blockchain

En este apartado se explica cómo el implementar una red Blockchain entre los participantes del CIDS asegurará el intercambio de alertas entre los nodos.

Un ejemplo de cómo sería el funcionamiento es el siguiente. La información generada en forma de alertas por los participantes de la red Blockchain será almacenada en la cadena Blockchain, replicándose a lo largo de la red por los distintos nodos. Los nodos se comunicarán por un **protocolo consensado** para garantizar la validez de las transacciones antes de añadirlas a la Blockchain. Así se garantiza que solo las alertas válidas, bajo un determinado formato y especificaciones, son añadidas a la Blockchain. Cada participante puede consultar la Blockchain de transacciones válidas.

De esta manera los participantes son **responsables** de sus acciones, dado que serán visibles por todos los participantes. Además, la **integridad** de la información también se garantiza mediante la Blockchain, no poseyendo la misma SPoF dado que la caída de un nodo en concreto no causaría un fallo en el sistema asegurando la **resistencia** del mismo. La solución al **mínimo coste** en las comunicaciones se solventa almacenando el hash de las transacciones en la Blockchain en vez de información en crudo. La **escalabilidad** del sistema es viable gracias al diseño de las redes Blockchain. Cada nodo puede gestionar las alertas que se envían a la Blockchain, pudiendo elegir cuáles compartir o no, asegurando la **privacidad** de los participantes.

Alexopoulos, Vasilomanolakis, Ivánkó y Mühlhäuser (2018) proponen un diseño de arquitectura de un sistema CIDR basado en Blockchain. El esquema es el siguiente:

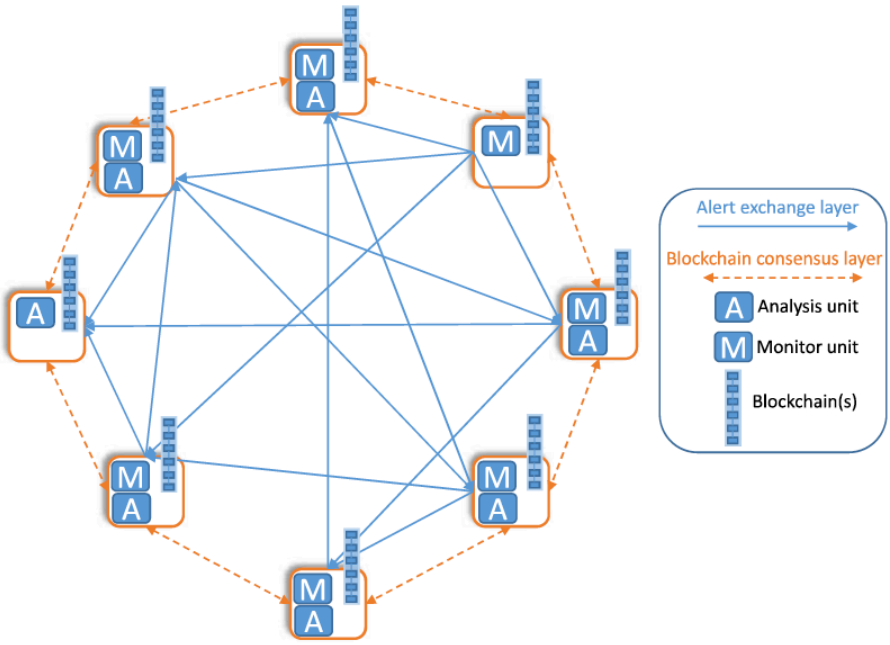

Ilustración 7 - Esquema CIDS - Alexopoulos, N., Vasilomanolakis, E., Ivánkó, N. R., & Mühlhäuser, M. (2018). Towards Blockchain-Based Collaborative Intrusion Detection Systems

En el diseño los nodos participantes de la red son, o unidades de monitor, o de análisis o ambas. La comunicación entre nodos se realiza mediante dos capas, "Alert Exchange layer" y "Blockchain consensus layer" [6]. En la "Alert Exchange layer", o capa de intercambio de alertas, es donde el sistema CIDR realiza el proceso de difusión de alertas. Cada nodo intercambia o recolecta información según su rol de unidad de monitor o de análisis. El protocolo usado para esto viene determinado por las necesidades del sistema CIDS, pudiéndose usar protocolos flooding o gossiping para difundir las alertas o también intercambiar las alertas por requerimiento de los nodos. En la capa de consenso, o "Consensus layer" un subconjunto de nodos, los nodos de análisis, mediante un protocolo de consenso deciden que transacciones deberían ser añadidas a la Blockchain. La combinación de la conexión entre las dos capas, el resultado de la capa de consenso y las propiedades intrínsecas de la red Blockchain aseguran responsabilidad de los participantes y garantía de la integridad de la información. En el caso en el que se quiera compartir alertas solo entre ciertos nodos, como podría ser una organización que no quiera relevar cierta información de sus alertas, estos nodos podrán crear una red colaborativa entre ellos separada de la otra red o cifrando la información con una clave que solo posean los nodos que deben de leer dicha información.

2.6 Consideraciones de diseño

2.6.1 Uso de Blockchain

El uso de la Blockchain como método de comunicación nos ofrece un grado de responsabilidad, integridad y resistencia para nuestro sistema. Aún así, hay que tener en cuenta el tipo de Blockchain escogida y el algoritmo de consenso elegido, ya que afectará al grado de cumplimiento de los requisitos anteriormente expuestos.

Otras características, como la escalabilidad de la red o la privacidad de los participantes podrán depender del tipo de red Blockchain usada (pública, consorcio o privada) y la forma y el formato en que se realiza el intercambio de información. También será de gran importancia las reglas o alertas que se intercambien en la red.

2.6.2 Tipo de red Blockchain

Las redes públicas de Blockchain proveen de autenticidad, integridad y resistencia al sistema garantizando una vista global de las transacciones de alertas y reglas. Sin embargo, tiene sus ventajas y desventajas. Las redes públicas proveen una red sin control en la que cualquiera puede formar parte de ella libremente, y donde todo el mundo puede leer y actualizar la cadena. Los CIDS pueden beneficiarse de esto, ya que la información almacenada tiene su integridad protegida y disponible para todos los participantes, mientras los participantes sean responsables de sus acciones. Aunque puede tener un efecto negativo, ya que al ser accesibles las alertas y reglas intercambiadas por cualquiera, los atacantes podrían utilizar dicha información para realizar ataques no detectables.

En las redes de consorcio, los permisos de acceso a la red están más controlados, así como los permisos de modificación y lectura que pueden estar restringidos a ciertos usuarios. En este tipo de redes hay un número de nodos preseleccionados que son los encargados de actualizar la cadena, pudiendo los mismos detectar comportamientos anómalos e intentos de falsificación de la cadena, revertiendo la cadena a su estado anterior y echando a dichos nodos de la red. Las redes privadas, poseen los permisos de acceso limitados al igual que en las redes de consorcio, internamente pueden comportarse como una red Blockchain pública o de consorcio, según las necesidades de dicha red. Estos dos tipos de redes son ideales

para organizaciones o instituciones que no quieran que su información sea pública y tener el control de los participantes del sistema.

2.6.3 Consenso entre nodos

La selección del algoritmo de consenso entre los participantes de la red es de gran importancia. Especialmente en diseños de Blockchain corporativos, hay posibilidad de escoger un conjunto de nodos que será los responsables de ejecutar el algoritmo de consenso, garantizando la integridad de la cadena al sistema. Aparte de esto, el algoritmo de consenso seleccionado afectará en gran medida a las garantías de seguridad del sistema. Tanto el algoritmo Proof-of-Work como Proof-of-Stake son muy usados por su eficacia [6].

2.6.4 Información guardada en la cadena

La información guardada en la cadena es otra cuestión a tener en cuenta, ya que lo ideal es guardar toda la información en crudo de las transacciones para poder consultar la información que contenía cada una de ellas. Esto puede causar un problema de almacenamiento, dependiendo de las cantidades de datos manejadas y de la capacidad económica de la organización para mantener esos datos almacenados. Una solución que se usa para redes que no son capaces de almacenar toda la información, es almacenar en la cadena solo los hashes de cada transacción para poder comprobar dicha información con posterioridad, pudiendo repartir la información de las transacciones por partes sin necesidad de que todos los nodos la almacenen.

2.6.5 Privacidad de la información

Las redes Blockchain, sobre todo, las públicas, necesitan proveer de un mecanismo para intercambiar información sensible entre nodos que no se quiera compartir con toda la red. Una solución puede ser encriptar la información con claves simétricas criptográficas, compartiendo las claves solo con los participantes que se quiere compartir dicha información [6]. Esto permitiría compartir información sensible entre solo ciertos nodos de la red.

2.7 Herramientas existentes

Se considera que existe un vacío en el estado del arte, en cuanto a herramientas CIDS Open Source en lo que respecta a su estudio. Sí se puede encontrar un artículo donde se explican algunas características de un sistema CIDS implementado para la seguridad de una red Blockchain. Esta herramienta no posee el mismo enfoque que lo que se quiere desarrollar en esta investigación, pero su estudio ayudará al desarrollo de lo que se presenta aquí.

2.7.1 BAD: the first Blockchain Anomaly Detection solution

Esta herramienta desarrollada por Signorini, Pontecorvi, Kanoun y Di Pietro (2018) se presenta como el primer detector de intrusos en Blockchain. Esta herramienta es privada pero los creadores de dicha herramienta han elaborado en su artículo "*BAD: a Blockchain Anomaly Detection solution*" explican cómo funciona y en que se basa *BAD*.

BAD aprovecha los metadatos de Blockchain y de los forks, con el fin de recolectar posibles actividades maliciosas en la red y/o los sistemas. Cuenta con las siguientes características:

- Es un sistema distribuido: evitando los SPoFs.
- A prueba de manipulaciones: evitando que el malware elimine o altere información.
- Es confiable: toda la información es verificada por la mayoría de la red.
- Es privada: evitando que cualquier tercero recolecte/analice/almacene información sensible.

BAD ha sido diseñada para ser una solución ad hoc, la razón es que BAD no confía en un tipo específico de Blockchain y puede ser configurado para detectar ataques en cualquier aplicación blockchain [8]. A continuación, se describen los distintos módulos de este CIDS, para comprender las funcionalidades del mismo:

- **Filtro de transacción** (Tx Filter): este módulo intercepta los mensajes estándar de la Blockchain y los reenvía tanto a los *miners* como *al chain manager*, no

interrumpiendo el funcionamiento estándar de Blockchain. Además, permite la recolección de metadatos de las transacciones.

- **Administrador de la cadena** (Chain manager): es el responsable de mantener la cadena Blockchain, conteniendo la información de los forks generados por los participantes en el sistema. Recibe los mensajes del filtro de transacción y recupera posible información perdida de la base de datos de la cadena, la cual almacena toda la información de la cadena. Por último, notifica al inspector de patrones si se ha actualizado la cadena y si se debe realizar un análisis de amenazas.

- **Inspector de patrones** (pattern inspector): usa la base de datos de la cadena para detectar actividad sospechosa.

- **Detector de amenazas** (Threat detector): recibe las anomalías detectadas por el inspector de patrones y analiza cual ha sido la causa raíz de esa actividad por medio de la explotación de las transacciones relacionadas con ella. Después la información del ataque se almacena en una base de datos de amenazas, la cual contiene información de todos los patrones que han sido considerados maliciosos para la Blockchain.

2.8 Tecnologías

Tras la investigación de las tecnologías disponibles para desarrollar una red Blockchain, se concluye que se usan distintos lenguajes de programación como puede ser java, python, js, php, entre otros. Lo mismo ocurre con las implementaciones de Blockchain, hay variedad para elegir ofreciendo características muy similares.

Teniendo en cuenta esto, dado que no hay un lenguaje claramente que sea mejor para trabajar con Blockchain, se usará Python 3.7 ya que es un lenguaje muy extendido, con una gran comunidad y muchas librerías nativas disponibles que ayudarán a la implementación de la solución CIDS.

2.8.1 Python 3.7

Guido van Rossum, un programador Holanes, creo Python en 1991. Este lenguaje esta basado en una simple filosofía: Simplicidad y Minimalismo [9]. Python es un lenguaje de programación no tipado, seguro y con una curva de aprendizaje baja. Posee estructuras de alto nivel eficientes y un enfoque simple, pero efectivo, en la programación orientada a objetos. La sintaxis de Python y el tipado dinámico, junto con la naturaleza de su interprete, lo hacen un lenguaje ideal para el scripting y el desarrollo rápido de aplicaciones en diversas áreas y la mayoría de las plataformas [10]. Además, hay un extenso conjunto de librerías de Python para distintas áreas, lo que lo hace un lenguaje de programación fácil y rápido para adentrarse en nuevas áreas en las que se posea poca experiencia.

Ilustración 8 - Python - https://www.python.org/

La red Blockchain será desarrollada desde cero mediante Python, así se tendrá total control y conocimiento sobre el código que se ejecuta y será más fácil modificar características de la Blockchain, como puede ser el algoritmo de consenso, la estructura de los bloques, entre otros. Existe una librería de blockchain para Python, pero está enfocada al minado de Bitcoin y otras criptomonedas. Por esa razón y lo expuesto en el párrafo anterior se escoge no usarla.

2.8.2 Flask

Flask es un mini-framework para el desarrollo de aplicaciones web WSGI. Esta librería está diseñada para empezar a desarrollar aplicaciones rápida y fácilmente [11]. Con la capacidad de escalar de pequeñas aplicaciones simples a proyectos más grandes y complejos. Comenzó como una simple capa que englobaba las librerías de Werkzeug y Jinja, y se ha convertido en una de los framework de aplicaciones web más famosos para Python.

Ilustración 9 - Flask - https://palletsprojects.com/p/flask/

Dado que se usarán en este estudio las conexiones http, se va a usar la librería Flask de Python que permitirá gestionar peticiones http con mucha facilidad. El protocolo de comunicación entre los nodos será TLS ya que es un protocolo seguro que nos permite una fácil comunicación entre nodos.

2.8.3 Snort IDS

El software a usar en cuanto a la herramienta IDS que se ha escogido ha sido Snort, debido a que es un proyecto de código libre con una comunidad muy extendida y con una amplia documentación. Snort es un sistema de detección de intrusos Open Source, con la capacidad de realizar análisis de tráfico de red en tiempo real y registro de paquetes en redes IP [12]. Ofrece análisis de protocolo, búsqueda o concordancia de contenido, y detecta gran variedad de ataques y sondeos, como buffer overflow, análisis de puertos, ataques CGI, sondeos de SMB, intentos de OS fingerprinting, entre otros. Una de las mejores características de Snort es la capacidad para definir reglas. En ciertos casos, es necesario configurar determinadas alertas para determinados tipos de paquetes. Es común, que frente a una vulnerabilidad explotable remotamente en algún equipo crítico, será necesario solventar la situación mediante la configuración de alguna regla hasta que la vulnerabilidad sea solucionada [13].

Ilustración 10 - Snort - https://www.snort.org/

Snort puede configurarse para que funcione en tres modos principales:

- **Sniffer:** en este modo el software leerá los paquetes de la red y los mostrará por consola.

- **Registro de paquetes:** el software además de leer los paquetes de red los almacenada en un registro o base de datos.

- **IDS:** si se configura en este modo, monitorizará y analizará el tráfico de red en base a las reglas definidas por el usuario.

En este proyecto se configurará Snort en modo IDS, ya que los distintos Snorts compartirán distintas reglas para mejorar su eficacia.

2.8.4 Gitlab

Gitlab es un software de desarrollo totalmente integrado que permite a los equipos de desarrollo ser transparentes, rápidos, efectivos y cohesivos desde la discusión de una nueva idea hasta la producción, todo en la misma plataforma [14].

Ilustración 11 - GitLab - https://gitlab.com/

Se ofrece como servicio web que permite el control de versiones y de desarrollo de software usando el protocolo git para ello.

En este trabajo se usará para actualizar el código en los distintos nodos y tener un histórico de la modificación del código.

2.8.5 Pycharm

Pycharm es un IDE usado para programar con Python. Proviene de análisis de código, debugger, integración con git, consola Python y consola bash, entre otras características [15].

Ilustración 12 - PyCharm - https://www.jetbrains.com/pycharm/

En este proyecto se usará para desarrollar el software en cuestión, ya que facilita en gran manera el desarrollo del código, el control de versiones con git y la realización de pruebas mediante el debbuger y las consolas.

2.8.6 Trello

Trello es un software de gestión de proyectos que proviene de una interfaz web, con tableros Kanban para la gestión y asignación de tareas. Ofrece integración con otras herramientas como Google Drive, Slack, Zapier, entre otras [16].

Ilustración 13 - Trello - https://trello.com/

En este proyecto se usará para llevar un control de las tareas que aún están por realizar y las ya realizadas.

2.8.7 Postman

Postman es un software que nos permite construir distintos tipos de peticiones HTTP/HTTPS. Empezó en el 2012 como un proyecto para simplificar el desarrollo de APIs y el realizado de pruebas con las mismas [17]. Esta aplicación permite la creación de peticiones a APIs internas o de terceros, elaboración de tests para validar el comportamiento de APIs, posibilidad de crear entornos de trabajo diferentes (con variables globales y locales), y la posibilidad de ser compartido mediante una exportación en formato JSON [18].

Ilustración 14 - Postman - https://www.getpostman.com/

En este proyecto se usará para realizar distintas peticiones a los nodos de la Blockchain, para comprobar el correcto funcionamiento de la herramienta.

2.9 Conclusiones del estudio del estado del arte

Tras lo expuesto en este capítulo se concluye que, hay un vacío en el estado del arte al no existir ninguna herramienta CIDS Open Source para poder testear y estudiar. Por tanto, se ha decidido desarrollar la red Blockchain con Python+Flask, usar Snort como solución IDS y crear un conector entre la Blockchain y los IDS para, de esta manera, crear un sistema CIDS. Dicho sistema CIDS debería cumplir mayoritariamente los requisitos para sistemas CIDS mencionados en este capítulo. Se desarrollará un mínimo producto viable con el fin de crear una base para este tipo de herramientas, distribuyendo dicho software como Open Source para que la comunidad pueda tanto utilizarlo como nutrirlo.

3. Alcance y Objetivos

3.1 Objetivo general

El objetivo general de este estudio es mejorar la seguridad de las organizaciones mediante un sistema de detección de intrusos colaborativo y seguro. Además de proveer una herramienta Open Source para que organizaciones pequeñas tengan accesibilidad a este tipo de herramientas y lo puedan implementar y adaptarlo a sus necesidades.

3.2 Alcance

El alcance del proyecto será definir una o varias arquitecturas de cómo podría estar diseñado un sistema real de CIDS basado en Blockchain y de código libre. También se desarrollará un mínimo producto viable construido a partir del diseño propuesto, con el fin de que en futuros desarrollos se cree una herramienta completa. Dicho sistema será un prototipo de CIDS para realizar pruebas en desarrollo, no debiendo llevar dicho sistema a producción.

3.3 Objetivos específicos

Para cumplimentar el alcance se definen dos objetivos principales:

- Describir al menos una arquitectura del sistema CIDS.
- Desarrollar un mínimo producto viable del sistema CIDS.

Y otros cuatro secundarios:

- Verificar la herramienta mediante pruebas y documentar el resultado de las mismas.
- Analizar y documentar el funcionamiento de la herramienta.
- Sintetizar un pequeño manual de usuario.
- Describir las ventajas de esta herramienta sobre otros sistemas CIDS.

3.4 Metodología de trabajo

La metodología que se va a usar se basará en una metodología de trabajo ágil, dividiendo las tareas generales en tareas más específicas para así poder alcanzar más fácilmente un producto mínimo viable. Se comenzará diseñando el sistema y describiendo los componentes del mismo. Una vez hecho esto se definirán los requisitos de la herramienta para satisfacerlos, con la implementación de los mismos. Durante el desarrollo del código se documentarán los avances del mismo. Cuando el desarrollo del código y la implementación de la herramienta estén listos, se realizarán pruebas para verificar que el sistema CIDS funciona correctamente. En caso contrario se documentarán los errores de la herramienta y se efectuará las propuestas de cómo se podrían solucionar dichos errores. Las pruebas se realizarán con maquinas virtuales o sistemas físicos con sistemas operativos Linux, debiendo contener al menos tres nodos IDS para comprobar la efectividad de la herramienta. Una vez realizadas y documentadas las pruebas, se sintetizará un pequeño manual de uso y se completará la memoria añadiendo las conclusiones y los futuros desarrollos.

4. Diseño del sistema

En este capítulo se empezará describiendo la arquitectura de alto nivel, para después seguir con la arquitectura de bajo nivel y terminar con la implementación de la herramienta.

4.1 Identificación de requisitos

A continuación, se describirán los requisitos de la herramienta y el sistema, basándose en lo expuesto en los capítulos anteriores de este trabajo. Los requisitos poseerán el identificador "**R-XX**" para posteriormente referenciarlos fácilmente en capítulos posteriores de esta memoria. Los requisitos son los siguientes:

- **R-01:** La herramienta debe ser desarrollada en Python junto con la librería Flask.
- **R-02:** La herramienta de código libre que se usará será Snort.
- **R-03:** La herramienta permitirá interconectar nodos Snort, con el fin de nutrirse los unos a los otros de reglas IDS.
- **R-04:** Las comunicaciones se realizarán mediante una red Blockchain.
- ***R-05*:** En la red Blockchain las transacciones deben ser asociadas a cada nodo para asegurar la responsabilidad de los mismos.
- **R-06:** La red Blockchain no poseerá un nodo máster para evitar puntos únicos de fallo del sistema (SPoFs), asegurando la resistencia del sistema.
- **R-07:** Los nodos de la red Blockchain deberán usar un algoritmo de consenso para asegurar la integridad y la calidad de la información.
- **R-08:** El sistema debe ser escalable a múltiples nodos y localizaciones.
- **R-09:** El sistema deberá usar la filosofía del mínimo coste posible, tanto para las comunicaciones como para la computación.

- **R-10:** Los nodos deberán poder elegir qué información/reglas compartir con los demás nodos en la red Blockchain.

- **R-11:** El protocolo para las comunicaciones de la red Blockchain será HTTPS/TLS.

- **R-12:** Todo el conjunto de transacciones de la red Blockchain (la chain de la blockchain) se debe almacenar de forma persistente en los nodos.

- **R-13:** La red blockchain será de acceso privado.

- **R-14:** Se permitirá la inclusión de nodos en la Blockchain mediante una llamada POST, que deberá incluir una contraseña solo conocida por la organización.

- **R-15:** La lista de nodos de la blockchain será almacenada de forma persistente en cada uno de los participantes de la red.

- **R-16:** El conector de Snort con la Blockchain deberá poder automatizar el envío de datos por la red Blockchain y la actualización de la cadena Blockchain. Ya sea internamente o mediante tareas cron/anacron.

Con este software se pretende abordar el problema del intercambio de información en los CIDS asegurando no repudio, confidencialidad, integridad y disponibilidad del mismo. Sin llegar a desarrollar un sistema apto para un sistema productivo, sino un producto mínimo viable para realizar pruebas y con el fin de que cualquier organización pueda tomarlo de base para desarrollar su propio sistema CIDS.

4.2 Diseño de alto nivel

Para explicar la arquitectura de alto nivel se mostrarán dos posibles arquitecturas de red, la primera a nivel local pensada en organismos pequeños que disponen de una sola localización para realizar sus operaciones, y la segunda una arquitectura de red con conexiones entre varias localizaciones pensada para organismos medianos o grandes.

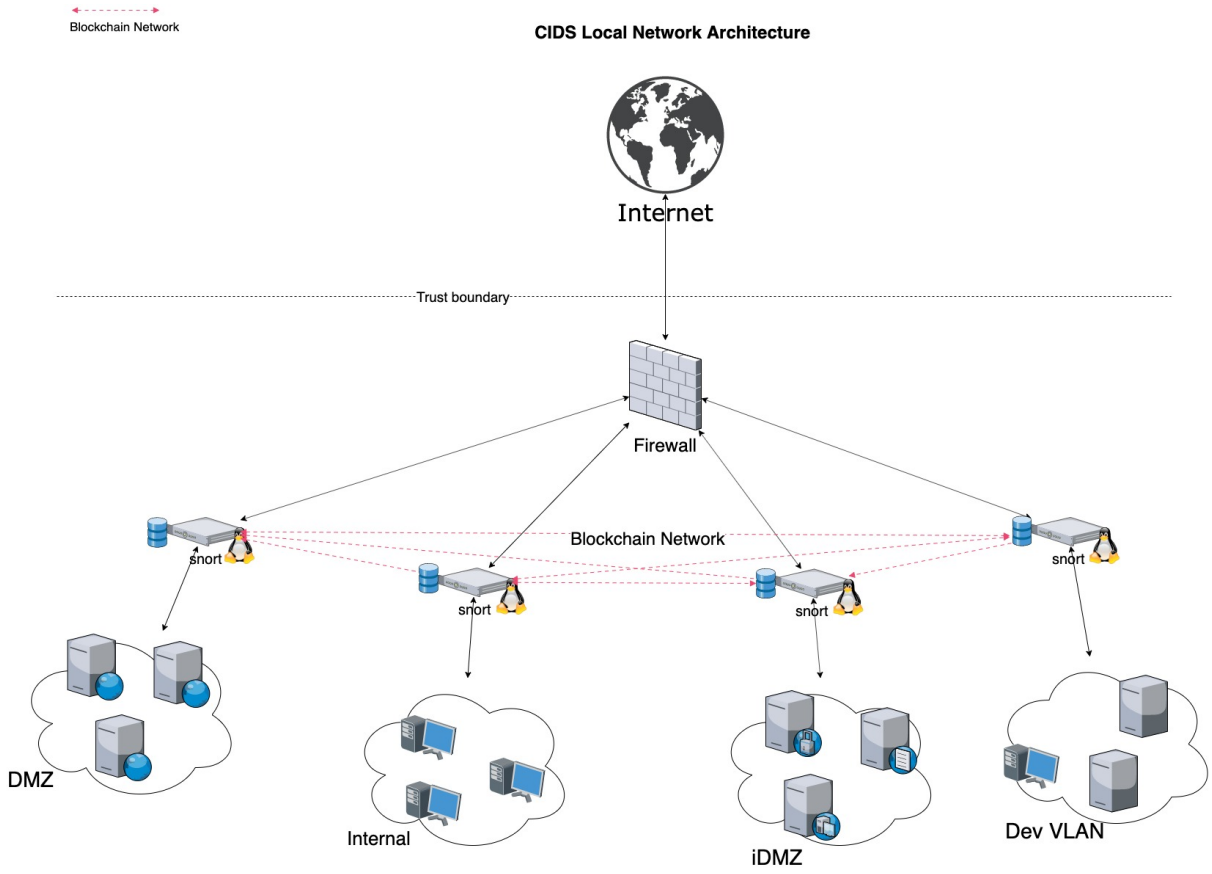

Ilustración 15 - Arquitectura de red CIDS local

En este ejemplo se establece una red interna de un organismo dividido en cuatro vlans diferenciadas. Cada una de estas subredes tiene un propósito distinto y las reglas del firewall no deben de ser las mismas para cada una de ellas. En la red DMZ estarán permitidas las conexiones desde el exterior al interior de la red dado que allí se alojarán los servidores en producción, la iDMZ tendrá reglas más restrictivas con respecto a las conexiones del exterior ya que alojará servidores de preproducción y en las redes interna y dev las conexiones desde el exterior no estarán permitidas.

Dado que cada una de estas subredes tienen fines distintos, también tendrán reglas distintas en los IDS enfocadas a las mismas. En este ejemplo se aprecia como detrás del firewall, que actúa a la vez de routing, se ha colocado cada uno de los IDS (Snort), uno por subred. Estos IDS estarán conectados por una red Blockchain privada en la que solo ellos serán participantes de la misma, entre ellos se compartirán las distintas reglas que se vayan implementando en cada uno de ellos, para así poder nutrirse unos de otros y crear un sistema de reglas más completo. Cada uno de estos nodos decidirá qué reglas implementar según unos criterios definidos por la naturaleza de su red. Está información compartida será inmutable, trazable, confidencial y estará siempre disponible gracias a la naturaleza de la red

Blockchain. La cadena de la Blockchain se almacenará de forma segura en cada uno de los sistemas, para que en caso de reinicio de la máquina siga teniendo la información de la cadena disponible, pudiendo actualizarla al unirse de nuevo a la red Blockchain.

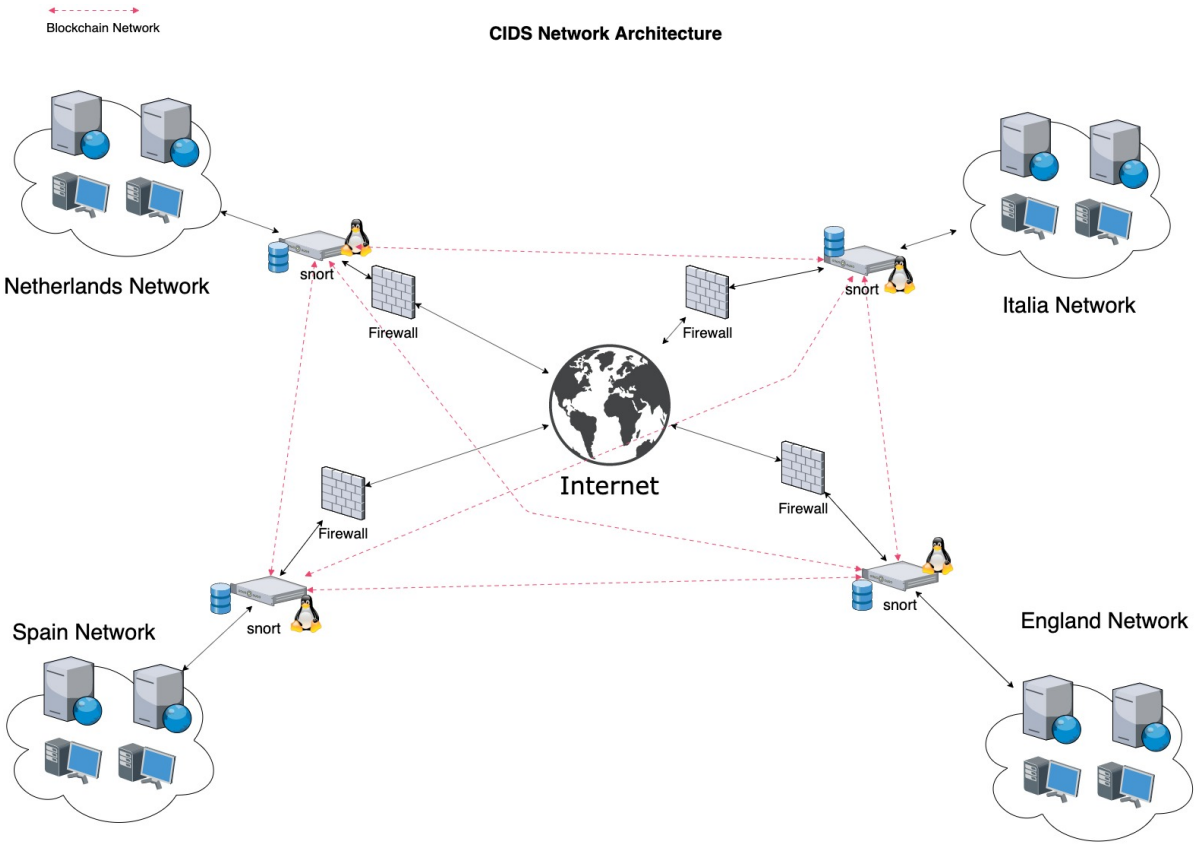

Ilustración 16 - Arquitectura de red CIDS

Esta segunda arquitectura representa una organización internacional con sedes en distintos países. Gracias a la herramienta CIDS las reglas que se creen a raíz de un ataque en una de las sedes, se compartirán con los demás IDS, evitando así que el ataque se repita en otra sede. Se ha simplificado el diagrama colocando un solo sistema IDS delante de cada red, realmente cada IDS junto con su red se representarían con el diagrama local anterior a este.

4.3 Diseño a bajo nivel

4.3.1 Diagrama de componentes

Para explicar el diseño a bajo nivel se va a exponer el siguiente diagrama de componentes.

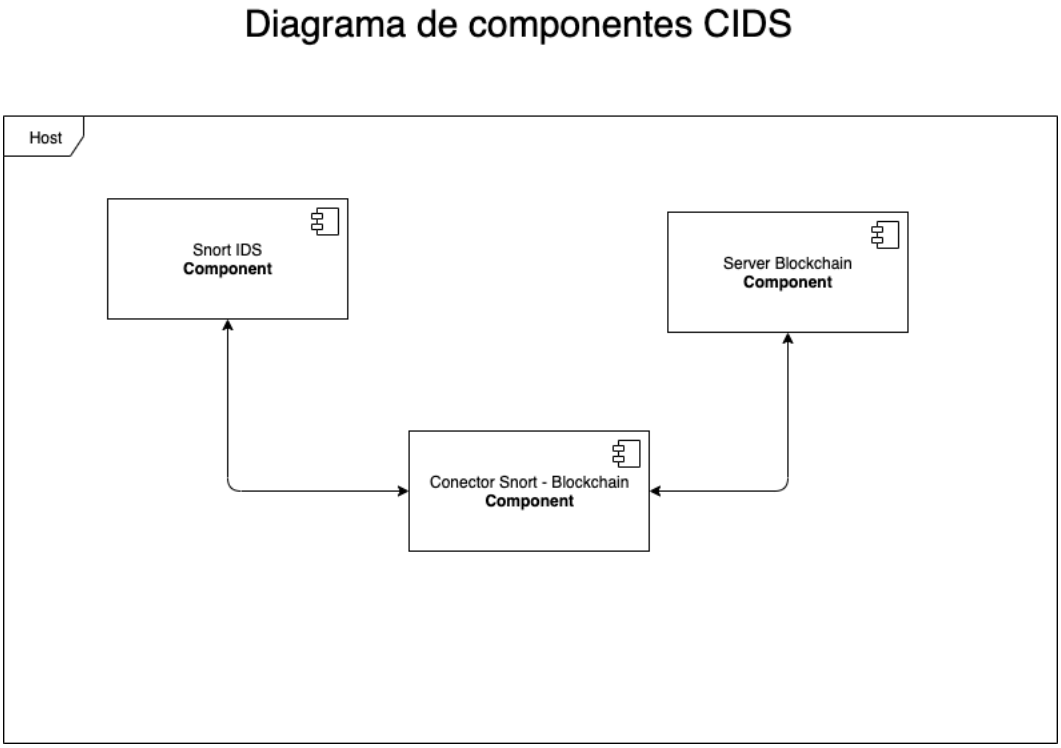

Ilustración 17 - Diagrama de componentes

Como se puede apreciar en la imagen el sistema consta de tres componentes:

1. **Snort IDS:** un servicio Snort que actuara como IDS y usara tanto las reglas definidas localmente como las intercambiadas mediante la Blockchain, según el criterio aplicado de reglas.

2. **Server Blockchain:** un servicio Blockchain, desarrollado en Python, que interconectará al nodo a la red CIDS y permitirá el intercambio de información.

3. **Conector Snort <-> Blockchain:** un conector, desarrollado en Python, que permitirá tanto enviar información para que sea enviada al servidor de Blockchain para la

difusión de dicha información, como transformar la información recibida de la red Blockchain a reglas de Snort.

4.3.2 Formato de datos intercambiados

Los datos intercambiados en la Blockchain serán bloques que contendrán distintas transacciones, por tanto, la definición del formato de ambos será el siguiente.

Bloques

Los bloques tendrán los siguientes campos:

- **index:** enumera el bloque, será una de las formas de identificar el bloque junto con el hash.
- **transaction:** contiene el payload o carga de información que se transmitirá con el bloque. Esta información será la ip del host que envía dicha información, su localización y las reglas IDS que quiera transmitir.
- **timestamp:** marca de tiempo en formato Linux, para dejar constancia de cuando se creó dicho bloque.
- **previous_hash:** contiene el hash válido del bloque anterior para mantener la correlación de los bloques y evitar falsificaciones en las transacciones.
- **nonce:** es una limitación que se añade para dificultar el “proof-of-work”, se define como un número de ceros, que hay que añadir delante del hash del bloque para que dicho bloque sea válido. A mayor nonce mayor dificultad de encontrar un hash válido.

Transacciones

Las transacciones tendrán los siguientes campos:

- **rules:** contendrá las reglas que se quieren transmitir con la Blockchain.
- **host:** contendrá la ip que realiza la transacción.

- **location:** contendrá la localización del nodo que envía dicha transacción.

5. Implementación

En este capítulo se mostrará y explicará la implementación realizada tanto del servidor Blockchain como del conector con el IDS (Snort).

5.1 Estructura de la aplicación

A continuación, se mostrará una imagen con los archivos que contiene la aplicación, mostrando la estructura de la misma, y explicando cada uno de ellos a continuación.

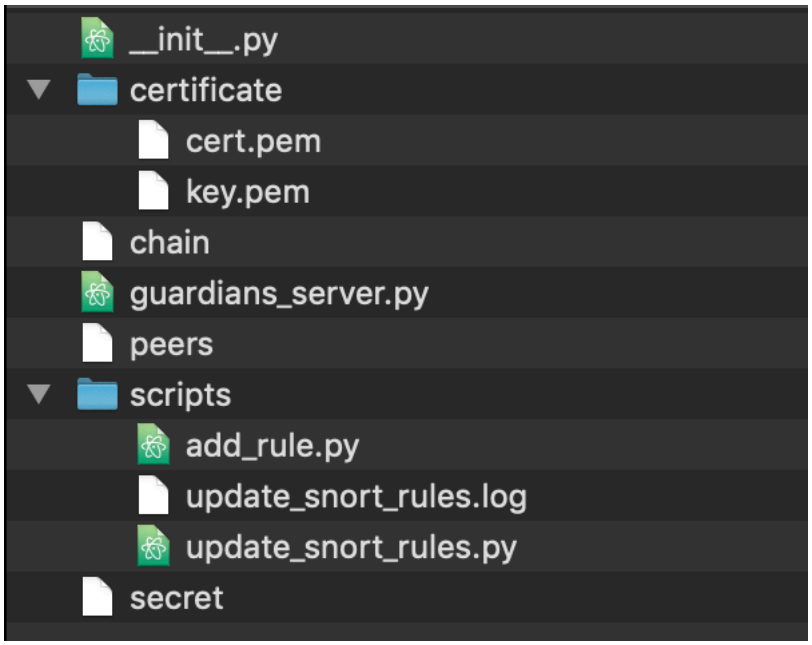

Ilustración 18 - Estructura de la herramienta

- **__init__.py:** archivo que se utiliza en Python para convertir el código en un módulo, y así permitir posteriormente su importación.

- **guardians_severr.py:** es el código principal de la aplicación, contiene el grueso del código de la herramienta.

- **chain:** contiene una copia actual de la cadena de la Blockchain.

- **peers:** contiene la lista de nodos de la Blockchain.

- **secret:** contiene una contraseña secreta utilizada en ciertas llamadas en la aplicación.

- **certificate/:**

    - **cert.pem:** contiene el certificado generado por el nodo.

    - **key.pem:** contiene el par de claves rsa pública-privada.

- **scripts/:**

    - **add_rule.py:** script usado para introducir nuevas reglas en la Blockchain.

    - **update_snort_rules.py:** script usado para actualizar las reglas del propio nodo.

    - **update_snort_rules.log:** registro de la ejecución del script "update_snort_rules.py", indicando si la actualización ha sido satisfactoria o, por el contrario, ha ocurrido algún error.

    - **join_in_blockchain.py:** script usado para añadir un nuevo nodo a la blockchain de forma semiautomática.

5.2 Implementación de la Blockchain

Tras distintas búsquedas de códigos existentes, escritos en Python, que pudieran dar una base para la creación de la red Blockchain. Finalmente se escogió el siguiente código de Github https://github.com/satwikkansal/python_blockchain_app, dado que, sin hacer uso de librerías externas, crea un conjunto de métodos para el funcionamiento de la Blockchain. Además, usa Flask la librería de Python comentada anteriormente en este estudio, para el intercambio de paquetes http.

Tras la realización de varias pruebas de funcionamiento con dicho código se comprobó que no funcionaba correctamente. Se decidió arreglar los fallos en el código para el correcto funcionamiento de la Blockchain, además de añadir las modificaciones necesarias para posteriormente, intercambiar reglas del CIDS a través de dicha Blockchain. A continuación, se mostrará y se explicará, con las correcciones de código y modificaciones ya aplicadas, las funciones más relevantes para el funcionamiento de la Blockchain.

5.2.1 Definición de las clases Block y Blockchain

```
9   class Block:
10      def __init__(self, index, transaction, timestamp, previous_hash, nonce=0):
11          self.index = index
12          self.transaction = transaction
13          self.timestamp = timestamp
14          self.previous_hash = previous_hash
15          self.nonce = nonce
```

Ilustración 19 - Clase Block

```
22  class Blockchain:
23      difficulty = 2
24
25      def __init__(self):
26          self.unconfirmed_transactions = []
27          self.chain = []
28          self.create_genesis_block()
```

Ilustración 20 - Clase Blockchain

```
37  @property
38  def last_block(self):
39      return self.chain[-1]
```

Ilustración 21 - Función last_block()

En estos fragmentos de código se definen las siguientes clases fundamentales de la Blockchain:

- **Block:** es la definición de la estructura de los bloques que se intercambiarán en la red Blockchain. Las propiedades, explicadas en el apartado de diseño, necesarias para esta clase son: index, transaction, timestamp, previous_hash y nonce.

- **Blockchain:** clase que define la Blockchain para el intercambio de bloques, en la misma se definen las funciones necesarias para el funcionamiento de la Blockchain, a excepción de la función que calcula el hash que pertenece a la clase Block. Estas son sus propiedades:

    - **unconfirmed_transactions:** es una lista donde se almacenan las transacciones cuando aún no han sido validadas para introducirlas en la cadena.

    - **chain:** la lista donde se almacenarán los bloques validados.

    - **create_genesis_block():** función que genera el primer bloque de la Blockchain o toma una cadena ya existente.

    - **last_block:** devuelve el último bloque de la cadena.

Con esta parte de la implementación se pretende cumplir el requisito **R-04**.

5.2.2 Cálculo del hash

```
17    def compute_hash(self):
18        block_string = json.dumps(self.__dict__, sort_keys=True)
19        return sha256(block_string.encode()).hexdigest()
```

Ilustración 22 - Función compute_hash()

Esta función, que es la única perteneciente a la clase “Block”, calcula el hash del bloque en cuestión con el algoritmo criptográfico sha256, el cual aún no se ha conseguido vulnerar, y devuelve el hash obtenido.

Con esta parte de la implementación se pretende cumplir el requisito **R-04**.

5.2.3 Definición del primer bloque o inicialización por medio de cadena existente

```
35    def create_genesis_block(self):
36        data = []
37        chain_blockchain = []
38        if os.path.isfile('./chain'):
39            with open('./chain', 'r') as file:
40                data = file.read()
41            chain = data.split('\n')
42            if len(chain) > 0:
43                for bck in chain:
44                    chn = eval(bck)
45                    if isinstance(chn['chain'], str):
46                        for dic in eval(chn['chain']):
47                            block = Block(dic['index'], dic['transaction'],
48                            dic['timestamp'], dic['previous_hash'])
49                            block.hash = block.compute_hash()
50                            chain_blockchain.append(block)
51                    else:
52                        for dic in chn['chain']:
53                            block = Block(dic['index'], dic['transaction'],
54                            dic['timestamp'], dic['previous_hash'])
55                            block.hash = block.compute_hash()
56                            chain_blockchain.append(block)
57                self.chain = chain_blockchain
58            else:
59                genesis_block = Block(0, [], time.time(), "0")
60                genesis_block.hash = genesis_block.compute_hash()
61                self.chain.append(genesis_block)
62        else:
63            genesis_block = Block(0, [], time.time(), "0")
64            genesis_block.hash = genesis_block.compute_hash()
65            self.chain.append(genesis_block)
```

Ilustración 23 - Función create_genesis_block()

Está función se ejecuta en la inicialización de la herramienta y realiza lo siguiente:

- Si existe el fichero “./chain” y su tamaño es mayor que uno, lo que quiere decir que el archivo no está vacío, obtiene los datos del mismo y actualiza la cadena de la Blockchain. De esta forma el servidor puede ser reiniciado sin que se pierda la información de la cadena en ese nodo.

- Si no existe tal fichero, genera el primer bloque con index cero, lista de transacciones vacía, el timestamp en el que se crea dicho bloque y el “previous_hash” que se define como cero. Acto seguido comprueba el hash con la función “compute_hash()” y añade el bloque a la cadena.

Con esta parte de la implementación se pretende cumplir el requisito **R-04**, **R-12** y **R-15**.

5.2.4 Proof of Work

```
63    def proof_of_work(self, block):
64        block.nonce = 0
65
66        compute_hash = block.compute_hash()
67        while not compute_hash.startswith('0' * Blockchain.difficulty):
68            block.nonce += 1
69            compute_hash = block.compute_hash()
70
71        return compute_hash
```

Ilustración 24 - Función proof_of_work()

Hay un problema que podría provocar una falsificación de la cadena, si se cambia el bloque anterior, se pueden volver a calcular los hashes de todos los bloques siguientes con bastante facilidad y crear una cadena de bloques válida diferente. Para prevenir esto, se hace que la tarea de calcular el hash sea difícil y aleatoria.

La forma en la que se consigue es la siguiente. En lugar de aceptar cualquier hash para el bloque, se le añade alguna restricción. Añadiendo una restricción de que el hash debe de comenzar con 2 ceros a la izquierda (propiedad “difficulty” de la clase “Blockchain”). Teniendo en cuenta que, a menos que cambie el contenido del bloque, el hash no va a cambiar.

Para esto se usa la propiedad "nonce" del bloque. Un nonce es un número que seguirá cambiando hasta que se obtenga un hash que satisfaga nuestra restricción. El número de ceros a la izquierda (el valor 2 en nuestro caso) determina la "dificultad" del algoritmo de Proof of Work.

Con esta parte de la implementación se pretende cumplir el requisito **R-04**.

5.2.5 Añadir nuevo bloque

```
64    def add_block(self, block, proof):
65
66        previous_hash = self.last_block.hash
67
68        if previous_hash != block.previous_hash:
69            return False
70
71        if not Blockchain.is_valid_proof(block, proof):
72            return False
73
74        block.hash = proof
75        self.chain.append(block)
76        try:
77            with open('./chain', 'w') as file:
78                file.write(get_my_chain())
79        except:
80            print("Cannot write in the chain's file")
81
82        return True
```

Ilustración 25 - Función add_block()

Esta función añade un nuevo bloque a la cadena. Como argumentos se le introduce el bloque en cuestión y el valor devuelto por el "proof_of_work()". La función comprueba que el valor de "previous_hash" coincide con el del hash del último bloque añadido a la cadena, si coinciden pasa los argumentos introducidos a esta función a "is_valid_proof()", que comprobará que es un bloque válido (está función se explicará en el siguiente apartado), si es así, el bloque se añadirá a la cadena. Actualizando también el archivo './chain'.

Con esta parte de la implementación se pretende cumplir el requisito **R-04**, **R-12** y **R-15**.

5.2.6 Validar un bloque

```
68    def is_valid_proof(cls, block, block_hash):
69        return (block_hash.startswith('0' * Blockchain.difficulty) and
70            block_hash == block.compute_hash())
```

Ilustración 26 - Función is_valid_proof()

Para validar un bloque hay que comprobar que su hash empieza por tantos ceros (dado que el nonce que se ha definido es un 0) como dificultad tiene nuestro algoritmo, en este caso 2.

También hay que validar que el hash que viene en el bloque es el calculado con la función "compute_hash()". La función retorna un valor True si esto se cumple, si no lo hace retorna el valor False.

Con esta parte de la implementación se pretende cumplir el requisito **R-04**.

5.2.7 Minado del pool de transacciones sin confirmar

```
93    def mine(self):
94        if not self.unconfirmed_transactions:
95            return False
96
97        last_block = self.last_block
98
99        new_block = Block(index=last_block.index+1,
100                          transaction=self.unconfirmed_transactions,
101                          timestamp=time.time(),
102                          previous_hash=last_block.hash)
103       proof = self.proof_of_work(new_block)
104       added = self.add_block(new_block, proof)
105
106       if added:
107           self.unconfirmed_transactions = []
108           announce_new_block(new_block)
109           return new_block.index
110       else:
111           return False
```

Ilustración 27 - Función mine()

Antes de ser añadidas a la cadena las transacciones son almacenadas en un pool de transacciones sin confirmar. El proceso por el cual las transacciones pasan desde dicho pool a la cadena junto con el computo del Proof-of-Work se denomina minado de bloques. Una vez que este proceso ha sido realizado, se dice que el bloque en cuestión ha sido minado.

La función de minado primeramente comprueba que el pool de transacciones sin confirmar no está vacío, seguidamente crea un nuevo bloque asignándole el siguiente index, la transacción sin confirmar, el timestamp y el hash del bloque anterior. A continuación realiza el "proof_of_work()" y llama a la función "add_block()" pasándole como parámetro el bloque y la prueba recibida por el Proof-of-Work si el valor devuelto por dicha función es True, vacía el pool de transacciones sin confirmar, anuncia el bloque a la red mediante la función "announce_new_block()", que se explicará posteriormente en este trabajo, y retorna el index del bloque, en caso contrario retorna el valor False.

Con esta parte de la implementación se pretende cumplir el requisito **R-04**.

5.2.8 Interconexión entre los nodos

```
113    app = Flask(__name__)
114
115    blockchain = Blockchain()
116    blockchain.create_genesis_block()
```

Ilustración 28 - Inicialización de la blockchain

Como se ha mencionado en el apartado de tecnologías las conexiones entre los nodos se realizarán por http/https, para ello se ha usado la librería Flask de Python anteriormente mencionada que permite crear endpoints para realizar las distintas acciones en la Blockchain.

En este fragmento de código se crea la aplicación con Flask, se inicializa la Blockchain mediante una llamada a su clase, se genera el primer bloque o se obtendrá una cadena ya almacenada mediante la función “create_genesis_block()”.

Los diferentes endpoints que posee la aplicación son los siguientes:

- **/new_transaction:** permite añadir una nueva transacción al pool de transacciones sin confirmar.
- **/mine:** permite minar un bloque con las transacciones del pool.
- **/add_block:** permite añadir un nuevo bloque a la cadena.
- **/chain:** devuelve la cadena actual de la Blockchain.
- **/register_node_with:** llamada interna para registrar un nuevo nodo en la red Blockchain.
- **/register_node:** llamada externa para registrar la información del nodo que envía la petición.

A continuación, se explicarán en detalle estas llamadas y se mostrará su código parcial o total.

Con esta parte de la implementación se pretende cumplir el requisito **R-01** y **R-04**.

5.2.9 Cifrado TLS

```
418  # --------------------RUN APPLICATION--------------------
419  if __name__ == '__main__':
420      app.run(host='0.0.0.0', port=8000, ssl_context=('./certificate/cert.pem',
421                                                      './certificate/key.pem'))
```

Ilustración 29 - Inicializador de la aplicación con TLS

La herramienta monta un servicio en escucha a internet en el puerto 8000, las conexiones establecidas con la aplicación irán cifradas mediante TLS. Para ello se añaden los archivos “cert.pem”, que contiene el certificado y “key.pem”, que contiene el par de claves, al contexto ssl. Cada nodo tendrá su propio certificado y par de claves.

Con esta parte de la implementación se pretende cumplir el requisito **R-04** y **R-11**.

5.2.10 Lista de nodos o peers

La lista de nodos o peers será almacenada en el archivo ‘./peers’, que contendrá la lista de nodos que pertenecen a la Blockchain y a los que se le enviará la información. Este archivo será usado por varias funciones del código. Cada uno de los nodos debe eliminar su propia IP de la lista para evitar bucles infinitos de peticiones https.

5.2.11 Solicitud para añadir nueva transacción

```
126  @app.route('/new_transaction', methods=['POST'])
127  def new_transaction():
128      data = request.get_json()
129      required_fields = ["rule", "location", "host"]
130
131      for field in required_fields:
132          if not data.get(field):
133              return "Invalid transaction data", 404
134
135      data["timestamp"] = time.time()
136      blockchain.add_new_transaction(data)
137
138      return "Success", 201
```

Ilustración 30 - Llamada /new_transaction

Para realizar una nueva transacción se envía una petición POST a "/new_transaction" con un valor para cada una de las siguientes propiedades:

- **Host:** contendrá la dirección ip de quien transmite la regla para asociar dicha transacción al nodo y así mantener la trazabilidad.

- **Rule:** contendrá la regla que quieres transmitir el nodo a la blockchain.

- **Location:** contendrá la localización de dicho nodo que podrá ser desde una coordenada al nombre de una sede de la organización.

La función recupera los datos, le añade un timestamp y añade la nueva transacción al pool de transacciones sin confirmar mediante una llamada a la función "add_new_transaction()", pasándole como parámetro la información recuperada de la petición junto con el timestamp.

Con esta parte de la implementación se pretende cumplir los requisitos **R-01**, **R-04** y **R-05**.

5.2.12 Solicitud de minado de transacciones

La solicitud de minado del pool de transacciones sin confirmar, se realiza enviando una solicitud POST a "/mine", la función internamente llama a la función "mine()" de la clase "Blockchain" e indica si el bloque fue minado correctamente o hubo un error en el proceso.

La función “mine()” anuncia el nuevo bloque a la red blockchain mediante la función “announce_new_block()”.

```
380  def announce_new_block(block):
381
382      headers = {"Content-Type": "application/json"}
383      with open('./peers', 'r') as file:
384          data_peers = file.read()
385      peers = data_peers.split('\n')
386      for peer in peers:
387          if peer:
388              requests.post("https://"+peer+"add_block", headers=headers,
389              data=json.dumps(block.__dict__, sort_keys=True), verify=False)
```

Ilustración 31 - Función announce_new_block()

Esta función envía una petición POST a cada uno de los nodos de la lista de peers, a la ruta “/add_block” para que verifiquen el bloque y lo añadan a la cadena Blockchain, para así tener la cadena actualizada en cada nodo.

```
230  @app.route('/add_block', methods=['POST'])
231  def verify_and_add_block():
232      block_data = request.get_json()
233      block = Blockchain(block_data["index"],
234                         block_data["transactions"],
235                         block_data["timestamp"],
236                         block_data["previous_hash"])
237      proof = block_data['hash']
238      added = blockchain.add_block(block, proof)
239
240      if not added:
241          return "The block was discarded by the node", 400
242
243      return "Block added to the chain", 201
```

Ilustración 32 - Llamada /add_block

La petición recibida por los nodos ejecuta la función “verify_and_add_block()” que obtendrá los datos de la petición POST recibida, creara un bloque y llamará a la función interna “add_block()” pasándole como parámetro el bloque creado y el hash recibido, esta función explicada anteriormente verificara el nuevo bloque y devolverá un valor True si el bloque es validado y un valor False si es rechazado. Finalmente envía la respuesta de la petición indicando si el nodo se ha añadido correctamente o no.

Con esta parte de la implementación se pretende cumplir los requisitos **R-01**, **R-04** y **R-10**.

5.2.13 Consenso entre nodos

```
#Consensus
def get_longest_chain():

    global blockchain

    my_chain = requests.post('https://127.0.0.1:8000/get_chain',
                             verify=False).json()
    current_len = my_chain['length']
    longest_chain = my_chain['chain']

    with open('./peers', 'r') as file:
        data_peers = file.read()
    peers = data_peers.split('\n')

    for node in peers:
        if node:
            print('https://{}get_chain'.format(node))
            response = requests.post('https://{}get_chain'.format(node),
                                     verify=False)
            if response:
                print(response.status_code)
                length = response.json()["length"]
                chain = response.json()["chain"]
                if length > current_len and
                    blockchain.check_chain_validity(chain):
                    current_len = length
                    longest_chain = chain
    if longest_chain:
        return longest_chain
    else:
        return False
```

Ilustración 33 - Función get_longest_chain()

Está función se usa para llegar a un consenso entre los nodos de cuál es la chain válida más larga. Para ello obtiene la chain de cada uno de los nodos, y la suya propia, mediante llamadas POST a "/get_chain". Para cada una de ellas comprueba su validez y su tamaño, la chain válida de mayor tamaño será almacenada y devuelta al final de la función.

Esta función es llamada cuando un nuevo nodo es añadido a la red Blockchain, para enviarle la chain validada por todos los nodos y así poder formar parte de la red Blockchain.

Con esta parte de la implementación se pretende cumplir el requisito **R-01**, **R-04**, **R-06** y **R-07.**

5.2.14 Registro de nuevos nodos

El registro de nuevos nodos se realiza mediante dos funciones, la primera "register_node_with()" es llamada internamente.

```
197  # THIS FUNCTION IS CALL INTERNALLY
198  @app.route('/register_with', methods=['POST'])
199  def register_with_existing_node():
200
201      node_address = request.get_json()["node_address"]
202      myip = request.get_json()["myip"]
203      secret = sha256(request.get_json()["secret"].encode()).hexdigest()
204
205      if not node_address or not secret:
206          return "Invalid data", 400
207
208      data = {"host_origin": myip, "secret": secret}
209      headers = {"Content-Type": "application/json"}
210
211      # Make a request to register with remote node and obtain information
212      response = requests.post("https://" + node_address + "register_node",
213                               data=json.dumps(data), headers=headers, verify=False)
214
215      if response.status_code == 200:
216          print(response.text)
217          global blockchain
218          chain_dump = response.json()['chain']
219
220          response2 = requests.post("https://" + node_address + "peers",
221                                    verify=False)
222          peers = response2.json()['peers']
223          peers.append(node_address)
224          file = open('./peers','w+')
225          for p in peers:
226              if p:
227                  file.write(p+'\r\n')
228          file.close()
229          blockchain = create_chain_from_dump(chain_dump, peers)
230          return "New node successfully registered", 200
231      else:
232          return response.content, response.status_code
```

Ilustración 34 - Llamada /register_with

Se realiza una llamada POST interna a "https://127.0.0.1/register_with" con tres parámetros, "node_address" que contendrá la dirección del nodo con el que se quiere sincronizar, "myip" que contendrá la ip del nodo origen y "secret" que es una clave secreta que solo poseerán los nodos de nuestra red privada de Blockchain.

La función obtiene dicha información de la petición POST, y realiza una llamada a la dirección indicada en la variable “node_address”, junto con su IP y el hash de la clave, a la ruta “/register_node”.

```
# THIS FUNCTION IS CALL EXTERNALLY
@app.route('/register_node', methods=['POST'])
def register_new_peers():
    headers = {"Content-Type": "application/json"}
    node_address = request.get_json()["host_origin"]
    secret = request.get_json()["secret"]
    if not node_address or not secret:
        return "Invalid data", 400

    with open('./secret', 'r') as file:
        code = file.read().replace('\n', '')
    with open('./peers', 'r') as file:
        data_peers = file.read()
    peers = data_peers.split('\n')
    for p in peers:
        if p:
            requests.post("https://" + p + "update_peers", data=json.dumps
            ({'peer': node_address}), headers=headers, verify=False)
    if secret == sha256(code.encode()).hexdigest():
        if node_address not in peers:
            with open('./peers', 'a') as file:
                file.write(node_address+'\n')
            chain = get_longest_chain()
            if chain:
                return json.dumps({"chain": str(chain)}), 200
            else:
                return "Error adding node", 500
        else:
            return "Node already in the peers's list", 200
    else:
        return "Error adding node", 500
```

Ilustración 35 - Llamada /register_node

Esta petición llama a la función “register_new_peers()” en el nodo receptor de la dicha petición. Este obtiene los datos de la petición, comprueba que el hash de la clave enviada es el mismo que el hash recibido en el campo “secret”, si dicho hash es correcto y el nodo no está en la lista actual de “peers” (dicha lista se almacena en un archivo en el servidor), se añade a la lista, se hace una llamada POST a “/update_peers” a cada uno de los nodos para que actualicen su lista de nodos y se le devuelve la cadena actual mediante una llamada a la función “get_longest_chain()”, explicada en el apartado anterior, para que pueda sincronizarse con la información actual de la Blockchain.

Volviendo a la ejecución de la función “register_with_existing_node()”, una vez recibida la respuesta de la petición POST del nodo con el que se quiere sincronizar, si el código de respuesta es 200 se obtiene la cadena de la respuesta de la petición y se llama a la función “create_chain_from_dump()”, pasándole como parámetro la cadena recibida, para actualizar su cadena y estar sincronizado con los otros nodos de la red. En caso contrario devuelve el código de la respuesta recibida y su contenido.

```
315  def create_chain_from_dump(chain_dump, peers):
316
317      chain_dump = chain_dump[1:]
318      chain_dump = chain_dump[:-1]
319      array = chain_dump.split('},')
320      array_dic = []
321      count = 0
322      length = len(array)
323
324      for a in array:
325          if count != length - 1:
326              array_dic.append(json.dumps(a + '}'))
327              count += 1
328          else:
329              array_dic.append(json.dumps(a))
330      chain_blockchain = []
331      blockchain = Blockchain()
332      for block_data in array_dic:
333          block_d = eval(json.loads(block_data.strip()))
334          block = Block(block_d["index"],
335                        block_d["transaction"],
336                        block_d["timestamp"],
337                        block_d["previous_hash"],
338                        block_d['hash'])
339          chain_blockchain.append(block)
340      blockchain.chain = chain_blockchain
341
342      file_chain = []
343      for c in chain_blockchain:
344          file_chain.append(c.__dict__)
345      dic = {"length": len(file_chain), "chain": json.dumps(file_chain),
346                                                 "peers": peers}
347      file = open('./chain', 'w+')
348      file.write(str(dic))
349      file.close()
```

Ilustración 36- Función create_chain_from_dump()

Con esta parte de la implementación se pretende cumplir el requisito **R-01**, **R-04**, **R-06**, **R-08**, **R-13** y **R-15.**

5.2.15 Solicitar cadena

```
328  @app.route('/get_chain', methods=['POST'])
329  def get_my_chain():
330      chain_data = []
331      for block in blockchain.chain:
332          chain_data.append(block.__dict__)
333      with open('./peers', 'r') as file:
334          data_peers = file.read()
335      peers = data_peers.split('\n')
336      return json.dumps({"length": len(chain_data),
337                         "chain": chain_data,
338                         "peers": peers})
```

Ilustración 37 - Llamada /get_chain

Mediante una llamada POST, que puede ser interna o externa, a "/chain" se ejecuta la función "get_chain()" que devolverá todos los bloques de la cadena actual, la lista de peers y el tamaño de la cadena. El tamaño de la cadena se utiliza para asegurar que se tiene la cadena más actualizada, ya que la cadena más larga validada será la más actualizada.

Con esta parte de la implementación se pretende cumplir el requisito **R-01**, **R-04** y **R-15.**

5.3 Configuración de IDS (Snort)

Se debe instalar y configurar Snort como NIDS, en este apartado no se explicará cómo realizar esta configuración ya que hay muchos tutoriales en internet para realizarlo. Se han encontrado tutoriales y múltiples documentaciones de instalación y configuración, según plataforma y versión en la documentación oficial de Snort:

https://www.snort.org/documents#OfficialDocumentation.

Una vez realizada dicha instalación y configurado para que corra como un servicio, lo único que se tiene que hacer es crear el archivo "/etc/snort/rules/blockchain.rules" y darle los permisos necesarios para que tanto Snort como la aplicación lo puedan leer, ejecutar y escribir. Este archivo contendrá las reglas que se compartirán con la Blockchain, dicho archivo será actualizado con las reglas que compartirán cada uno de los IDS.

Ilustración 38 - Configuración de reglas en snort.conf

Este archivo se debe añadir a la configuración de Snort “/etc/snort/snort.conf”, en la sección de reglas específicas como se indica en la imagen.

Con esta parte de la implementación se pretende cumplir el requisito **R-02.**

5.4 Implementación del conector IDS <-> Blockchain

El conector está formado por una pequeña implementación en el servidor Blockchain y dos scripts de Python, para añadir nuevas reglas y para actualizar las reglas de Snort.

5.4.1 Script para añadir nuevas reglas

```
2   import requests
3   import json
4
5   host = input('Introduce your IP connected to blockchain: ')
6   location = input('Introduce your location: ')
7   rule = input('Add new rule to pool: ')
8
9   data = {"host": host, "location": location, "rule": rule}
10  headers = {"Content-Type": "application/json"}
11
12  response_nt = requests.post('https://127.0.0.1:8000/new_transaction',
13                              data=json.dumps(data), headers=headers, verify=False)
14  print(response_nt)
15  print(response_nt.status_code)
16
17  if response_nt.status_code == 200:
18      print('Transaction added to unconfirmed transactions.')
19      print('Sending request to mine the block.')
20      response_mine = requests.post('https://127.0.0.1:8000/mine', verify=False)
21
22      if response_mine.status_code == 200:
23          print('The block has been mined.')
24          print('Writing new rule into blockchain.rules temporarily, until update script will run.')
25          try:
26              with open('/etc/snort/rules/blockchain.rules', 'a') as file:
27                  file.write(rule)
28          except:
29              print('Error writing temporary rules.')
30
31          print('Done!')
32  else:
33      print(response_nt.status_code)
34      print('The transaction has not been accepted')
```

Ilustración 39 - Script add_rule.py

Este script se ejecuta desde consola mediante la orden “python3 add_rule.py”. Al ejecutarlo realiza las siguientes acciones.

1. Solicita que se introduzca el host que realiza la petición, la localización del nodo y la regla que se quiere compartir en la Blockchain.

2. Con esta información se realiza una petición local POST a “/new_transaction”.

3. Si la respuesta de dicha petición incluye el código 200, indicando que todo ha ido bien, realiza otra petición local POST a “/mine”. Esta minará el bloque y llamará a la función “announce_new_block()” que realizará a su vez una petición POST con destino “/add_block” a cada uno de los nodos de la lista de peers, para que estos añadan el nuevo bloque a la cadena.

4. Si todo ha ido bien el script escribe la regla temporalmente en “/etc/snort/rules/blockchain.rules”, hasta que se realice la actualización de las mismas.

5. Si alguno de los pasos anteriores no funciona correctamente, el script devolverá un error y parará la ejecución del mismo.

Con esta parte de la implementación se pretende cumplir el requisito **R-02**, **R-03**, **R-04**, **R-05** y **R-10.**

5.4.2 Script de actualización de reglas IDS

```
import requests
from datetime import datetime
import os.path

if not os.path.isfile('update_snort_rules.log'):
    file = open('./update_snort_rules.log', 'w+')
    file.close()

response = requests.post('https://127.0.0.1:8000/update_snort_rules',
                         verify=False)
if response.status_code == 200:
    time = datetime.now()
    with open('./update_snort_rules.log', 'a') as file:
        file.write(time.strftime("%d/%m/%Y %H:%M:%S") + '
                                 200 - Update was successfully done.\r\n')
else:
    time = datetime.now()
    with open('./update_snort_rules.log', 'a') as file:
        file.write(time.strftime("%d/%m/%Y %H:%M:%S") + " " +
        str(response.status_code) + ' - ERROR update was not done.\r\n')

# To automate this script add the following line to your crontab.
# This task will throw this script everyday at 04:30
# 30 04 * * * python $ROUTE_SCRIPT & systemctl restart snortd
```

Ilustración 40 - Script update_snort_rules.py

Este script se ejecuta desde consola mediante la orden “python3 update_snort_rules.py”. Al ejecutarlo realiza las siguientes acciones.

1. Realiza una petición local POST a “/update_snort_rules”, que se explicará en la siguiente sección.

2. Si la respuesta contiene un 200, indicando que todo ha ido bien, escribe en el archivo “./update_snort_rules.log” una marca de tiempo, el código de la respuesta y un mensaje indicando que todo ha funcionado correctamente.

3. Si la contiene un código distinto, indicando que algo no esperado ha ocurrido, escribe en el archivo “./update_snort_rules.log” una marca de tiempo, el código de la respuesta y un mensaje indicando que la actualización ha fallado.

```
2   05/09/2019 20:10:52 200 - Update was successfully done.
3   05/09/2019 20:13:26 200 - Update was successfully done.
4   05/09/2019 20:15:21 200 - Update was successfully done.
5   05/09/2019 20:18:29 200 - Update was successfully done.
6   05/09/2019 20:43:56 500 - ERROR update was not done.
7   05/09/2019 20:45:52 500 - ERROR update was not done.
8   05/09/2019 20:46:27 500 - ERROR update was not done.
9   05/09/2019 20:46:57 500 - ERROR update was not done.
10  05/09/2019 20:47:30 200 - Update was successfully done.
11  05/09/2019 20:49:34 200 - Update was successfully done.
12  06/09/2019 13:56:33 200 - Update was successfully done.
13  06/09/2019 15:16:30 200 - Update was successfully done.
14  06/09/2019 15:20:02 200 - Update was successfully done.
```

Ilustración 41- Log update_snort_rules.log

La intención de este script es automatizarlo, para que se ejecute cada un periodo de tiempo, mediante cron. Cron es un administrador de procesos en segundo plano que permite planificar la ejecución de scripts. Añadiendo la siguiente línea al crontab, “30 04 * * * python $ROUTE_SCRIPT & systemctl restart snortd”, el script se ejecutará todos los días a las 04:30 y reiniciará el servicio de Snort para que actualice las reglas.

Con esta parte de la implementación se pretende cumplir el requisito **R-02**, **R-03**, **R-04**, **R-05**, **R-10** y **R-16.**

5.3.3 Solicitud de actualización de reglas IDS

```
392  # --------------CONNECTOR IDS <-> BLOCKCHAIN--------------
393  @app.route('/update_snort_rules', methods=['POST'])
394  def update_snort_rules():
395
396      if request.remote_addr == "127.0.0.1":
397          global blockchain
398          rules_data = []
399          for block in blockchain.chain:
400              rules_data.append(block.transaction)
401          shutil.copy2('/etc/snort/rules/blockchain.rules',
402          '/etc/snort/rules/blockchain.backup')
403          os.remove('/etc/snort/rules/blockchain.rules')
404          file = open('/etc/snort/rules/blockchain.rules', 'w+')
405
406          for r in rules_data:
407              if r:
408                  file.write(r[0]['rule']+"\r\n")
409          file.close()
410
411          return 'Update was succesfully done', 200
412      else:
413          return 'Only local request will attend.', 500
414
415  # ------------------------------------------------------
```

Ilustración 42 - Llamada /update_snort_rules

Al recibir la petición POST comprueba que esta provenga de localhost o “127.0.0.1”, para asegurar que solo el mismo nodo puede solicitar la actualización de las reglas. Así cada nodo puede elegir cuando actualizar sus reglas. Seguidamente recorre la cadena de la Blockchain y añade todos los bloques a una lista. Realiza una copia de “blockchain.rules” a “blockchain.backup”, para tener una copia de seguridad local, acto seguido elimina el archivo “blockchain.rules” y lo vuelve a crear para introducir las reglas. Finalmente recorre la lista con los bloques y extrae las reglas de cada uno de ellos, para ir escribiéndolas en “blockchain.rules”. Si todo ha ido bien devuelve el código 200 y un mensaje indicando que todo ha ido bien, en caso contrario devuelve un 500 y un mensaje indicando que un error ha ocurrido.

Con esta parte de la implementación se pretende cumplir el requisito **R-02**, **R-03**, **R-04**, **R-05** y **R-10.**

5.4.4 Script para añadir nuevas reglas

```
3   import requests
4   import json
5   from getpass import getpass
6
7   host = input('Introduce IP to connect: ')
8   myip = input('Introduce your IP: ')
9   print('Introduce secret: \n')
10  password = getpass()
11
12  data = {"node_address": host, "myip": myip, "secret": password}
13  headers = {"Content-Type": "application/json"}
14
15  response_nt = requests.post('https://127.0.0.1:8000/register_with',
16                              data=json.dumps(data), headers=headers,
17                                                     verify=False)
18  print(response_nt)
19  print(response_nt.status_code)
```

Ilustración 43 - Script join_in_blockchain.py

Con este script se solicita la entrada a la red Blockchain. Al ejecutarlo solicita la IP del nodo de la red Blockchain, al que se le va a solicitar la entrada, la IP del nodo que solicita la entrada y la contraseña para acceder a la red (la almacenada en el archivo secret). Una vez introducidos los datos se realiza una llamada POST a https://127.0.0.1:8000/register_with, con dicha información. El script devuelve el texto y el código de estado de la respuesta.

Con esta parte de la implementación se pretende cumplir el requisito **R-03**, **R-04**, **R-06**, **R-08**, **R-11 R-13** y **R-14.**

6. Evaluación

En este capítulo se explicará el entorno usado para las pruebas y los resultados obtenidos de las mismas.

6.1 Entorno de pruebas

La red en la que se han realizado las pruebas, ha sido una red privada con direccionamiento 192.168.56.0/24. Para el desarrollo de las pruebas se han usado tres nodos, uno físico y dos virtuales, con las siguientes características:

- **Nodo 1:** sistema físico con sistema operativo macOS Mojave 10.14.6**,** 16 CPUs y 32 GB RAM. IP: 192.168.56.1.

- **Nodo 2:** sistema virtualizado con sistema operativo CentOS 7.6**,** 4 CPUs y 8 GB RAM. IP: 192.168.56.101.

- **Nodo 3:** sistema virtualizado con sistema operativo CentOS 7.6**,** 4 CPUs y 8 GB RAM. IP: 192.168.56.103.

A continuación, se mostrarán los resultados de las pruebas, paso a paso. En las capturas se puede ver un terminal dividido en tres partes, cada una de ellas es uno de los nodos.

6.2 Resultados de las pruebas

6.2.1 Configuración previa a las pruebas

Antes de que los nodos puedan comunicarse correctamente hay que realizar unas pequeñas configuraciones:

- El archivo peers de cada nodo debe contener las direcciones IP de los otros nodos de la Blockchain, pero no la suya propia.

- Se debe generar una chain inicial, ya sea creándola manualmente o iniciando un nodo sin ninguna chain almacenada, para que así la herramienta generé el primer bloque de la chain. Esta chain con el primer bloque debe estar almacenada en cada uno de los nodos en ‘./chain’, ya que es necesario que el primer bloque sea igual en todos los nodos o ninguno aceptara nuevos bloques de otros nodos.

- Snort tiene que estar instalado en todos los nodos y la ruta “/etc/snort/rules” debe de ser accesible por la herramienta, permitiéndola escribir y leer en la misma.

- La clave almacenada en ‘./secret’ debe de ser la misma en todos los nodos.

- Cada uno de los nodos tiene que generar un certificado TLS y una par de claves criptográficas, situándolos en ‘./certificate’. Se recomienda que la clave sea de RSA 4096 bits.

Con estas configuraciones realizadas, se inician las pruebas.

6.2.2 Comunicación entre nodos

Para empezar, se comprueba que todos los nodos poseen la misma chain, antes de ejecutar la herramienta.

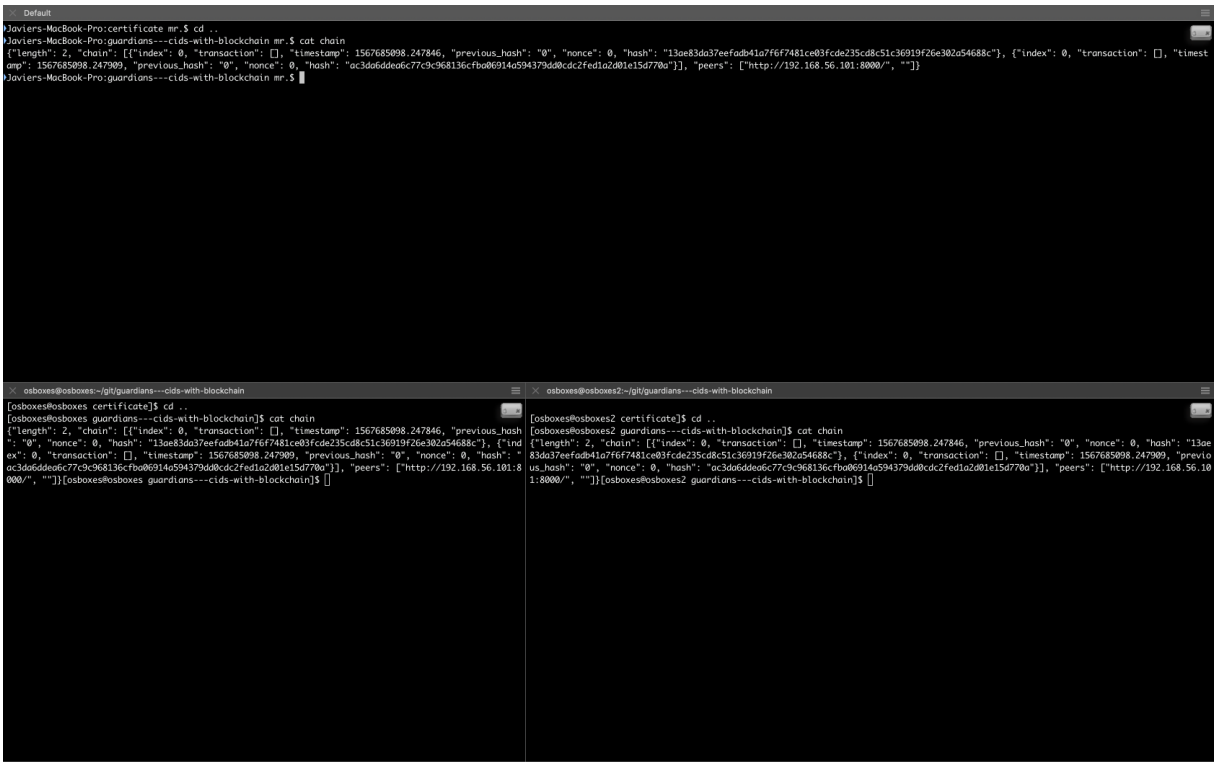

Ilustración 44 - Prueba de comunicación entre nodos 1

Se comprueba que cada uno de los nodos posee su propio certificado TLS, en la ruta ‘./certificate’.

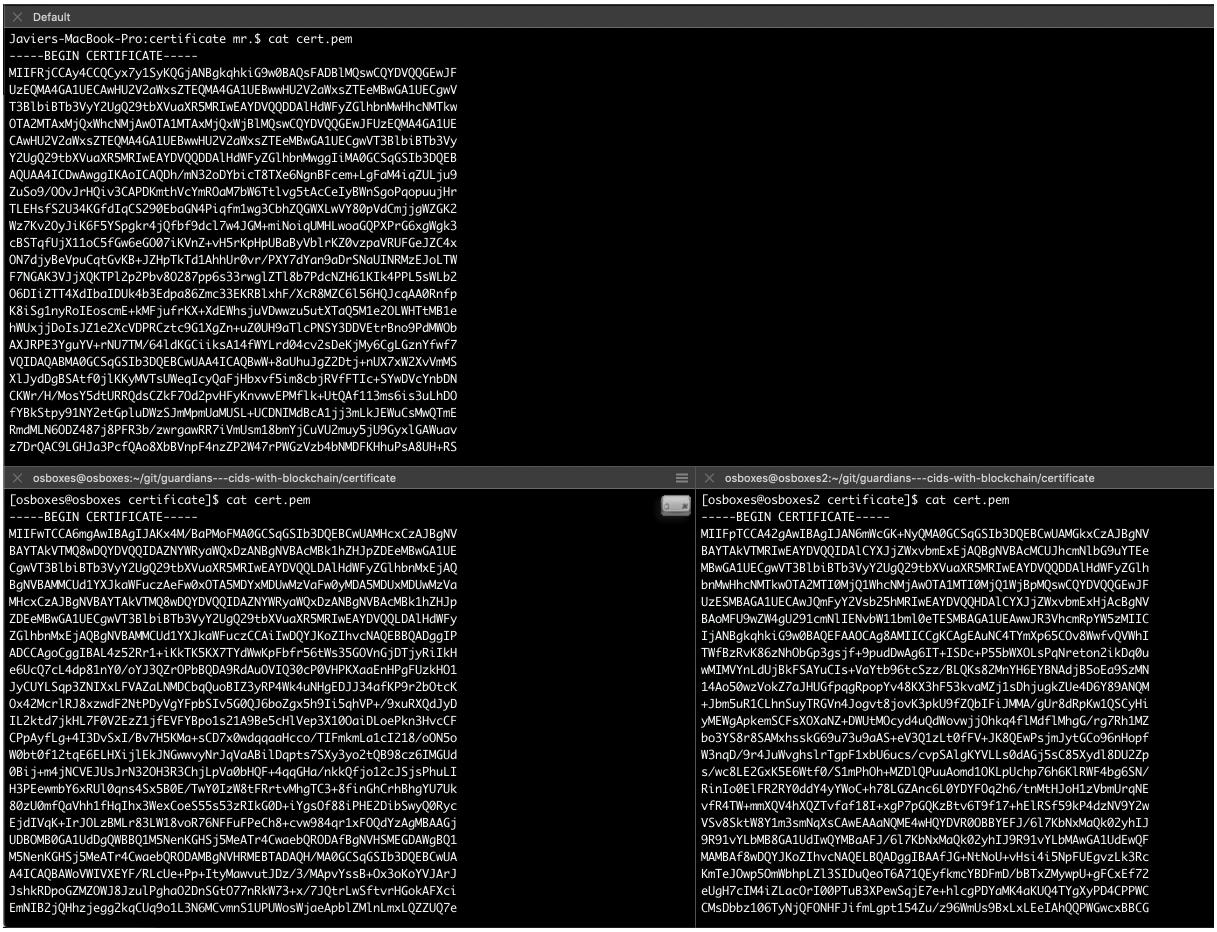

Ilustración 45 - Prueba de comunicación entre nodos 2

Con estas comprobaciones realizadas, se inicia la herramienta en cada uno de los nodos con el comando “python3 guardians_server.py”.

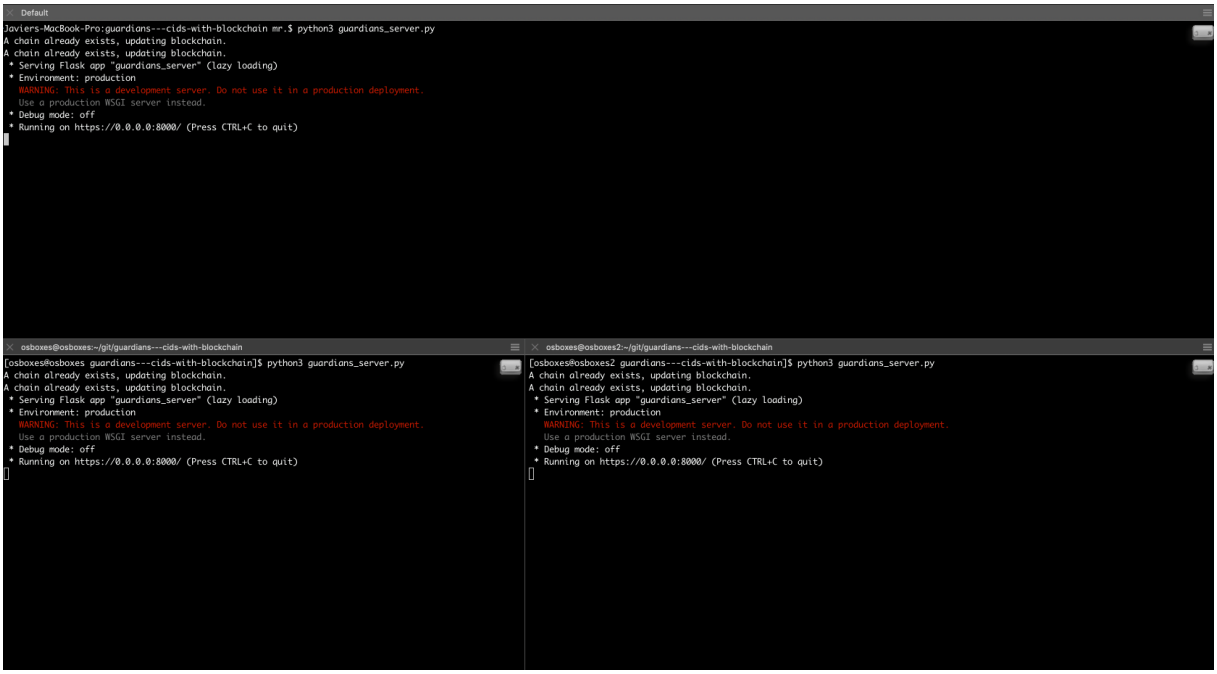

Ilustración 46 - Prueba de comunicación entre nodos 3

Como se puede ver en la imagen, cada uno de los nodos levanta un servicio en el puerto 8000, escuchando en 0.0.0.0.

Se comprueba que la herramienta ha obtenido correctamente la chain almacenada en el archivo ‘./chain’. Para ello desde el nodo 1, se solicita la chain con la llamada ‘/get_chain’, a cada uno de los nodos.

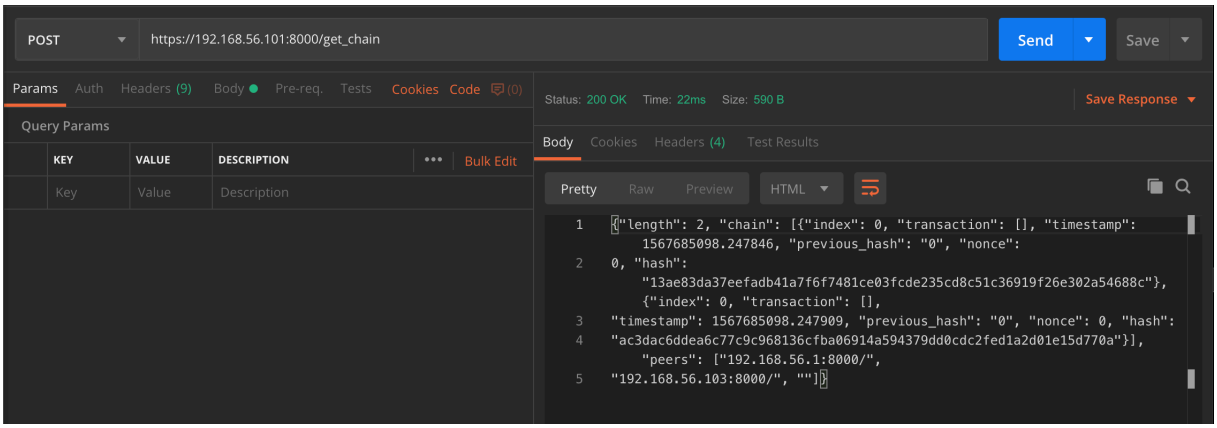

Ilustración 47 - Prueba de comunicación entre nodos 4

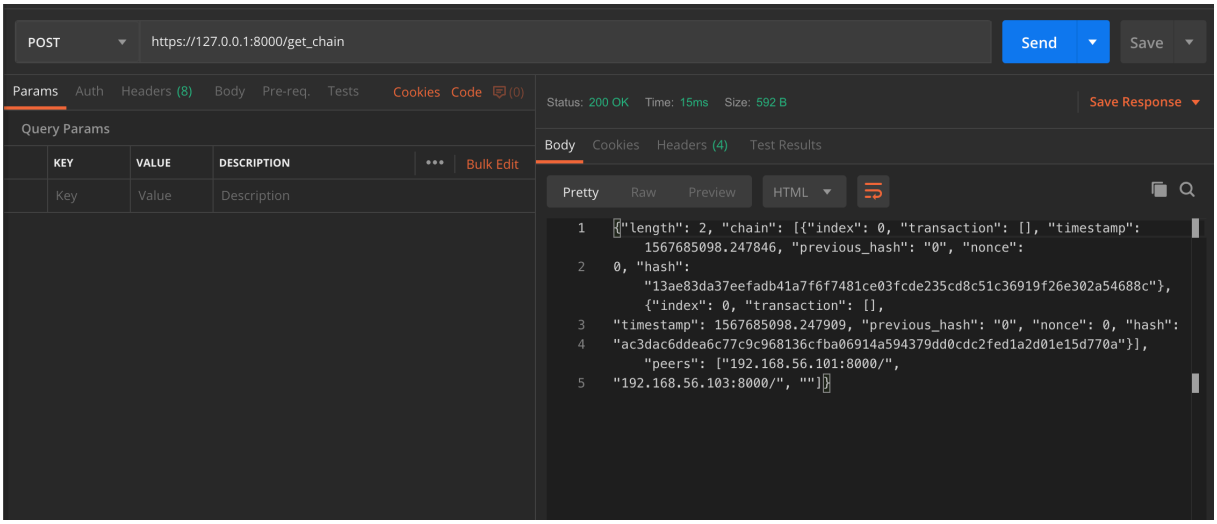

Ilustración 48 - Prueba de comunicación entre nodos 5

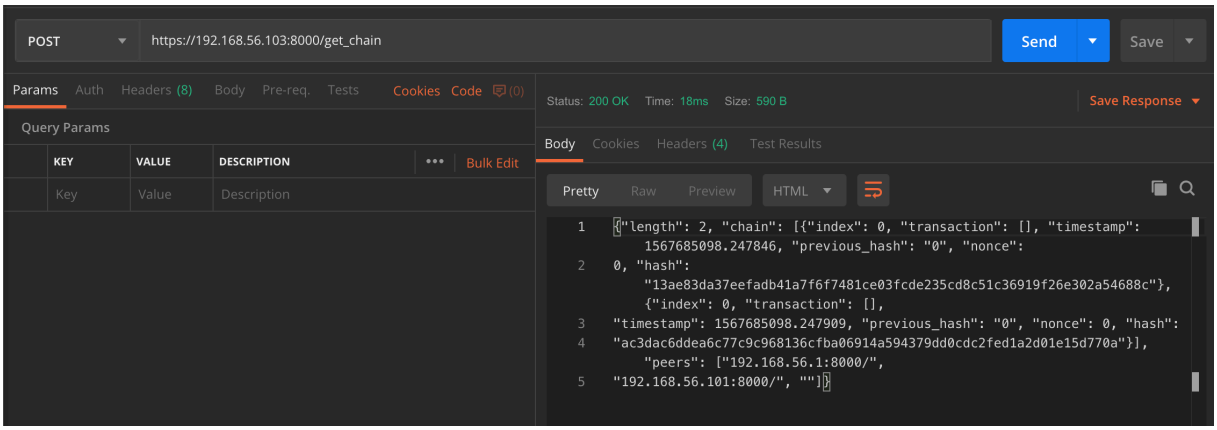

Ilustración 49 - Prueba de comunicación entre nodos 6

Se reciben las respuestas correctamente con el contenido de la chain de cada uno de los nodos. Se verifica que la chain es actualmente la misma en todos los nodos y, por tanto, no deberían tener ningún problema en añadir bloques recibidos por otros nodos.

Si se comprueba el log mostrado por consola en cada uno de los servidores, se puede apreciar las distintas llamadas realizadas entre los nodos.

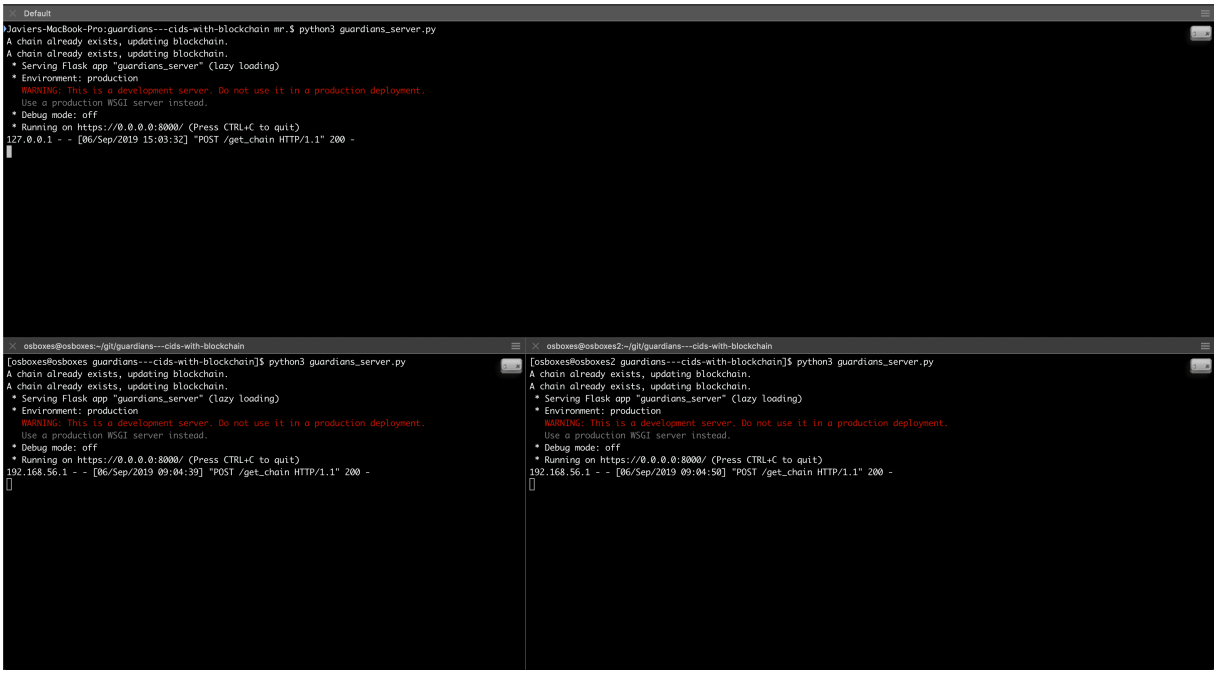

Ilustración 50 - Prueba de comunicación entre nodos 7

6.2.3 Añadir nuevo bloque mediante llamadas POST

En este apartado, se mostrará cómo añadir nuevas reglas mediante llamadas POST para seguir paso a paso el camino que recorre la información, desde su creación en el nodo local hasta su envío a los otros nodos. En un apartado posterior, se mostrará como añadir nuevas reglas mediante un script, dado que la herramienta está pensada para añadir estas nuevas reglas fácilmente mediante dicho script, sin tener que usar herramientas de llamadas http/https.

Se va a hacer uso de la herramienta Postman, mencionada en la sección de tecnologías de este trabajo, para enviar las peticiones a los servidores.

Se va a construir una llamada POST a la dirección ‘https://127.0.0.1:8000/new_transaction’, en el cuerpo de petición se introducen los valores correspondientes a los campos host, rule y location. Esta llamada se realiza localmente y almacenara la transacción en el pool de transacciones sin confirmar.

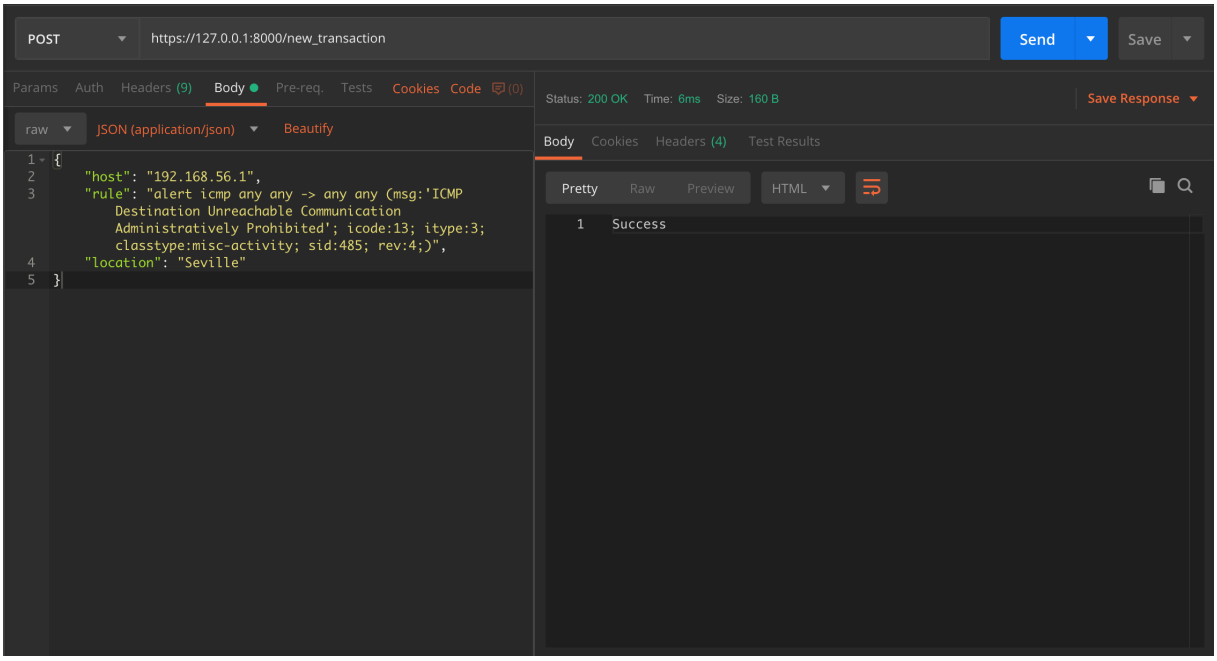

Ilustración 51 - Prueba de añadir nuevo bloque 1

Para que esta transacción sea minada y comunicada a los demás nodos, hay que realizar otra petición local POST a https://127.0.0.1:8000/mine. Esto hará que el nodo compruebe la validez del bloque y si es correcta, lo enviará a los demás nodos, mediante la llamada POST https://$IP:8000/add_block, para que ellos también comprueben su validez y lo añadan a la cadena.

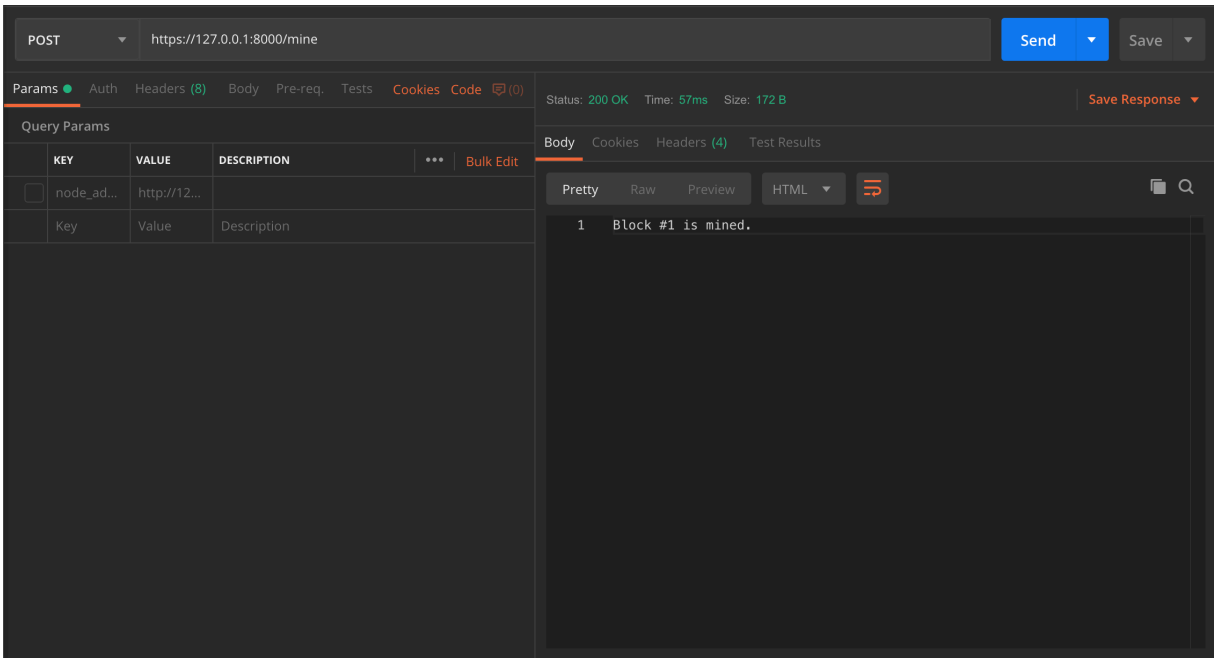

Ilustración 52 - Prueba de añadir nuevo bloque 2

Se aprecia que el bloque se ha minado correctamente, si se comprueba la consola se pueden ver las distintas llamadas intercambiadas entre los nodos. Se puede apreciar también un aviso de solicitud insegura, esto es debido a que los certificados TLS son autofirmados, esto no afecta a la encriptación de las comunicaciones.

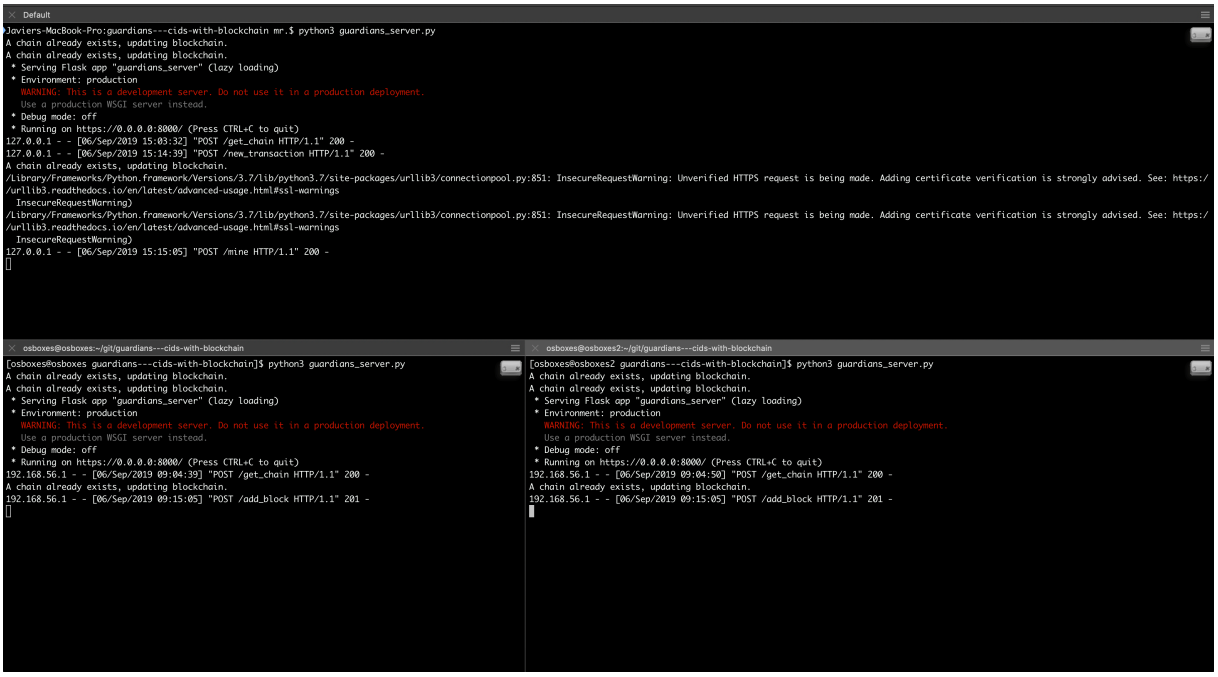

Ilustración 53 - Prueba de añadir nuevo bloque 3

Si se comprueba la cadena en cada uno de los bloques, se puede comprobar que todas se han actualizado correctamente con la información del último bloque enviado.

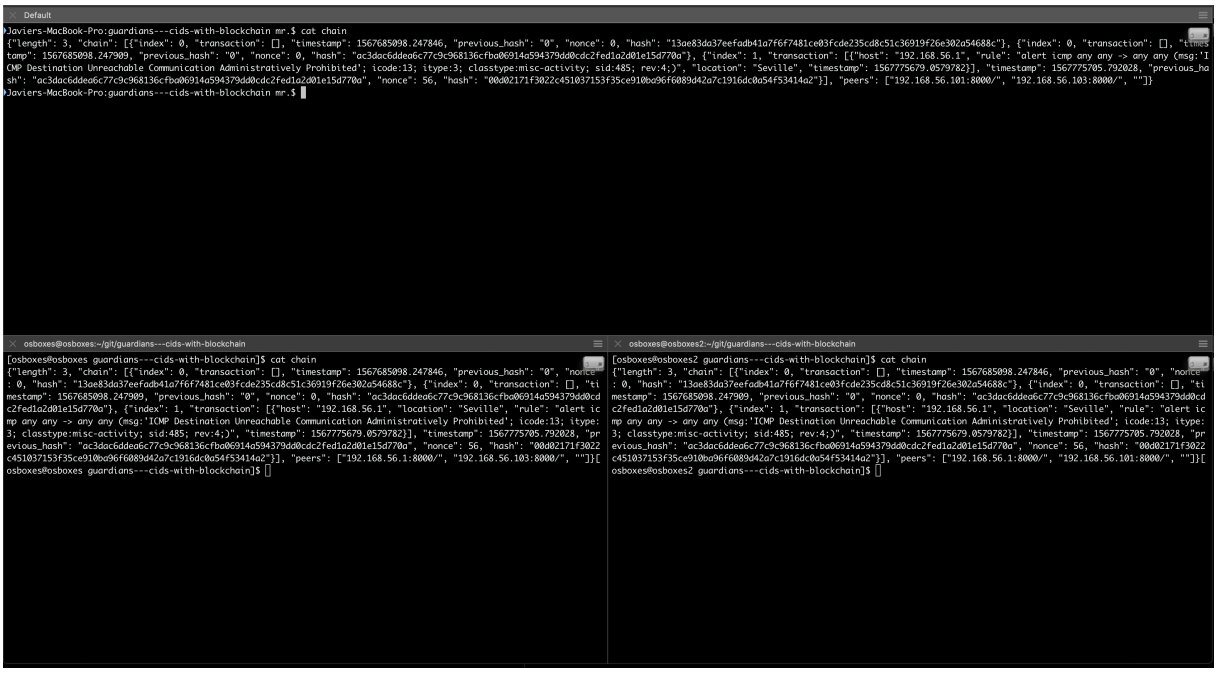

Ilustración 54 - Prueba de añadir nuevo bloque 4

6.2.4 Actualización de reglas mediante script

Para actualizar las reglas hay dos opciones, la primera si la tarea cron está programada para ejecutar el script en un momento concreto, esperar a que se ejecute o ejecutarlo mediante el comando “python3 scripts/update_snort_rules.py”. En este caso se ejecutará el script manualmente.

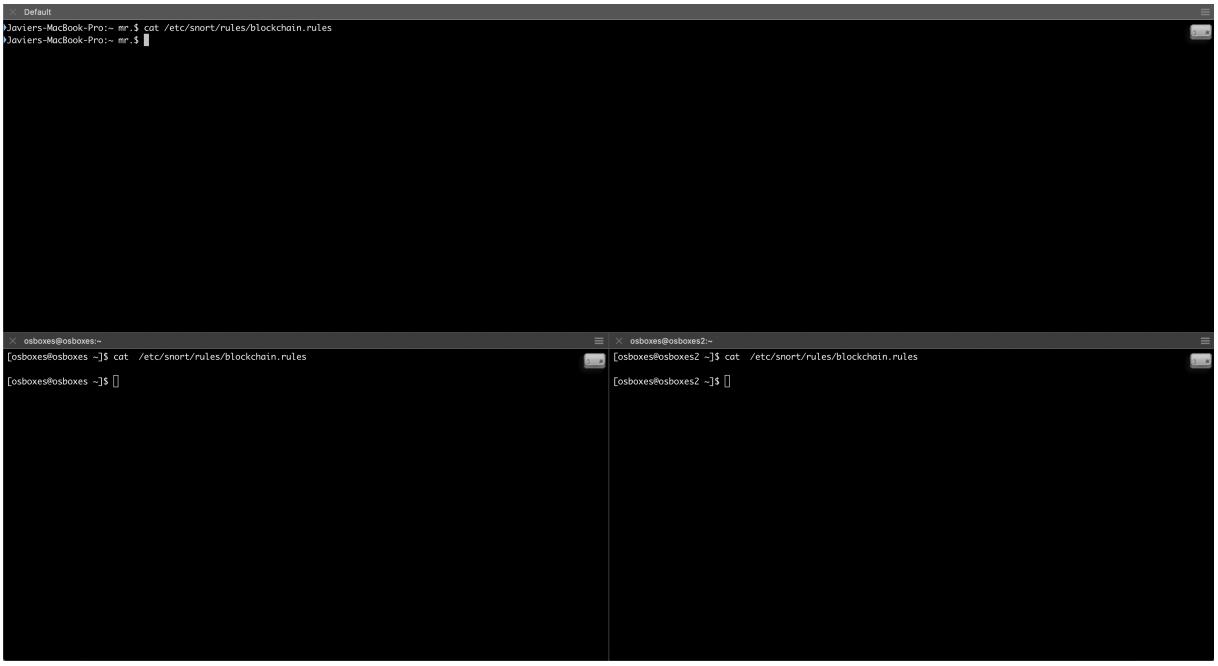

Ilustración 55 - Prueba update_snort_rules.py 1

Se comprueba el estado del archivo de reglas y se ejecuta el script.

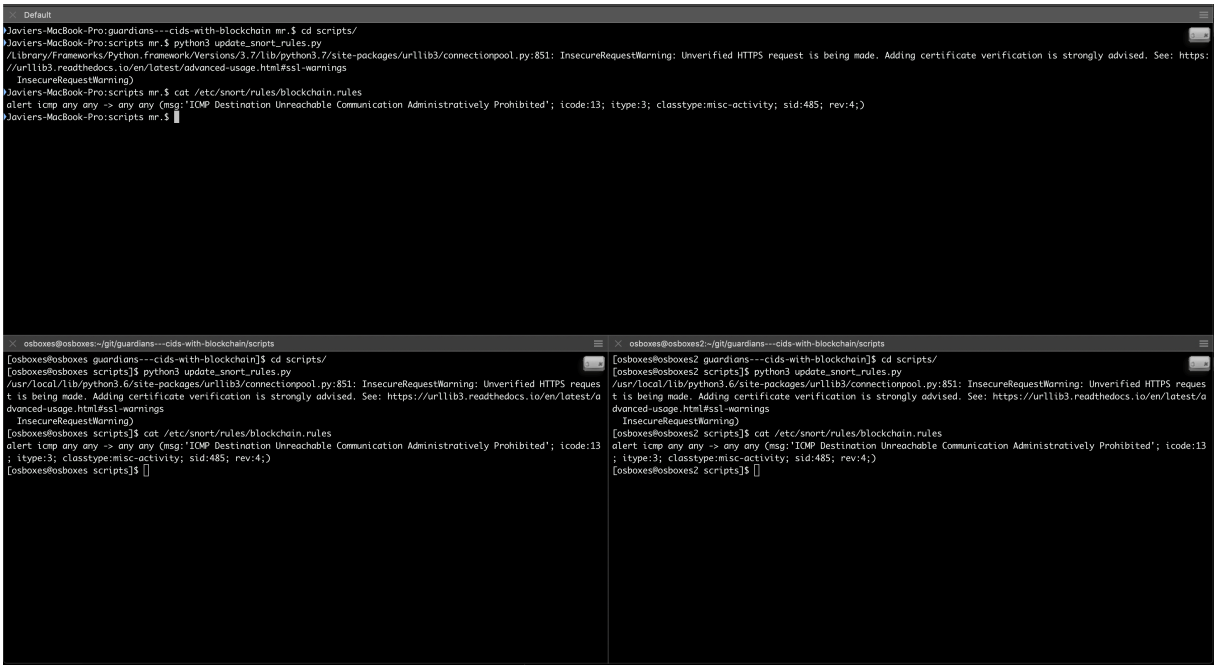

Ilustración 56 - Prueba update_snort_rules.py 2

Al ejecutarlo el nodo hace una llamada local a '/update_snort_rules' y actualiza las reglas en el archivo "/etc/snort/rules/blockchain.rules", como se puede apreciar en la imagen.

Para añadir una tarea al crontab que haga que se actualicen las reglas todos los días a las 04:30 y reinicie el servicio snort, añadimos la siguiente línea al crontab del sistema.

```
[osboxes@osboxes scripts]$ crontab -l
30 04 * * * python /home/osboxes/git/guardians---cids-with-blockchain/scripts/update_snort_rules.py & systemctl restart snortd
[osboxes@osboxes scripts]$
```

Ilustración 57 - Prueba update_snort_rules.py 3

6.2.5 Añadir nueva regla mediante script

El script './scripts/add_rule.py' es la herramienta que proporciona este software para añadir reglas rápidamente a la Blockchain, mediante la línea de comandos. Al ejecutarlo solicitará la IP el nodo, su localización y la regla a compartir.

```
Javiers-MacBook-Pro:scripts mr.$ python3 add_rule.py
Introduce your IP connected to blockchain: 192.168.56.1
Introduce your location: Seville
Add new rule to pool: alert tcp $HOME_NET 20432 -> $EXTERNAL_NET any (msg:"DDOS shaft client login to handler"; flow:from_server,established; co
ntent:"login|3A|"; reference:arachnids,254; reference:url,security.royans.net/info/posts/bugtraq_ddos3.shtml; classtype:attempted-dos; sid:230;
rev:5;)
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages/urllib3/connectionpool.py:851: InsecureRequestWarning: Unverified
HTTPS request is being made. Adding certificate verification is strongly advised. See: https://urllib3.readthedocs.io/en/latest/advanced-usage.h
tml#ssl-warnings
  InsecureRequestWarning)
<Response [200]>
200
Transaction added to unconfirmed transactions.
Sending request to mine the block.
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages/urllib3/connectionpool.py:851: InsecureRequestWarning: Unverified
HTTPS request is being made. Adding certificate verification is strongly advised. See: https://urllib3.readthedocs.io/en/latest/advanced-usage.h
tml#ssl-warnings
  InsecureRequestWarning)
The block has been mined.
Writing new rule into blockchain.rules temporarily, until update script will run.
Done!
Javiers-MacBook-Pro:scripts mr.$
```

Ilustración 58 - Prueba add_rule.py 1

El script realizará dos peticiones locales a "/new_transaction", y si todo ha ido bien a "/mine", esta se encargará de enviar las solicitudes "/add_block" a los otros nodos. Además, la regla será escrita temporalmente en "blockchain.rules" en el nodo origen de la regla, hasta que se produzca la actualización de reglas.

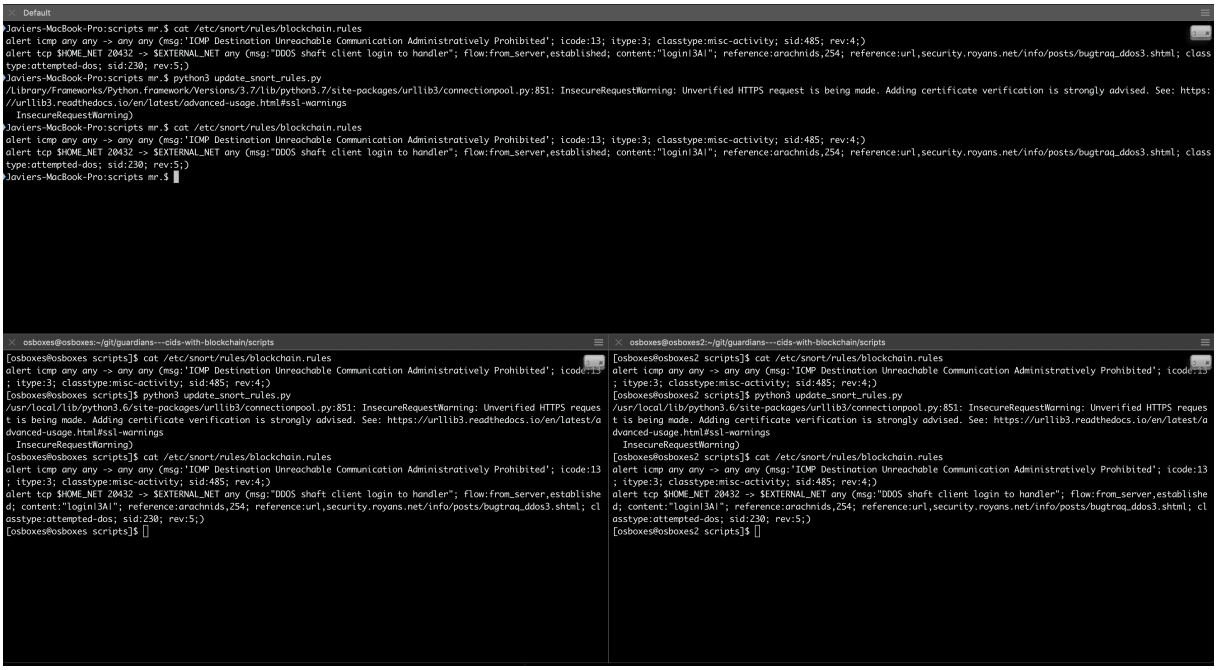

Ilustración 59 - Prueba add_rule.py 2

Los otros nodos no poseerán la regla escrita en su archivo de reglas hasta que se produzca la actualización de reglas, ya sea de forma automática o manual.

6.2.6 Añadir nuevo nodo mediante llamada HTTPS

Con esta prueba se pretende validar la acción de añadir un nuevo nodo a la red Blockchain.

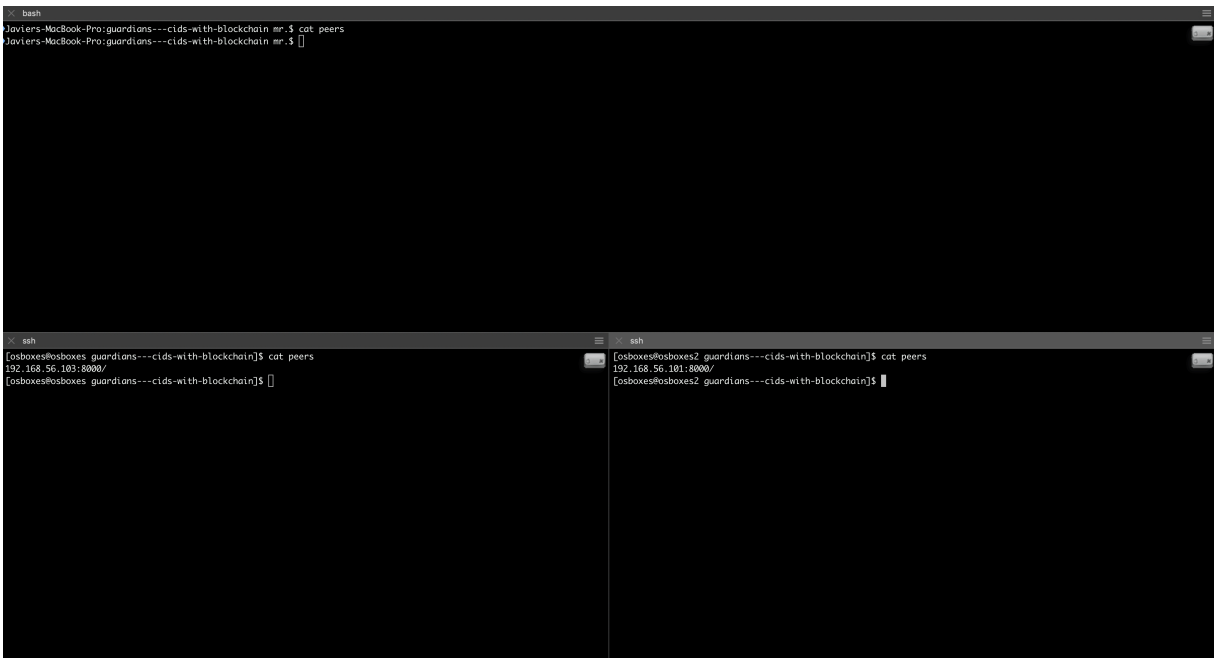

Ilustración 60 - Prueba join_in_blockchain.py 1

El nodo 1 (pantalla superior), será el elegido para ser añadido a la red Blockchain. En la imagen se aprecia como la lista de peers del nodo 1 está vacía, mientras que el nodo 2 y 3, se tienen mutuamente en la lista de nodos.

Ilustración 61 - Prueba join_in_blockchain.py 2

Se comprueba también que el nodo 1 tiene una cadena distinta a la de los nodos 2 y 3.

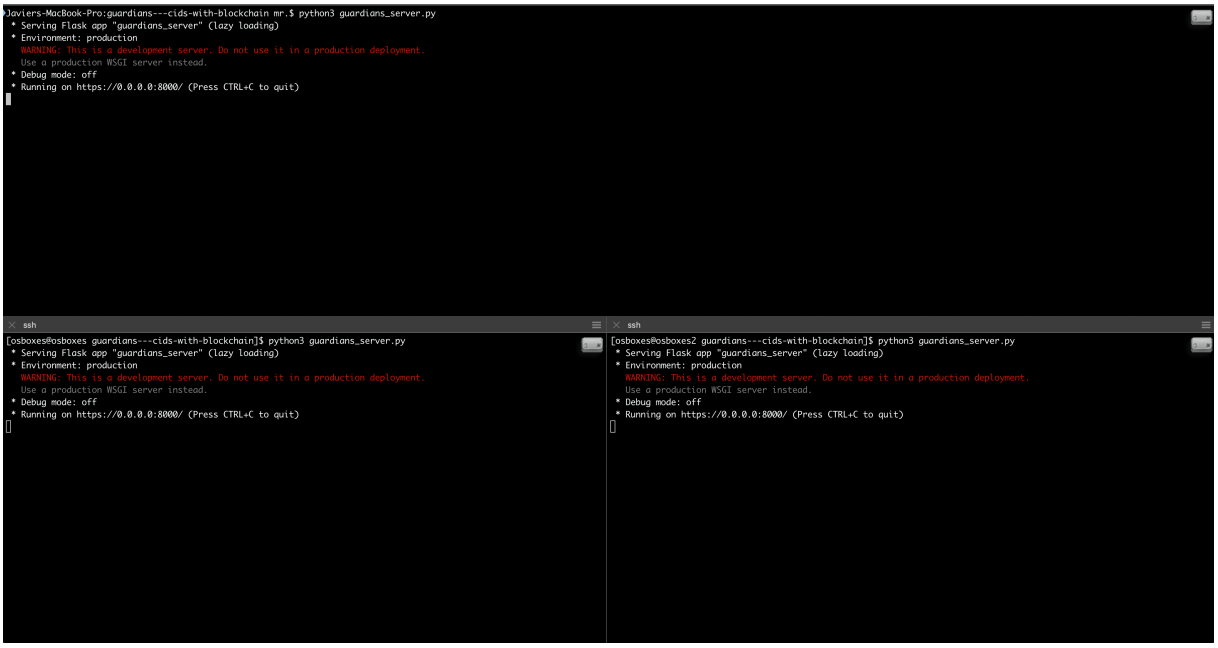

Ilustración 62 - Prueba join_in_blockchain.py 3

Se inicia el servidor Blockchain mediante el comando “python3 guardians_server.py”, en cada uno de los nodos.

El siguiente paso es realizar la llamada a https://127.0.0.1:8000/register_with desde el nodo 1. Este paso se puede realizar tanto mediante una llamada manual, con alguna herramienta como Postman.

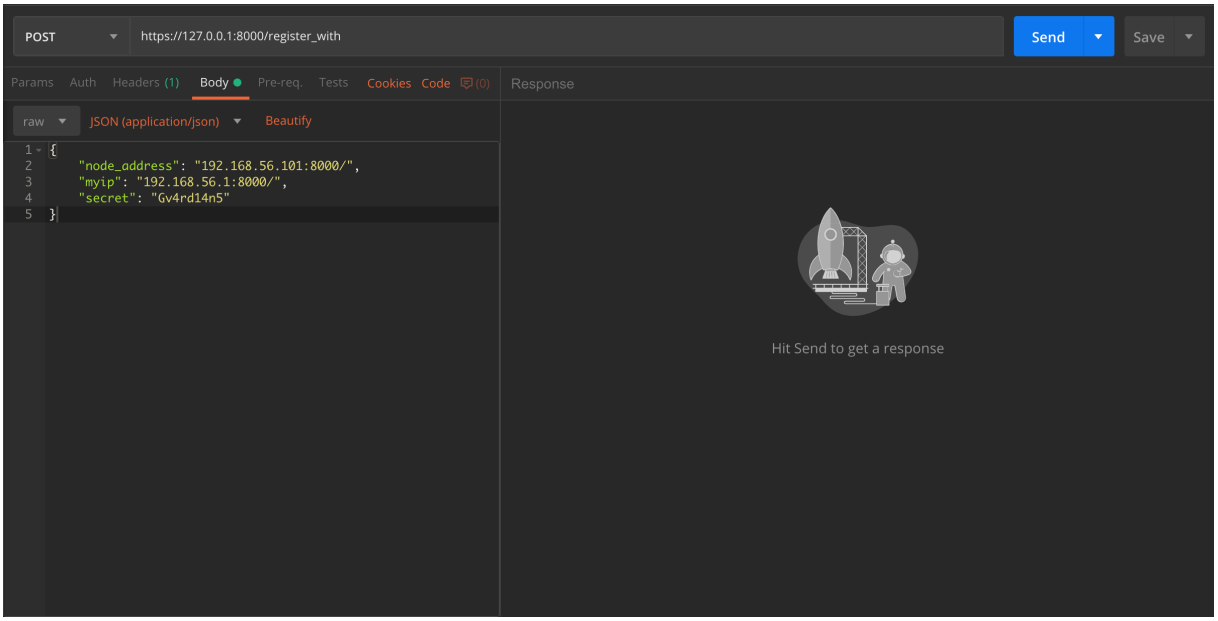

Ilustración 63 - Prueba join_in_blockchain.py 4

Como mediante la ejecución del script “join_in_blockchain.py”.

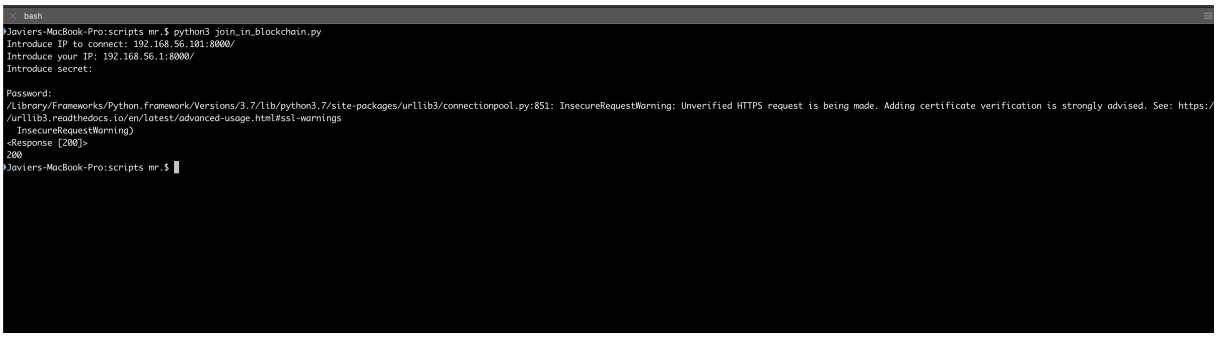

Ilustración 64 - Prueba join_in_blockchain.py 5

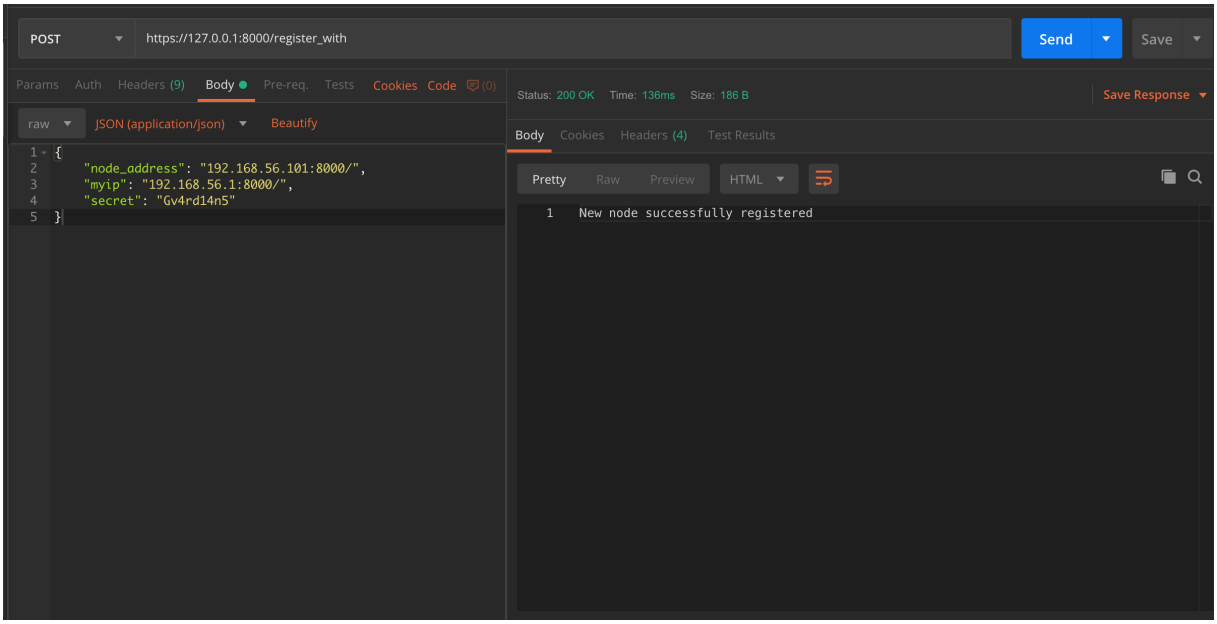

Ilustración 65 - Prueba join_in_blockchain.py 6

Se puede apreciar en las consolas como se han realizado las distintas llamadas POST necesarias entre ellos.

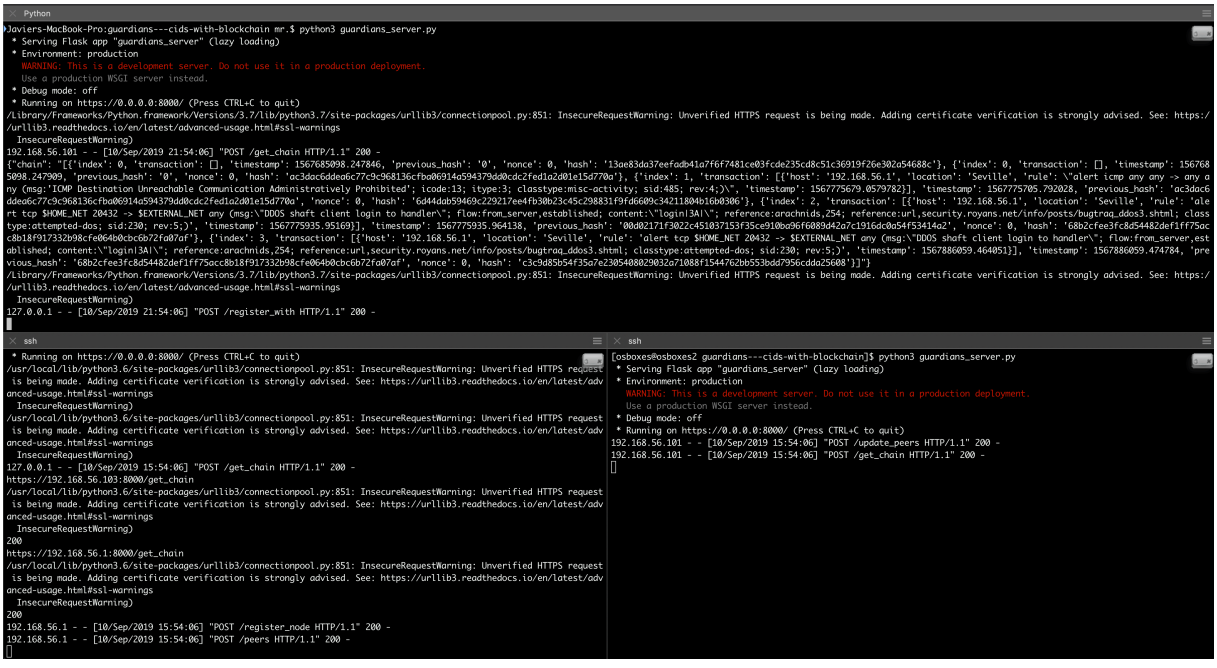

Ilustración 66 - Prueba join_in_blockchain.py 7

Al comprobar otra vez la lista de peers se ve que se han actualizado correctamente.

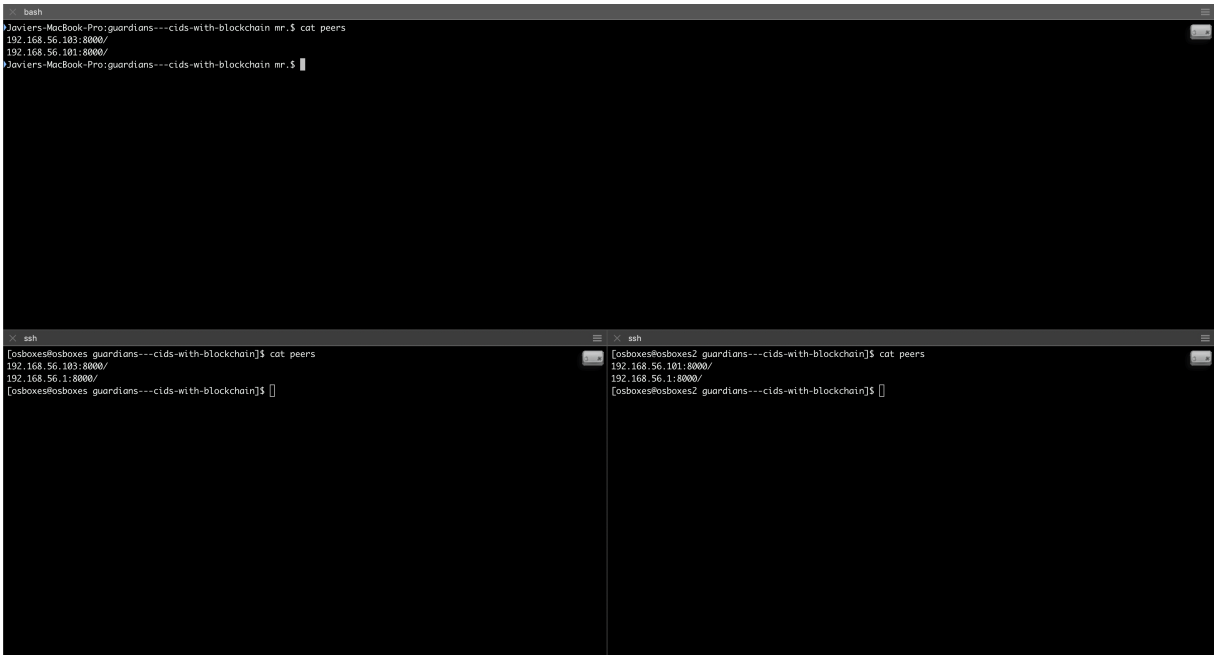

Ilustración 67 - Prueba join_in_blockchain.py 8

Al igual que la chain en el nodo 1.

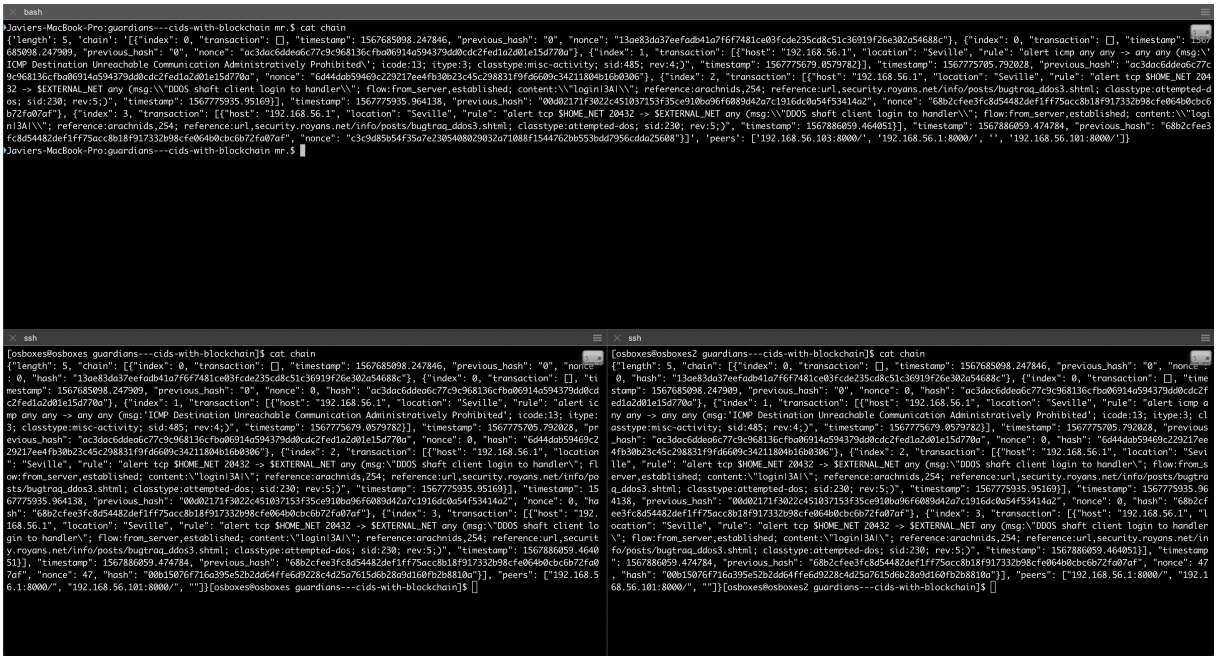

Ilustración 68 - Prueba join_in_blockchain.py 9

6.2.7 Conclusiones de las pruebas

Las pruebas realizadas han tenido un resultado satisfactorio, cumpliendo los objetivos que se querían alcanzar, demostrando que esta herramienta es una base o un producto mínimo viable para el desarrollo de una herramienta más funcional que se pueda implantar en sistemas reales en productivo. Esto es así porque la herramienta permite el envío de información segura, garantizando la integridad de la misma mediante los hashes de la chain, la trazabilidad de las transacciones mediante el campo host y los logs de las transacciones, la confidencialidad de qué información compartir, ya que solo se comparten las reglas del archivo “blockchain.rules”, y la disponibilidad de la misma mediante la descentralización de la información y de la red.

7. Conclusiones y trabajo futuro

7.1 Conclusiones

Esta investigación de fin de master fue direccionada por un objetivo general: desarrollar un mínimo producto viable de una herramienta CIDS con sus comunicaciones basadas en Blockchain. Además de definir una o varias arquitecturas de red posibles.

Este objetivo surge de la necesidad de fortalecer la seguridad de las distintas organizaciones, tanto de ámbito privado como público debido a la cada vez mayor ciberdelincuencia en la red. Se ha comprobado que ya no es suficiente con poseer un firewall para estar seguros, hacen falta muchas más medidas de seguridad, entre ellas los detectores de intrusos o IDS, que son capaces de detectar comportamientos anómalos en la red y bloquearlos si están configurados para ello. Y a partir de estos surgen los CIDS, que se trata de un sistema que se compone de varios IDS que se comunican entre sí. Estos sistemas CIDS requieren de ciertas características, mencionadas en este estudio, que ayudan a cumplir las comunicaciones en las redes Blockchain.

De la necesidad de estas herramientas y de la carencia de software Open Source, que cualquier organismo pueda implementar, surge este proyecto que incluye varios objetivos específicos que se desglosan, en concreto, en dos principales y en cuatro secundarios.

El primer objetivo principal: describir al menos una arquitectura del sistema CIDS. Esto se cumple en el capítulo de diseño, en concreto en el punto 4.2. En el cual se definen dos posibles arquitecturas del sistema. La primera define una arquitectura para una sola localización y la segunda, una arquitectura para varias sedes en distintas localizaciones.

El segundo objetivo principal: desarrollar un mínimo producto viable del sistema CIDS. Esto se logra en el código adjunto en el anexo 1 de este trabajo. El funcionamiento del código se expone en el capítulo de implementación, y en el capítulo de evaluación se comprueba su correcto funcionamiento. Dado que la herramienta permite interconectar varios sistemas IDS Snort, a través de un sistema Blockchain, que se usa para compartir la información de forma segura, garantizando la privacidad, la confidencialidad, la integridad y la disponibilidad de la misma. Cada nodo elige qué información comparte con la Blockchain, y actualiza las reglas que se comparten en dicha red en el momento que se elija, a través de tareas cron. Además,

el sistema permite añadir nuevos nodos mediante una petición HTTPS, incluyendo una contraseña que solo poseerán los nodos de la red Blockchain.

Los cuatro objetivos secundarios se cumplen a lo largo de este trabajo de la siguiente forma. El primero de estos objetivos “Verificar la herramienta mediante pruebas y documentar el resultado de las mismas”, se cumple en el capítulo de evaluación. El segundo, “Analizar y documentar el funcionamiento de la herramienta”, se cumple en el capítulo de implementación, el tercero “Sintetizar un pequeño manual de usuario”, se cumple en el anexo 2 de este documento y el último de estos objetivos “Describir las ventajas de esta herramienta sobre otros sistemas CIDS”, se expondrá en el párrafo final de este apartado.

Además, los distintos requisitos expuestos en el capítulo de diseño se relacionan con las porciones de código que las satisfacen a lo largo del capítulo de implementación. A excepción del requisito “R-09: El sistema deberá usar la filosofía del mínimo coste posible, tanto para las comunicaciones como para la computación.”, que no se cumple completamente, ya que se podrían almacenar solo los hashes de las transacciones en la chain para así ahorrar espacio.

Las principales ventajas que tiene este sistema CIDS sobre otras herramientas del mercado son dos: la primera, se trata de una herramienta Open Source (dado que el código de este trabajo se publicará en GitHub de forma libre para su uso) que cualquier organización pueda usar, y/o modificar esta herramienta para mejorar la seguridad de su arquitectura de red. La segunda: esta herramienta realiza sus comunicaciones mediante una red Blockchain, lo que dota de ciertas características de seguridad expuestas en este trabajo y de necesidad para este tipo de sistemas.

Esta investigación nutre a la comunidad de una base para el estudio y desarrollo de sistemas colaborativos de detección de intrusos, usando Blockchain como medio de comunicación. Debido a la falta de herramientas CIDS Open Source, este trabajo también pretende suplir este vacío y sirva de base y de guía para para que de la comunidad puedan surgir nuevas herramientas de uso libre aptas para su uso en entornos productivos.

7.2 Futuras líneas de trabajo

Las futuras líneas de trabajo que se proponen en base al TFM, se basan en cómo mejorar esta herramienta para poder ser usada en entornos en producción. Serían las siguientes:

- Creación de una interfaz gráfica implementada con Flask, que mejore la usabilidad de la herramienta.
- Securizar las conexiones entre los nodos para que se realicen por pares de claves PKI, para así mejorar la seguridad del sistema.
- El algoritmo de consenso usado en esta herramienta, es un algoritmo muy básico y permite margen de mejora. Con esto se conseguirá un mejor consenso y seguridad de la información en el sistema.
- Añadir el uso de base de datos para mejorar la accesibilidad y la persistencia de los datos intercambiados en la Blockchain.
- Automatización de creación de reglas según eventos que concuerden con reglas de comportamientos anómalos.
- Sanetización de la información que es introducida en las peticiones HTTPS. Esto mejorar la seguridad del sistema.
- Correr la aplicación mediante un servicio, almacenando sus logs.

Esta herramienta actualmente se debe usar en entornos de desarrollo para comprobar su correcto funcionamiento, e implementar mejoras a la misma antes de usarla en sistemas en productivo.

8. Bibliografía

[1] CCN-CERT, “CCN-CERT IA-09/18,” *https://www.ccn-cert.cni.es/*, 2018.

[2] W. Pandini, “Sep IDS: Historia, concepto y terminología,” *https://ostec.blog/es/seguridad-perimetral/ids-conceptos*, 2019.

[3] D. Shinder, “SolutionBase: Understanding how an intrusion detection system (IDS) works,” *techrepublic.com*, 2005.

[4] M. Rouse, “Unified threat management devices: Understanding UTM and its vendors,” *searchsecurity.techtarget.com/definition/intrusion-detection-system*, 2014.

[5] Techopedia, “Intrusion Prevention System (IPS),” *techopedia.com*, 2019.

[6] N. Alexopoulos, E. Vasilomanolakis, N. R. Ivánkó, and M. Mühlhäuser, “Towards blockchain-based collaborative intrusion detection systems,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 10707 LNCS, pp. 107–118, 2018.

[7] W. Meng, E. W. Tischhauser, Q. Wang, Y. Wang, and J. Han, “When intrusion detection meets blockchain technology: A review,” *IEEE Access*, vol. 6, pp. 10179–10188, 2018.

[8] M. Signorini, M. Pontecorvi, W. Kanoun, and R. Di Pietro, “BAD: Blockchain Anomaly Detection,” 2018.

[9] C. Severance, “Python para informáticos,” *Creat. Common Attrib. NonCommercial-ShareAlike 3.0 Unported*, 2009.

[10] G. van Rossum, “Python documentation,” *https://docs.python.org*, 2019.

[11] The Pallets Project, “Flask micro python frmework,” *https://palletsprojects.com/p/flask/*, 2019.

[12] M. Roesch, “Snort,” *www.snort.org*, 2019.

[13] J. Llopis Polvoreda, “Sistema de monitorización IDS Snort,” *Univ. Politécnica Val.*, 2017.

[14] V. Zaporozhets, Dmitriy y Sizov, “Gitlab,” *https://gitlab.com/*, 2019.

[15] Jet Brain, “Pycharm,” *https://www.jetbrains.com/pycharm/*, 2019.

[16] Atlassian, “Trello,” *https://trello.com/*, 2019.

[17] Postman, “Postman,” *https://www.getpostman.com/*, 2019.

[18] Postman, “Postman Learning Center,” *https://learning.getpostman.com/concepts/*, 2019.

9. Anexos

9.1 Anexo 1: Código de la aplicación

A continuación, se mostrará el Código de la aplicación por archivos.

9.1.1 guardians_server.py

```
import json
import os
import requests
import time
from hashlib import sha256
import shutil

from flask import Flask, request


class Block:
    def __init__(self, index, transaction, timestamp, previous_hash,
nonce=0):
        self.index = index
        self.transaction = transaction
        self.timestamp = timestamp
        self.previous_hash = previous_hash
        self.nonce = nonce

    def compute_hash(self):
        block_string = json.dumps(self.__dict__, sort_keys=True)
        return sha256(block_string.encode()).hexdigest()


class Blockchain:

    difficulty = 2

    def __init__(self):
        self.unconfirmed_transactions = 3
        self.chain = []
        self.create_genesis_block()

    def create_genesis_block(self):
        data = []
        chain_blockchain = []
```

```
        if os.path.isfile('./chain'):
            with open('./chain', 'r') as file:
                data = file.read()
            chain = data.split('\n')
            if len(chain) > 0:
                for bck in chain:
                    chn = eval(bck)
                    if isinstance(chn['chain'], str):
                        for dic in eval(chn['chain']):
                            block = Block(dic['index'],
dic['transaction'], dic['timestamp'], dic['previous_hash'])
                            block.hash = block.compute_hash()
                            chain_blockchain.append(block)
                    else:
                        for dic in chn['chain']:
                            block = Block(dic['index'],
dic['transaction'], dic['timestamp'], dic['previous_hash'])
                            block.hash = block.compute_hash()
                            chain_blockchain.append(block)
                self.chain = chain_blockchain
            else:
                genesis_block = Block(0, [], time.time(), "0")
                genesis_block.hash = genesis_block.compute_hash()
                self.chain.append(genesis_block)
        else:
            genesis_block = Block(0, [], time.time(), "0")
            genesis_block.hash = genesis_block.compute_hash()
            self.chain.append(genesis_block)

    @property
    def last_block(self):
        return self.chain[-1]

    def add_block(self, block, proof):

        previous_hash = self.last_block.hash

        if previous_hash != block.previous_hash:
            return False

        if not Blockchain.is_valid_proof(block, proof):
            return False

        block.hash = proof
        self.chain.append(block)
        try:
            with open('./chain', 'w') as file:
```

```
                file.write(get_my_chain())
        except:
            print("Cannot write in the chain's file")

        return True

    def proof_of_work(self, block):

        block.nonce = 0

        compute_hash = block.compute_hash()
        while not compute_hash.startswith('0' *
Blockchain.difficulty):
            block.nonce += 1
            compute_hash = block.compute_hash()

        return compute_hash

    def add_new_transaction(self, transaction):
        self.unconfirmed_transactions.append(transaction)

    @classmethod
    def is_valid_proof(cls, block, block_hash):

        b = Blockchain()
        return (block_hash.startswith('0' * Blockchain.difficulty)
and
                block_hash == b.proof_of_work(block))

    @classmethod
    def check_chain_validity(cls, chain):
        result = True
        previous_hash = "0"

        for block in chain:
            block_hash = block.hash

            delattr(block, "hash")

            if not cls.is_valid_proof(block, block.hash) or \
                    previous_hash != block.previous_hash:
                result = False
            if result:
                block.hash, previous_hash = block_hash, block_hash

            return result
```

```
    def mine(self):

        if not self.unconfirmed_transactions:
            return False

        last_block = self.last_block

        new_block = Block(index=last_block.index + 1,
                          transaction=self.unconfirmed_transactions,
                          timestamp=time.time(),
                          previous_hash=last_block.hash)
        proof = self.proof_of_work(new_block)
        added = self.add_block(new_block, proof)

        if added:
            self.unconfirmed_transactions = []
            announce_new_block(new_block)
            return new_block.index
        else:
            return False


app = Flask(__name__)

blockchain = Blockchain()
blockchain.create_genesis_block()


@app.route('/new_transaction', methods=['POST'])
def new_transaction():
    data = request.get_json()
    required_fields = ["rule", "location", "host"]

    for field in required_fields:
        if not data.get(field):
            return "Invalid transaction data", 404

    data["timestamp"] = time.time()
    blockchain.add_new_transaction(data)

    return "Success", 200


@app.route('/mine', methods=['POST'])
def mine_unconfirmed_transactions():
    result = blockchain.mine()
    if not result:
```

```
        return "No transactions to mine", 500
    return "Block #{} is mined.".format(result), 200

@app.route('/peers', methods=['POST'])
def get_peers():
    with open('./peers', 'r') as file:
        data = file.read()
    peers = data.split('\n')
    return json.dumps({'peers': peers})

@app.route('/update_peers', methods=['POST'])
def add_peer():
    new_peer = request.get_json()['peer']
    with open('./peers', 'a') as file:
        file.write(new_peer+'\n')
    return "Peer: {} added".format(new_peer)


# THIS FUNCTION IS CALL INTERNALLY
@app.route('/register_with', methods=['POST'])
def register_with_existing_node():

    node_address = request.get_json()["node_address"]
    myip = request.get_json()["myip"]
    secret =
sha256(request.get_json()["secret"].encode()).hexdigest()

    if not node_address or not secret:
        return "Invalid data", 400

    data = {"host_origin": myip, "secret": secret}
    headers = {"Content-Type": "application/json"}

    # Make a request to register with remote node and obtain
information
    response = requests.post("https://" + node_address +
"register_node",
                             data=json.dumps(data), headers=headers,
verify=False)

    if response.status_code == 200:
        print(response.text)
        global blockchain
        chain_dump = response.json()['chain']

        response2 = requests.post("https://" + node_address +
"peers", verify=False)
```

```
        peers = response2.json()['peers']
        peers.append(node_address)
        file = open('./peers','w+')
        for p in peers:
            if p:
                file.write(p+'\r\n')
        file.close()
        blockchain = create_chain_from_dump(chain_dump, peers)
        return "New node successfully registered", 200
    else:
        return response.content, response.status_code


# THIS FUNCTION IS CALL EXTERNALLY
@app.route('/register_node', methods=['POST'])
def register_new_peers():
    headers = {"Content-Type": "application/json"}
    node_address = request.get_json()["host_origin"]
    secret = request.get_json()["secret"]
    if not node_address or not secret:
        return "Invalid data", 400

    with open('./secret', 'r') as file:
        code = file.read().replace('\n', '')
    with open('./peers', 'r') as file:
        data_peers = file.read()
    peers = data_peers.split('\n')
    for p in peers:
        if p:
            requests.post("https://" + p + "update_peers",
                          data=json.dumps({'peer': node_address}),
headers=headers, verify=False)
    if secret == sha256(code.encode()).hexdigest():
        if node_address not in peers:
            with open('./peers', 'a') as file:
                file.write(node_address+'\n')
            chain = get_longest_chain()
            if chain:
                return json.dumps({"chain": str(chain)}), 200
            else:
                return "Error adding node", 500
        else:
            return "Node already in the peers's list", 200
    else:
        return "Error adding node", 500

#Consensus
```

```
def get_longest_chain():

    global blockchain
    my_chain = requests.post('https://127.0.0.1:8000/get_chain',
verify=False).json()
    current_len = my_chain['length']
    longest_chain = my_chain['chain']

    with open('./peers', 'r') as file:
        data_peers = file.read()
    peers = data_peers.split('\n')

    for node in peers:
        if node:
            print('https://{}get_chain'.format(node))
            response =
requests.post('https://{}get_chain'.format(node), verify=False)
            if response:
                print(response.status_code)
                length = response.json()["length"]
                chain = response.json()["chain"]
                if length > current_len and
blockchain.check_chain_validity(chain):
                    current_len = length
                    longest_chain = chain

    if longest_chain:
        return longest_chain
    else:
        return False


def create_chain_from_dump(chain_dump, peers):

    chain_dump = chain_dump[1:]
    chain_dump = chain_dump[:-1]
    array = chain_dump.split('},')

    array_dic = []
    count = 0
    length = len(array)

    for a in array:
        if count != length - 1:
            array_dic.append(json.dumps(a + '}'))
            count += 1
        else:
```

```
            array_dic.append(json.dumps(a))

    chain_blockchain = []
    blockchain = Blockchain()
    for block_data in array_dic:
        block_d = eval(json.loads(block_data.strip()))
        block = Block(block_d["index"],
                      block_d["transaction"],
                      block_d["timestamp"],
                      block_d["previous_hash"],
                      block_d['hash'])
        chain_blockchain.append(block)
    blockchain.chain = chain_blockchain

    file_chain = []
    for c in chain_blockchain:
        file_chain.append(c.__dict__)
    dic = {"length": len(file_chain), "chain":
json.dumps(file_chain), "peers": peers}
    file = open('./chain', 'w+')
    file.write(str(dic))
    file.close()

    return blockchain


@app.route('/add_block', methods=['POST'])
def verify_and_add_block():
    block_data = request.get_json()
    block = Block(block_data["index"],
                  block_data["transaction"],
                  block_data["timestamp"],
                  block_data["previous_hash"])
    proof = block_data['hash']
    added = blockchain.add_block(block, proof)

    if not added:
        return "The block was discarded by the node", 400

    return "Block added to the chain", 201


@app.route('/pending_tx')
def get_pending_tx():
    return json.dumps(blockchain.unconfirmed_transactions)
```

```
@app.route('/get_chain', methods=['POST'])
def get_my_chain():
    chain_data = []
    for block in blockchain.chain:
        chain_data.append(block.__dict__)
    with open('./peers', 'r') as file:
        data_peers = file.read()
    peers = data_peers.split('\n')
    return json.dumps({"length": len(chain_data),
                       "chain": chain_data,
                       "peers": peers})



@app.route('/update_chain', methods=['POST'])
def update_chain():
    global blockchain
    chain = request.get_json()["chain"]
    if blockchain.check_chain_validity(chain):
        blockchain.chain = chain
        return 200
    else:
        return "No valid chain", 500


def announce_new_block(block):

    headers = {"Content-Type": "application/json"}
    with open('./peers', 'r') as file:
        data_peers = file.read()
    peers = data_peers.split('\n')
    for peer in peers:
        if peer:
            requests.post("https://"+peer+"add_block",
headers=headers, data=json.dumps(block.__dict__, sort_keys=True),
verify=False)


# --------------CONNECTOR IDS <-> BLOCKCHAIN--------------
@app.route('/update_snort_rules', methods=['POST'])
def update_snort_rules():
    if request.remote_addr == "127.0.0.1":
        global blockchain
        rules_data = []
        for block in blockchain.chain:
            rules_data.append(block.transaction)
```

```
        shutil.copy2('/etc/snort/rules/blockchain.rules',
'/etc/snort/rules/blockchain.backup')
        os.remove('/etc/snort/rules/blockchain.rules')
        file = open('/etc/snort/rules/blockchain.rules', 'w+')

        for r in rules_data:
            if r:
                file.write(r[0]['rule']+"\r\n")
        file.close()

        return 'Update was succesfully done', 200
    else:
        return 'Only local request will attend.', 500

# ------------------------------------------------------

# -------------------RUN APPLICATION-------------------
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8000,
ssl_context=('./certificate/cert.pem', './certificate/key.pem'))
```

9.1.2 add_rule.py

```
import requests
import json

host = input('Introduce your IP connected to blockchain: ')
location = input('Introduce your location: ')
rule = input('Add new rule to pool: ')

data = {"host": host, "location": location, "rule": rule}
headers = {"Content-Type": "application/json"}

response_nt =
requests.post('https://127.0.0.1:8000/new_transaction',
                              data=json.dumps(data), headers=headers,
verify=False)
print(response_nt)
print(response_nt.status_code)

if response_nt.status_code == 200:
    print('Transaction added to unconfirmed transactions.')
    print('Sending request to mine the block.')
    response_mine = requests.post('https://127.0.0.1:8000/mine',
verify=False)

    if response_mine.status_code == 200:
```

```
        print('The block has been mined.')
        print('Writing new rule into blockchain.rules temporarily,
until update script will run.')
        try:
            with open('/etc/snort/rules/blockchain.rules', 'a') as
file:
                file.write(rule)
        except:
            print('Error writing temporary rules.')

        print('Done!')
else:
    print(response_nt.status_code)
    print('The transaction has not been accepted')
```

9.1.3 update_snort_rules.py

```
import requests
from datetime import datetime
import os.path

if not os.path.isfile('update_snort_rules.log'):
    file = open('./update_snort_rules.log', 'w+')
    file.close()

response =
requests.post('https://127.0.0.1:8000/update_snort_rules',
verify=False)
if response.status_code == 200:
    time = datetime.now()
    with open('./update_snort_rules.log', 'a') as file:
        file.write(time.strftime("%d/%m/%Y %H:%M:%S") + ' 200 -
Update was successfully done.\r\n')
else:
    time = datetime.now()
    with open('./update_snort_rules.log', 'a') as file:
        file.write(time.strftime("%d/%m/%Y %H:%M:%S") + " " +
str(response.status_code) + ' - ERROR update was not done.\r\n')


# To automate this script add the following line to your crontab.
# This task will throw this script everyday at 04:30
# 30 04 * * * python $ROUTE_SCRIPT & systemctl restart snortd
```

9.1.4 join_in_blockchain.py

```
import requests
import json
from getpass import getpass

host = input('Introduce IP to connect: ')
myip = input('Introduce your IP: ')
print('Introduce secret: \n')
password = getpass()

data = {"node_address": host, "myip": myip, "secret": password}
headers = {"Content-Type": "application/json"}

response_nt = requests.post('https://127.0.0.1:8000/register_with',
                            data=json.dumps(data), headers=headers,
verify=False)
print(response_nt)
print(response_nt.status_code)
```

9.2 Anexo 2: Manual de usuario

Antes de iniciar los distintos nodos de la Blockchain, se debe realizar las siguientes configuraciones:

- Establecer un passphrase común en el archivo ‘./secret’.
- Añadir la lista de peers al archivo ‘./peers’ sin incluir la ip del propio nodo.
- Generar certificados TLS propios en cada uno de los nodos.
- Crear un primer bloque común a todos los nodos de la red.
- Tener instalado Snort configurado como NIDS y corriendo como servicio.
- Crear el archivo “/etc/snort/rules/blockchain.rules” y concederle los permisos necesarios.

Una vez realizadas estas configuraciones, para iniciar la aplicación se debe iniciar una shell dentro del directorio “./guardians—cids-with-blockchain” y ejecutar el siguiente comando.

```
python3 guardians_server.py
```

Una vez ejecutado, la red está lista para empezar a compartir información. Para crear nuevas reglas se puede hacer uso del script “./scripts/add_rule.py” que solicitará distintos parámetros y enviara la información a través de la blockchain.

Para actualizar las reglas de Snort se puede hacer uso del script “./scripts/update_snort_rules.py”, que actualizará el archivo “/etc/snort/rules/blockchain.rules” con las reglas que contenga la cadena de la Blockchain.

Para añadir un nuevo nodo a la Blockchain y que se sincronice con la misma, se puede hacer uso del script “./scripts/join_in_blockchain.py”, que solicitará distintos parámetros de conexión, entre ellos el passphrase para incorporarse a la Blockchain.

Todo lo anterior se puede realizar también mediante llamadas HTTPS con cualquier herramienta que permita la creación de este tipo de llamadas.