Universidad Nacional del Sur

Proyecto Final de Carrera
Ingeniería en Computación

Seguridad en redes LAN: utilizando Docker para mejorar la navegación web.

Salvador Catalfamo

Bahía Blanca – Argentina
2021

Agradecimientos

Al concluir una etapa maravillosa quiero extender un profundo agradecimiento a mis formadores y directores de proyecto Alejandro y Leonardo, personas de gran sabiduría y conocimiento quienes se han esforzado por ayudarme a llegar al punto en el que me encuentro.

Mi gratitud hacia mi familia, en especial a mis padres y hermanos, por haberme sostenido y apoyado en cada paso de esta carrera con amor, comprensión y alegría.

Y para finalizar, agradezco a todos los que fueron mis compañeros de clase durante este camino universitario, ya que gracias al compañerismo y amistad han aportado en un alto porcentaje a mis ganas de seguir adelante en mi carrera profesional.

Resumen

¿Qué información circula dentro de una organización cuando utilizamos los sistemas internos de la misma? ¿Se toman las medidas de seguridad suficientes para proveernos confidencialidad en las comunicaciones? ¿Cuándo podemos afirmar que estamos utilizando sistemas seguros? Las respuestas a estas preguntas estarán explicadas en este proyecto.

Se realizó un trabajo de investigación teórico/práctico donde se muestran las consecuencias que tiene utilizar sistemas que no implementan mecanismos de seguridad, además, se desarrollaron posibles soluciones a las problemáticas anteriormente mencionadas. En varios casos, se montaron escenarios de pruebas para un mejor entendimiento del lector.

Palabras Clave:

Seguridad e Infraestructura - Navegación segura - Redes Internas - Entidad Certificante CA - Protocolo SSL - PKI - GNU/Linux - Máquinas Virtuales - Kali - Docker

Índice general

Capítulo 1

Introducción

A lo largo de la carrera, y, particularmente, en una de mis materias preferidas, Seguridad en Sistemas, nos han explicado la importancia la protección de los datos, como así también la de los canales de comunicación. Al desplegar una red segura, se deben considerar los siguientes aspectos centrales de la seguridad en sistemas, tales como la confidencialidad, la integridad y la disponibilidad (CIA).

En nuestra corta experiencia hemos observado como las organizaciones abordan los temas de seguridad en sus sistemas informáticos y en las redes de datos . En general, se concentran en los equipos que están expuestos a la red pública, dejando de lado los que se encuentran aislados de la misma. Muchas veces se piensa que es suficiente, sin embargo, puede traer graves inconvenientes. Es por eso que realizaremos un estudio teórico/práctico sobre las consecuencias de la navegación en redes internas sin ningún tipo de cifrado de datos ni certificaciones, lo cual puede traer graves inconvenientes en cuanto a la seguridad interna. Entre las consecuencias podemos ejemplificar con sistemas internos que no proveen conexiones seguras y descuidan la autenticación confidencial de los usuarios.

1.1. Objetivos

Se plantearon principalmente los siguientes objetivos:

- Aplicar los conocimientos en seguridad en redes para desplegar aplicaciones de red.

- Aprender el uso de nuevas herramientas de administración, automatización y seguridad en sistemas.
- Montar un escenario virtual que sirva de pruebas frente a la gestión de certificados, administración de la infraestructura y la detección de debilidades dentro una red privada.
- Se desarrollarán medidas de seguridad en una red interna utilizando *Docker* como centro de pruebas.

1.2. Metodología de trabajo

El presente trabajo final de carrera se dividirá en tres etapas:

- Primera: Se mostrarán escenarios donde la navegación insegura puede traer problemas asociados dentro de la organización.
- Segunda: Se plantearán escenarios de trabajo buscando adquirir experiencia en la utilización de la herramienta *Docker*, para poder obtener parámetros que nos permitan realizar comparaciones con tecnologías similares y sus áreas de aplicación.
- Tercera: Se definirán posibles soluciones para afrontar los aspectos planteados en la primera etapa.

Capítulo 2

Navegación web en redes internas

La conexión en red es la tecnología clave para una amplia variedad de aplicaciones dentro de una organización, como por ejemplo: correo electrónico, transferencia de archivos, sistemas de intranet para gestión de tareas dentro de la organización (departamento de compras, relación con el cliente ERP, deparmentos de ventas, gestión de recursos humanos, gestión de seguridad industrial, dashboards, etc.). Sin embargo, existe una falta significativa en la aplicación de métodos de seguridad para estas aplicaciones.

2.1. Protocolo HTTP

El Protocolo de transferencia de hipertexto (*HTTP*) es un protocolo de solicitud/respuesta *stateless* (sin estado) que utiliza semántica extensible y cargas útiles de mensajes autodescriptivos para una interacción flexible con sistemas basados en red.

HTTP es un protocolo genérico para sistemas de información. Está diseñado para ocultar los detalles de cómo se implementa un servicio mostrando una interfaz a los clientes que es independiente de los tipos de recursos proporcionados. Del mismo modo, los servidores no necesitan conocer el propósito de cada cliente: una solicitud *HTTP* puede aislada en vez de estar asociada con un tipo específico de cliente o una secuencia predeterminada de pasos de la aplicación.

El resultado es un protocolo que se puede utilizar de forma eficaz en muchos contextos diferentes y cuyas implementaciones pueden evolucionar a lo largo del

tiempo. Una consecuencia de esta flexibilidad es que el protocolo no se puede definir en términos de lo que ocurre detrás de la interfaz: estamos limitados a definir la sintaxis de la comunicación, los mensajes y el comportamiento esperado de los destinatarios. Si la comunicación se considera de forma aislada, las acciones exitosas deben reflejarse correspondientemente. Sin embargo, dado que varios clientes pueden actuar en paralelo y quizás con propósitos cruzados, no podemos exigir que tales cambios sean observables más allá del alcance de una única respuesta.

2.1.1. Arquitectura

HTTP fue creado para la arquitectura World Wide Web (WWW) y ha evolucionado con el tiempo para soportar las necesidades de escalabilidad de un sistema mundial. Gran parte de esa arquitectura se refleja en la terminología y definiciones de sintaxis utilizadas para definir *HTTP*.

Mensajes Cliente/Servidor

HTTP es un protocolo de solicitud/respuesta que opera intercambiando mensajes a través de una "conexión" en la capa de sesión o transporte. En el contecto de este protocolo, un "cliente" es un programa que establece una conexión a un servidor con el propósito de enviar una o más solicitudes. Un "servidor" es un programa que acepta conexiones para atender solicitudes mediante el envío de respuestas. La mayoría de las comunicaciones *HTTP* consisten en una solicitud (GET) de algún recurso identificado por un *URI*. En el caso más simple, esto podría lograrse mediante una única conexión entre un usuario y el servidor.

Un cliente envía una solicitud *HTTP* a un servidor en forma de mensaje de solicitud, comenzando con una línea que incluye un método, *URI* y versión del protocolo, seguida de campos de encabezado que contienen modificadores de solicitud, información del cliente y metadatos de representación, una línea vacía para indicar el final de la sección del encabezado, y finalmente un cuerpo del mensaje que contiene el cuerpo de la carga útil (si lo hay).

Un servidor responde a la solicitud de un cliente enviando uno o más mensajes de respuesta *HTTP*, cada uno de los cuales comienza con una línea de estado que

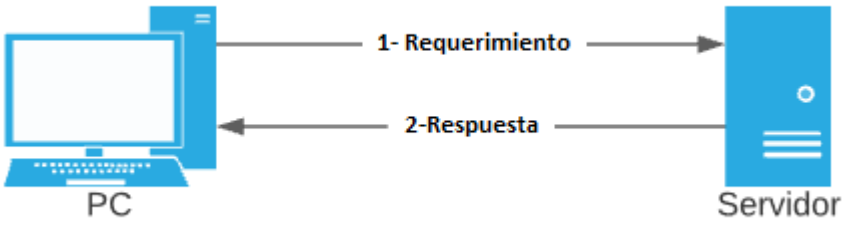


Figura 2.1: Comunicación básica en HTTP

incluye la versión del protocolo, un código de estado (éxito o error) y una descripción en forma de texto asociada al mismo, posiblemente seguida de campos de encabezado con información del servidor y metadatos de recursos, una línea vacía para indicar el final de la sección del encabezado y finalmente, un cuerpo del mensaje la carga útil del mismo

Ejemplo

El siguiente ejemplo ilustra un intercambio de mensajes típico para una solicitud GET a la dirección *cs.uns.edu.ar/index.php*:

Requerimiento del Cliente:

```
GET /index.php HTTP/1.1
User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
Host: www.cs.uns.edu.ar
Accept-Language: en, mi
```

Respuesta del Servidor:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
```

```
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain

Departamento de Ciencias e Ingeniería de la Computación -
            Universidad Nacional del Sur
```

2.1.2. Métodos más importantes del protocolo HTTP

El protocolo *HTTP* contiene varios métodos, como por ejemplo PUT, HEAD, DELETE, etc. Sin embargo, para nuestro trabajo explicaremos los dos más utilizados GET y POST, lo que nos permitirá tener una base a la hora de presentar el caso de estudio de la sección 3.5.

GET

El método GET solicita al servidor la transferencia de un recurso. GET es el mecanismo principal de recuperación de información y el foco de casi todas las optimizaciones de rendimiento. Por lo tanto, cuando las personas hablan de recuperar información identificable a través de *HTTP*, generalmente se refieren a realizar una solicitud GET.

Se puede pensar que a la hora de solicitar un recurso, este sea un archivo dentro de un directorio, y la respuesta sea el mismo archivo. Sin embargo, no existen tales limitaciones en la práctica. De hecho, se puede configurar un servidor para ejecutar o interpretar el contenido de esos archivos y enviar la salida en lugar de transferir los archivos directamente. Independientemente de la solicitud, el servidor solo necesita saber cómo tratar a cada uno de sus recursos.

POST

El método POST solicita que un recurso del servidor sea procesado con los datos que el cliente le envía. Por ejemplo, POST se utiliza para las siguientes funciones (entre otras):

- Proporcionar un bloque de datos, como los campos ingresados en un formulario HTML, a un proceso de manejo de datos.
- Publicar un mensaje grupo de noticias, lista de correo, *blog* o grupo similar de artículos.
- Crear un nuevo recurso que aún no ha sido identificado por el servidor.
- Agregar datos a las representaciones existentes de un recurso.

2.1.3. Códigos de respuesta

El código de estado es un número entero de tres dígitos que da el resultado del intento de comprender y satisfacer la solicitud. Los códigos de estado *HTTP* son extensibles. No se requiere que los clientes *HTTP* comprendan el significado de todos los códigos de estado registrados, aunque se espera una mínima comprensión.

Por ejemplo, si un cliente recibe un código de estado no reconocido de 471, el cliente puede asumir que hubo algo mal con su solicitud y tratar la respuesta como si hubiera recibido un código de estado 400 (Solicitud incorrecta). El mensaje de respuesta generalmente contendrá una representación que explica el estado.

El primer dígito del código de estado define la clase de respuesta. Los dos últimos dígitos no tienen ninguna función de categorización. Hay cinco valores para el primer dígito:

- Informativo (1xx): Se recibió la solicitud, se continúa procesando.
- Satisfactoria (2xx): La solicitud se recibió, comprendió y aceptó correctamente.
- Redireccionamiento (3xx): Se deben realizar más acciones para completar la solicitud.
- Error del cliente (4xx): La solicitud contiene una sintaxis incorrecta o no se puede cumplir.
- Error del servidor (5xx): El servidor no cumplió con una solicitud aparentemente válida.

2.1.4. Infraestructura de clave pública: una Introducción a la navegación segura

Gracias a la criptografía de clave pública, podemos comunicarnos de forma segura y confidencial con las personas cuyas claves públicas tengamos, pero hay una serie de

problemas que siguen sin resolverse. Por ejemplo, ¿cómo podemos comunicarnos con personas que nunca hemos conocido? ¿Cómo almacenamos las claves públicas y las revocamos? Más importante aún, ¿cómo lo hacemos a escala mundial, con millones de servidores y miles de millones de personas y dispositivos? Es una tarea difícil, pero para eso se creó la infraestructura de clave pública (PKI).

El objetivo de PKI es permitir una comunicación segura entre partes que nunca se han conocido antes. El modelo que usamos hoy se basa en "trusted third parties" llamados autoridades de certificación (*CA*, a veces también llamadas autoridades de certificación) para emitir certificados en los que confiamos siempre. Una infraestructura PKI está formado por los siguientes agentes.

Suscriptor

El suscriptor (o entidad final) es la parte que desea brindar servicios seguros, que requieren un certificado. Por ejemplo: una empresa que desea publicar su sitio web.

Autoridad de Registro

La autoridad de registro (RA) lleva a cabo determinadas funciones de gestión relacionadas con la emisión de certificados. Por ejemplo, una RA puede realizar la validación de identidad necesaria antes de solicitar a una *CA* que emita un certificado. En algunos casos, las RA también se denominan autoridades de registro local (LRA). En la práctica, muchas *CA* también realizan tareas de RA.

Autoridad de certificación

Una autoridad de certificación (*CA*) es un agente en el que confiamos para emitir certificados que confirman las identidades de los suscriptores. También están obligados a proporcionar información de revocación actualizada en línea para que las partes que confían puedan verificar que los certificados siguen siendo válidos.

Consumidor de certificados

Técnicamente, se trata de los navegadores web, de programas y de sistemas operativos que realizan la validación de certificados. Para ello, contienen almacenes que contienen los certificados de confianza de algunas *CA*. En un sentido más amplio,

los agentes que confían son los usuarios finales que dependen de certificados para una comunicación segura en Internet.

Certificados

Un certificado es un documento digital que contiene una clave pública, cierta información sobre la entidad asociada y una firma digital del emisor del certificado. En otras palabras, es una herramienta que nos permite intercambiar, almacenar y usar claves públicas. Con eso, los certificados se convierten en el componente básico de PKI.

Cadenas de certificados

En la mayoría de los casos, un certificado de una entidad final por sí solo es insuficiente para una validación exitosa. En la práctica, cada servidor debe proporcionar una cadena de certificados que conduzca a una raíz de confianza. Las cadenas de certificados se utilizan por motivos de seguridad, técnicos y administrativos.

Autoridades de certificación

Las autoridades de certificación (*CA*) son la parte más importante del modelo actual de confianza en Internet. Pueden emitir un certificado para cualquier nombre de dominio, lo que significa que todo lo que digan es válido. Durante mucho tiempo, el costo de los certificados era bastante elevado. Sin embargo, en estos días, el precio se redujo drásticamente, impulsado por una fuerte competencia, adicionando a varias organizaciones que proveen estos servicios de manera gratuita.

Ciclo de vida del certificado

El ciclo de vida del certificado comienza cuando un suscriptor prepara una Solicitud de firma de certificado (CSR) y la envía a la *CA* de su elección. El propósito principal del CSR es poner a disposición de la *CA* la clave pública, así como demostrar la propiedad de la clave privada correspondiente (mediante una firma).

Figura 2.2: Solicitud de un certificado

2.1.5. HTTP con Seguridad SSL (HTTPS)

Con una comprensión mínima de los conceptos la criptografía, podemos observar cómo funciona el protocolo *Secure Sockets Layer* (*SSL*). Aunque *SSL* no es un protocolo extremadamente complicado, ofrece varias opciones y variaciones.

El protocolo *SSL* consiste en un conjunto de mensajes y reglas sobre cuándo enviar (y no enviar) cada mensaje. En esta sección, mostraremos cuales son esos mensajes, la información general que contienen y cómo los sistemas usan los diferentes mensajes en una sesión de comunicaciones.

Roles SSL

El protocolo *Secure Sockets Layer* define dos roles diferentes para las partes que se comunican. Por un lado, tenemos un cliente, y por el otro un servidor. La distinción es muy importante, porque *SSL* requiere que los dos sistemas se comporten de manera muy diferente.

El cliente es el sistema que inicia las comunicaciones seguras; el servidor responde a la solicitud del cliente. En el uso más común de *SSL*, la navegación *web* segura, el navegador *web* es el cliente *SSL* y el sitio *web* es el servidor *SSL*. Para *SSL* en sí, las

distinciones más importantes entre clientes y servidores son sus acciones durante la negociación de los parámetros de seguridad.

Dado que el cliente inicia una comunicación, tiene la responsabilidad de proponer un conjunto de opciones *SSL* para usar en el intercambio. El servidor selecciona entre las opciones propuestas por el cliente y decide que utilizarán realmente los dos sistemas. Aunque la decisión final recae en el servidor, el servidor solo puede elegir entre las opciones que el cliente propuso originalmente.

Mensajes SSL

Cuando los clientes y servidores *SSL* se comunican, lo hacen intercambiando mensajes *SSL*. Esta sección mostrará cómo los sistemas utilizan estos mensajes en sus comunicaciones. La función más básica (y uno de los propósitos más importantes) que realiza un cliente y un servidor *SSL* es establecer la seguridad a través de un canal para comunicaciones cifradas. Los primeros tres mensajes SYN, SYN ACK y SYN correspondientes al protocolo TCP, Luego, inician los mensajes pertenecientes a la comunicación *SSL*.

ClientHello El mensaje *ClientHello* inicia la comunicación *SSL* entre las dos partes. El cliente usa este mensaje para pedirle al servidor que comience a negociar los servicios de seguridad usando *SSL*.

El mensaje este compuesto por ciertos campos:

- Versión: refiere a la versión más alta de *SSL* que el cliente puede admitir.
- RandomNumber: proporciona la semilla para cálculos criptográficos críticos.
- SessionID: es opcional, y muchas veces no es utilizado.
- CipherSuites: permite a un cliente enumerar los diversos servicios criptográficos que el cliente puede admitir
- CompressionMethods: es utilizado por el cliente para enumerar todos los diversos métodos de compresión de datos que puede admitir. Los métodos de compresión son una parte importante de *SSL* porque el cifrado tiene una secuencia significativa en la efectividad de cualquier técnica de compresión de datos.

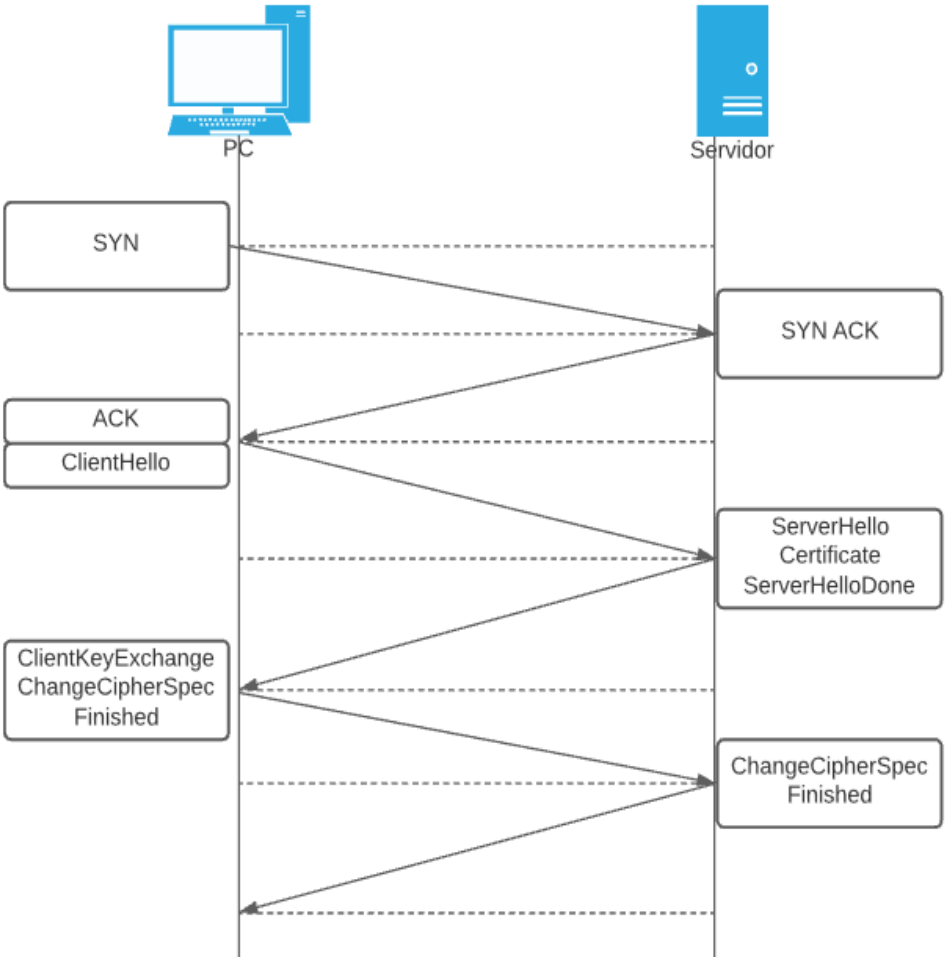


Figura 2.3: Mensajes SSL

ServerHello Este mensaje complementa el campo *CipherSuite* del *ServerHello*. Si bien el campo *CipherSuite* indica los algoritmos criptográficos y los tamaños de clave, este mensaje contiene la información de la clave pública en sí. Tenga en cuenta que el mensaje *ServerKeyExchange* se transmite sin cifrado, por lo que solo se puede incluir de forma segura información de clave pública. El cliente utilizará la clave pública del servidor para cifrar una clave de sesión.

ServerKeyExchange Este mensaje complementa el campo *CipherSuite* del *ServerHello*. Si bien el campo *CipherSuite* indica los algoritmos criptográficos y los tamaños de clave, este mensaje contiene la información de la clave pública en sí.

ServerHelloDone El mensaje *ServerHelloDone* le dice al cliente que el servidor ha terminado con sus mensajes iniciales de negociación. El mensaje en sí no contiene otra información, pero es importante para el cliente, porque una vez que el cliente recibe un *ServerHelloDone*, puede pasar a la siguiente fase para establecer las comunicaciones seguras.

ClientKeyExchange Cuando el servidor ha terminado su parte de la negociación *SSL* inicial, el cliente responde con un mensaje *ClientKeyExchange*. Así como *ServerKeyExchange* proporciona la información clave para el servidor, *ClientKeyExchange* le dice al servidor la información clave del cliente. En este caso, sin embargo, la información clave es para el algoritmo de cifrado simétrico que ambas partes usarán para la sesión. Además, la información del mensaje del cliente se cifra mediante la clave pública del servidor. Este cifrado protege la información de la clave a medida que atraviesa la red y permite al cliente verificar que el servidor realmente posee la clave privada correspondiente a su clave pública. De lo contrario, el servidor no podrá descifrar este mensaje. Esta operación es una protección importante contra un atacante que intercepta mensajes de un servidor legítimo y finge ser ese servidor reenviando los mensajes a un cliente desprevenido. Dado que un servidor falso no conocerá la clave privada del servidor real, no podrá descifrar el mensaje *ClientKeyExchange*. Sin la información en ese mensaje, la comunicación entre las dos partes no puede tener éxito.

ChangeCipherSpec Una vez que el cliente envía información clave en un mensaje *ClientKeyExchange*, se completa la negociación *SSL* preliminar. En ese momento, las partes están listas para comenzar a utilizar los servicios de seguridad que han negociado.

El protocolo *SSL* define un mensaje especial, *ChangeCipherSpec*, para indicar explícitamente que ahora se deben invocar los servicios de seguridad. Para cualquier sistema dado, ya sea un cliente o un servidor, *SSL* define un estado de escritura y un estado de lectura. El estado de escritura define la información de seguridad de los datos que envía el sistema y el estado de lectura define la información de seguridad de los datos que recibe el sistema.

Finished Inmediatamente después de enviar sus mensajes *ChangeCipherSpec*, cada sistema también envía un mensaje *Finished*. Los mensajes *Finished* permiten que ambos sistemas verifiquen que la negociación se ha realizado correctamente y que la seguridad no se ha visto comprometida.

Cada mensaje *Finished* contiene un *hash* criptográfico de información importante sobre la negociación recién finalizada. Esto protege contra un atacante que logra insertar mensajes ficticios o eliminar mensajes legítimos de la comunicación. Si un atacante pudiera hacerlo, los cálculos de *hash* del cliente y del servidor no coincidirían y detectarían el compromiso.

Finalización de las comunicaciones seguras Aunque, en la práctica, rara vez se utiliza (principalmente debido a la naturaleza de las sesiones *web*), *SSL* tiene un procedimiento definido para finalizar una comunicación segura entre dos partes. En este procedimiento, los dos sistemas envían cada uno una alerta de cierre especial al otro.

Autenticar la identidad del servidor

Anteriormente se explicó cómo *SSL* puede establecer comunicaciones cifradas entre dos partes, lo que puede no agregar mucha seguridad a la comunicación. Con el cifrado solo, ninguna de las partes puede estar realmente segura de la identidad de la otra. La razón típica para usar el cifrado en primer lugar es mantener la información en secreto de algún tercero. Pero si ese tercero pudiera hacerse pasar con éxito como

el destinatario previsto de la información, entonces el cifrado no serviría de nada. Los datos estarían encriptados, pero el atacante tendría todos los datos necesarios para desencriptarlos. Para evitar este tipo de ataques, *SSL* incluye mecanismos que permiten a cada parte autenticar la identidad de la otra. Con estos mecanismos, cada parte puede estar segura de que la otra es genuina y no un atacante enmascarado. En esta sección, veremos cómo *SSL* permite que un servidor se autentique.

Certificate Al autenticar su identidad, el servidor envía un mensaje de certificado en lugar del mensaje *ServerKeyExchange* descrito anteriormente. El mensaje *Certificate* simplemente contiene una cadena de certificados que comienza con el certificado de clave pública del servidor y termina con el certificado raíz de la autoridad certificadora.

El cliente tiene la responsabilidad de asegurarse de que puede confiar en el certificado que recibe del servidor. Esa responsabilidad incluye verificar las firmas del certificado, los tiempos de validez y el estado de revocación. También significa asegurarse de que la autoridad de certificación sea una en la que el cliente confíe. Normalmente, los clientes toman esta determinación conociendo la clave pública de las autoridades de certificación confiables de antemano, a través de algunos medios confiables. Microsoft, por ejemplo, carga las claves públicas en el repositorio de claves del Sistema Operativo para autoridades de certificación conocidas. Por otro lado, Firefox tiene su propio repositorio, incluido en el Software

ClientKeyExchange El mensaje *ClientKeyExchange* del cliente también difiere en la autenticación del servidor, aunque la diferencia no es importante. Cuando solo se va a utilizar cifrado, el cliente cifra la información en el mensaje *ClientKeyExchange* utilizando la clave pública que el servidor proporciona en su mensaje *ServerKeyExchange*. En este caso, por supuesto, el servidor se está autenticando y, por lo tanto, ha enviado un mensaje de certificado en lugar de un *ServerKeyExchange*. El cliente, por lo tanto, encripta su información usando la clave pública contenida en el certificado del servidor. Este paso es importante porque le permite al cliente asegurarse de que la parte con la que se está comunicando realmente posee la clave privada del servidor. Solo un sistema con la clave privada real podrá descifrar este mensaje y continuar con éxito la comunicación.

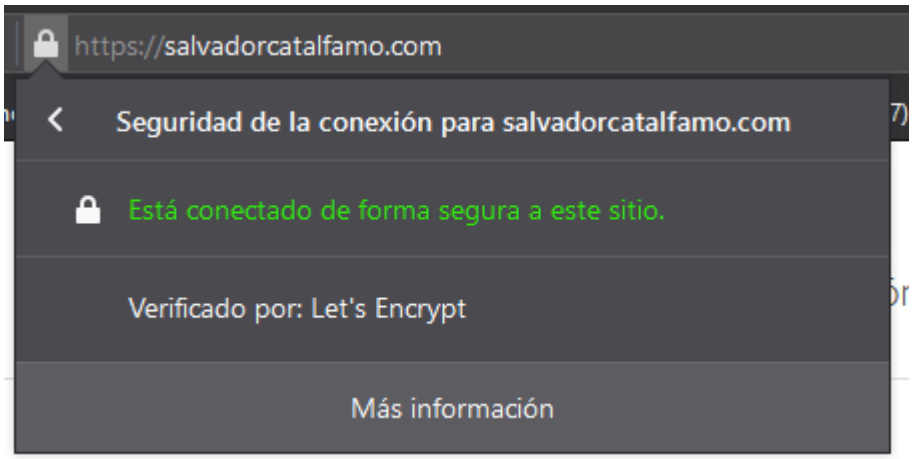


Figura 2.4: Validación de dominio

Niveles de validación

Hay tres tipos de certificados *SSL* disponibles en la actualidad: validación por dominio (DV), validación por organización (OV) y validación extendida (EV). Los niveles de cifrado son los mismos para cada certificado, lo que difiere son los procesos de investigación y verificación necesarios para obtener el certificado.

Validación de dominio (DV) Validación de dominio *SSL* o DV *SSL* representa el nivel base para los tipos de *SSL*. Estos son perfectos para sitios *web* que solo necesitan cifrado y nada más. Los certificados DV *SSL* suelen ser económicos y se pueden emitir en cuestión de minutos. Eso es porque el proceso de validación está completamente automatizado. Simplemente es necesario demostrar que es propietario de su dominio y que el certificado DV es suyo.

Validación de la organización (OV) Validación de organización u OV *SSL* representa el término medio para los tipos de certificados *SSL*. Para obtener OV *SSL*, su empresa u organización debe someterse a un examen comercial ligero. Esto puede demorar hasta tres días hábiles porque alguien tiene que verificar la información de su empresa. OV *SSL* muestra los mismos indicadores visuales que DV *SSL*, pero proporciona una forma de ver la información comercial verificada en la sección de detalles del certificado.

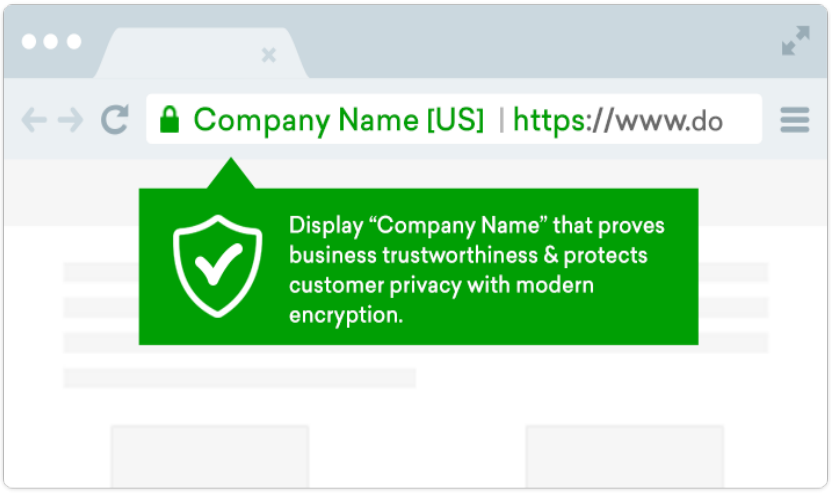


Figura 2.5: Validación extendida (Captura antigua)

Extended Validation (EV) *SSL* de validación extendida o *SSL* con EV requiere un exhaustivo examen comercial. Esto puede parecer mucho, pero en realidad no lo es si su empresa tiene registros disponibles públicamente. EV *SSL* activaba un indicador visual único: el nombre de su organización verificado que se muestra en el navegador. Esto en la actualidad ya no sucede, por lo que no es posible a simple vista identificarlo.

Tipos de certificados

Dominio Simple Como sugiere el nombre, un certificado *SSL* de un solo dominio solo se puede usar en un solo dominio o IP. Este se considera el tipo de certificado *SSL* predeterminado.

Multi-Dominio Este tipo de *SSL* es un certificado para todos los usos. Permiten cifrar hasta 250 dominios diferentes y subdominios ilimitados.

Wildcard Los *wildcard* están diseñados específicamente para cifrar un dominio y todos los subdominios que lo acompañan (también representado como *.dominio.com). Los *wildcard* solo están disponibles en los niveles DV y OV.

Multi-Dominio Wildcard Los *wildcard* multidominio pueden cifrar hasta 250 dominios diferentes y subdominios ilimitados.

Validación de propiedad de dominio

Los certificados se utilizan con mayor frecuencia para autenticar nombres de dominio. Por lo tanto, se confía en las autoridades de certificación (*CA*) para verificar que un solicitante de un certificado represente legítimamente el nombre de dominio en el certificado.

Los diferentes tipos de certificados reflejan diferentes tipos de verificación de *CA*. Los certificados de "Validación de dominio" (DV) son el tipo más común. La única validación que debe realizar la *CA* en el proceso de emisión de un certificado DV es verificar que el solicitante tiene un control efectivo del dominio. La *CA* no está obligada a intentar verificar la identidad real del solicitante. Esto difiere en los certificados de "Validación de la organización" y "Validación extendida", donde el proceso está destinado a verificar también la identidad real del solicitante.

ACME (*Automatic Certificate Management Environment*) permite a un cliente la automatización de gestión de certificados mediante un conjunto de mensajes transmitidos a través de HTTPS. La emisión de certificados mediante el protocolo *ACME* se asemeja al proceso de emisión de una *CA* tradicional, en el que un usuario crea una cuenta, solicita un certificado y demuestra el control de los dominios con el certificado para que la *CA* emita el certificado solicitado.

ACME utiliza un *framework* de desafío/respuesta extensible para la validación de dominios. El servidor envía al cliente un conjunto de desafíos, y el cliente responde enviando la respuesta al mismo en una solicitud POST a una *URL* de desafío.

Los diferentes desafíos permiten al servidor obtener pruebas de diferentes aspectos del control sobre un dominio. En los desafíos como *HTTP* y *DNS*, el cliente demuestra directamente su capacidad para hacer ciertas acciones relacionadas con el dominio. Es de gran utilidad explicar los diferentes tipos de desafíos que se puede ofrecer a un cliente, ya que uno es el más común, sin embargo, al hablar de redes internas, no lo podremos utilizar, e iremos por la otra opción, un tanto menos conocida.

Desafío HTTP Con la validación *HTTP*, el cliente prueba su control sobre un nombre de dominio al demostrar que puede guardar recursos *HTTP* en un servidor accesible bajo ese nombre de dominio. El servidor *ACME* desafía al cliente solicitándole un archivo en una ruta específica, con una cadena específica como contenido.

Este es el tipo de desafío más común en la actualidad. El servidor le da un *token* al cliente *ACME* y éste coloca un archivo en su servidor *web*. Por ejemplo, para el servidor de la Universidad la ubicación sería: *http://cs.uns.edu.ar/.well-known/acme-challenge/*TOKEN. Ese archivo contiene el *token* más una huella digital de la clave de su cuenta.

Una vez que el cliente le informa al servidor que el archivo está listo, el servidor intenta recuperarlo. Al recibir una respuesta, el servidor construye y almacena la autorización de la clave a partir del valor del *token* de desafío y la clave de la cuenta del cliente actual.

Dado un par de desafío/respuesta, el servidor verifica el control del dominio por parte del cliente verificando que el recurso se aprovisionó como se esperaba.

Ventajas:

- Es fácil de automatizar sin conocimientos adicionales sobre la configuración de un dominio.
- Funciona con servidores *web* estándar.

Desventajas:

- No funciona si su ISP bloquea el puerto 80 (esto es raro, pero algunos ISP residenciales lo hacen).
- Let's Encrypt no le permite utilizar este desafío para emitir certificados Wildcard.
- Si tiene varios servidores web, debe asegurarse de que el archivo esté disponible en todos ellos.

Este desafío está fuera de nuestro alcance, ya que partimos de la premisa de que el tráfico que queremos proteger nunca saldrá a *Internet*, lo que implica que no tendremos ni puertos ni direcciones expuestas para que un servidor externo pueda verificar el recurso mencionado anteriormente.

Desafío DNS Cuando el identificador que se está validando es un nombre de dominio, el cliente puede demostrar el control de ese dominio proporcionando un registro *DNS* de tipo TXT que contenga un valor designado.

Un cliente cumple este desafío construyendo una clave de autorización a partir del valor de un *token* proporcionado y la clave de la cuenta del cliente. A continuación, el cliente calcula un *hash* SHA-256 de la clave de autorización. El registro proporcionado al *DNS* contiene la codificación de *URL* base64 de este *hash*.
Ejemplo:

Si se desea validar el nombre de dominio *www.cs.uns.edu.ar*, el cliente proporcionaría el siguiente registro *DNS*:

```
_acme-challenge.www.cs.uns.edu.ar: "gfj9Xq...Rg85nM"
```

Al recibir una respuesta, el servidor construye y almacena la llave de autorización clave a partir del valor del *token* del desafío y la clave de la cuenta del cliente actual. Para validar un desafío de *DNS*, el servidor realiza los siguientes pasos:

1. Calcula el *hash* SHA-256 de la clave de autorización almacenada.
2. Consulta los registros TXT para el nombre de dominio de validación.
3. Verifica que el contenido de uno de los registros TXT coincida con el valor de *hash*.

Si todas las verificaciones anteriores tienen éxito, entonces la validación es exitosa. Si no se encuentra ningún registro *DNS*, o si el registro *DNS* y el contenido del mismo no pasan estas comprobaciones, la validación falla.

Capítulo 3

Debilidades del protocolo HTTP

HTTP originalmente no fue un protocolo pensado en la seguridad. Los mensajes *HTTP* se envían a través de *Internet* sin cifrar y, por lo tanto, cualquiera puede leer el mismo cuando se dirige a su destino. *Internet*, como su nombre indica, es una red de computadoras (*interconnected computer networks*), no un sistema punto a punto. Hablando específicamente de una red interna, no sabemos cómo se enrutan los mensajes y nosotros, como usuarios, no tenemos idea de que otras partes podrían captarlos. Debido a que *HTTP* va a través de texto plano, los mensajes se pueden interceptar, leer, e incluso alterar en el camino.

En este capítulo veremos por qué el protocolo *HTTP* es inseguro, adicionando luego un caso de estudio donde se demuestra un pequeño ataque.

3.1. Riesgos de intermediarios

Un intermediario es alguien que puede acceder al contenido de lo que circula por la red. A esta actividad o ataque se lo conoce como *man-in-the-middle* u hombre en el medio. El intermediario puede realizar ciertos actos: interceptar, modificar o fabricar datos. Desde este punto de vista, la confidencialidad puede verse afectada si alguien intercepta datos, y la integridad puede fallar si alguien o un programa modifica o fabrica datos falsos.

Como mencionamos anteriormente, *HTTP* no fue pensado para ser seguro, por lo que el protocolo en sí mismo no puede resolver este problema. En este proyecto, explotaremos esta vulnerabilidad para demostrar la facilidad con la que un agente

puede hacerse de nuestro tráfico.

3.2. Confidencialidad del mensaje

La definición de confidencialidad es sencilla: solo las personas o los sistemas autorizados pueden acceder a los datos protegidos. Sin embargo, como veremos en capítulos posteriores, garantizar la confidencialidad puede resultar difícil. Por sí solo, el protocolo no cifra los mensajes, sin embargo, dado que *HTTP* se ha diseñado para ser independiente del protocolo de transporte, de modo que se puede utilizar en muchas formas diferentes de conexión cifrada.

3.3. Integridad de los mensajes

Integridad refiere a distintos significados en diferentes contextos. Cuando examinamos la forma en la que usamos este término, encontramos varios significados diferentes. Por ejemplo, si decimos que hemos conservado la integridad de un elemento, podemos querer decir que el elemento está sin modificar, modificado solo de formas aceptables, solo por personas autorizadas, solo por procesos autorizados, etc.

Hablando de datos, se reconocen tres aspectos particulares de la integridad: acciones autorizadas, protección de recursos y detección y corrección de errores.

De la misma manera que en la integridad del mensaje, *HTTP* no define un mecanismo específico para garantizar la integridad de los mensajes, sino que se basa en la capacidad de detección de errores de los protocolos de transporte subyacentes y en el uso de tramas delimitadas por longitud. Históricamente, la falta de un mecanismo de integridad único se ha justificado por la naturaleza informal de la mayoría de las comunicaciones *HTTP*. Sin embargo, el predominio de *HTTP* como mecanismo de acceso a la información ha dado como resultado la necesidad de verificar la integridad en ciertos entornos.

3.4. Tipos de ataques

Existen principalmente dos tipos de ataques a la red: ataques pasivos y ataques activos. La motivación detrás de los atacantes pasivos y los atacantes activos son totalmente diferentes. Mientras que la motivación de los atacantes pasivos es simplemente robar información sensible y analizar el tráfico para robar mensajes futuros, la motivación de los atacantes activos es impedir la comunicación normal entre dos entidades legítimas.

3.4.1. Ataques Pasivos

Los ataques pasivos ocurren cuando se monitorea y analiza información sensible, posiblemente comprometiendo la seguridad de las empresas y sus clientes.

Estos ataques están principalmente interesados en robar información confidencial. Esto sucede sin el conocimiento de la víctima. Como tales, son difíciles de detectar y, por lo tanto, es difícil de proteger la red de los mismos. Entre los ataques más comunes están los de análisis y monitoreo de tráfico, como así también las escuchas de comunicaciones telefónicas.

3.4.2. Ataques Activos

Los ataques activos ocurren cuando la información se modifica, se altera o se destruye por completo. Aquí el intruso inicia instrucciones para perturbar la comunicación regular de la red. Algunos de estos se muestran a continuación:

- Modificación: el nodo malicioso realiza algunas alteraciones en el enrutamiento. Esto da como resultado que el remitente envíe mensajes a través de la ruta larga, lo que provoca un retraso en la comunicación. Este es un ataque a la integridad.
- Fabricación: un nodo malicioso genera un mensaje de enrutamiento falso que provoca la generación de información incorrecta sobre la ruta entre dispositivos. Este es un ataque a la autenticidad.
- *Spoofing*: un nodo malicioso presenta incorrectamente su identidad para que el remitente cambie su topología, y por consiguiente el destino de sus mensajes.

Figura 3.1: Escritorio de Kali Linux

- Denegación de servicios: un nodo malicioso envía un mensaje al nodo y consume el ancho de banda de la red en cómputo desperdiciado.
- *Man-in-the-middle*: también llamado ataque de secuestro, es un ataque en el que se altera y transmite en secreto las comunicaciones entre dos partes legítimas sin su conocimiento. Estas partes, a su vez, desconocen lo que sucede, pues no perciben un cambio en la comunicación.

3.5. Caso de estudio: Interceptando la red para obtener credenciales

3.5.1. Herramienta Utilizadas

Kali Linux

Kali Linux es una distribución de Linux (basada en Debian) centrada en la seguridad. Es una versión renombrada de la famosa distribución de Linux conocida como Backtrack, que venía con un enorme repositorio de herramientas de piratería de código abierto, para pruebas de penetración de aplicaciones *web*, inalámbricas y de red.

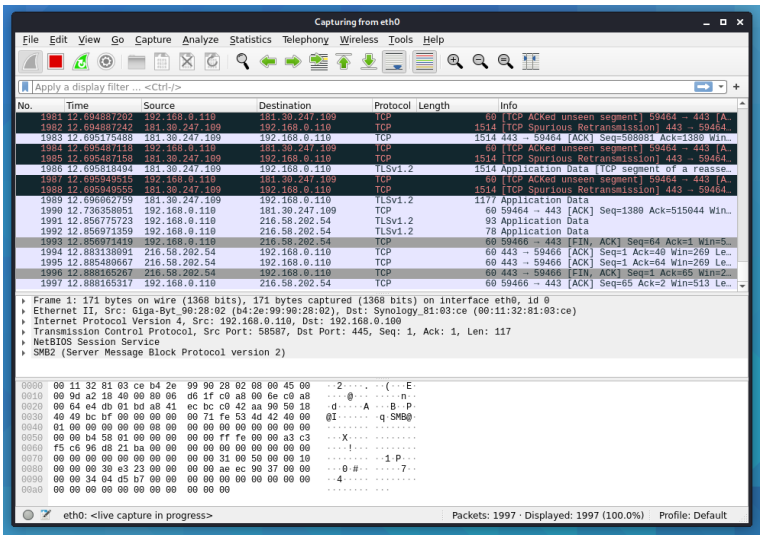


Figura 3.2: Interface del Wireshark

Kali Linux contiene muchas herramientas preinstaladas con todas las dependencias y ya está lista para usar. Esto nos permite tener que prestar más atención a las pruebas y no a la instalación de la herramienta. Las actualizaciones para las herramientas instaladas en Kali Linux se publican con mayor frecuencia, lo que le ayuda a mantener las a las mismas actualizadas.

Esta distribución contiene las herramientas necesarias para realizar nuestro ataque.

Wireshark

Wireshark es uno de los analizadores de protocolos de red más populares, es de código abierto y gratuito. Wireshark está preinstalado en Kali y es ideal para la resolución de problemas de red, análisis y, para este caso de estudio, una herramienta perfecta para monitorear el tráfico de posibles objetivos. Wireshark usa un kit de herramientas para implementar su interfaz de usuario y para capturar paquetes. Funciona de manera muy similar a un comando *tcpdump*; sin embargo, nos brinda una interfaz gráfica, posee opciones integradas de clasificación y filtrado.

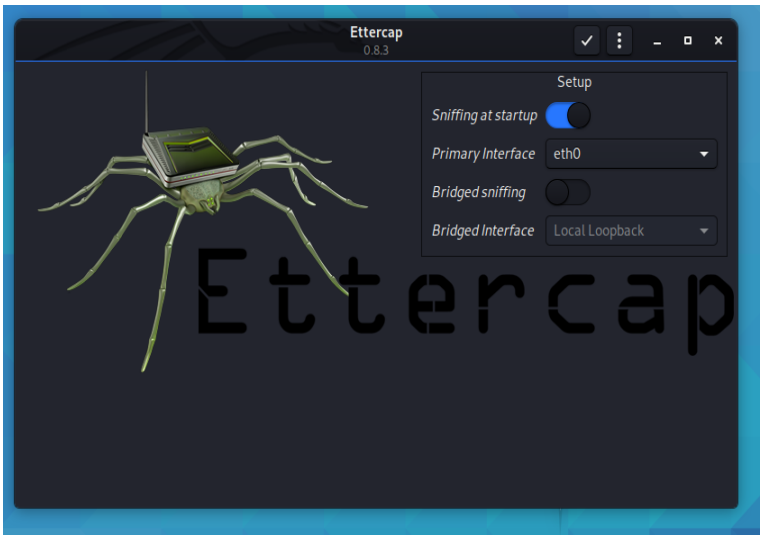


Figura 3.3: Ettercap

Ettercap

Ettercap es un paquete completo gratuito y de código abierto para ataques basados en intermediarios. Ettercap se puede utilizar para análisis de protocolos de redes informáticas y auditorías de seguridad, con funciones de rastreo de conexiones en tiempo real y filtrado de contenido. Ettercap funciona configurando la interfaz de red del atacante en modo promiscuo y *ARP* para envenenar las máquinas víctimas.

3.5.2. Realización del ataque

La idea principal de esta sección es demostrar que, encontrándose en una red interna y con ciertas herramientas, es posible realizar un ataque sin necesidad de conocer a fondo la implementación de la misma ni de tener mayores privilegios.

El escenario montado (figuras 3.4 y 3.5) consiste en crear una página *web* con un formulario donde se debe completar con usuario y contraseña, y un submit el cual envía esta información desde el cliente hasta el servidor *web*. El envío de este formulario contiene la información confidencial, por lo que en un escenario seguro ningún intermediario podría obtener estos datos. Dado que este tráfico circula utilizando el protocolo *HTTP*, mostraremos como nos podemos hacer de las credenciales ingresadas por el usuario.

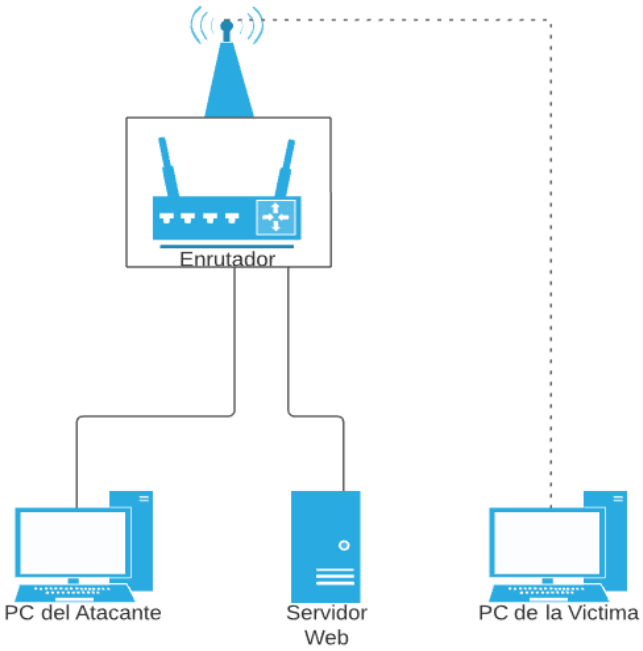


Figura 3.4: Escenario montado

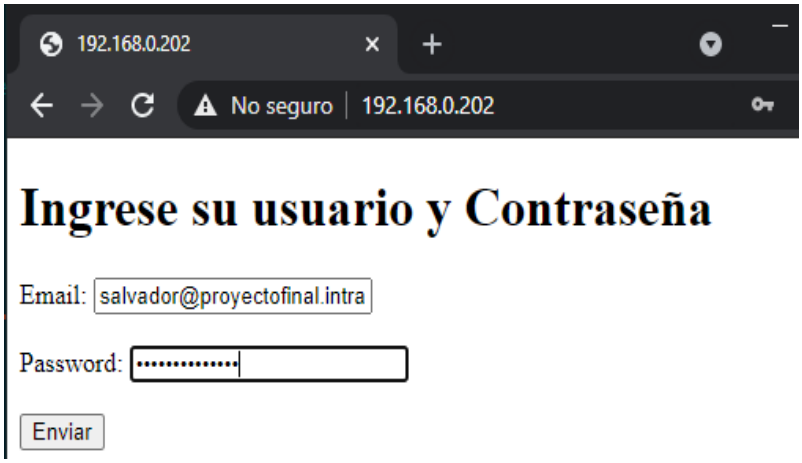


Figura 3.5: Formulario montado

3.5.3. Preparando Ettercap para el ataque ARP Poisoning

Lo primero que debemos hacer, en la lista de aplicaciones, es buscar el apartado *Sniffing* y *Spoofing*, ya que es allí donde encontraremos las herramientas necesarias para llevar a cabo este ataque. A continuación, abriremos Ettercap.

El siguiente paso es seleccionar la tarjeta de red con la que vamos a trabajar. Para ello, en el menú superior de Ettercap seleccionaremos Sniff > Unified Sniffing y, cuando nos lo pregunte, seleccionaremos nuestra tarjeta de red (por ejemplo, en nuestro caso, eth0).

Luego buscaremos todos los hosts conectados a nuestra red local. Para ello, seleccionaremos Hosts del menú de la parte superior y seleccionaremos la primera opción, Hosts List (Figura 3.6).

Allí veremos todos los hosts o dispositivos conectados a nuestra red. Sin embargo, en caso de que no estén todos, podemos realizar una exploración completa de la red simplemente abriendo el menú Hosts y seleccionando la opción Scan for hosts. Tras unos segundos, la lista de antes se debería actualizar mostrando todos los dispositivos, con sus respectivas *IPs* y *MACs*, conectados a nuestra red.

3.5.4. Nuestro Ettercap ya está listo. Ya podemos empezar con el ataque ARP Poisoning

En caso de querer realizar un ataque dirigido contra un solo host, por ejemplo, suplantar la identidad de la puerta de enlace para monitorear las conexiones de la víctima, antes de empezar con el ataque debemos establecer los dos objetivos.

Para ello, debajo de la lista de hosts podemos ver tres botones, aunque nosotros prestaremos atención a los dos últimos:

Target 1 – Seleccionamos la *IP* del dispositivo a monitorear, en este caso, la computadora de la víctima, y pulsamos sobre dicho botón.

Target 2 – Pulsamos la *IP* que queremos suplantar, en este caso, la de la puerta de enlace y la del servidor *web*.

Debemos elegir el menú MITM de la parte superior y, en él, escoger la opción ARP Poisoning. Nos aparecerá una pequeña ventana de configuración, en la cual

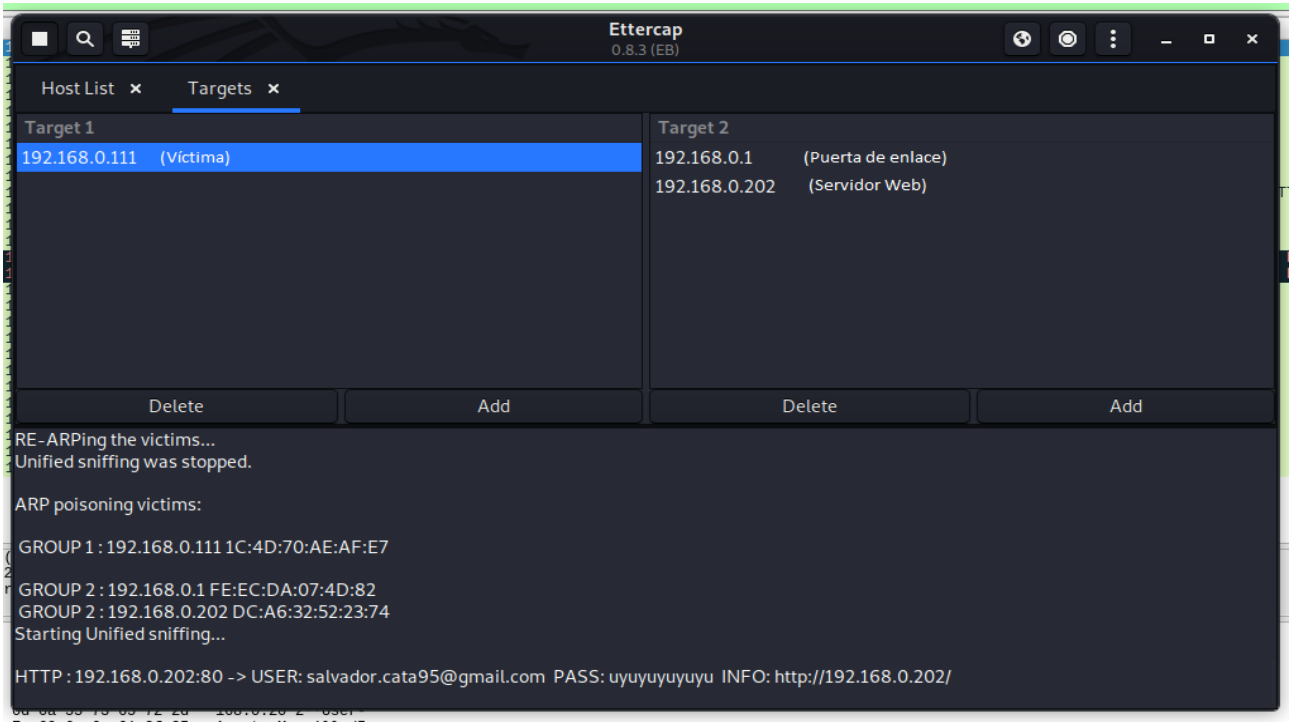


Figura 3.6: Ettercap

debemos asegurarnos de marcar SNIFF REMOTE CONNECTIONS. Pulsamos sobre OK y el ataque dará lugar. Ahora ya podemos tener el control sobre el host que hayamos establecido como *Target 1*. Lo siguiente que debemos hacer es, ejecutar Wireshark para capturar todos los paquetes de red y analizarlos en busca de información interesante.

Como se puede ver en la figura 3.7, Wireshark nos permite filtrar el tráfico, y con el simple hecho de decirle que queremos mostrar los requerimientos GET pudimos dar con el paquete que queríamos, en el request podemos ver el usuario `salvador@proyectofinal.test` y la contraseña `PASSWORD SEGURO`

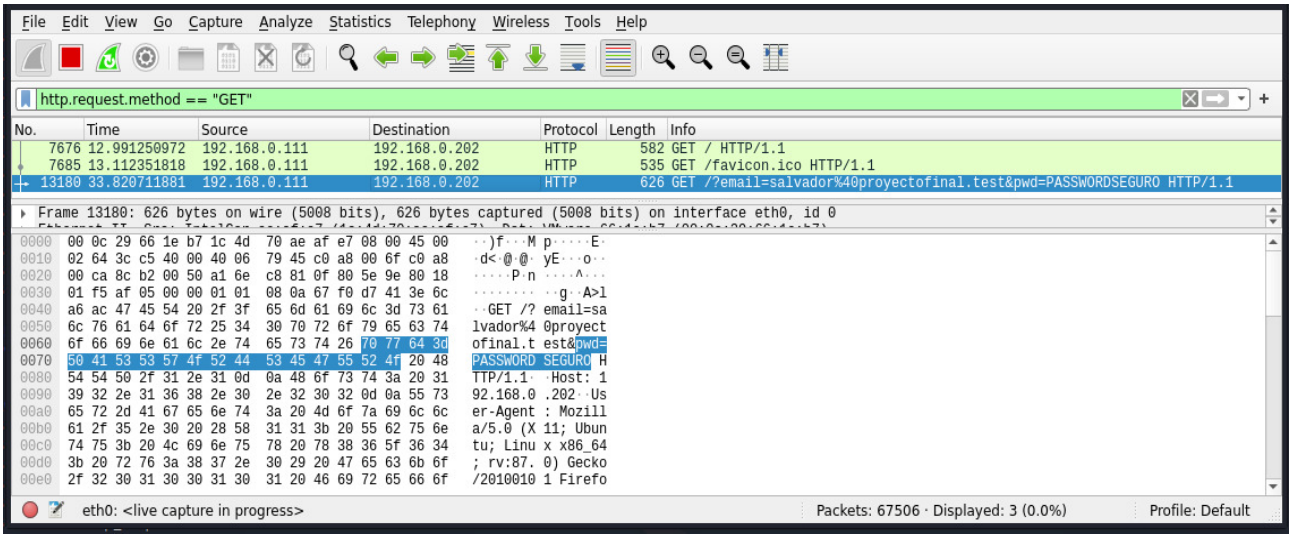


Figura 3.7: Paquetes capturados

Capítulo 4

Soluciones estudiadas

Luego de haber explicado las debilidades del protocolo *HTTP*, vamos a explorar las distintas soluciones encontradas, empezando con un pequeño marco teórico, el cual nos permitirá probar estas implementaciones y aplicarlas en un escenario de prueba.

La herramienta utilizada es *Docker*, con ella es posible simular escenarios sin necesidad de tener cada uno de los servidores de manera física. Para entenderlo un poco mejor, explicaremos lo que es la virtualización y profundizaremos en la virtualización basada en contenedores.

4.1. Máquinas virtuales

La virtualización brinda la capacidad de ejecutar aplicaciones, sistemas operativos o servicios del sistema en un entorno distinto. Obviamente, los mismos tienen que estar ejecutándose en un determinado sistema informático en un momento dado, pero la virtualización proporciona un nivel de abstracción lógica que libera las aplicaciones, los servicios del sistema e incluso el sistema operativo de estar vinculado a una pieza específica de *hardware*. El enfoque de la virtualización hace que todo esto sea portátil a través de diferentes sistemas informáticos físicos.

El enfoque basado en *VM* virtualiza el sistema operativo completo. Con esto nos referimos a la virtualización de discos, *CPU* y *NIC*. En otras palabras, podemos afirmar que se trata de virtualizar la arquitectura de conjunto de instrucciones completa, como ejemplo, la arquitectura *x86*.

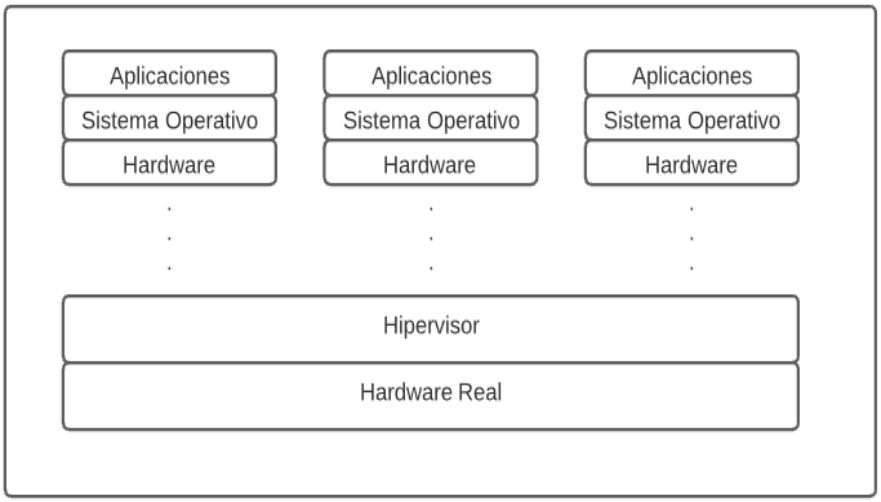


Figura 4.1: Virtualización

Para virtualizar un sistema operativo, se utiliza un *software* especial, denominado *hipervisor*, y es el encargado, entre otras cosas, de aislar el sistema operativo de las máquinas virtuales y permitir crearlas y gestionarlas.

Una *VM* debe satisfacer tres propiedades:

- Aislamiento: debe aislar a los invitados entre sí.
- Equivalencia: debe comportarse igual, con o sin virtualización. Esto significa que se deben ejecutar la mayoría de las instrucciones en el *hardware* físico sin ninguna traducción hacia el *hardware* virtual.
- Rendimiento: Debería funcionar tan bien como lo hace sin ninguna virtualización. Esto nuevamente significa que la sobrecarga de ejecutar una VM debe ser mínima.

4.2. Virtualización basada en Contenedores

Esta forma de virtualización no abstrae el *hardware*, sino que utiliza técnicas dentro del *kernel* de Linux para aislar las rutas de acceso para diferentes recursos. Establece un límite lógico dentro del mismo sistema operativo. Como resultado, obtenemos un sistema de archivos raíz separado, un árbol de procesos separado, un subsistema de red separado, etc.

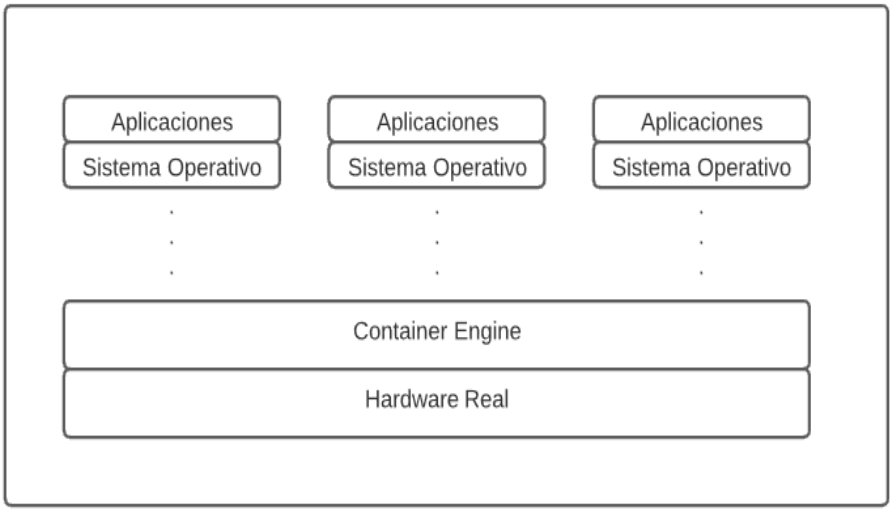


Figura 4.2: Virtualización basada en Contenedores

El *kernel* de Linux se compone de varios componentes y funcionalidades; los relacionados con contenedores son los siguientes:

- Grupos de control (*Cgroups*)
- Espacios de nombres (Namespaces)
- Linux con seguridad mejorada (SELinux)

Cgroups

La funcionalidad de *cgroup* permite limitar y priorizar recursos, como *CPU*, RAM, la red, el sistema de archivos, etc. El objetivo principal es no exceder los recursos, para evitar desperdiciar recursos que podrían ser necesarios para otros procesos.

Espacios de nombres

La funcionalidad del espacio de nombres permite particionar los recursos del kernel, de modo que un conjunto de procesos ve un conjunto de recursos, mientras que otro conjunto de procesos ve un conjunto diferente de recursos. La característica funciona al tener el mismo espacio de nombres para estos recursos en los distintos conjuntos de procesos, pero esos nombres se refieren a recursos distintos. Algunos

nombres de recursos que pueden existir en varios espacios son los ID de proceso, nombres de host, ID de usuario, nombres de archivo y algunos nombres asociados con el acceso a la red y la comunicación entre procesos.

Cuando se inicia un sistema Linux, solo se crea un espacio de nombres. Los procesos y recursos se unirán al mismo espacio de nombres, hasta que se cree un espacio de nombres diferente, se le asignen recursos y los procesos se unan a él.

SELinux

SELinux es un módulo del *kernel* de Linux que proporciona un mecanismo para hacer cumplir la seguridad del sistema, con políticas específicas. Básicamente, SELinux puede limitar el acceso de los programas a archivos y recursos de red. La idea es limitar los privilegios de los programas y demonios al mínimo, de modo que pueda limitar el riesgo de que el sistema se detenga.

Lo contenedores utilizan los recursos directamente y no necesitan de un emulador en absoluto, cuantos menos recursos, mayor eficiencia. Se pueden ejecutar diferentes aplicaciones en el mismo host: aisladas a nivel de *kernel* y aisladas por espacios de nombres y *cgroups*. El *kernel*, es decir, el Sistema Operativo, lo comparten todos los contenedores.

Contenedores

Cuando hablamos de contenedores, nos referimos indirectamente a dos conceptos principales: una imagen de contenedor y una imagen de contenedor en ejecución. Una imagen de contenedor es la definición del contenedor, en donde el *software* restante se instala como capas adicionales, como se muestra en el diagrama.

Una imagen de contenedor suele estar formada por varias capas (figura 4.3). La primera capa está dada por la imagen base, que proporciona las funcionalidades centrales del sistema operativo, con todas las herramientas necesarias para comenzar. Los equipos de desarrolladores a menudo trabajan construyendo sus propias capas sobre estas imágenes base. Los usuarios también pueden crear imágenes de aplicaciones más avanzadas, que no solo tienen un sistema operativo, sino que también incluyen herramientas de depuración y bibliotecas.

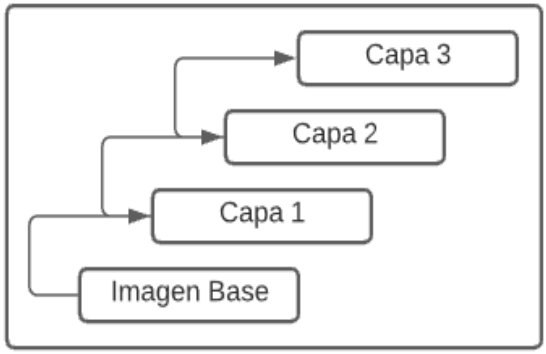


Figura 4.3: Capas de contenedores

Los contenedores brindan aislamiento al aprovechar las tecnologías del *kernel*, como *cgroups*, espacios de nombres del *kernel* y SELinux. Dado que utilizan un *kernel* compartido y un host de contenedor, se reduce la cantidad de recursos necesarios para el contenedor en sí y son más livianos en comparación con las máquinas virtuales.

Por lo mencionado anteriormente, podemos afirmar que los contenedores brindan una agilidad que no es factible con las *VM*, y una prueba de esto es que solo se necesitan unos segundos para iniciar un nuevo contenedor. Además, los mismos admiten un modelo más flexible en lo que respecta a la utilización de la *CPU* y los recursos de memoria, y permiten modos de ráfaga de recursos, donde las aplicaciones pueden consumir más recursos cuando es requerido, dentro de los límites definidos.

4.3. *Docker*

Docker es una herramienta de código abierto que automatiza la implementación de aplicaciones en contenedores. Fue escrito por el equipo de *Docker* y publicado por ellos bajo la licencia Apache 2.0. Está diseñado para proporcionar un entorno ligero y rápido para implementar escenarios determinados, así como un flujo de trabajo eficiente para llevar ese código desde una computadora portátil a su entorno de prueba y luego a producción. De hecho, se puede comenzar con *Docker* en un host mínimo que no ejecute nada más que un *kernel* de Linux compatible y un binario de *Docker*.

4.3.1. Imágenes

Las imágenes son los componentes básicos del mundo de *Docker*. El uso común es lanzando los contenedores a partir de imágenes. Tienen un formato en capas, que se forman paso a paso utilizando una serie de instrucciones.

Se puede considerar a las imágenes como el "código fuente" de los contenedores. Son muy portátiles y se pueden compartir, almacenar y actualizar. .

4.3.2. Registros

Docker almacena las imágenes que se crean en registros. Hay dos tipos de registros: públicos y privados. La herramienta opera el registro público de imágenes, llamado *Docker* Hub. Se puede crear una cuenta y usarla para compartir y almacenar nuestras propias imágenes. También es posible almacenar las imágenes que desee y mantenerlas privadas en *Docker* Hub. Estas imágenes pueden incluir código fuente u otra información de propiedad que se quiera mantener segura o solo compartir con otros miembros de su equipo u organización.

4.3.3. Contenedores

El *software* permite construir e implementar contenedores dentro de los cuales se puede empaquetar aplicaciones y servicios. Como acabamos de mencionar, los contenedores se lanzan a partir de imágenes y estos pueden contener uno o más procesos en ejecución.
Un contenedor es:

- Un formato de imagen.
- Un conjunto de operaciones estándar.
- Un entorno de ejecución.

Se puede hacer una analogía entre los contenedores de *Docker* y los contenedores de envío estándar, utilizado para transportar mercancías a nivel mundial. En lugar de enviar mercancías, los contenedores de *Docker* envían *software*. Cada contenedor contiene una imagen de *software*, su "carga", y, al igual que su contraparte física, permite realizar un conjunto de operaciones. Por ejemplo, se puede crear, iniciar, detener, reiniciar y destruir. Como un contenedor de envío, *Docker* no se preocupa

por el contenido del contenedor cuando realiza estas acciones; por ejemplo, si un contenedor es un servidor *web*, una base de datos o un servidor de aplicaciones. Cada contenedor se carga igual que cualquier otro. A *Docker* tampoco le importa dónde envía su contenedor: se puede compilar en una computadora portátil, cargarlo en un registro, luego descargarlo en un servidor físico o virtual, probarlo, implementarlo en un clúster de una docena de *hosts* y ejecutarlo.

4.4. Propuestas: Introducción

Se han investigado tres alternativas enfocadas en redes internas para mejorar la seguridad de las mismas, luego de eso se eligió la que más ventajas nos ofreció. Las alternativas son las siguientes: Los certificados auto-firmados (*Self-signed Certificates*), implementar una entidad certificante interna, y la utilización de un certificado emitido por una entidad certificante conocida.

4.5. Propuesta 1: Self-signed Certificates

Un certificado autofirmado es un certificado digital que no está firmado por una autoridad de certificación (*CA*) de confianza pública (o privada). La razón por la que se consideran diferentes de los certificados tradicionales es que quien los crea, emite y firma es la empresa o el desarrollador responsable del sitio.

Los certificados autofirmados son los menos útiles de los tres. Firefox facilita su uso de forma segura; crea una excepción en la primera visita, después de lo cual el certificado autofirmado se considera válido en las conexiones posteriores. Otros navegadores hacen que haga clic en una advertencia de certificado cada vez. A menos que esté comprobando la huella digital del certificado cada vez, no es posible hacer que ese certificado autofirmado sea seguro. Incluso con Firefox, puede resultar difícil utilizar estos certificados de forma segura.

Por ejemplo, para solicitar un certificado *SSL* de una *CA* de confianza como Verisign o GoDaddy, se debe enviar una Solicitud de firma de certificado (CSR) y te dan un certificado a cambio, que firmaron con su certificado raíz y clave privada. Todos

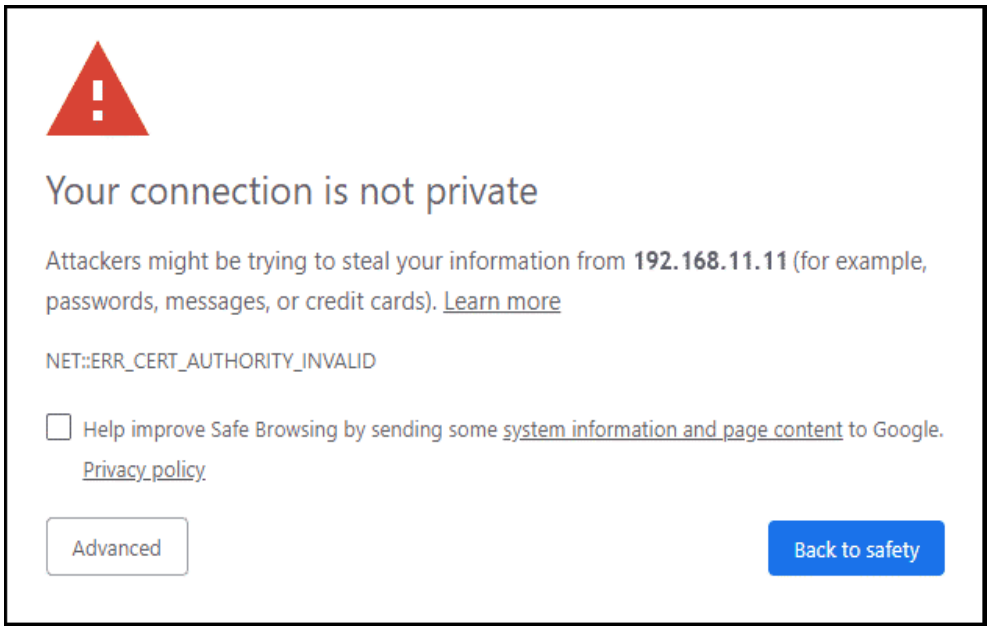


Figura 4.4: Advertencia Certificado Autofirmado

los navegadores tienen una copia (o acceden a una copia desde el sistema operativo) del certificado raíz, por lo que el navegador puede verificar que su certificado fue firmado por una *CA* confiable.

En la infraestructura de clave pública (PKI), ambas partes deben confiar en la autoridad de certificación. es decir, navegadores y servidores. En este caso, no existe tal confianza, con lo cual, los certificados tendrán la confianza inherente a la manera en la que se generaron.

La principal dificultad es que los usuarios siempre encontrarán una advertencia donde el navegador diga que se encuentra en un sitio con un certificado autofirmado (figura 4.4) . En la mayoría de los casos, no verificarán que el certificado es el correcto, por lo que generará desconfianza en los usuarios.

En prácticamente todos los casos, un enfoque mucho mejor es utilizar una *CA* privada, que es nuestra próxima propuesta. Requiere un poco más de trabajo por adelantado, pero una vez que la infraestructura está establecida y la clave raíz se distribuye de manera segura a todos los usuarios, dichas implementaciones son tan seguras como el resto del ecosistema PKI.

4.6. Propuesta 2: Internal CA

Como se explicó anteriormente, una entidad de certificación es un agente en el que confiamos para emitir certificados que confirman las identidades de los suscriptores, o bien de los sitios a los cuales visitamos.

Esta propuesta de solución implica establecer una entidad de certificación interna a la red privada, y por consiguiente hacer que los equipos que se encuentran dentro de la misma confíen el ella. Esto se hace mediante un servidor dedicado que firme los certificados que se le solicitan.

4.6.1. Caso de estudio: Creando nuestra Entidad Certificante privada

Servidor DNS con Docker

Para montar esta propuesta de solución, se debió crear un servidor *DNS*, que lo que hace es básicamente mapear direcciones IP a nombres de dominio. Por ejemplo: nuestra página que se encuentra en la dirección *192.168.0.202* se pasará a llamar pagina1.salvadorcatalfamo.intra, donde salvadorcatalfamo.intra abarca toda nuestra organización y las páginas que se encuentren dentro de ella. Otra razón por la cual se debe crear un servidor *DNS*, es que los certificados no se pueden otorgar a direcciones IP, por lo cual es un requisito obligatorio tener las páginas *web* direccionadas con nombres de dominio.

Se eligió el servidor *DNS* CoreDNS ya que es amigable con *Docker*; sucede que, por cada versión del programa, se generan las imágenes de *Docker* correspondientes. Estas imágenes son públicas y oficiales, lo que da confiabilidad y seguridad extra a la hora de utilizarlas. La configuración está formada por los siguientes componentes: los archivos de configuración de CoreDNS (corefile) y los sitios que nosotros deseemos, en nuestro caso *pagina1.salvadorcatalfamo.intra*
El Corefile es el archivo de configuración de CoreDNS. Este define:

- Qué servidores escuchan en que puertos y que protocolo.
- Para qué zona tiene autoridad cada servidor.
- Qué *plugins* (complementos) se cargan en un servidor.

El formato es el siguiente

```
ZONE: [PORT] {
    [PLUGIN] ...
}
```

ZONE: define la zona de este servidor. El puerto por defecto es el 53, o bien el valor que se le indique con el flag -dns.port.
PLUGIN: define los complementos que queremos cargar. Cada plugin puede tener varias propiedades, por lo que también podrían tener argumentos de entrada.
Nuestro archivo de configuración es el siguiente:

```
.:53 {
    forward . 8.8.8.8 9.9.9.9
    log
}

salvadorcatalfamo.intra:53 {
    file /etc/coredns/salvadorcatalfamo
    log
    reload 10s
}
```

A grandes rasgos, lo que indica esta configuración es que va a existir una zona "salvadorcatalfamo.intra", que estará definida por el archivo que se encuentra en */etc/coredns/salvadorcatalfamo*. Por otro lado, el tráfico restante (dominios externos a nuestra red) será fordwardeado a servidores *DNS* externos (*8.8.8.8* y *9.9.9.9*). Además se establecieron algunos plugins de logeo y de refresco de configuración.

Nuestro archivo */etc/coredns/salvadorcatalfamo* contiene la siguiente información.

```
salvadorcatalfamo.intra.          IN  SOA ns1.salvadorcatalfamo.intra. ...
pagina1.salvadorcatalfamo.intra.  IN  A   192.168.0.124
```

Esto en principio es suficiente para nuestro sitio interno, y contiene las direcciones ip de los servidores *web* y entidades certificantes.

Por el lado de *Docker*, se utilizó un archivo Docker-compose.yml, y un Dockerfile. Docker-compose es una herramienta para definir y ejecutar aplicaciones *Docker* de

varios contenedores. Compose usa un archivo YAML para configurar los servicios de la aplicación. Luego, con un solo comando, se crean e inician todos los servicios desde este archivo de configuración. En nuestro caso, se definió de la siguiente manera

```
version: '3.1'
services:
    coredns:
        build: .
        container_name: coredns
        restart: always
        expose:
            - '53'
            - '53/udp'
        ports:
            - '53:53'
            - '53:53/udp'
        volumes:
            - './config:/etc/coredns'
```

Por otro lado, el fichero dockerfile está compuesto por las siguientes líneas:

```
FROM coredns/coredns:1.7.0
ENTRYPOINT ["/coredns"]
CMD ["-conf", "/etc/coredns/Corefile"]
```

En conjunto, establecen la imagen de CoreDNS que se utilizará, los archivos de configuración y los puertos que se expondrán, entre otras configuraciones.

Creación de nuestra CA en *Docker*

La estrategia para crear nuestra *CA* será seguir los pasos que se deberían realizar en un servidor habitual, pero partiendo desde una imagen de Docker de Ubuntu (de stock), y luego realizando un commit de estas configuraciones. Luego, archivos importantes como el certificado root y la llave privada deberán resguardarse, o simplemente resguardar el contenedor creado.

Se utilizó esta estrategia ya que no había imágenes oficiales que nos sirva para tal fin, por el simple hecho de que únicamente se requiere tener instalado OpenSSL y tenerlo configurado.
Como primer paso, corremos una imagen del sistema operativo Ubuntu

```
docker run -it -v $PWD/ca:/root/ca ubuntu
```

Dentro del contenedor, ejecutamos los siguientes comandos:

```
apt-get update
apt-get install ntp
apt-get install openssl
```

Establecer el hostname al contenedor, hay una línea con la ip del contenedor y el nombre del mismo, que es utilizado como hostname, en nuestro caso

```
172.17.0.2      080dec560726
```

Lo cambiamos por un hostname con el dominio incluido

```
172.17.0.2      ca.salvadorcatalfamo.intra
```

Creamos las carpetas para mejor organización

```
mkdir /root/newcerts
mkdir /root/certs
mkdir /root/crl
mkdir /root/private
mkdir /root/requests
```

Creamos un archivo vacío y un archivo que contiene el primer número de serie para los certificados

```
touch index.txt
echo '1234' > serial
```

Luego hay que crear la llave privada y el certificado root, en este caso nos pedirá una contraseña, si este servidor se usará en un ambiente de producción, deberá ser una contraseña compleja.

```
openssl genrsa -aes256 -out private/cakey.pem
```

Una vez que generamos la llave privada, la misma será utilizada como entrada en la creación de nuestro certificado root. Nos pedirá algunos datos de localización y relacionados a la organización

```
openssl req -new -x509 -key /root/ca/private/cakey.pem
        -out cacert.pem -days 3650
```

Cambiamos los permisos de los archivos que creamos

```
chmod 600 -R /root/ca
```

Realizamos unas modificaciones en el archivo de configuración, donde indicaremos la dirección de los certificados, y algunas opciones de configuración adicionales

```
vim /usr/lib/ssl/openssl.cnf
```

Una vez que realizamos estos pasos, estamos listos para realizar el commit de la imagen, con esto, todos los pasos que realizamos (instalar los paquetes, modificar los archivos de configuración, etc) no son necesarios que se ejecuten nuevamente.

Para realizar un commit, y que nuestro contenedor sea fácilmente identificable, deberemos seguir los siguientes pasos

```
docker ps -a        #identificamos el último contenedor utilizado
docker commit {id_del_contenedor}
docker image ls     #Identificamos la imagen recién creada
                    #no tendrá ni repositorio ni tag
docker image tag {id_de_imagen} ca:1.0  #nombramos nuestra imagen
```

Ahora cada vez que queramos correr nuestra *CA*, lo haremos de la siguiente manera:

```
docker run -it -v $PWD/ca:/root/ca ca:1.0
```

Em el comando anterior, estamos asumiendo que queremos compartir los archivos de la *CA* con el servidor *host*.

Nginx con Docker

Para probar nuestro certificado, utilizaremos una imagen oficial de Nginx, los archivos de configuración son los siguientes:

`docker-compose.yml`

```
web:
  image: nginx
  volumes:
   - ./pagina1:/usr/share/nginx/html:ro
  ports:
   - "80:80"
  environment:
   - NGINX_HOST=pagina1.salvadorcatalfamo.intra
   - NGINX_PORT=80
```

En el archivo mostrado, le decimos a *Docker* que utilice la imagen Nginx, que nuestros archivos fuentes van a estar en el directorio *./pagina1* y que exponga el puerto 80, entre otras cosas.
Para ejecutar este contenedor, se debe ejecutar el siguiente comando:

```
docker-compose up -d
```

Creando nuestro certificado

Para firmar un certificado, tenemos que seguir los pasos mostrados en la figura 2.2 : el servidor donde se alojará la web debe realizar una solicitud, luego nuestra *CA* retornará el certificado firmado. Desde el servidor web en cuestión, se debe crear una llave privada:

```
openssl genrsa -aes256 -out pagina1.pem 2048
```

Luego, se deberá crear la solicitud de firma de certificado:

```
openssl req -new -key pagina1.pem -out pagina1.csr
```

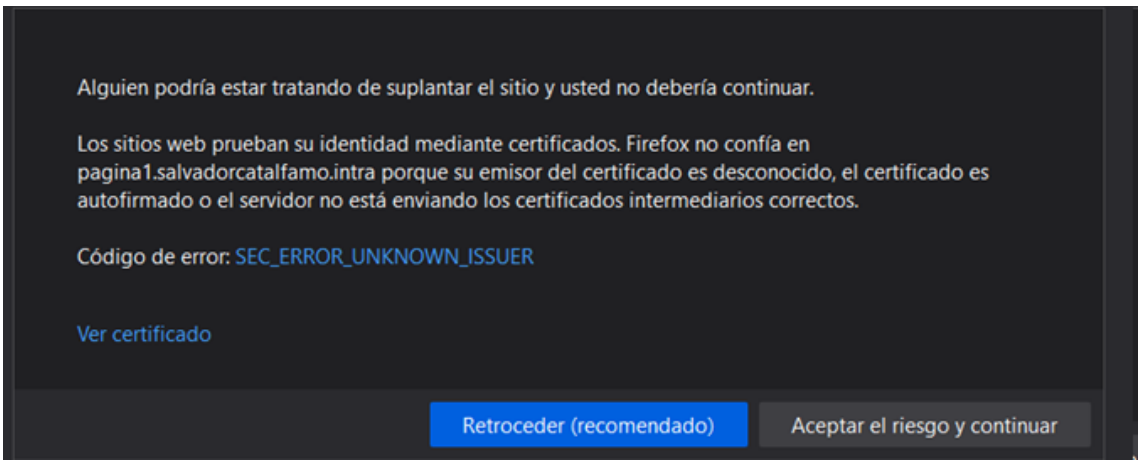


Figura 4.5: CA Desconocida

Luego enviamos esta solicitud y la firmamos en nuestra *CA*, esta solicitud la vamos a colocar en el directorio */root/ca/requests*

```
openssl ca -in pagina1.csr -out pagina1.crt
```

Configuración del certificado en el servidor web

Una vez que tenemos este certificado, lo colocamos en el servidor web y lo configuramos. Para el caso de Nginx, se debe editar el archivo de configuración correspondiente a nuestra web, que en este caso es *pagina1.salvadorcatalfamo.intra* con el fin de que el mismo pueda localizar correctamente los certificados firmados recientemente. Adicionalmente, se puede obligar a que cada requerimiento sea redirigido a una conexión segura mediante *SSL*.

Como se puede ver en la figura 4.5, aunque el certificado esté configurado, el mismo no es confiable ya que nuestra entidad certificante no está importada a nuestro almacén local.

Luego de establecer en nuestra computadora (particularmente en el navegador Mozilla) que la entidad certificante creada es confiable, es posible ver que nuestra conexión es segura, como se muestra en la figura 4.6.

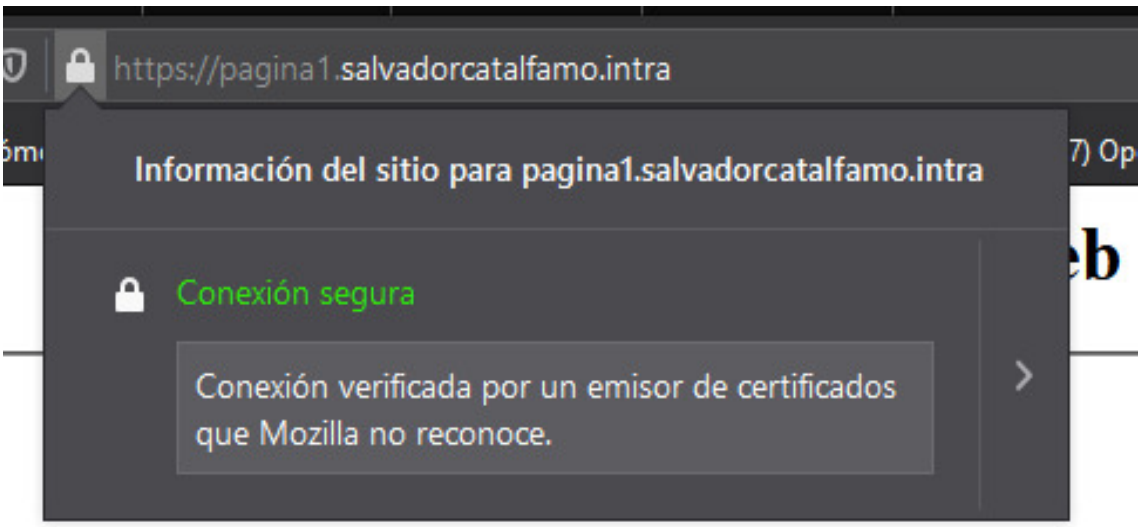


Figura 4.6: Resultado de la solución

Como ventajas de esta propuesta tenemos que no es necesario depender de un agente externo para la obtención de certificados, ya que en muchos casos no es necesario, y en otros, los certificados provistos por estos no son gratuitos.

Por otro lado, al implementar esta solución, se vio una mayor complejidad en la implementación: se requiere un alto entendimiento técnico de todo el ciclo de certificación, y un conocimiento sobre las particularidades que pueden llegar a existir en la generación de certificados y de llaves privadas. Además, hay organizaciones en las que no se tiene el control de todos los equipos que se conectan a la red; por ejemplo, empleados externos, o bien proveedores de servicios, con lo cual, existe la posibilidad de no poder configurar como confiable la entidad certificante en los mismos.

Pese a que esta propuesta es de las más implementadas, decidimos buscar una opción que nos permita desligarnos de ciertas responsabilidades, como así también no tener que realizar configuraciones individuales a los *hosts* de nuestra organización.

4.7. Propuesta 3: Certificación con Let's Encrypt

Esta estrategia consiste en generar un certificado *wildcard*, y utilizarlo en cada sitio de la organización. Para esto se debe tener un dominio (en mi caso, salvadorcatalfamo.com) y demostrar la propiedad del mismo. Como se vio anteriormente, hay dos maneras que utiliza Let's Encrypt para demostrar la propiedad de un dominio, pero la que nos sirve en el caso de una red interna es la que intervienen los registros *DNS*. La verificación que ofrecen con este tipo de certificado es la mínima (DV) y el proceso es explicado a continuación.

4.7.1. Pasos a seguir

Obtener un dominio

El primer paso es conseguir un dominio, en mi caso ya tenía uno: salvadorcatalfamo.com. Este dominio apunta a nuestra ip pública. La configuración *DNS* es la siguiente:

Tipo	**Nombre**	**Contenido**	**Prioridad**	**TTL**
A	salvadorcatalfamo.com	181.228.121.12	0	14400
NS	salvadorcatalfamo.com	ns1.donweb.com	0	14400
SOA	salvadorcatalfamo.com	ns2.donweb.com	0	14400
SOA	salvadorcatalfamo.com	ns3.hostmar.com root.hostmar.com 2021010700 28800 7200 2000000 86400 ns2.donweb.com	0	14400

Instalando Let's Encrypt en el servidor

```
sudo add-apt-repository ppa:certbot/certbotsudo
apt-get update
sudo apt-get install python-certbot-nginx
```

Instalando Nginx

```
sudo apt-get update
sudo apt-get install nginx
```

Obteniendo un certificado *SSL* de tipo *wildcard* desde Let's Encrypt

```
sudo certbot --server https://acme-v02.api.letsencrypt.org/directory
-d *.salvadorcatalfamo.com --manual --preferred-challenges dns-01 certonly
```

Configuración DNS

Luego de ejecutar el comando anterior, Let's Encrypt nos da un contenido que se debe agregar a un registro *DNS*. El tipo de registro es TXT y se muestra en la siguiente tabla.

Tipo	Nombre	Contenido	Prioridad	TTL
TXT	_acme-challenge.salvadorcatalfamo.com	11UZJD27bPDb_jFs6f...	0	14400

Configuración de Nginx para servir a subdominios

Se debe modificar el siguiente archivo de configuración */etc/nginx/sites-available/salvadorcatalfamo.com* como se muestra a continuación:

```
server {
   listen 80;
   listen [::]:80;
   server_name *.salvadorcatalfamo.com;
   return 301 https://$host$request_uri;
}
server {
   listen 443 ssl;
   server_name *.example.com;  ssl_certificate /.../.../fullchain.pem;
   ssl_certificate_key /.../privkey.pem;
   include /etc/letsencrypt/options-ssl-nginx.conf;
```

```
   ssl_dhparam /.../ssl-dhparams.pem;  root /.../salvadorcatalfamo.com;
   index index.html;
   location / {
      try_files $uri $uri/ =404;
   }
}
```

Test la configuración y reinicio del servicio

La configuración se puede testear con el siguiente comando:

```
sudo nginx -t
```

Si tiene éxito, se debe volver a cargar Nginx usando

```
sudo /etc/init.d/nginx reload
```

Ahora Nginx contiene un certificado de tipo *wildcard*, un certificado *SSL* con respaldo de una entidad certificante como Let's Encrypt

Le hemos visto un gran potencial a esta solución, aunque también es poco implementada. Tenemos la gran ventaja de no tener que instalar ningún requerimiento en las computadoras dentro de la organización. Con esto, no se mostrarán mensajes de seguridad en los navegadores, no importa cuál sea el programa, ya que Let's Encrypt es una entidad de confianza para diversos navegadores y sistemas operativos. Todo esto y la seguridad extra al saber que nuestros datos van por un canal seguro gracias al protocolo *SSL*.

Otra gran ventaja que vimos es la facilidad con la que se llevó a cabo, en este proyecto se logró implementar antes esta solución que la entidad certificante, y con mucha mayor facilidad.

Como desventajas podemos decir que no tenemos la posibilidad de obtener la validación extendida (EV), ya que no está disponible actualmente. Por otro lado, que las llaves privadas y el certificado (que es único para todo el dominio) estén en diversos servidores a la vez, implica que se deben tener mayores recaudos a la hora de utilizarlos, ya que se debe asegurar el control de éstos. Aunque a nosotros no nos

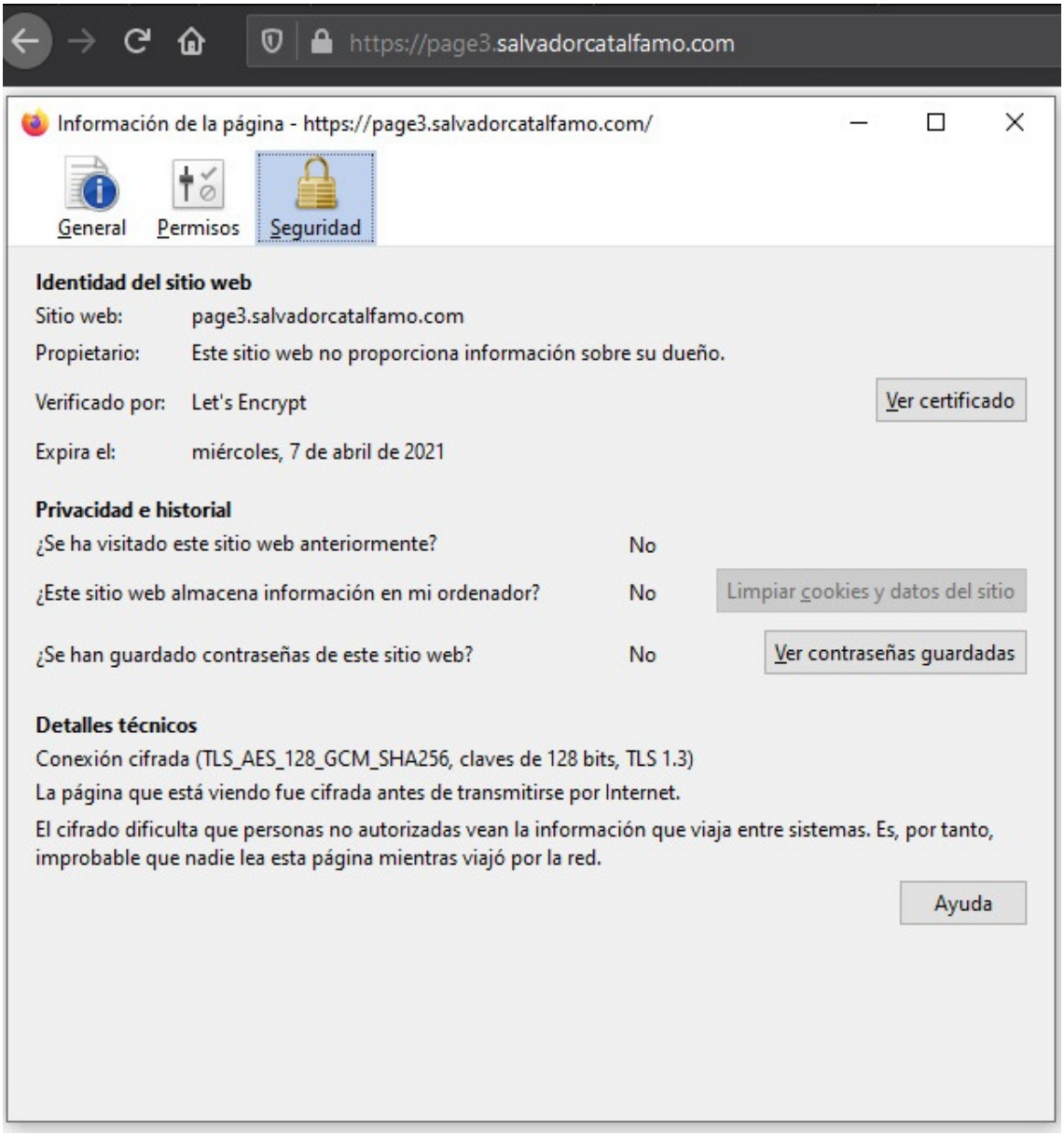


Figura 4.7: Web interna con certificado de Let's Encrypt

sucedió, puede llegar a suceder que si se pierde conexión a *Internet*, el certificado no se pueda validar, producto de no tener toda la cadena de certificación hasta llegar a la raíz.

Aunque se puede llegar a pensar, administrar los certificados y las llaves puede llegar a ser un gran desafío para los administradores de sistema, sin embargo, hay muchas herramientas que nos proveen automatización y monitoreo para realizar esta clase de tareas.

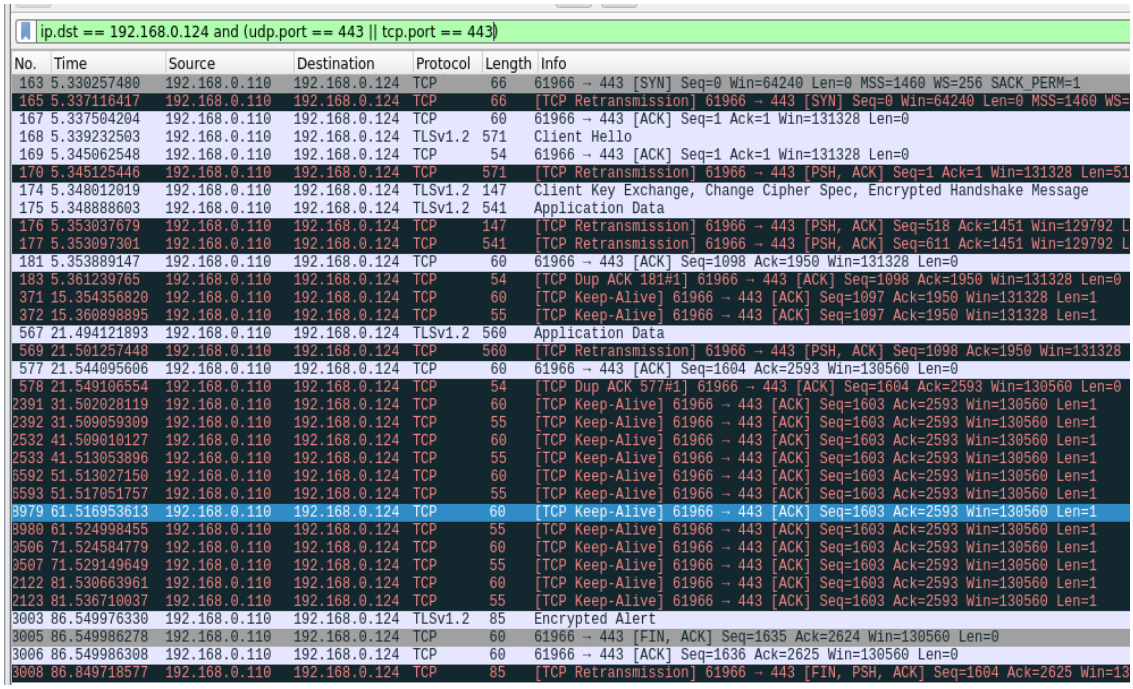
4.8. Caso de estudio: Buscando credenciales en tráfico seguro

Luego de ver las diversas soluciones propuestas, una parte importante de nuestro proyecto fue verificar que verdaderamente aumenta la seguridad cuando nuestro tráfico va encriptado. Para este caso de estudio, se utilizó el mismo formulario propuesto en la sección 3.5, lo único que con servidores en distintas direcciones.

Dado que el proceso de capturar el tráfico en una red interna fue explicado previamente, se van a mostrar únicamente los paquetes capturados desde la primera solicitud hasta el envío del formulario.

En la figura 4.8 podemos observar que:

- No es posible determinar a simple vista cual es el paquete en el cual se envía la información crítica.
- Abriendo e investigando el contenido de cada uno de los paquetes mostrados, tampoco es posible ver las credenciales completadas en el formulario, que obviamente son de nuestro conocimiento.
- Se establece una conexión segura a través del protocolo TCP, algo que no vimos en el caso de estudio de *HTTP*.



ip.dst == 192.168.0.124 and (udp.port == 443 || tcp.port == 443)

No.	Time	Source	Destination	Protocol	Length	Info
163	5.330257480	192.168.0.110	192.168.0.124	TCP	66	61966 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
165	5.337116417	192.168.0.110	192.168.0.124	TCP	66	[TCP Retransmission] 61966 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=
167	5.337504204	192.168.0.110	192.168.0.124	TCP	60	61966 → 443 [ACK] Seq=1 Ack=1 Win=131328 Len=0
168	5.339232503	192.168.0.110	192.168.0.124	TLSv1.2	571	Client Hello
169	5.345062548	192.168.0.110	192.168.0.124	TCP	54	61966 → 443 [ACK] Seq=1 Ack=1 Win=131328 Len=0
170	5.345125446	192.168.0.110	192.168.0.124	TCP	571	[TCP Retransmission] 61966 → 443 [PSH, ACK] Seq=1 Ack=1 Win=131328 Len=51
174	5.348012019	192.168.0.110	192.168.0.124	TLSv1.2	147	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
175	5.348888603	192.168.0.110	192.168.0.124	TLSv1.2	541	Application Data
176	5.353037679	192.168.0.110	192.168.0.124	TCP	147	[TCP Retransmission] 61966 → 443 [PSH, ACK] Seq=518 Ack=1451 Win=129792 L
177	5.353097301	192.168.0.110	192.168.0.124	TCP	541	[TCP Retransmission] 61966 → 443 [PSH, ACK] Seq=611 Ack=1451 Win=129792 L
181	5.353889147	192.168.0.110	192.168.0.124	TCP	60	61966 → 443 [ACK] Seq=1098 Ack=1950 Win=131328 Len=0
183	5.361239765	192.168.0.110	192.168.0.124	TCP	54	[TCP Dup ACK 181#1] 61966 → 443 [ACK] Seq=1098 Ack=1950 Win=131328 Len=0
371	15.354356820	192.168.0.110	192.168.0.124	TCP	60	[TCP Keep-Alive] 61966 → 443 [ACK] Seq=1097 Ack=1950 Win=131328 Len=1
372	15.360898895	192.168.0.110	192.168.0.124	TCP	55	[TCP Keep-Alive] 61966 → 443 [ACK] Seq=1097 Ack=1950 Win=131328 Len=1
567	21.494121893	192.168.0.110	192.168.0.124	TLSv1.2	560	Application Data
569	21.501257448	192.168.0.110	192.168.0.124	TCP	560	[TCP Retransmission] 61966 → 443 [PSH, ACK] Seq=1098 Ack=1950 Win=131328
577	21.544095606	192.168.0.110	192.168.0.124	TCP	60	61966 → 443 [ACK] Seq=1604 Ack=2593 Win=130560 Len=0
578	21.549106554	192.168.0.110	192.168.0.124	TCP	54	[TCP Dup ACK 577#1] 61966 → 443 [ACK] Seq=1604 Ack=2593 Win=130560 Len=0
2391	31.502028119	192.168.0.110	192.168.0.124	TCP	60	[TCP Keep-Alive] 61966 → 443 [ACK] Seq=1603 Ack=2593 Win=130560 Len=1
2392	31.509059309	192.168.0.110	192.168.0.124	TCP	55	[TCP Keep-Alive] 61966 → 443 [ACK] Seq=1603 Ack=2593 Win=130560 Len=1
2532	41.509010127	192.168.0.110	192.168.0.124	TCP	60	[TCP Keep-Alive] 61966 → 443 [ACK] Seq=1603 Ack=2593 Win=130560 Len=1
2533	41.513053896	192.168.0.110	192.168.0.124	TCP	55	[TCP Keep-Alive] 61966 → 443 [ACK] Seq=1603 Ack=2593 Win=130560 Len=1
6592	51.513027150	192.168.0.110	192.168.0.124	TCP	60	[TCP Keep-Alive] 61966 → 443 [ACK] Seq=1603 Ack=2593 Win=130560 Len=1
6593	51.517051757	192.168.0.110	192.168.0.124	TCP	55	[TCP Keep-Alive] 61966 → 443 [ACK] Seq=1603 Ack=2593 Win=130560 Len=1
8979	61.516953613	192.168.0.110	192.168.0.124	TCP	60	[TCP Keep-Alive] 61966 → 443 [ACK] Seq=1603 Ack=2593 Win=130560 Len=1
8980	61.524998455	192.168.0.110	192.168.0.124	TCP	55	[TCP Keep-Alive] 61966 → 443 [ACK] Seq=1603 Ack=2593 Win=130560 Len=1
9506	71.524584779	192.168.0.110	192.168.0.124	TCP	60	[TCP Keep-Alive] 61966 → 443 [ACK] Seq=1603 Ack=2593 Win=130560 Len=1
9507	71.529149649	192.168.0.110	192.168.0.124	TCP	55	[TCP Keep-Alive] 61966 → 443 [ACK] Seq=1603 Ack=2593 Win=130560 Len=1
2122	81.530663961	192.168.0.110	192.168.0.124	TCP	60	[TCP Keep-Alive] 61966 → 443 [ACK] Seq=1603 Ack=2593 Win=130560 Len=1
2123	81.536710037	192.168.0.110	192.168.0.124	TCP	55	[TCP Keep-Alive] 61966 → 443 [ACK] Seq=1603 Ack=2593 Win=130560 Len=1
3003	86.549976330	192.168.0.110	192.168.0.124	TLSv1.2	85	Encrypted Alert
3005	86.549986278	192.168.0.110	192.168.0.124	TCP	60	61966 → 443 [FIN, ACK] Seq=1635 Ack=2624 Win=130560 Len=0
3006	86.549986308	192.168.0.110	192.168.0.124	TCP	60	61966 → 443 [ACK] Seq=1636 Ack=2625 Win=130560 Len=0
3008	86.849718577	192.168.0.110	192.168.0.124	TCP	85	[TCP Retransmission] 61966 → 443 [FIN, PSH, ACK] Seq=1604 Ack=2625 Win=13

Figura 4.8: Tráfico capturado

Capítulo 5

Conclusiones y Trabajos a Futuro

En este proyecto abordamos una amplia cantidad de temas de manera resumida, con un gran potencial de estudio por delante. La navegación *web* nos permitió encontrar una extensa cantidad conceptos para desarrollar, tales como los protocolos *HTTP*, *HTTPS*, *DNS*, y los posibles procesos que se pueden realizar para segurizar nuestra red.

Con respecto a *Docker*, quiero destacar la amplia comunidad existente, ya que ésta nos brinda muchas facilidades, tales como ejemplo e instructivos que nos ayuda a la hora de trabajar. El tiempo que nos ahorramos con *Docker* fue destinado a entender el funcionamiento de la seguridad en las redes, tales como el uso de los certificados *SSL* y las entidades certificantes.

A pesar del contenido teórico propuesto, el trabajo se concentró en las soluciones presentadas en el capítulo 4. Allí se explicó brevemente como con un corto conocimiento en el funcionamiento del protocolo *SSL* se pueden establecer mecanismos para que el tráfico *web* vaya de manera segura; uno de nuestros principales objetivos, junto con el de concientizar a las personas de la importancia de conocer los sitios por los que se navega. Aunque en nuestro caso abordamos las redes pertenecientes a una organización, el contenido de la información es válido para cualquier sitio *web* que se visita, ya sea interno o externo.

Como continuación de este trabajo, existen diversos temas de investigación en los que es posible continuar trabajando. Como mencionamos al comienzo del manuscrito, en las redes internas, además de los datos que circulan producto de la navegación web, también existe otro tipo de tráfico, como correos electrónicos, archivos com-

pletos de configuración o bien multimedia, como así también las consultas DNS que se realizar a estos servidores dedicados. En diversas ocasiones, este tipo de tráfico va sin cifrar, y, hablando particularmente del correo electrónico, es más común de lo que parece, ya que tampoco fue diseñado pensando en la seguridad. El estudio de todo el circuito que conlleva desde el envío de un correo, hasta la recepción del mismo quedará pendiente para trabajos futuros, ya que puede llegar a tomar tanto como este proyecto mismo.

Con respecto a las propuestas de solución, una de ellas fue la creación de una entidad certificante interna. La gestión de la creación de los certificados fue completamente manual, esto, es fácilmente automatizable, estableciendo estándares de seguridad, tales como herramientas de cifrado de datos, autenticación y registro de los certificados generados.

Bibliografía

[1] Ansari, J. A. *Web Penetration Testing with Kali Linux, Second Edition.* Packt Publishing, 2015.

[2] Charles P. Pfleeger, Shari Lawrence Pfleeger, J. M. *Security in Computing, Fifth Edition.* Pearson Education, 2015.

[3] Jain, S. M. *Linux Containers and Virtualization.* Apress Media, 2020.

[4] Lee Allen, Tedi Heriyanto, S. A. *Kali Linux – Assuring Security by Penetration Testing.* Packt Publishing, 2014.

[5] Mukhopadhyay, M. C. M. S. V. E. B. I. *The Essence of Network Security: An End-to-End Panorama.* Springer Nature Singapore, 2021.

[6] Pollard, B. *HTTP/2 in Action.* Manning Publications, 2019.

[7] RFC-2660. *The Secure HyperText Transfer Protocol.* IETF, 1999.

[8] RFC-7230. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing.* IETF, 2014.

[9] RFC-8555. *Automatic Certificate Management Environment (ACME).* IETF, 2019.

[10] Ristić, I. *Bulletproof SSL and TLS.* Feisty Duck, 2014.

[11] Thomas, S. A. *SSL and TLS Essentials.* John Wiley and Sons, Inc., 2000.

[12] Vocale, L. F. M. *Hands-On Cloud-Native Microservices with Jakarta EE.* Packt Publishing, 2019.

[13] VON HAGEN, W. *Professional Xen Virtualization*. Wiley Publishing, Inc, 2008.