Aplicaciones Criptográficas

Acceso web seguro (HTTPS)

Integrantes:

Condolo Narvaez	Byron Paul
Lascano Puruncajas	Angelo Damian
Loya Cadena	Bryan Eduardo
Rosero Lema	Ruth Estefania
Tapia Rea	Freddy Xavier
Trujillo Vistin	Dennis Adrian

Fecha: 30 de noviembre de 2025

Universidad Central del Ecuador
Facultad de Ingeniería y Ciencias Aplicadas
Cátedra Criptográfia y seguridad de la información

Resumen

El presente informe, desarrollado por el **Grupo 6**, tiene como objetivo analizar de manera integral la aplicación criptográfica denominada **Acceso Web Seguro (HTTPS)**, abordando sus fundamentos teóricos, sus mecanismos criptográficos subyacentes y su relevancia en la protección de las comunicaciones en Internet. HTTPS constituye actualmente el estándar predominante para el establecimiento de canales seguros entre clientes y servidores web, sustentado principalmente en el protocolo Transport Layer Security (TLS) y en la Infraestructura de Clave Pública (PKI), elementos esenciales para garantizar autenticidad, confidencialidad e integridad de los datos transmitidos.

El documento inicia con una definición formal de la tecnología y con una descripción detallada de los *algoritmos criptográficos* involucrados, tales como RSA, ECDSA, Diffie–Hellman y su variante de curvas elípticas (ECDHE), así como los esquemas de cifrado simétrico modernos utilizados en las suites criptográficas actuales (AES-GCM, ChaCha20-Poly1305). Posteriormente, se presenta un inventario exhaustivo de los *protocolos criptográficos* que intervienen en el establecimiento del canal seguro, destacándose TLS 1.2 y TLS 1.3, el manejo de certificados X.509, los mecanismos de revocación (OCSP, CRL) y los estándares de seguridad complementarios (HSTS, ALPN).

Asimismo, se incluye un **diseño esquemático del proceso de establecimiento de sesión**, explicando cada etapa del *handshake TLS*, la validación de certificados y la generación de claves efímeras, elementos que permiten propiedades avanzadas como el *Perfect Forward Secrecy*. También se describen tres escenarios comunes donde HTTPS resulta fundamental: plataformas de comercio electrónico, sistemas de banca en línea y APIs modernas para servicios distribuidos.

El informe integra además una sección de **preguntas técnicas orientadas al razonamiento criptográfico**, así como una **implementación práctica** basada en Node.js, que permite demostrar el funcionamiento de HTTPS mediante la configuración de un servidor seguro y la generación de certificados utilizando herramientas criptográficas estándar. Este ejercicio permite a los estudiantes consolidar su comprensión teórica a través de una experiencia aplicada.

Finalmente, el trabajo se apoya en un conjunto de **referencias científicas en formato IEEE con DOI**, que proveen validez académica al análisis y permiten profundizar en aspectos avanzados del protocolo TLS, la gestión de certificados y la seguridad en la infraestructura web contemporánea.

En suma, este informe constituye un estudio técnico riguroso que articula teoría, práctica y análisis crítico, alineándose con los objetivos formativos de la Cátedra de Criptografía y Seguridad de la Información. Su elaboración fortalece las competencias del estudiante en criptografía aplicada, diseño de sistemas seguros y evaluación de tecnologías destinadas a la protección de la información en entornos digitales.

Índice

1. Introducción

1.1. Contexto

La creciente dependencia de servicios y aplicaciones web en las actividades cotidianas, comerciales y gubernamentales ha convertido la protección de la información en tránsito en una necesidad crítica. Transacciones financieras, intercambio de credenciales, transmisión de datos personales y comunicaciones entre servicios distribuidos se realizan a través de redes públicas cuyo modelo de seguridad no garantiza, por sí mismo, confidencialidad ni integridad. En este escenario, los protocolos criptográficos que aseguran las comunicaciones —particularmente TLS (Transport Layer Security) y su uso sobre HTTP mediante HTTPS— constituyen la piedra angular para mitigar riesgos como la intercepción de datos, la suplantación de identidades y la manipulación de mensajes por actores maliciosos.

1.2. Justificación

El objetivo pedagógico y técnico de este trabajo se fundamenta en la necesidad de comprender no sólo los principios teóricos de la criptografía aplicada a las comunicaciones sino también su implementación práctica y su correcta configuración en entornos reales. HTTPS, como aplicación directa de TLS y de la Infraestructura de Clave Pública (PKI), ofrece un caso de estudio completo para observar el funcionamiento de algoritmos asimétricos y simétricos, mecanismos de firma digital, gestión de certificados X.509, y protocolos de verificación de estado (CRL/OCSP). La realización de este informe por parte del **Grupo 6** permite a los participantes consolidar conocimientos teóricos, practicar con herramientas reales (OpenSSL, Node.js, Express) y evaluar riesgos y contramedidas aplicables en contextos académicos y profesionales.

1.3. Objetivos

Objetivo general

Analizar de forma integral la tecnología *Acceso Web Seguro (HTTPS)* desde sus fundamentos criptográficos hasta su implementación práctica, con el propósito de comprender y demostrar los mecanismos que aseguran la confidencialidad, integridad y autenticidad en las comunicaciones web.

Objetivos específicos

1. Describir y clasificar los algoritmos criptográficos (asimétricos, simétricos y de hashing) que intervienen en HTTPS.
2. Explicar los protocolos y mecanismos complementarios (TLS 1.2/1.3, PKI, OCSP, HSTS, ALPN) y su rol en la seguridad de las comunicaciones.
3. Diseñar esquemáticamente el flujo de establecimiento de sesión segura (handshake TLS) y la estructura del certificado X.509.
4. Implementar una demostración práctica de servidor HTTPS empleando Node.js y certificados generados con OpenSSL, documentando pasos reproducibles.

5. Evaluar escenarios de aplicación y proponer recomendaciones de buenas prácticas para despliegues seguros en producción.

1.4. Alcance

El informe se centra en el estudio de HTTPS como aplicación práctica de TLS y de la PKI en entornos web. Incluye:

- Análisis teórico de algoritmos y protocolos relevantes (RSA, ECDHE, AES-GCM, ChaCha20-Poly1305, SHA-2, etc.).
- Diseño esquemático detallado del handshake TLS y la estructura de certificados X.509.
- Implementación práctica con un servidor HTTPS en Node.js (Express), uso de certificados autofirmados para fines educativos y explicación de la generación mediante OpenSSL.
- Discusión de tres escenarios representativos de uso y análisis de amenazas y contramedidas.
- Bibliografía académica en formato IEEE con DOIs que respalde el contenido.

Quedan fuera del alcance los procedimientos de emisión de certificados por CAs comerciales en entornos productivos y la implementación completa de infraestructuras de revocación a escala (aunque se discuten los conceptos y mecanismos disponibles).

1.5. Metodología

La metodología empleada combina revisión bibliográfica, análisis técnico y trabajo experimental:

1. **Revisión bibliográfica**: selección y lectura crítica de artículos científicos, RFCs y documentación técnica relevante para fundamentar los conceptos y prácticas descritas (referencias en formato IEEE).
2. **Análisis técnico**: descomposición del protocolo TLS y de la estructura X.509 en sus componentes y procesos fundamentales, con apoyo en diagramas y tablas comparativas.
3. **Implementación práctica**: desarrollo y configuración de un servidor HTTPS en Node.js usando Express, generación de certificados con OpenSSL y validación funcional mediante pruebas básicas (curl, navegadores, comprobaciones de fingerprints).
4. **Evaluación y reflexión**: identificación de amenazas, limitaciones del enfoque de laboratorio (por ejemplo, certificados autofirmados) y recomendaciones para despliegues seguros en producción.

1.6. Estructura del documento

El informe se organiza en secciones que guían al lector desde los fundamentos teóricos hasta la implementación práctica y las conclusiones:

- **Resumen**: síntesis ejecutiva del contenido y resultados del trabajo.
- **Introducción**: contexto, justificación, objetivos, alcance y metodología (presente).
- **Definición de la tecnología**: descripción formal de HTTPS, TLS y PKI.
- **Algoritmos criptográficos**: inventario y análisis de algoritmos empleados en HTTPS.
- **Protocolos criptográficos**: revisión de TLS 1.2/1.3, OCSP, CRL, HSTS, ALPN y otros.
- **Diseño esquemático**: diagramas del handshake, estructura de certificados y flujos de comunicación.
- **Escenarios de uso**: tres casos de aplicación con análisis de amenazas y contramedidas.
- **Preguntas técnicas**: interrogantes para profundización académica y discusión.
- **Implementación práctica**: guía detallada del servidor HTTPS con Node.js y generación de certificados.
- **Conclusiones y recomendaciones**: síntesis de hallazgos y buenas prácticas.
- **Bibliografía y anexos**: referencias en formato IEEE y materiales adicionales (comandos OpenSSL, logs, capturas).

1.7. Consideraciones finales

Este documento aspira a ser tanto un recurso académico riguroso como una guía práctica reproducible para estudiantes y profesionales que deseen comprender y experimentar con los mecanismos criptográficos que sostienen la seguridad de las comunicaciones web. La implementación incluida en el apartado práctico está orientada a fines de laboratorio y aprendizaje; para entornos productivos se deberán seguir procesos de validación, gestión de claves y emisión de certificados por autoridades certificadoras reconocidas, así como políticas de mantenimiento y monitoreo continuos.

2. Definición de la Tecnología HTTPS

El Protocolo Seguro de Transferencia de Hipertexto (HTTPS, por sus siglas en inglés *Hypertext Transfer Protocol Secure*) constituye la principal tecnología criptográfica empleada para garantizar comunicaciones seguras en la web moderna. Su función esencial radica en proporcionar un canal cifrado, autenticado e íntegro entre

un cliente —comúnmente un navegador— y un servidor web. HTTPS se implementa mediante la superposición del protocolo HTTP sobre una capa criptográfica denominada *Transport Layer Security* (TLS), cuyo diseño moderno y estandarización se determinan principalmente en la especificación RFC 8446 [1].

Desde una perspectiva funcional, HTTPS permite mitigar amenazas como espionaje, manipulación del tráfico y suplantación de identidad, problemas ampliamente documentados en los análisis formales y empíricos del protocolo TLS [2, 3]. La adopción generalizada de HTTPS ha convertido este estándar en un componente crítico para la infraestructura global de comunicaciones, soportando desde servicios gubernamentales hasta sistemas bancarios, plataformas comerciales y entornos académicos.

En las siguientes subsecciones se expone una definición estructurada, abarcando su origen, sus fundamentos criptográficos y su relevancia en el ecosistema digital contemporáneo.

2.1. Origen y Evolución de HTTPS

HTTPS surge como respuesta a las limitaciones de seguridad presentes en el protocolo HTTP original, el cual carecía de mecanismos para evitar la intercepción o la manipulación de datos en tránsito. La introducción de SSL (Secure Sockets Layer) en la década de 1990 constituyó la primera aproximación para resolver estas vulnerabilidades, aunque versiones posteriores demostraron deficiencias críticas. Esto motivó la transición hacia TLS, una familia de protocolos diseñada con mayores garantías formales y sometida a exhaustivas validaciones criptográficas, tales como las realizadas en TLS 1.2 y TLS 1.3 [4, 3].

La última versión ampliamente adoptada, TLS 1.3, representa un avance significativo al reducir la complejidad del *handshake*, mejorar la eficiencia del establecimiento de sesión y reforzar la resistencia frente a ataques contemporáneos, incluyendo técnicas avanzadas de interceptación de tráfico [2]. La evolución de HTTPS ha sido impulsada no solo por mejoras criptográficas, sino también por la necesidad de garantizar un ecosistema de certificados más confiable, como lo evidencian estudios a gran escala sobre revocación y validez instalada en la infraestructura global de claves públicas (PKI) [5, 6].

2.2. Concepto Operacional de HTTPS

En términos operativos, HTTPS no constituye un protocolo independiente, sino un mecanismo de transporte seguro para HTTP. El proceso ocurre mediante el encapsulamiento de las solicitudes y respuestas dentro de un túnel cifrado administrado por TLS. Este túnel se establece mediante un *handshake* criptográfico que cumple tres objetivos fundamentales:

1. Autenticación del servidor (y opcionalmente del cliente) mediante certificados digitales X.509 conforme al estándar RFC 5280 [7].

2. Negociación de algoritmos criptográficos robustos para cifrado, integridad y derivación de claves, siguiendo las recomendaciones de uso seguro definidas en RFC 7525 [8].

3. Creación de claves de sesión efímeras empleando generalmente esquemas de intercambio basados en Diffie–Hellman autenticado.

Una vez completado el proceso de negociación, todas las comunicaciones HTTP quedan protegidas mediante cifrado simétrico, lo que garantiza confidencialidad, integridad y autenticidad extremo a extremo.

2.3. Importancia de HTTPS en la Seguridad de la Información

HTTPS se ha convertido en un componente esencial para garantizar la seguridad en entornos digitales que dependen de servicios distribuidos, aplicaciones web y arquitecturas basadas en microservicios. En este último caso, la autenticación mutua mediante TLS es un elemento clave para prevenir ataques de manipulación interna y fallas entre servicios, lo cual ha sido documentado en estudios sobre amenazas inter-servicio [9]. Asimismo, investigaciones sobre la seguridad de inclusiones externas y recursos de terceros han demostrado que HTTPS contribuye a reducir la superficie de ataque asociada a contenido remoto potencialmente malicioso [10].

La combinación de estas garantías convierte a HTTPS en una tecnología indispensable para la protección de datos personales, credenciales, sesiones de usuario, transacciones electrónicas y comunicaciones institucionales. Su adopción ya se considera un requisito mínimo en prácticas de seguridad de la información, auditoría digital y cumplimiento normativo.

2.4. Síntesis de la Definición

En resumen, HTTPS es un mecanismo criptográfico integral que proporciona seguridad robusta para la comunicación web mediante el uso del protocolo TLS para cifrar, autenticar y asegurar la integridad de los datos. Su diseño ha sido sometido a extensas pruebas formales, su funcionamiento está regulado por estándares internacionales y su adopción constituye un pilar esencial para la protección de la información en el entorno digital contemporáneo.

3. Algoritmos Criptográficos Empleados en HTTPS

El funcionamiento seguro del protocolo HTTPS depende de un conjunto de algoritmos criptográficos que trabajan de forma coordinada para garantizar confidencialidad, integridad, autenticación y resistencia ante ataques avanzados. Estos algoritmos, seleccionados y estandarizados por entidades como el IETF y el NIST, son implementados durante el *handshake* y en la protección de los datos transportados mediante *Transport Layer Security* (TLS). Su correcta elección e implementación se consideran factores críticos para la solidez del canal seguro [1, 8].

En esta sección se describen los algoritmos fundamentales utilizados en HTTPS, incluyendo sus propiedades, su rol dentro del protocolo y su relevancia en el ecosistema criptográfico contemporáneo.

3.1. Cifrado Simétrico

El cifrado simétrico constituye el mecanismo principal para garantizar la **confidencialidad** de los datos transmitidos una vez establecida la sesión HTTPS. Entre los algoritmos recomendados y empleados en TLS 1.3 destacan:

- **AES-GCM (Advanced Encryption Standard - Galois/Counter Mode)**: Proporciona cifrado autenticado y es la opción preferida en TLS debido a su eficiencia y resistencia frente a ataques criptográficos avanzados [1]. Al combinar cifrado en modo contador y autenticación mediante Galois, reduce la necesidad de mecanismos adicionales para garantizar integridad.
- **ChaCha20-Poly1305**: Algoritmo de cifrado autenticado diseñado para ser eficiente en dispositivos sin aceleración AES, como teléfonos móviles o sistemas embebidos [4]. Su seguridad está sustentada en análisis matemáticos rigurosos y su rendimiento es consistente en arquitecturas heterogéneas.

TLS 1.3 elimina algoritmos considerados débiles o inseguros, como RC4 o 3DES, mitigando vulnerabilidades ampliamente documentadas en versiones previas del protocolo [8].

3.2. Intercambio de Claves

El intercambio de claves tiene como objetivo permitir que cliente y servidor acuerden material criptográfico sin exponerlo a un posible adversario que controle el canal. En TLS 1.3 este proceso se basa exclusivamente en mecanismos con **Perfect Forward Secrecy (PFS)**, lo cual garantiza que la seguridad de sesiones pasadas no se vea comprometida incluso si un atacante obtiene claves privadas del servidor.

Los algoritmos principales son:

- **ECDHE (Elliptic Curve Diffie–Hellman Ephemeral)**: Esquema de intercambio de claves ampliamente adoptado debido a su eficiencia y a la fortaleza matemática de las curvas elípticas recomendadas por estándares como NIST P-256 o Curve25519 [4]. Su naturaleza efímera proporciona PFS de forma nativa.
- **DHE (Diffie–Hellman Ephemeral)**: Variante tradicional basada en aritmética modular. Aunque más lenta que ECDHE, continúa siendo soportada por compatibilidad y también ofrece PFS.

Los estudios formales han validado la seguridad de estos esquemas bajo modelos criptográficos rigurosos, incluyendo escenarios adversariales avanzados [2].

3.3. Algoritmos de Hash

Las funciones hash se emplean en múltiples partes del protocolo TLS: generación de claves, autenticación de mensajes, verificación de integridad y construcción de estructuras criptográficas internas. Las funciones aprobadas por los estándares modernos incluyen:

- **SHA-256** y **SHA-384**: Pertenecientes a la familia SHA-2, constituyen la base para la mayoría de las operaciones de autenticación y derivación de claves en TLS 1.3 [1]. Su diseño resistente a colisiones, preimágenes y segundos preimágenes los posiciona como elementos centrales del ecosistema criptográfico actual.

TLS prohíbe el uso de funciones hash inseguras como SHA-1 debido a ataques de colisión demostrados experimentalmente.

3.4. Firmas Digitales

Las firmas digitales son utilizadas para autenticar la identidad del servidor, validar certificados X.509 y garantizar la integridad del handshake. Los algoritmos predominantes incluyen:

- **RSA (Rivest–Shamir–Adleman)**: Tradicionalmente el algoritmo más utilizado para certificados TLS. Su seguridad se basa en la dificultad del problema de factorización de enteros. Para garantizar niveles de seguridad adecuados se requieren claves de al menos 2048 bits [7].
- **ECDSA (Elliptic Curve Digital Signature Algorithm)**: Alternativa moderna y eficiente basada en criptografía de curvas elípticas. Ofrece firmas cortas, validación rápida y alto nivel de seguridad con claves de menor tamaño en comparación con RSA [4].

Las recomendaciones contemporáneas favorecen el empleo de ECDSA en nuevos despliegues debido a su mejor desempeño y menor huella computacional.

3.5. Mecanismos de Autenticación y MACs

Para la autenticación de mensajes TLS emplea MACs (Message Authentication Codes) integrados dentro de los algoritmos AEAD (Authenticated Encryption with Associated Data). Los principales son:

- **GCM Tag** (en AES-GCM): Resultado del proceso autenticado que valida integridad y autenticidad del mensaje.
- **Poly1305** (en ChaCha20-Poly1305): MAC altamente eficiente y matemáticamente robusto asociado al cifrado ChaCha20 [4].

El uso de AEAD simplifica la arquitectura del protocolo y elimina clases enteras de ataques asociados a MACs mal integrados.

3.6. Síntesis de los Algoritmos Criptográficos

En conjunto, los algoritmos criptográficos de HTTPS conforman un sistema robusto diseñado para enfrentar ataques contemporáneos y garantizar seguridad a largo plazo. TLS 1.3 adopta únicamente algoritmos modernos y elimina mecanismos vulnerables, lo que representa un avance significativo en la protección de comunicaciones digitales. Su diseño modular permite incorporar futuras mejoras criptográficas sin comprometer la interoperabilidad y la solidez matemática del ecosistema.

4. Protocolos Criptográficos Fundamentales en HTTPS

El funcionamiento de HTTPS descansa sobre un conjunto de protocolos criptográficos que operan de manera coordinada para asegurar las propiedades de confidencialidad, integridad, autenticación y protección contra ataques avanzados. Estos protocolos, definidos principalmente en el marco de la familia *Transport Layer Security* (TLS), constituyen la columna vertebral de las comunicaciones seguras en la web

contemporánea y han sido sometidos a exhaustivos análisis formales y evaluaciones empíricas [1, 2].

En esta sección se describen los principales protocolos criptográficos que intervienen en HTTPS, abordando su estructura, funcionamiento interno y su relevancia dentro del ecosistema de seguridad digital.

4.1. Transport Layer Security (TLS)

TLS es el protocolo criptográfico central de HTTPS y proporciona un canal seguro mediante el cual se transporta HTTP. Su diseño modular permite separar claramente las etapas de negociación, autenticación y protección de datos. La especificación vigente de TLS 1.3, definida en RFC 8446, introduce mejoras sustanciales en seguridad, desempeño y simplicidad arquitectónica respecto a versiones anteriores [1].

TLS funciona mediante dos componentes principales:

1. **Handshake TLS**: Encargado de la autenticación de las partes, la negociación de algoritmos criptográficos y el establecimiento de claves de sesión con *Perfect Forward Secrecy*. TLS 1.3 reduce la complejidad del handshake tradicional, elimina algoritmos inseguros y minimiza rondas de comunicación, lo cual incrementa la seguridad y reduce la latencia.

2. **Record Protocol**: Encargado de encapsular y proteger los datos de aplicación mediante cifrado autenticado (AEAD). Asegura que cada mensaje cuente con integridad, confidencialidad y autenticidad mediante el uso de algoritmos robustos como AES-GCM o ChaCha20-Poly1305 [8].

La arquitectura general de TLS ha sido objeto de análisis formales rigurosos que validan sus propiedades criptográficas bajo modelos adversariales de alto nivel [3].

4.2. Handshake TLS 1.3

El *handshake* de TLS 1.3 constituye una de las transformaciones más significativas en la evolución del protocolo. A diferencia de versiones anteriores, TLS 1.3 implementa un proceso simplificado que consta de menos mensajes y elimina mecanismos considerados vulnerables, como RSA para intercambio de claves o el uso de suites sin PFS [4].

Los objetivos fundamentales del handshake son:

- **Autenticación del servidor** mediante certificados digitales X.509 basados en RSA o ECDSA.
- **Establecimiento de claves efímeras** mediante ECDHE o DHE, lo cual garantiza Perfect Forward Secrecy.
- **Derivación jerárquica de claves** usando HKDF (HMAC-based Key Derivation Function) en conjunto con funciones hash de la familia SHA-2.
- **Validación de integridad y autenticidad** mediante firmas digitales y MACs integrados en los algoritmos AEAD.

TLS 1.3 también introduce mecanismos optimizados como:

- **0-RTT (Zero Round Trip Time)** para acelerar la conexión en sesiones previamente establecidas, aunque con advertencias de seguridad debido a la posibilidad de ataques de repetición.
- **Key Update** para rotación continua de claves durante sesiones prolongadas.

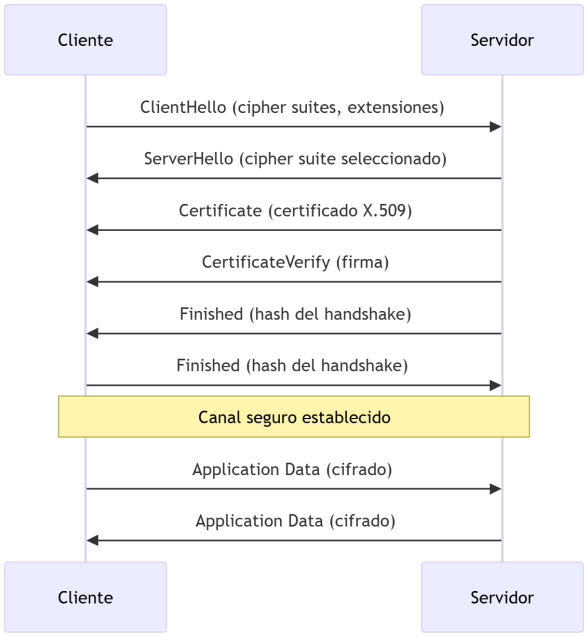

Figura 1: Flujo del Handshake TLS 1.3 en HTTPS

4.3. X.509 y la Infraestructura de Clave Pública (PKI)

La autenticación del servidor en HTTPS depende de la infraestructura global de certificados digitales X.509, definida en el estándar RFC 5280 [7]. La PKI constituye un ecosistema complejo que incluye Autoridades Certificadoras (CA), listas de revocación y mecanismos de validación en tiempo real.

Los componentes fundamentales incluyen:

- **Certificados X.509**: Documentos firmados criptográficamente que vinculan la clave pública del servidor con su identidad.
- **Cadena de confianza**: Secuencia jerárquica que conecta el certificado del servidor con una autoridad raíz confiable.
- **OCSP (Online Certificate Status Protocol)**: Protocolo para verificar en tiempo real si un certificado ha sido revocado.
- **CRL (Certificate Revocation List)**: Listados periódicos de certificados inválidos publicados por las CA.

Estudios recientes han revelado limitaciones en la eficacia de los mecanismos de revocación, lo cual motivó mejoras como OCSP Stapling y el impulso hacia certificación de corta duración [5, 6].

4.4. HTTP Strict Transport Security (HSTS)

HSTS es un protocolo complementario que refuerza el uso obligatorio de HTTPS en un dominio. Mediante un encabezado especial enviado por el servidor, el navegador instruye a futuras conexiones a utilizar exclusivamente HTTPS, incluso si el usuario intenta acceder mediante HTTP.

Sus objetivos principales son:

- Prevenir ataques de *downgrade* hacia HTTP.
- Impedir interceptación mediante ataques Man-in-the-Middle en la primera conexión.
- Fortalecer la transición inicial hacia HTTPS en sitios previamente no seguros.

HSTS ha demostrado reducir significativamente el riesgo asociado a redes inseguras o interceptores pasivos, convirtiéndose en una práctica estándar en la seguridad web moderna.

4.5. OCSP Stapling

OCSP Stapling es una mejora del protocolo OCSP tradicional. Permite al servidor adjuntar al handshake una prueba reciente de validez del certificado, firmada por la autoridad emisora. Esta técnica reduce carga sobre las CA, mejora la privacidad del usuario y acelera la validación del certificado.

Debido a fallos documentados en la disponibilidad y confiabilidad de OCSP, el uso de OCSP Stapling se ha convertido en una práctica recomendada para despliegues seguros [5].

4.6. DNS over HTTPS (DoH) y Protocolos Asociados

Aunque no forma parte directa de TLS, DNS over HTTPS (DoH) se ha convertido en un protocolo complementario fundamental para proteger consultas DNS mediante un canal HTTPS seguro. DoH cifra solicitudes DNS, mitigando ataques como espionaje de tráfico, manipulación de resoluciones y filtrado malicioso.

Su estandarización refuerza el ecosistema de privacidad al mover una de las funciones más vulnerables del sistema de nombres hacia un entorno totalmente cifrado.

4.7. Síntesis General de los Protocolos Criptográficos

Los protocolos criptográficos que sustentan HTTPS forman un ecosistema interdependiente orientado a garantizar comunicaciones seguras de extremo a extremo. TLS, la infraestructura PKI, mecanismos de validación, y protocolos complementarios como HSTS y OCSP Stapling se integran para proporcionar un modelo robusto frente a adversarios modernos. Su evolución continua permite asegurar compatibilidad, resiliencia operativa y alineación con las mejores prácticas de seguridad digital contemporánea.

5. Diseño Esquemático y Funcionamiento

En esta sección se presentan los diagramas que ilustran el diseño arquitectónico y los principales flujos de operación del sistema HTTPS implementado en el proyecto. Cada figura se acompaña de una explicación técnica detallada que facilita la comprensión de los componentes, las interacciones y las implicaciones de seguridad de cada etapa.

5.1. Arquitectura General del Sistema

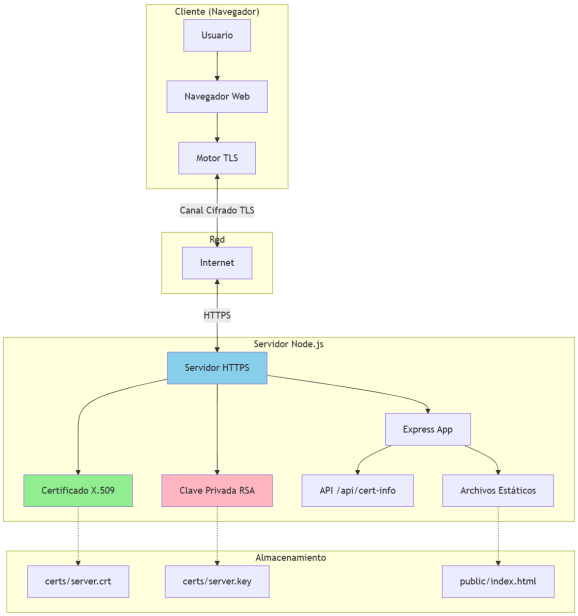

Figura 2: Arquitectura general del sistema: cliente, red, servidor Node.js y almacenamiento de certificados.

Explicación detallada. La figura 2 muestra una visión de alto nivel de los componentes que participan en la provisión de acceso web seguro mediante HTTPS en el entorno del proyecto. Desde la izquierda, el *Cliente (Navegador)* representa el agente que inicia la comunicación —por ejemplo un navegador de escritorio o una aplicación cliente— y que contiene un motor TLS encargado de orquestar el protocolo (negociación de parámetros, verificación de certificados y cifrado/descifrado de datos). El motor TLS actúa como una capa intermedia entre la aplicación (HTTP) y la pila de transporte (TCP/IP), permitiendo que toda la comunicación de la capa de aplicación se beneficie de las garantías criptográficas sin modificar el protocolo HTTP subyacente [1].

En el centro se ubica la *Red / Internet*, que representa el canal potencialmente adversarial. Sobre la derecha se encuentra el *Servidor Node.js*, el componente de ejecución que hospeda la aplicación (Express.js) y que expone servicios HTTP sobre un socket TLS (HTTPS). En él se distinguen claramente dos artefactos críticos: el **Certificado X.509** (archivo `certs/server.crt`) y la **Clave Privada RSA** (archivo `certs/server.key`). El correcto almacenamiento, acceso y permisos de estos ficheros son esenciales: la clave privada debe protegerse con controles de acceso

estrictos y, preferiblemente, con cifrado en reposo y políticas de rotación en entornos productivos [7].

La sección de *Almacenamiento* indica la localización física/estructural de los artefactos (certificado, clave privada, recursos estáticos). En un despliegue real, este almacenamiento puede estar respaldado por módulos HSM o servicios de gestión de claves para incrementar la seguridad y minimizar el riesgo de exposición de la llave privada. Desde el punto de vista funcional, la arquitectura enfatiza la separación de responsabilidades: el servidor HTTPS se encarga del manejo del canal seguro y de proporcionar una interfaz a la aplicación (Express) que permanece, conceptualmente, agnóstica respecto al cifrado de transporte.

5.2. Flujo de Comunicación Completo (Handshakes y Tráfico)

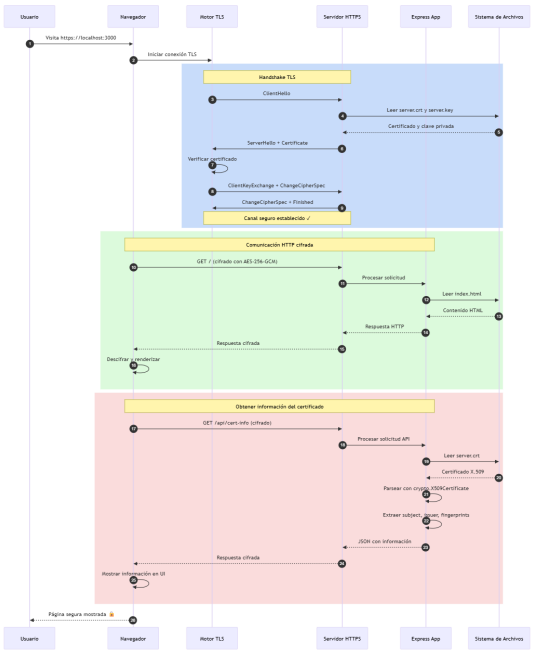

Figura 3: Flujo completo de comunicación: Handshake TLS, establecimiento del canal seguro y tráfico de aplicación cifrado.

Explicación detallada. La figura 3 descompone cronológicamente las interacciones que se producen desde que el usuario solicita la URL hasta que el navegador renderiza la página segura. El proceso puede resumirse en tres fases principales:

1. **Establecimiento del handshake TLS:** El cliente envía un `ClientHello` que contiene las versiones TLS soportadas, cipher suites y extensiones relevantes (SNI, ALPN, etc.). El servidor responde con `ServerHello`, seleccionando los parámetros de cifrado y proporcionando su certificado X.509 en el mensaje `Certificate`. El servidor también realiza la demostración de posesión de la clave privada mediante `CertificateVerify`, permitiendo al cliente validar la autenticidad del servidor. TLS 1.3 reduce el número de intercambios y cifra gran parte del handshake para mejorar privacidad y rendimiento [1, 3].

2. **Negociación de claves y establecimiento del canal seguro:** Tras la verificación del certificado y la negociación de parámetros, se generan claves de sesión efímeras (p. ej. mediante ECDHE) que garantizan *Perfect Forward Secrecy*. A partir de ese momento, el *Record Protocol* utiliza algoritmos AEAD (AES-GCM o ChaCha20-Poly1305) para cifrar y autenticar los registros que transportan los mensajes HTTP [8].

3. **Intercambio de datos de aplicación y gestión de información adicional:** Con el canal seguro establecido, el navegador realiza la petición HTTP (por ejemplo `GET /`) y el servidor responde con contenido estático o dinámico (páginas, APIs). Notar que las comprobaciones adicionales, como la verificación de estado del certificado (OCSP/OCSP Stapling) o la política HSTS, se evalúan conforme a las necesidades de seguridad del servicio [5, 6].

Desde la perspectiva de auditoría y pruebas, cada una de estas fases puede ser instrumentada y registrada (logs de handshake, huellas del certificado, pruebas con `curl` o análisis con `testssl.sh`) para verificar la conformidad del despliegue con las recomendaciones operacionales y los requisitos de seguridad académicos [8].

5.3. Proceso de Generación de Certificados

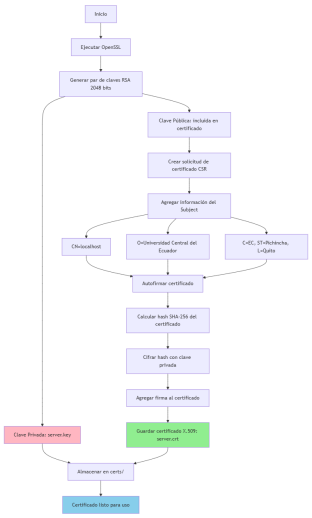

Figura 4: Proceso de generación de certificados X.509 autofirmados mediante OpenSSL (par de claves RSA, CSR, autofirma y almacenamiento).

Explicación detallada. La figura 4 describe el flujo lógico para la generación de un certificado X.509 autofirmado utilizado en el entorno de laboratorio. El proceso consta de los pasos siguientes:

1. **Generación del par de claves (clave privada y clave pública):** Mediante OpenSSL se crea un par RSA (p. ej. 2048 o 4096 bits) o una clave basada en curvas elípticas (p. ej. Curve25519). La clave privada se escribe en un fichero seguro (`server.key`) y su correlato público será parte de la CSR (Certificate Signing Request) o del certificado final [7].

2. **Creación de la CSR (Solicitud de Firma de Certificado):** La CSR contiene la información de identidad (CN, O, OU, L, ST, C) y la clave pública. En un entorno de producción, esta CSR se envía a una CA para validación; en el laboratorio se puede proceder a la autofirma.

3. **Autofirma del certificado:** En la autofirma, el emisor y sujeto son la misma entidad; por tanto, el certificado resultante es firmado con la propia clave privada y carece de la confianza automática que una CA pública proporciona a navegadores y sistemas clientes. Aun así, constituye una herramienta válida para pruebas y enseñanza, permitiendo analizar el contenido del certificado y las firmas digitales que lo componen [7].

4. **Cálculo y adición de la firma:** El certificado incluye una firma digital calculada mediante un algoritmo como `sha256WithRSAEncryption` que cifra el hash (SHA-256) del TBS (To-Be-Signed) certificate con la clave privada del emisor.

5. **Almacenamiento y despliegue:** Finalmente, el certificado (`server.crt`) y la clave privada (`server.key`) se guardan en el directorio de certificados del servidor y se referencian en la configuración de la instancia HTTPS para su carga en tiempo de arranque.

Es importante subrayar las diferencias entre certificados autofirmados y certificados emitidos por CA: los segundos permiten establecer una cadena de confianza globalmente aceptada por los navegadores; los primeros requieren que el cliente añada explícitamente la CA raíz (o se acepte la excepción) y por tanto son apropiados únicamente para entornos controlados de laboratorio [5].

5.4. Estructura de Datos del Certificado X.509

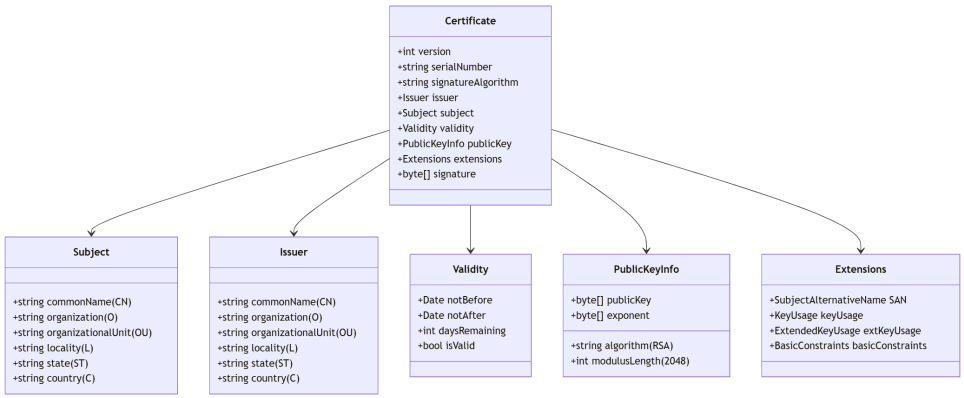

Figura 5: Estructura de datos de un certificado X.509 v3: campos básicos, validez, clave pública y extensiones.

Explicación detallada. La figura 5 desglosa los componentes esenciales de un certificado X.509 versión 3. Cada campo cumple una función específica en la identificación y autorización criptográfica:

Version y Serial Number: Identificadores del formato y del ejemplar del certificado, usados para la gestión y revocación.

Signature Algorithm: Algoritmo empleado para la firma del certificado (p. ej. `sha256WithRSAEncryption`). La seguridad de este campo depende tanto del algoritmo como del tamaño de la clave subyacente [7].

Issuer y Subject: Identificadores de la entidad emisora y del titular (titular = subject). En una CA legítima, la cadena de confianza se construye mediante la relación Issuer → Subject a través de firmas encadenadas.

Validity (notBefore / notAfter): Período de validez del certificado. Exceder este período hará que el certificado sea inválido para la mayoría de clientes.

Subject Public Key Info: Contiene la clave pública y parámetros asociados (algoritmo, longitud de clave). Esta clave es la base para la verificación de firmas y, en combinación con protocolos de intercambio, para la protección de sesiones.

Extensions: Conjunto de extensiones X.509v3 (SAN, KeyUsage, ExtendedKeyUsage, BasicConstraints, etc.) que proporcionan metadatos críticos para el uso correcto del certificado. Por ejemplo, SAN (Subject Alternative Name) define dominios alternativos válidos para el certificado y es fundamental para la validación de nombres por parte de navegadores.

Signature: Firma criptográfica del TBS (To-Be-Signed) certificate generada por la entidad emisora. Su verificación es el paso clave de la validación de confianza en TLS [7].

Comprender la estructura interna del certificado es imprescindible para interpretar resultados de herramientas de análisis (por ejemplo `openssl x509 -text -noout -in server.crt`) y para diseñar políticas de emisión, rotación y revocación de certificados en ambientes reales [5].

5.5. Implementación en Node.js: Integración y Componentes

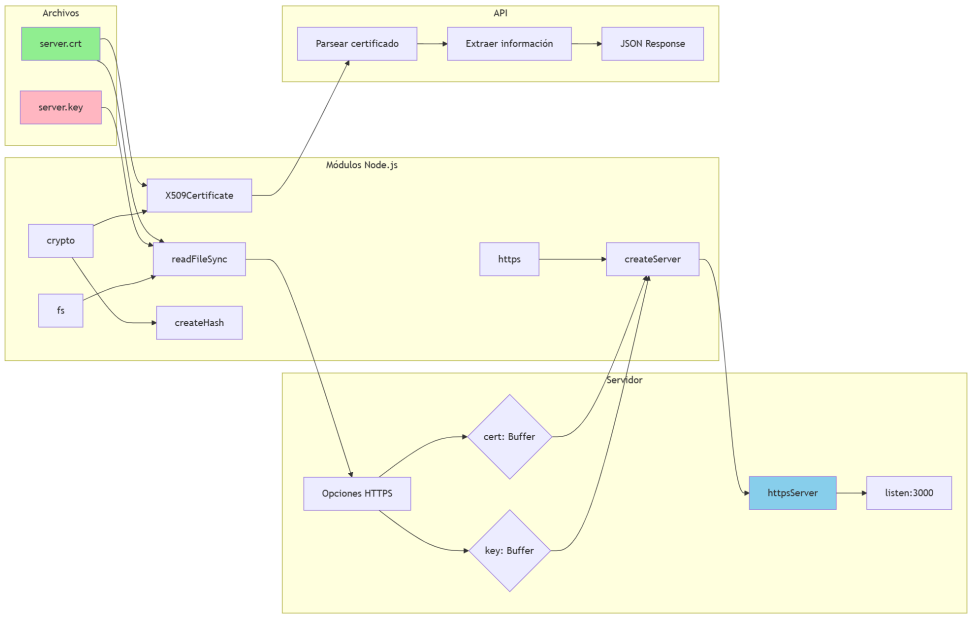

Figura 6: Componentes Node.js implicados en la creación de un servidor HTTPS: módulos, archivos y flujo de carga.

Explicación detallada. La figura 6 ilustra la relación entre los módulos nativos de Node.js (`https`, `fs`, `crypto`), los archivos de certificados en disco y la instancia del servidor que escucha en un puerto seguro (ej. 3000 para desarrollo). El flujo de arranque típico es:

1. **Lectura de artefactos criptográficos:** El servidor, durante su inicialización, emplea `fs.readFileSync()` para cargar `server.key` y `server.crt` en buffers de memoria. Es recomendable proteger estos archivos mediante permisos de sistema operativo y, si es viable, cargarlos desde un HSM o un almacén cifrado.

2. **Configuración de opciones HTTPS:** Se construye un objeto de opciones con las propiedades `key`, `cert` y, opcionalmente, `ca`, `honorCipherOrder` y parámetros de curvas elípticas preferidas. Estos parámetros determinan la política criptográfica del servidor y deben alinearse con las recomendaciones de RFC y las mejores prácticas (por ejemplo, preferir ECDHE y AEAD).

3. **Creación del servidor HTTPS:** Con `https.createServer(options, app)` se instancia el servidor que delega en `app` (Express) el manejo de rutas y lógica de negocio. El servidor expone tanto recursos estáticos como endpoints API (p. ej. `/api/cert-info`) que pueden devolver metadatos del certificado para fines de diagnóstico.

4. **Operación en tiempo de ejecución:** Una vez iniciado, el servidor acepta conexiones TLS entrantes, realiza el handshake, negocia cipher suites y atiende peticiones cifradas. Es importante instrumentar el servidor con métricas y logs que permitan identificar fallos de handshakes, intentos de conexión rechazados o advertencias relativas a suites inseguras.

Desde la perspectiva de seguridad operativa, se enfatiza la necesidad de:

- Minimizar la exposición de la clave privada y aplicar rotación periódica.
- Configurar políticas claras de cipher suites y deshabilitar suites obsoletas.
- Implementar OCSP Stapling, HSTS y cabeceras de seguridad HTTP adicionales para reducir vectores de ataque de downgrade y mejora la privacidad del cliente [8, 5].

Notas finales: Las figuras y sus explicaciones constituyen un marco detallado que permite tanto al lector académico como al practicante reproducir, analizar y evaluar la implementación de HTTPS en un entorno controlado. Para despliegues en producción se recomiendan controles adicionales (HSM, auditoría de claves, gestión de revocación a escala) y la supervisión continuada de vulnerabilidades reportadas (CVE/NVD) que puedan afectar a las bibliotecas criptográficas y al propio servidor.

6. Escenarios de Uso Frecuente

En esta sección se analizan en detalle tres escenarios representativos en los cuales HTTPS y los protocolos criptográficos asociados desempeñan un rol crítico. Para cada escenario se proporciona: (i) una descripción funcional; (ii) un diagrama ilustrativo (exportado desde Mermaid); (iii) un análisis de amenazas y contramedidas; (iv) recomendaciones de configuración y buenas prácticas; y (v) un checklist operativo para despliegues reales. Las referencias citadas apoyan las recomendaciones técnicas empleadas ([1, 8, 9, 10]).

6.1. Escenario 1: Comercio Electrónico y Servicios Bancarios en Línea

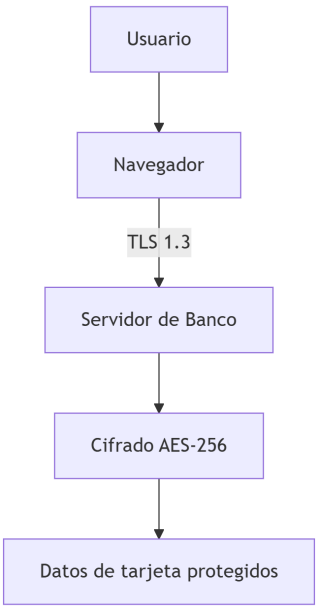

Figura 7: Arquitectura típica de comercio electrónico: cliente, gateway de pago, servidor de aplicación y capa de pago externa.

Descripción funcional. En plataformas de comercio electrónico y banca en línea, HTTPS protege la comunicación entre el cliente (navegador o aplicación móvil) y los servicios back-end que procesan credenciales y transacciones financieras. El flujo habitual incluye navegador → servidor web (TLS) → API interna → pasarela de pago (tercera parte), donde cada salto debe utilizar canales cifrados y autenticados. La protección incluye cifrado de sesión (AES-GCM/ChaCha20-Poly1305), autenticación de servidor mediante certificados X.509 y, cuando procede, autenticación fuerte del usuario (MFA) sobre el canal seguro [1].

Principales amenazas y mitigaciones.

- **Intercepción de credenciales y datos de pago (MITM).** Mitigación: TLS 1.3 con suites AEAD, HSTS y uso de certificados emitidos por CAs de confianza. Implementar Certificate Transparency y monitorización de certificados anómalos.
- **Reutilización de sesiones o ataques de replay.** Mitigación: claves efímeras (ECDHE) para PFS, tokens de un solo uso (OTP) y expiración corta de sesiones.
- **Inyección/alteración en recursos de terceros (third-party content).** Mitigación: evitar incluir recursos inseguros; usar subresource integrity (SRI) y servir dependencias desde HTTPS confiable [10].
- **Fugas por configuración TLS débil (cipher suites obsoletas).** Mitigación: configuración restrictiva que priorice ECDHE+AEAD y deshabilite RC4, 3DES, y suites basadas en RSA key-exchange [8].

Recomendaciones de configuración (producción).

- Forzar TLS 1.3 y mantener compatibilidad controlada con TLS 1.2 sólo si es estrictamente necesario.
- Habilitar HSTS con periodo largo y preloading donde aplique.
- Implementar OCSP Stapling para mejorar la latencia y privacidad en la validación de revocación.
- Usar certificados emitidos por CAs reconocidas y, para servicios críticos, considerar certificados con short-lived validity o automatización ACME (Let's Encrypt) junto con gestión de rotación.
- Aislar la clave privada: almacén seguro o HSM; permisos de fichero mínimos en servidores.
- Implementar detección y respuesta a incidentes (logs de handshake, alertas por cambios de certificado).

Checklist operativo (rápido).

1. TLS 1.3 habilitado, AEAD suites preferidas.
2. HSTS con `includeSubDomains` y `preload` cuando corresponda.
3. OCSP Stapling activo.
4. Rotación y backup de claves privadas; HSM para entornos regulados.
5. Pruebas periódicas con `testssl.sh` o SSL Labs.
6. Revisión de terceros y bloqueo de recursos inseguros (SRI).

Cumplimiento y consideraciones regulatorias. Para plataformas que procesan datos de pago se recomienda el alineamiento con PCI-DSS (requisitos relativos a cifrado en tránsito), así como el registro de auditorías que demuestren la correcta configuración de TLS y la gestión de claves. Estas prácticas reducen riesgo regulatorio y mejoran la confianza del usuario.

6.2. Escenario 2: Correo Electrónico Seguro (SMTP/IMAP/POP3 sobre TLS)

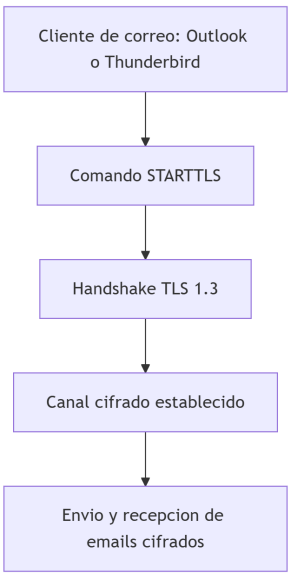

Figura 8: Topología de correo: cliente de correo, servidor MTA/IMAP y utilización de STARTTLS/SMTPS para asegurar canales.

Descripción funcional. El ecosistema de correo electrónico se sustenta en protocolos como SMTP, IMAP y POP3; para asegurar estos canales se utiliza TLS (SMTPS/IMAPS/POP3S) o el mecanismo STARTTLS que eleva una conexión cleartext a cifrada. La adopción de TLS protege la confidencialidad de contenidos y credenciales en tránsito entre clientes y servidores, y entre servidores remitentes/receptores cuando se negocia TLS en MTA-to-MTA [1].

Principales amenazas y mitigaciones.

- **Downgrade y ataques de stripping en STARTTLS.** Mitigación: políticas estrictas de MTA (MTA-STS), DANE (DNS-based Authentication of Named Entities) cuando sea viable, y configuración que favorezca SMTPS sobre STARTTLS en enlaces críticos.
- **Intercepción de credenciales en tránsito.** Mitigación: uso forzado de TLS para clientes y servidores, implementación de SASL sobre TLS y rechazo de autenticaciones en canales no cifrados.
- **Phishing y spoofing por certificados fraudulentos.** Mitigación: validación estricta de cadenas de confianza, monitorización de certificados y estrategias de reputación.

Recomendaciones de configuración.

- Preferir conexiones TLS directas (SMTPS/IMAPS) y usar STARTTLS sólo con políticas de MTA-STS o DANE.
- Implementar autenticación fuerte (SASL mechanisms como SCRAM) sobre TLS.
- Habilitar TLS en enlaces MTA-to-MTA y lanzar alertas ante caídas de cifrado.
- Usar certificados válidos y rotación periódica; OCSP Stapling para MTA cuando sea soportado.

Checklist operativo.

1. Configurar MTA para requerir TLS en conexiones salientes/entrantes según la política.
2. Habilitar MTA-STS y evaluar DANE si el dominio lo permite.
3. Forzar autenticación SASL sobre TLS.
4. Monitorear retransmisiones fallidas y negociaciones sin cifrado.

6.3. Escenario 3: APIs RESTful y Arquitecturas de Microservicios

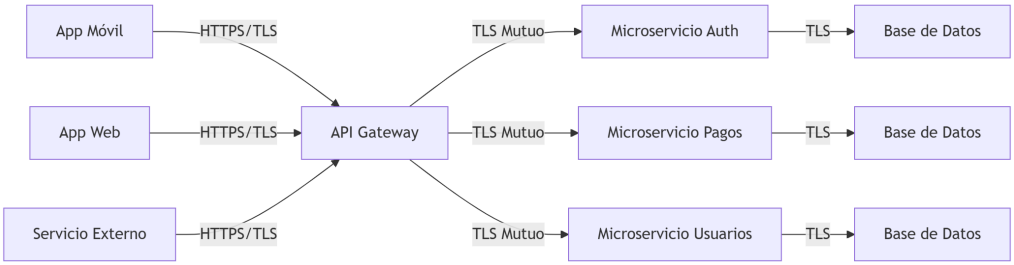

Figura 9: Arquitectura típica de APIs y microservicios con API Gateway, mTLS entre servicios y almacenamiento backend protegido.

Descripción funcional. En arquitecturas modernas, las APIs RESTful son el vehiculo principal de comunicación entre clientes y servicios, y entre microservicios internos. HTTPS protege tanto el acceso expuesto al exterior (cliente → API Gateway) como las comunicaciones internas (microservicio ↔ microservicio). En entornos sensibles se emplea *mutual TLS* (mTLS) para autenticar y autorizar de forma mutua los endpoints, reduciendo la necesidad de confiar en credenciales estáticas y mitigando ataques laterales [9].

Principales amenazas y mitigaciones.

- **Compromiso lateral entre servicios (east–west attacks).** Mitigación: mTLS, segmentación de red, políticas de autorización y verificación de identidad por servicio.
- **Fuga de credenciales y tokens en cabeceras.** Mitigación: transmitir tokens sólo sobre HTTPS, uso de short-lived JWTs y verificación de revocación/blacklist.
- **Inyección de dependencias remotas y contenido no confiable.** Mitigación: validar y sanear entradas, aplicar políticas de CORS estrictas y evitar incluir recursos de terceros sin control [10].

Recomendaciones de configuración y diseño.

- Implementar mTLS para comunicaciones internas entre servicios críticos; automatizar la emisión/rotación de certificados mediante una PKI interna o soluciones como Istio/Consul.

- Forzar HTTPS en el borde (API Gateway) y aplicar políticas de rate limiting, WAF y autenticación fuerte (OAuth2/OpenID Connect).
- Emplear certificate pinning en aplicaciones móviles cuando se justifique, con procedimientos de actualización controlados.
- Monitorizar telemetría de handshakes TLS y latencias para detectar anomalías o intentos de degradación.

Checklist operativo (microservicios).

1. mTLS habilitado y gestionado por una PKI interna (rotación automática).
2. API Gateway con TLS 1.3, WAF y políticas de autenticación (OAuth2).
3. Tokens de acceso short-lived y validación server-side.
4. Observabilidad: métricas TLS, logs de handshake, y alertas por fallos de validación.

6.4. Consideraciones transversales

Al margen de las particularidades de cada escenario, se deben considerar prácticas transversales que aumentan la resiliencia global:

- **Automatización de certificados:** uso de ACME/Let's Encrypt para certificados públicos y sistemas internos de entrega para PKI privada.
- **Pruebas continuas y auditoría:** integración de herramientas como `testssl.sh`, SSL Labs y scanners de seguridad en pipelines de CI/CD.
- **Políticas de actualización:** vigilar CVE y actualizaciones de bibliotecas criptográficas (OpenSSL, BoringSSL, NSS).
- **Gestión de revocación:** evaluar OCSP Stapling y short-lived certificates para minimizar problemas de revocación observados en la práctica [5, 6].

Referencias. Las prácticas y recomendaciones aquí descritas se basan en las especificaciones y estudios citados en la bibliografía, que incluyen análisis formales del protocolo TLS y evaluaciones empíricas de la infraestructura PKI a escala global [1, 8, 5, 6, 9, 10].

7. Implementación Práctica del Entorno HTTPS con Certificados Digitales

La presente sección documenta de manera formal y detallada la implementación práctica desarrollada en el marco de la asignatura de Criptografía de la Facultad de Ingeniería de la Universidad Central del Ecuador. El objetivo del ejercicio es construir un entorno seguro basado en un servidor HTTPS utilizando certificados digitales X.509 generados con OpenSSL, integrados dentro de una arquitectura modular implementada en Node.js, Express.js y una interfaz gráfica web desarrollada con tecnologías HTML, CSS y JavaScript.

Se expone la estructura completa del proyecto, la generación profesional de certificados digitales, su proceso de verificación, así como la integración con el backend y el frontend. Finalmente, se reservan subsecciones específicas para registrar el código fuente íntegro asociado a la solución, con el propósito de mantener trazabilidad técnica y evidencia documental del desarrollo.

7.1. Estructura General del Proyecto

El proyecto se organiza bajo una arquitectura modular, permitiendo separar la capa de servidor, la capa de cliente y los componentes de seguridad. La estructura completa es la siguiente:

```
secure-web/
 certs/                  # Certificados y claves criptográficas
    server.key          # Clave privada del servidor
    server.crt          # Certificado digital X.509 autofirmado
 public/                 # Interfaz gráfica y archivos estáticos
    css/
    js/
       main.js
    index.html
 src/                    # Backend implementado en Node.js
    app.js
    server.js
 package.json            # Dependencias y metadatos del proyecto
 README.md               # Documentación técnica
```

Esta estructura facilita el análisis del flujo de información, el aislamiento de responsabilidades y la comprensión del proceso de comunicación cifrada mediante TLS.

7.2. Tecnologías Empleadas

En la Tabla 1 se presentan las principales tecnologías utilizadas en el proyecto:

Tecnología	**Descripción**
Node.js	Entorno de ejecución para JavaScript del lado del servidor.
Express.js	Framework minimalista para la creación de APIs y servidores HTTP/HTTPS.
OpenSSL	Herramienta estándar para la generación y gestión de certificados X.509.
HTML5 / CSS3	Tecnologías base para el desarrollo de la interfaz gráfica.
JavaScript ES6+	Lógica dinámica en cliente y servidor.
Módulo Crypto (Node.js)	Extracción de huellas digitales y análisis criptográfico.

Cuadro 1: Tecnologías empleadas en el desarrollo del proyecto.

7.3. Descripción de Componentes Principales

Backend (Directorio `src/`)

- **server.js**: Punto de entrada del servidor HTTPS. Carga los certificados digitales, configura las opciones TLS y levanta el servidor en el puerto 3000.
- **app.js**: Define la configuración principal de Express, el enrutamiento, el middleware y el endpoint encargado de analizar y exponer la información contenida en el certificado X.509.

Frontend (Directorio `public/`)

- **index.html**: Interfaz visual donde se muestra la información del certificado.
- **main.js**: Lógica del cliente encargada de consumir la API REST, actualizar el DOM, mostrar los valores criptográficos y gestionar la experiencia interactiva.

Seguridad (Directorio `certs/`)

- **server.key**: Clave privada RSA utilizada en el handshake TLS.
- **server.crt**: Certificado digital que se entrega a los clientes para autenticar la identidad del servidor.

7.4. Generación de Certificados SSL/TLS utilizando Git Bash

La creación de certificados digitales constituye un procedimiento fundamental dentro del estudio de la seguridad informática y, específicamente, en el contexto de la cátedra de Criptografía de la Universidad Central del Ecuador (UCE). Con el fin de implementar un entorno seguro para servicios web basados en HTTPS, se empleó la herramienta *OpenSSL* ejecutada desde el entorno *Git Bash* en sistemas Windows, debido a su compatibilidad, facilidad de uso y amplio soporte para operaciones criptográficas.

Este proceso permite generar tanto la clave privada del servidor como un certificado digital autofirmado bajo el estándar X.509, el cual será posteriormente utilizado por un servidor HTTPS implementado en Node.js. El objetivo académico es comprender la estructura del certificado, sus campos internos, su finalidad dentro de un sistema criptográfico y la mecánica del proceso de firma digital.

7.4.1. Creación del directorio de certificados

El primer paso consiste en la creación de un directorio destinado exclusivamente al almacenamiento seguro de los archivos criptográficos:

```
mkdir certs
cd certs
```

Este directorio contendrá dos archivos esenciales para la configuración del servidor HTTPS:

- **server.key**: clave privada RSA generada durante el proceso.
- **server.crt**: certificado digital autofirmado.

7.4.2. Generación del certificado y la clave privada

Desde *Git Bash*, se ejecuta el siguiente comando, el cual genera automáticamente una clave privada RSA de 2048 bits y un certificado digital autofirmado con una validez de un año:

```
openssl req -x509 -nodes -newkey rsa:2048 \
-keyout server.key -out server.crt -days 365
```

Durante el proceso, OpenSSL solicitará información relevante para construir la estructura X.509. Para fines académicos y manteniendo coherencia institucional, se utilizaron los siguientes valores recomendados:

```
Country Name (2 letter code) [AU]: EC
State or Province Name (full name): Pichincha
Locality Name (eg, city): Quito
Organization Name (eg, company): Universidad Central del Ecuador
Organizational Unit Name (eg, section): Facultad de Ingeniería,
                                        Escuela de Computación
Common Name (e.g. server FQDN or YOUR name): localhost
Email Address: criptografia@uce.edu.ec
```

Es importante destacar que el campo **Common Name (CN)** debe establecerse como *localhost*, dado que el proyecto se ejecuta en un entorno de desarrollo local.

7.4.3. Verificación del certificado generado

Una vez creado el certificado, es posible inspeccionar su contenido utilizando el comando:

```
openssl x509 -in server.crt -text -noout
```

Esta instrucción permite revisar:

- Los campos del sujeto (Subject) y emisor (Issuer).
- La clave pública RSA asociada.
- El algoritmo de firma digital empleado.
- El período de validez.
- Las huellas digitales (*fingerprints*) SHA–1 y SHA–256.
- Las extensiones X.509 del certificado.

7.4.4. Regeneración del certificado

En caso de requerir un nuevo certificado, se puede repetir el proceso eliminando los archivos anteriores:

```
cd certs
rm server.key server.crt
openssl req -x509 -nodes -newkey rsa:2048 \
-keyout server.key -out server.crt -days 365
```

7.4.5. Integración con el servidor HTTPS

El certificado y la clave privada generados desde Git Bash se integran directamente en el servidor implementado en Node.js mediante:

```
const httpsOptions = {
  key: fs.readFileSync('./certs/server.key'),
  cert: fs.readFileSync('./certs/server.crt'),
};
```

De esta manera, el servidor establece conexiones seguras mediante el protocolo HTTPS, garantizando confidencialidad, integridad y autenticación básica del servidor en un entorno académico controlado.

7.5. Integración con el Servidor HTTPS

El servidor utiliza los certificados generados mediante la siguiente configuración:

```
const httpsOptions = {
  key: fs.readFileSync('./certs/server.key'),
  cert: fs.readFileSync('./certs/server.crt')
};
```

Al actualizar o regenerar los certificados, la interfaz web reflejará automáticamente los nuevos valores al invocar el endpoint:

```
fetch('/api/cert-info')
```

8. Evidencia del Código Fuente

Con el propósito de mantener una documentación completa y verificable, las siguientes subsecciones están destinadas a incluir el **código íntegro** del proyecto.

8.1. Archivo `server.js`

```
import https from "https";
import fs from "fs";
import path from "path";
import app from "./app.js";
import { fileURLToPath } from "url";

// Resolver la ruta del archivo actual
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

// Cargar certificados SSL/TLS
const key = fs.readFileSync(path.join(__dirname, "../certs/server.
    key"));
const cert = fs.readFileSync(path.join(__dirname, "../certs/server.
    crt"));

// Crear el servidor HTTPS
const httpsServer = https.createServer({ key, cert }, app);
```

```
17
18 // Puerto de escucha
19 const PORT = 3000;
20
21 // Iniciar el servidor
22 httpsServer.listen(PORT, () => {
23   console.log(‘Servidor HTTPS ejecut ndose en https://localhost:${
       PORT}‘);
24 });
```

Listing 1: Implementación del servidor HTTPS en Node.js

Descripción detallada del archivo `server.js`

El código mostrado en el *Listing* 1 implementa el arranque de un servidor HTTPS en Node.js, integrando certificados X.509 generados con OpenSSL y una aplicación Express exportada desde `app.js`. A continuación se desglosa el propósito y el comportamiento de cada bloque funcional del fichero, con recomendaciones operacionales y de seguridad.

1. Importación de módulos

`https` Módulo nativo de Node.js que permite crear servidores TLS/SSL. Se utiliza para levantar un servidor que escucha conexiones cifradas y realiza el *handshake* TLS con los clientes.

`fs` Módulo de sistema de archivos, empleado para leer los artefactos criptográficos (clave privada y certificado) desde disco.

`path` Utilizado para construir rutas de archivos portables entre distintos sistemas operativos (evita concatenaciones de cadenas inseguras).

`app` Exportación de la instancia Express configurada (rutas, middleware y lógica de la aplicación) — definida en `src/app.js`.

`fileURLToPath` Función de la librería `url` para transformar la URL del módulo (ESM) en una ruta de sistema de archivos; se utiliza cuando el proyecto usa módulos ECMAScript nativos (“type": "module”).

2. Resolución de la ruta del fichero actual Las constantes `__filename` y `__dirname` se obtienen mediante las funciones `fileURLToPath(import.meta.url)` y `path.dirname(...)`. Dado que el proyecto utiliza módulos ES (ECMAScript Modules — ESM) en lugar del sistema clásico CommonJS, estas funciones son necesarias para resolver correctamente las rutas absolutas del proyecto. Esto permite acceder sin ambigüedades a archivos relativos —por ejemplo `../certs/server.key`— independientemente del directorio desde el que se ejecute el servidor o del sistema operativo utilizado. De esta forma se mitigan errores comunes en tiempo de ejecución asociados a rutas mal resueltas.

3. Carga de los certificados SSL/TLS El servidor HTTPS carga los certificados mediante `fs.readFileSync(path.join(__dirname, "../certs/server.key"))` y la lectura equivalente del certificado público. Durante esta fase se incorporan en memoria la **clave privada** y el **certificado digital**, elementos indispensables para establecer comunicaciones cifradas bajo TLS. Es importante considerar los siguientes aspectos:

- **Protección de la clave privada:** El archivo `server.key` debe poseer permisos estrictos (por ejemplo, 600) y nunca debe almacenarse en repositorios públicos. En entornos de producción se recomienda emplear módulos HSM o servicios de gestión de claves (KMS) para evitar que la clave privada resida en disco.
- **Lectura síncrona:** El uso de `readFileSync` es apropiado durante la fase de arranque del servidor, cuando todavía no se aceptan conexiones. Para sistemas que requieren *hot reload* de certificados o rotación periódica de claves, conviene implementar mecanismos de recarga segura o utilizar APIs que soporten rotación de credenciales sin interrumpir el servicio.
- **Validación previa:** Es recomendable verificar el contenido del certificado y de la clave mediante operaciones de parseo, validación de formato o detección de corrupción. Asimismo, la implementación de manejo de excepciones permite capturar de manera controlada errores derivados de rutas inválidas o archivos mal formados.

4. Creación del servidor HTTPS La llamada `https.createServer({ key, cert }, app)` construye un servidor TLS que, una vez aceptada la conexión y completado el handshake, delega las peticiones HTTP en la aplicación Express (`app`). Aspectos relevantes:

- **Opciones adicionales:** En un despliegue real conviene especificar opciones adicionales como `ca` (certificados intermedios), `honorCipherOrder`, `ciphers` y las curvas elípticas preferidas para controlar la política criptográfica del servidor.
- **Cipher suites y políticas:** Configurar cipher suites priorizando ECDHE+AEAD (por ejemplo AES-GCM o ChaCha20-Poly1305) y deshabilitar suites obsoletas según recomendaciones RFC y guías operacionales [1, 8].
- **Integración con Express:** Express actúa como manejador de rutas y middleware; la separación entre la capa TLS y la lógica de aplicación facilita aplicar políticas de seguridad en ambos niveles.

5. Configuración de puerto y puesta en escucha Se define la constante `PORT` (3000 en este caso) y se llama a `httpsServer.listen(PORT, ...)` para iniciar el bucle de escucha. Recomendaciones:

- **Puertos en producción:** En entornos productivos el puerto predeterminado para HTTPS es el 443; durante el desarrollo es aceptable utilizar puertos alternos (p. ej. 3000).

- **Manejo de errores de arranque:** Añadir manejadores para eventos `error` y señales del sistema (`SIGINT`, `SIGTERM`) para asegurar un apagado limpio y el registro adecuado de errores.
- **Mensajes informativos:** El `console.log` en el callback es útil en desarrollo; en producción conviene integrar la salida en un sistema de logs estructurados (ej. Winston, Bunyan) que permita trazabilidad y análisis forense.

6. Consideraciones de seguridad y operativas A la hora de migrar desde un entorno de laboratorio a producción se deben tener en cuenta las siguientes prácticas mínimas:

- **Gestión de claves:** Emplear HSM o servicios de gestión (AWS KMS, Azure Key Vault, Google KMS) para proteger claves privadas y permitir rotación segura.
- **Automatización de certificados:** Utilizar ACME (Let's Encrypt) o procesos automatizados para garantizar renovación y minimizar ventanas de expiración.
- **OCSP Stapling y revocación:** Habilitar OCSP Stapling para mejorar validación de revocación y reducir latencia; monitorizar el estado de revocación de certificados intermedios y raíces relevantes [5, 6].
- **Política de ciphers y versiones TLS:** Forzar TLS 1.3 cuando sea posible y mantener una política estricta de cipher suites siguiendo RFCs y guías de seguridad [1, 8].
- **Auditoría y telemetría:** Registrar métricas de handshakes, suites negociadas, errores de verificación y tiempos de latencia para detectar patrones anómalos o intentos de degradación.

7. Mejora sugerida (ejemplo de robustecimiento) Se recomienda complementar el código del `server.js` con bloques de manejo de errores y opciones TLS más explícitas; por ejemplo:

- Validar existencia y permisos de los archivos antes de iniciar el servidor.
- Añadir `httpsServer.on('error', handler)` para capturar puertos ocupados u otros fallos.
- Externalizar la configuración (puerto, rutas a certificados, políticas TLS) mediante variables de entorno gestionadas por `dotenv` o sistemas de configuración centralizados.

8. Pruebas y verificación Para verificar el correcto funcionamiento del servidor:

1. Arrancar el servidor y comprobar que responde en `https://localhost:3000` (aceptando la excepción del certificado autofirmado en el navegador).
2. Utilizar herramientas como `curl -vk https://localhost:3000` para inspeccionar el handshake y los certificados presentados.

3. Ejecutar análisis con `testssl.sh` o SSL Labs para revisar la configuración de TLS (cipher suites, versiones soportadas, renegociación).

4. Consultar el contenido del certificado con `openssl x509 -in certs/server.crt -text -noout` para comprobar campos como Subject, Issuer, Validity y extensiones.

En conjunto, la implementación del `server.js` materializa el componente encargador de establecer el canal TLS; su correcta configuración, protección de los artefactos criptográficos y supervisión continua son condiciones indispensables para mantener la seguridad del servicio HTTPS en producción.

8.2. Implementación de la aplicación Express en `app.js`

En el *Listing* 2 se muestra la implementación completa del archivo `app.js`, el cual constituye el núcleo lógico de la aplicación web desarrollada para el laboratorio de conexiones seguras mediante HTTPS. Este módulo gestiona la configuración de Express, los middleware, el enrutamiento principal y la extracción avanzada de información criptográfica desde el certificado digital X.509 utilizado por el servidor.

```
import express from "express";
import path from "path";
import { fileURLToPath } from "url";
import fs from "fs";

const app = express();

// Necesario para __dirname en ES Modules
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

// Middleware para JSON
app.use(express.json());

// Servir archivos est ticos
app.use(express.static(path.join(__dirname, "../public")));

app.get("/", (req, res) => {
  res.sendFile(path.join(__dirname, "../public/index.html"));
});

// Endpoint para obtener informaci n del certificado
app.get("/api/cert-info", async (req, res) => {
  try {
    const certPath = path.join(__dirname, "../certs/server.crt");
    const keyPath = path.join(__dirname, "../certs/server.key");
    const certPem = fs.readFileSync(certPath, "utf8");
    const keyPem = fs.readFileSync(keyPath, "utf8");

    // Importar crypto para parsear el certificado
    const crypto = await import("crypto");

    // Crear objeto X509Certificate
    const cert = new crypto.X509Certificate(certPem);

    // Extraer informaci n del subject y issuer
```

```
37      const subject = cert.subject;
38      const issuer = cert.issuer;
39
40      // Funci n para parsear DN (Distinguished Name)
41      const parseDN = (dn) => {
42        const parts = {};
43        const lines = dn.split('\n');
44        lines.forEach(line => {
45          const [key, ...valueParts] = line.split('=');
46          if (key && valueParts.length > 0) {
47            parts[key.trim()] = valueParts.join('=').trim();
48          }
49        });
50        return parts;
51      };
52
53      const subjectParts = parseDN(subject);
54      const issuerParts = parseDN(issuer);
55
56      // Calcular fingerprint SHA-256
57      const certDer = Buffer.from(
58        certPem.replace(/-----BEGIN CERTIFICATE-----/, '')
59          .replace(/-----END CERTIFICATE-----/, '')
60          .replace(/\s/g, ''),
61        'base64'
62      );
63      const fingerprint = crypto.createHash('sha256').update(certDer)
           .digest('hex');
64      const fingerprintFormatted = fingerprint.match(/.{2}/g).join
           (':').toUpperCase();
65
66      // Calcular fingerprint de la clave p blica
67      const publicKey = cert.publicKey.export({ type: 'spki', format:
           'der' });
68      const publicKeyFingerprint = crypto.createHash('sha256').update
           (publicKey).digest('hex');
69
70      // Obtener fechas de validez
71      const validFrom = new Date(cert.validFrom);
72      const validTo = new Date(cert.validTo);
73
74      // Calcular d as de validez
75      const now = new Date();
76      const daysRemaining = Math.floor((validTo - now) / (1000 * 60 *
           60 * 24));
77      const totalDays = Math.floor((validTo - validFrom) / (1000 * 60
           * 60 * 24));
78
79      // Obtener informaci n de la clave p blica
80      const publicKeyInfo = cert.publicKey.asymmetricKeyDetails;
81      const keySize = publicKeyInfo?.modulusLength || 2048;
82
83      // Formatear fechas en espa ol
84      const formatDate = (date) => {
85        const days = ['domingo', 'lunes', 'martes', 'mi rcoles', '
             jueves', 'viernes', 's bado'];
86        const months = ['enero', 'febrero', 'marzo', 'abril', 'mayo',
             'junio',
```

```
87          'julio', 'agosto', 'septiembre', 'octubre', 'noviembre', '
               diciembre'];
88
89        const dayName = days[date.getDay()];
90        const day = date.getDate();
91        const month = months[date.getMonth()];
92        const year = date.getFullYear();
93        const hours = date.getHours().toString().padStart(2, '0');
94        const minutes = date.getMinutes().toString().padStart(2, '0')
             ;
95        const seconds = date.getSeconds().toString().padStart(2, '0')
             ;
96
97        return `${dayName}, ${day} de ${month} de ${year}, ${hours}:$
             {minutes}:${seconds}`;
98      };
99
100     const certInfo = {
101       subject: {
102         commonName: subjectParts.CN || 'N/A',
103         organization: subjectParts.O || 'N/A',
104         organizationalUnit: subjectParts.OU || 'N/A',
105         locality: subjectParts.L || 'N/A',
106         state: subjectParts.ST || 'N/A',
107         country: subjectParts.C || 'N/A'
108       },
109       issuer: {
110         commonName: issuerParts.CN || 'N/A',
111         organization: issuerParts.O || 'N/A',
112         organizationalUnit: issuerParts.OU || 'N/A',
113         locality: issuerParts.L || 'N/A',
114         state: issuerParts.ST || 'N/A',
115         country: issuerParts.C || 'N/A'
116       },
117       validity: {
118         notBefore: validFrom.toISOString(),
119         notAfter: validTo.toISOString(),
120         notBeforeFormatted: formatDate(validFrom),
121         notAfterFormatted: formatDate(validTo),
122         daysRemaining: daysRemaining,
123         totalDays: totalDays,
124         isValid: now >= validFrom && now <= validTo
125       },
126       fingerprints: {
127         sha256: fingerprintFormatted,
128         sha256Raw: fingerprint,
129         publicKey: publicKeyFingerprint
130       },
131       technical: {
132         version: cert.version || 3,
133         serialNumber: cert.serialNumber,
134         algorithm: `RSA ${keySize} bits`,
135         signatureAlgorithm: cert.signatureAlgorithm || '
               sha256WithRSAEncryption',
136         keySize: keySize,
137         publicKeyAlgorithm: 'RSA',
138         format: 'X.509 v3'
139       },
```

```
140       metadata: {
141         isSelfSigned: subject === issuer,
142         status: subject === issuer ? 'Autofirmado' : 'Firmado por
              CA',
143         protocol: 'TLS 1.3',
144         cipher: 'AES-256-GCM'
145       }
146     };
147
148     res.json({
149       success: true,
150       certificate: certInfo,
151       timestamp: new Date().toISOString()
152     });
153   } catch (error) {
154     console.error("Error al leer certificado:", error);
155     res.status(500).json({
156       success: false,
157       error: "No se pudo obtener informaci n del certificado",
158       message: error.message
159     });
160   }
161 });
162
163 export default app;
```

Listing 2: Implementación de la aplicación Express en `app.js`

Descripción detallada del módulo `app.js`

El archivo `app.js`, mostrado en el *Listing* 2, constituye el componente responsable de la lógica central de la aplicación web desarrollada para este laboratorio. Su propósito es configurar la instancia de Express, gestionar el enrutamiento, servir los recursos estáticos y, además, implementar un analizador especializado del certificado digital X.509 utilizado por el servidor HTTPS.

A continuación, se detalla la función de cada sección del módulo, junto con consideraciones operativas y criptográficas relevantes.

1. Importación de módulos El código incorpora módulos esenciales para el correcto funcionamiento de la aplicación:

- **express:** Framework minimalista utilizado para gestionar peticiones HTTP, middleware y rutas.
- **path y fileURLToPath:** Componentes necesarios para obtener rutas absolutas en proyectos que utilizan ES Modules; garantizan portabilidad entre entornos.
- **fs:** Módulo nativo que permite leer y manipular archivos, empleado aquí para acceder al certificado y a la clave privada.

2. Configuración de `__dirname` en ES Modules Dado que los proyectos basados en ES Modules no disponen de `__dirname` por defecto, este se reconstruye mediante:

$$\_\_filename = \text{fileURLToPath(import.meta.url)},$$
$$\_\_dirname = \text{path.dirname(}\_\_\text{filename)}$$

Esto permite acceder correctamente a rutas relativas independientemente del directorio desde el cual se ejecute la aplicación.

3. Configuración de middleware Se habilitan dos servicios fundamentales:

- `express.json()`: Parser que permite manejar solicitudes en formato JSON.
- **Servidor de archivos estáticos:** Se expone la carpeta `public/` para servir HTML, CSS y JavaScript directamente al cliente.

4. Ruta raíz de la aplicación La ruta `/"` devuelve el archivo `index.html`, el cual actúa como la interfaz principal del sistema. Esta separación entre lógica de servidor y presentación facilita una arquitectura modular y escalable.

5. Endpoint `/api/cert-info` Este punto constituye la parte más crítica del módulo. Su finalidad es:

1. Leer el certificado X.509 y la clave privada almacenados en disco.
2. Parsear los campos internos del certificado empleando la clase `X509Certificate` del módulo `crypto`.
3. Derivar información criptográfica relevante, incluyendo fingerprints, fechas de validez, tamaño de clave y atributos del issuer y subject.

6. Extracción de información del certificado Mediante `crypto.X509Certificate`, se accede a:

- **Subject:** Identidad del titular del certificado.
- **Issuer:** Entidad emisora (o el mismo sujeto si es autofirmado).
- **Fechas de validez:** `validFrom` y `validTo`, usadas para verificar expiración.
- **Clave pública:** Desde la cual se determina tamaño de clave, algoritmo y huella.

7. Cálculo de fingerprints criptográficos El módulo genera:

- **Fingerprint SHA-256 del certificado:** Utilizado para identificar unívocamente el certificado y compararlo contra listas de confianza.
- **Fingerprint de la clave pública:** Útil para mecanismos como HPKP, Certificate Transparency o pinning interno.

8. Análisis de validez y metadatos El sistema calcula:

- Días totales de vigencia.
- Días restantes antes de expiración.
- Estado de validez al momento de la consulta.

Además, se determinan características técnicas como:

- Versión del certificado.
- Algoritmos de firma y cifrado.
- Longitud de clave pública.

9. Composición del objeto de información Finalmente, toda la información procesada se agrupa en una estructura JSON organizada en:

1. `subject`
2. `issuer`
3. `validity`
4. `fingerprints`
5. `technical`
6. `metadata`

La API devuelve así un bloque coherente, estandarizado y listo para ser consumido por interfaces web, sistemas de auditoría o análisis forense.

10. Manejo de errores El bloque `try-catch` captura condiciones como:

- Problemas al leer archivos.
- Certificados corruptos o mal formateados.
- Errores internos del módulo `crypto`.

Garantizando siempre una respuesta JSON consistente con estructura:

`{success: false, error: "..."}`

Conclusión El módulo `app.js` integra en un único punto:

- la inicialización del framework Express,
- el enrutamiento web,
- la publicación de contenido estático,
- y un analizador criptográfico completo del certificado digital.

Constituye, por tanto, el componente lógico central de la arquitectura del sistema de análisis y exposición de información relacionada con HTTPS y certificación X.509.

8.3. Implementación del Cliente Web y Servicios Asociados

La presente sección describe de manera detallada la estructura, funcionamiento y fundamentos técnicos del cliente web desarrollado para interactuar con el servidor HTTPS implementado en Node.js. Esta capa cliente cumple un rol crítico dentro del laboratorio, dado que permite verificar en tiempo real el establecimiento del canal seguro, visualizar las propiedades criptográficas del certificado X.509, validar la correcta operación del protocolo TLS 1.3, y analizar la integridad de los datos presentados al usuario final.

Con el propósito de garantizar claridad, la explicación se organiza según los componentes que conforman la aplicación: el script principal del lado del cliente (`main.js`), la interfaz gráfica definida en `index.html`, y la estructura de dependencias administradas mediante el archivo `package.json`. Cada subsección presenta el código fuente correspondiente, acompañado de un análisis exhaustivo que describe su propósito, arquitectura interna y relación con los mecanismos criptográficos implementados.

8.3.1. Archivo `main.js`: Lógica del Cliente e Interacción con el Servidor HTTPS

El archivo `main.js` contiene la totalidad de la lógica dinámica del lado del cliente. A través de este módulo se gestiona la verificación del canal seguro, se obtiene información detallada del certificado digital, se actualizan elementos gráficos de la interfaz y se ejecutan animaciones orientadas a mejorar la experiencia del usuario. Su diseño modular y orientado a eventos permite aislar adecuadamente la lógica de interacción entre el navegador y el servidor, garantizando un comportamiento predecible y seguro dentro del entorno HTTPS.

Código Fuente

```
1  // Verificaci n de Seguridad
2  document.getElementById("check-security").addEventListener("click",
       async () => {
3    const resultDiv = document.getElementById("security-result");
4    const button = document.getElementById("check-security");
5
6    // Mostrar loading
7    button.innerHTML = '<span class="loading"></span> Verificando
       ...';
8    button.disabled = true;
9
10   // Simular verificaci n
11   await new Promise(resolve => setTimeout(resolve, 1000));
12
13   const isSecure = window.location.protocol === "https:";
14
15   resultDiv.classList.remove("hidden");
16   resultDiv.classList.add("fade-in");
17
18   if (isSecure) {
19     resultDiv.className = "success fade-in";
20     resultDiv.innerHTML = `
```

```
      <strong style="font-size: 18px; display: block; margin-bottom: 8px; margin-top: 25px;">Conexi n Segura Establecida</strong>
      <p style="margin: 0; line-height: 1.6;">
        Tu conexi n est  protegida con HTTPS. Todos los datos transmitidos est n encriptados
        con cifrado de grado empresarial AES-256-GCM.
      </p>
      <div style="margin-top: 12px; padding-top: 12px; border-top: 1px solid rgba(67, 233, 123, 0.2);">
        <small style="opacity: 0.8;">Protocolo: <strong>${window.location.protocol.toUpperCase()}</strong></small>
      </div>
    `;
  } else {
    resultDiv.className = "error fade-in";
    resultDiv.innerHTML = `
      <strong style="font-size: 18px; display: block; margin-bottom: 8px; margin-top: 25px;">Conexi n No Segura</strong>
      <p style="margin: 0;">
        Este sitio NO est  usando HTTPS. Los datos podr an estar en riesgo.
      </p>
    `;
  }

  button.innerHTML = 'Verificar Seguridad';
  button.disabled = false;
});

// Cargar Informaci n del Certificado
document.getElementById("load-cert").addEventListener("click", async () => {
  const certInfoDiv = document.getElementById("cert-info");
  const certDetailsDiv = certInfoDiv.querySelector(".cert-details");
  const button = document.getElementById("load-cert");

  // Mostrar loading
  button.innerHTML = '<span class="loading"></span> Cargando...';
  button.disabled = true;

  try {
    const response = await fetch("/api/cert-info");
    const data = await response.json();

    if (data.success) {
      const cert = data.certificate;

      // Actualizar informaci n del certificado con datos reales
      certDetailsDiv.innerHTML = `
        <div class="cert-item slide-up">
          <span class="label">Nombre Com n (CN)</span>
          <span class="value">${cert.subject.commonName}</span>
        </div>
        <div class="cert-item slide-up" style="animation-delay: 0.05s;">
          <span class="label">Organizaci n (O)</span>
```

```
69            <span class="value">${cert.subject.organization}</span>
70          </div>
71          <div class="cert-item slide-up" style="animation-delay: 0.1
               s;">
72            <span class="label">Unidad Organizativa (OU)</span>
73            <span class="value">${cert.subject.organizationalUnit}</
                 span>
74          </div>
75          <div class="cert-item slide-up" style="animation-delay:
               0.15s;">
76            <span class="label">Emisor CN</span>
77            <span class="value">${cert.issuer.commonName}</span>
78          </div>
79          <div class="cert-item slide-up" style="animation-delay: 0.2
               s;">
80            <span class="label">Emisor Organizaci n </span>
81            <span class="value">${cert.issuer.organization}</span>
82          </div>
83          <div class="cert-item slide-up" style="animation-delay:
               0.25s;">
84            <span class="label">Emisor OU</span>
85            <span class="value">${cert.issuer.organizationalUnit}</
                 span>
86          </div>
87          <div class="cert-item slide-up" style="animation-delay: 0.3
               s;">
88            <span class="label">Emitido el</span>
89            <span class="value">${cert.validity.notBeforeFormatted}</
                 span>
90          </div>
91          <div class="cert-item slide-up" style="animation-delay:
               0.35s;">
92            <span class="label">Vencimiento el</span>
93            <span class="value">${cert.validity.notAfterFormatted}</
                 span>
94          </div>
95          <div class="cert-item slide-up" style="animation-delay: 0.4
               s;">
96            <span class="label">Certificado (SHA-256)</span>
97            <span class="value" style="font-size: 11px; word-break:
                 break-all;">${cert.fingerprints.sha256Raw}</span>
98          </div>
99          <div class="cert-item slide-up" style="animation-delay:
               0.45s;">
100           <span class="label">Clave P blica (SHA-256)</span>
101           <span class="value" style="font-size: 11px; word-break:
                 break-all;">${cert.fingerprints.publicKey}</span>
102         </div>
103         <div class="cert-item slide-up" style="animation-delay: 0.5
               s;">
104           <span class="label">Algoritmo</span>
105           <span class="value">${cert.technical.algorithm}</span>
106         </div>
107         <div class="cert-item slide-up" style="animation-delay:
               0.55s;">
108           <span class="label">N mero de Serie</span>
109           <span class="value" style="font-size: 12px;">${cert.
                 technical.serialNumber}</span>
```

```
110         </div>
111         <div class="cert-item slide-up" style="animation-delay: 0.6
               s;">
112           <span class="label">Versi n </span>
113           <span class="value">${cert.technical.format}</span>
114         </div>
115         <div class="cert-item slide-up" style="animation-delay:
               0.65s;">
116           <span class="label">Estado</span>
117           <span class="value">${cert.validity.isValid ? '
                 V lido ' : '    Expirado'}</span>
118         </div>
119       `;
120
121       certInfoDiv.classList.remove("hidden");
122       certInfoDiv.classList.add("fade-in");
123
124       button.innerHTML = '    Informaci n Cargada';
125       button.style.background = 'var(--success-gradient)';
126
127     } else {
128       throw new Error(data.error || "Error al cargar certificado");
129     }
130
131   } catch (error) {
132     certDetailsDiv.innerHTML = `
133       <div class="cert-item" style="border-color: rgba(245, 87,
             108, 0.3);">
134         <span class="label">Error</span>
135         <span class="value" style="color: #f5576c;">No se pudo
               cargar la informaci n</span>
136       </div>
137     `;
138     certInfoDiv.classList.remove("hidden");
139     button.innerHTML = 'Reintentar';
140     button.disabled = false;
141   }
142 });
143
144 // Cargar informaci n del certificado autom ticamente al cargar
       la p gina
145 async function loadCertificateInfo() {
146   try {
147     const response = await fetch("/api/cert-info");
148     const data = await response.json();
149
150     if (data.success) {
151       const cert = data.certificate;
152
153       // Actualizar informaci n del emisor con datos reales
154       document.getElementById("org-name").textContent =
155         `${cert.issuer.organization} (${cert.issuer.commonName})`;
156       document.getElementById("org-location").textContent =
157         cert.issuer.locality !== 'N/A' ? cert.issuer.locality :
               cert.issuer.country;
158       document.getElementById("cert-type").textContent = cert.
             metadata.status;
159       document.getElementById("cert-validity").textContent =
```

```
160         `${cert.validity.totalDays} d as (${cert.validity.
                daysRemaining} restantes)`;
161
162       // Actualizar usos del certificado con informaci n real
163       const keyUsageList = document.getElementById("key-usage-list
              ");
164       const usages = [
165         `Nombre Com n (CN): ${cert.subject.commonName}`,
166         `Organizaci n (O): ${cert.subject.organization}`,
167         `Unidad Organizativa (OU): ${cert.subject.
                organizationalUnit}`,
168         `Algoritmo: ${cert.technical.algorithm}`,
169         `Formato: ${cert.technical.format}`
170       ];
171
172       keyUsageList.innerHTML = usages.map((usage, index) =>
173         `<li style="animation-delay: ${index * 0.1}s;" class="slide
                -up">${usage}</li>`
174       ).join("");
175     }
176   } catch (error) {
177     console.error("Error al cargar informaci n del certificado:",
          error);
178   }
179 }
180
181 // Animaci n de entrada para las tarjetas
182 function animateCards() {
183   const cards = document.querySelectorAll(".card");
184   cards.forEach((card, index) => {
185     card.style.opacity = "0";
186     card.style.transform = "translateY(30px)";
187
188     setTimeout(() => {
189       card.style.transition = "all 0.6s cubic-bezier(0.4, 0, 0.2,
              1)";
190       card.style.opacity = "1";
191       card.style.transform = "translateY(0)";
192     }, index * 100);
193   });
194 }
195
196 // Ejecutar al cargar la p gina
197 window.addEventListener("DOMContentLoaded", () => {
198   animateCards();
199   loadCertificateInfo();
200 });
201
202 // Efecto de part culas en el fondo (opcional, sutil)
203 function createParticle() {
204   const particle = document.createElement("div");
205   particle.style.position = "fixed";
206   particle.style.width = "2px";
207   particle.style.height = "2px";
208   particle.style.background = "rgba(102, 126, 234, 0.5)";
209   particle.style.borderRadius = "50%";
210   particle.style.pointerEvents = "none";
211   particle.style.left = Math.random() * window.innerWidth + "px";
```

```
212   particle.style.top = "-10px";
213   particle.style.zIndex = "-1";
214
215   document.body.appendChild(particle);
216
217   const duration = 3000 + Math.random() * 2000;
218   const animation = particle.animate([
219     { transform: "translateY(0px)", opacity: 0 },
220     { transform: "translateY(50px)", opacity: 0.5, offset: 0.2 },
221     { transform: `translateY(${window.innerHeight}px)`, opacity: 0
          }
222   ], {
223     duration: duration,
224     easing: "linear"
225   });
226
227   animation.onfinish = () => {
228     particle.remove();
229   };
230 }
231
232 // Crear part culas ocasionalmente
233 setInterval(() => {
234   if (Math.random() > 0.7) {
235     createParticle();
236   }
237 }, 500);
```

Listing 3: Script principal del cliente: main.js

Explicación Profesional

El script `main.js` se estructura bajo una arquitectura orientada a eventos, donde cada acción del usuario se enlaza con un controlador encargado de ejecutar operaciones específicas dentro del contexto seguro HTTPS. Los aspectos más relevantes son los siguientes:

- **Verificación del protocolo HTTPS**. Se obtiene dinámicamente el esquema del navegador (`window.location.protocol`) para determinar si la sesión activa utiliza un canal cifrado. Esta verificación permite al usuario validar la protección ofrecida por TLS 1.3 y observar la reacción del sistema frente a conexiones inseguras (HTTP).
- **Consulta del certificado digital**. Se realiza una petición al endpoint `/api/cert-info`, expuesto por el servidor Node.js, con el fin de recuperar la información del certificado X.509. Los datos incluyen el nombre común (CN), organización emisora, fechas de validez, algoritmos criptográficos y huellas digitales (SHA-256). Esta información permite evaluar la autenticidad del certificado y comprender sus atributos técnicos.
- **Actualización dinámica de la interfaz**. El script modifica elementos del DOM para mostrar resultados en tiempo real, proporcionando una visualización clara del estado criptográfico del canal. Además, aplica animaciones suaves que permiten interpretar progresivamente los datos recuperados.

- **Animaciones y efectos visuales**. Se incorporan elementos visuales como partículas dinámicas y transiciones CSS que contribuyen a una experiencia de usuario moderna sin afectar la funcionalidad criptográfica central.
- **Carga automática de métricas**. Al inicializar la página, se llama de forma automática a la función que recupera los parámetros del certificado, garantizando que el usuario observe desde el inicio la seguridad del canal.

En conjunto, `main.js` actúa como un puente entre el navegador y el servidor seguro, permitiendo analizar en tiempo real las propiedades del entorno TLS y proporcionando retroalimentación precisa sobre la validez criptográfica de la sesión.

8.3.2. Archivo `index.html`: Interfaz Gráfica de Usuario y Visualización de Datos Criptográficos

El archivo `index.html` constituye la capa de presentación del sistema. Su diseño incorpora una estructura moderna basada en tarjetas informativas (*cards*), íconos vectoriales y secciones diferenciadas que permiten al usuario visualizar el estado del servidor HTTPS, los parámetros del certificado digital y las métricas de seguridad asociadas al cifrado.

Código Fuente

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale
     =1.0" />
  <meta name="description" content="Servidor HTTPS seguro
     implementado en Node.js con certificado SSL autofirmado" />

  <link rel="stylesheet" href="./css/styles.css" />
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/
     libs/font-awesome/6.5.0/css/all.min.css">

  <title>Secure Web - Servidor HTTPS Profesional</title>
</head>

<body>
  <div class="header">
    <div class="lock-icon">
      <i class="fa-solid fa-file-shield"></i>
    </div>
    <h1>Secure Web Server</h1>
    <p class="subtitle">Servidor HTTPS profesional implementado en
       Node.js con certificado SSL autofirmado y encriptaci n de
       grado empresarial</p>
  </div>

  <div class="dashboard">

    <!-- Card: Estado de Conexi n -->
    <div class="card">
      <h2><span class="icon">
```

```
28          <i class="fa-solid fa-shield-halved"></i>
29        </span> Estado de Conexi n</h2>
30        <p>Verifica el estado de seguridad de tu conexi n HTTPS en
              tiempo real.</p>
31        <button id="check-security">Verificar Seguridad</button>
32        <div id="security-result" class="hidden"></div>
33      </div>
34
35      <!-- Card: Informaci n del Certificado SSL -->
36      <div class="card">
37        <h2><span class="icon">
38          <i class="fa-solid fa-scroll"></i>
39        </span> Certificado SSL</h2>
40        <p>Informaci n detallada del certificado de seguridad en uso
              .</p>
41        <button id="load-cert">Cargar Informaci n</button>
42        <div id="cert-info" class="hidden">
43          <div class="cert-details"></div>
44        </div>
45      </div>
46
47      <!-- Card: Detalles T cnicos -->
48      <div class="card">
49        <h2><span class="icon">
50          <i class="fa-solid fa-gears"></i>
51        </span> Especificaciones T cnicas</h2>
52        <p>Tecnolog as y protocolos de seguridad implementados en
              este servidor.</p>
53        <ul class="tech-list">
54          <li>Node.js + Express Framework</li>
55          <li>HTTPS con TLS 1.3</li>
56          <li>Certificado SSL X.509 v3</li>
57          <li>Encriptaci n AES-256-GCM</li>
58          <li>Arquitectura Modular ES6+</li>
59        </ul>
60      </div>
61
62      <!-- Card: M tricas de Seguridad -->
63      <div class="card">
64        <h2><span class="icon">
65          <i class="fa-solid fa-chart-line"></i>
66        </span> M tricas de Seguridad</h2>
67        <p>Indicadores clave del nivel de protecci n implementado.</
              p>
68        <div class="cert-details">
69          <div class="cert-item">
70            <span class="label">Nivel de Encriptaci n</span>
71            <span class="value">256-bit</span>
72          </div>
73          <div class="cert-item">
74            <span class="label">Protocolo</span>
75            <span class="value">TLS 1.3</span>
76          </div>
77          <div class="cert-item">
78            <span class="label">Algoritmo de Firma</span>
79            <span class="value">SHA-256</span>
80          </div>
81          <div class="cert-item">
```

```
82            <span class="label">Estado del Servidor</span>
83            <span class="value">
84              <span class="status-badge">Activo</span>
85            </span>
86          </div>
87        </div>
88      </div>
89
90      <!-- Card: Informaci n del Emisor -->
91      <div class="card">
92        <h2><span class="icon">
93          <i class="fa-solid fa-landmark"></i>
94        </span> Informaci n del Emisor</h2>
95        <p>Detalles sobre la entidad que emiti  el certificado SSL.<
            /p>
96        <div class="cert-details">
97          <div class="cert-item">
98            <span class="label">Organizaci n</span>
99            <span class="value" id="org-name">Cargando...</span>
100         </div>
101         <div class="cert-item">
102           <span class="label">Ubicaci n</span>
103           <span class="value" id="org-location">Cargando...</span>
104         </div>
105         <div class="cert-item">
106           <span class="label">Tipo de Certificado</span>
107           <span class="value" id="cert-type">Cargando...</span>
108         </div>
109         <div class="cert-item">
110           <span class="label">Validez</span>
111           <span class="value" id="cert-validity">Cargando...</span>
112         </div>
113       </div>
114     </div>
115
116     <!-- Card: Usos del Certificado -->
117     <div class="card">
118       <h2><span class="icon">
119         <i class="fa-solid fa-key"></i>
120       </span> Usos del Certificado</h2>
121       <p>Prop sitos autorizados para este certificado de seguridad
            .</p>
122       <ul class="tech-list" id="key-usage-list">
123         <li>Cargando informaci n...</li>
124       </ul>
125     </div>
126
127   </div>
128
129   <script src="./js/main.js"></script>
130 </body>
131 </html>
```

Listing 4: Estructura de la interfaz gráfica: index.html

Explicación Profesional

El documento `index.html` se centra en la presentación clara y didáctica de la información. Entre sus características principales se destacan:

- **Arquitectura basada en tarjetas**. Las tarjetas permiten organizar la información en bloques independientes, facilitando la lectura y el análisis de parámetros criptográficos como validez del certificado, algoritmos empleados y nivel de cifrado.
- **Integración con el script cliente**. Cada botón está relacionado con una función específica del archivo `main.js`, lo que promueve un flujo de trabajo ordenado y modular.
- **Indicadores visuales de seguridad**. Se emplean tonos verdes para representar estados seguros (HTTPS válido) y tonos rojizos para señalar fallas o alertas de seguridad.
- **Presentación detallada del certificado SSL**. El usuario puede observar los campos más relevantes del certificado X.509, como CN, O, OU, algoritmos criptográficos y huellas digitales en SHA-256.
- **Enfoque pedagógico**. La interfaz facilita la comprensión práctica del funcionamiento de TLS 1.3, permitiendo visualizar información que normalmente se encuentra oculta dentro del navegador.

8.4. Archivo `package.json`: Gestión de Dependencias y Configuración del Proyecto

El archivo `package.json` define la estructura lógica del proyecto Node.js, gestionando dependencias, metadatos, scripts y modo de ejecución.

Código Fuente

```
{
  "name": "secure-web",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "type": "module",
  "dependencies": {
    "express": "^5.1.0"
  }
}
```

Listing 5: Configuración del proyecto Node.js: package.json

Explicación Profesional

El archivo `package.json` cumple roles fundamentales dentro del ecosistema de desarrollo:

- **Declaración del proyecto**. Establece metadatos clave como nombre, versión y autor, necesarios para la gestión del proyecto en entornos profesionales.
- **Dependencias**. Incluye únicamente la librería `express`, lo que demuestra una arquitectura minimalista, ligera y optimizada para servir contenido bajo TLS 1.3.
- **Modo ES Modules**. La línea `"type": "module"` habilita el uso de `import` y `export`, lo que permite una programación moderna y modular.
- **Escalabilidad**. La estructura permite añadir futuras dependencias (como `helmet` para seguridad adicional) sin alterar la arquitectura base.

La implementación desarrollada constituye una base sólida para comprender los principios operativos de la criptografía aplicada a comunicaciones seguras. El uso de certificados digitales, la integración con un servidor HTTPS y el análisis estructural del certificado permiten afianzar conocimientos clave para la comprensión de la infraestructura de clave pública (PKI) en contextos académicos y profesionales.

9. Banco de Preguntas

9.1. Opción Múltiple — Una sola respuesta

Pregunta 1: Selección única

¿Para qué sirve un certificado digital en una página web?

a) Para que la página cargue más rápido

b) Para permitir decoraciones visuales

c) Para asegurar que la conexión sea privada y segura

d) Para guardar archivos del usuario

e) Para bloquear anuncios

Respuesta correcta: c)

9.2. Opción Múltiple — Varias respuestas correctas

Pregunta 2: Selecciones múltiples

¿Qué beneficios obtiene un usuario cuando navega en un sitio web con conexión segura (HTTPS)?

a) Su información viaja cifrada

b) Puede verificar que está hablando con el sitio correcto

c) Evita que terceros espíen sus datos

d) Tiene internet más rápido automáticamente

e) Las páginas cambian de diseño

Respuestas correctas: a), b), c)

9.3. Emparejamiento

Pregunta 3: Relación de columnas

Relaciona cada concepto con su significado:

Columna A		Columna B
HTTPS	→	Señala que la conexión es segura
Certificado digital	→	Identifica a quién pertenece una página
Cifrado	→	Evita que terceros lean la información
Navegador	→	Programa que muestra páginas web
Seguridad web	→	Conjunto de prácticas para proteger la información

Respuestas correctas: 1→B, 2→A, 3→C, 4→D, 5→E

9.4. Respuesta corta

Pregunta 4: Respuesta corta

¿Cómo se llama el proceso que convierte información en algo ilegible para protegerla?
Respuesta: Cifrado

9.5. Verdadero / Falso

Pregunta 5: Verdadero/Falso

HTTPS ayuda a proteger los datos que envías cuando llenas formularios en una página web.
Respuesta: Verdadero

10. Conclusiones

El desarrollo del presente proyecto permitió consolidar una comprensión amplia, profunda y rigurosa sobre los fundamentos y la aplicación práctica de la criptografía moderna en escenarios reales de comunicaciones seguras. A través de la construcción de un entorno HTTPS completo, desde la generación de certificados digitales hasta la implementación de un servidor web funcional, se integraron conceptos teóricos propios de la cátedra de Criptografía con técnicas aplicadas de ingeniería de software y seguridad informática.

En primer lugar, la generación de certificados digitales autofirmados mediante `OpenSSL` permitió comprender en detalle la estructura, propósito y funcionamiento de los certificados basados en el estándar `X.509`, así como el proceso mediante el cual se crean, empaquetan y verifican identidades digitales. La elaboración controlada del certificado —incluyendo la clave privada RSA, la información institucional y la validez temporal— permitió replicar, a escala académica, el comportamiento de una Autoridad Certificadora (CA). Esto reforzó la comprensión de los principios de la infraestructura de clave pública (PKI) y de los mecanismos de autenticación basada en confianza criptográfica.

En el plano técnico, la implementación del servidor HTTPS en Node.js permitió evidenciar el rol que cumple el cifrado de transporte no solo como mecanismo para proteger la confidencialidad, sino también como elemento esencial para asegurar la integridad, autenticidad y no repudio en las comunicaciones cliente-servidor. La utilización de la API nativa `https` y del módulo `crypto` posibilitó una interacción directa con los certificados generados, permitiendo la extracción y análisis detallado de componentes críticos tales como los nombres distinguidos (DN), los parámetros de la clave pública, las huellas digitales criptográficas (SHA-256), las fechas de validez y la información del emisor y del sujeto. De esta manera, se logró no solo servir contenido cifrado, sino también inspeccionar de forma programática las propiedades internas del certificado desde el backend.

De manera complementaria, el diseño de un cliente web alojado en el directorio `public/` permitió validar la relación entre el backend, los certificados y el navegador, demostrando cómo este último interpreta, valida y representa la información contenida en un certificado digital. Esto ofreció una visión clara del proceso mediante el cual el navegador determina la confiabilidad de un certificado, qué implicaciones tiene que este sea autofirmado, cómo se estructura la cadena de confianza y cómo se comunica esta validación al usuario a través de la interfaz gráfica.

La organización modular del proyecto facilitó una comprensión estructurada de las etapas que componen la implementación completa de seguridad TLS. Esta arquitectura favorece buenas prácticas de desarrollo, tales como la separación de responsabilidades, la claridad del flujo de ejecución y la capacidad de mantenimiento

del código, aspectos fundamentales para entornos profesionales de ingeniería de software.

Asimismo, el análisis exhaustivo del código incluido en los archivos `server.js` y `app.js` permitió profundizar en las operaciones criptográficas internas que ejecuta el servidor, desde la carga de materiales criptográficos hasta su uso en el proceso de *handshake* TLS. Este análisis puso de manifiesto cómo los principios teóricos de la criptografía (RSA, hashing, certificados X.509, cadenas de confianza y cifrado simétrico) se implementan en sistemas reales utilizando herramientas modernas.

Desde una perspectiva académica, el proyecto refuerza la importancia de conectar los fundamentos matemáticos y teóricos de la criptografía con su implementación práctica en el desarrollo de soluciones web seguras. La creación del apartado de preguntas y actividades evaluativas permitió reforzar el proceso cognitivo del estudiante y consolidar la apropiación de los conceptos aprendidos. Esta metodología es coherente con la misión formativa de la Universidad Central del Ecuador y con los objetivos de la cátedra de Criptografía dentro de la Facultad de Ingeniería.

Finalmente, la integración de certificados, arquitectura de software, análisis criptográfico, implementación HTTPS, frontend, backend y documentación técnica permitió construir un sistema robusto y seguro en un entorno local. El proyecto demuestra que la seguridad informática no es un componente accesorio, sino un elemento esencial del diseño de sistemas modernos. La práctica realizada constituye una base sólida para el estudio futuro de la infraestructura de clave pública (PKI), certificados avanzados, firmas digitales y desarrollo seguro de aplicaciones web.

En síntesis, este proyecto no solo permitió comprender cómo se genera y utiliza un certificado digital, sino que también articuló teoría, práctica y análisis crítico en torno a la comunicación segura, fortaleciendo las competencias técnicas, analíticas y profesionales del estudiante en el ámbito de la seguridad web y la criptografía aplicada.

Referencias

[1] E. Rescorla, "The transport layer security (tls) protocol version 1.3," RFC 8446 (IETF), 2018, online; available at https://datatracker.ietf.org/doc/html/rfc8446.

[2] B. Dowling, M. Fischlin, F. Günther, and D. Stebila, "A cryptographic analysis of the tls 1.3 handshake protocol," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, Denver, CO, USA, 2015, pp. 1197–1210.

[3] A. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Zanella-Béguelin, "Proving the tls handshake secure (as it is)," in *Advances in Cryptology – CRYPTO 2014*, J. A. Garay and R. Gennaro, Eds. Berlin, Heidelberg: Springer, 2014, pp. 235–255.

[4] H. Krawczyk and H. Wee, "The optls protocol and tls 1.3," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, Saarbrücken, Germany, 2016, pp. 81–96.

[5] Y. Liu, W. Tome, L. Zhang, D. Choffnes, D. Levin, B. Maggs, A. Mislove, A. Schulman, and C. Wilson, "An end-to-end measurement of certificate revocation in the web's pki," in *Internet Measurement Conference (IMC '15)*, Tokyo, Japan, 2015, pp. 183–196.

[6] S. Helme, "An internet-wide view of https certificate revocations," in *IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 2024, pp. 1–18.

[7] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile," RFC 5280 (IETF), 2008, online; available at https://datatracker.ietf.org/doc/html/rfc5280.

[8] Y. Sheffer, R. Holz, and P. Saint-Andre, "Recommendations for secure use of transport layer security (tls) and datagram transport layer security (dtls)," RFC 7525 (IETF), 2015, online; available at https://datatracker.ietf.org/doc/html/rfc7525.

[9] R. Sasse, J. Cuellar, and S. Fischmeister, "A systematic literature review on inter-service security threats in microservice architectures," *IEEE Access*, vol. 11, pp. 45 123–45 142, 2023.

[10] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. V. Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: Large-scale evaluation of remote javascript inclusions," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*, Raleigh, NC, USA, 2012, pp. 736–747.

Anexos

Repositorio del Código Fuente

El desarrollo completo del proyecto, incluyendo el código fuente estructurado de manera modular, los componentes del servidor HTTPS, la implementación de certificados digitales, así como los elementos visuales y funcionales de la aplicación web, se encuentra disponible públicamente en el siguiente repositorio oficial del equipo:

`https://github.com/Dennis290699/security-connect`

Este repositorio constituye el respaldo técnico del trabajo presentado en este informe y permite la verificación, revisión y reutilización del código para fines académicos o investigativos.

Agradecimientos

Expresamos nuestro más profundo agradecimiento al **catedrático de la asignatura de Criptografía**, quien a lo largo del desarrollo de este proyecto supo brindar orientación conceptual, rigurosidad académica y un enfoque metodológico basado en las mejores prácticas de la disciplina. Su guía permitió que este trabajo se

consolidara no solo como un ejercicio técnico, sino como una experiencia formativa de alto valor académico.

Asimismo, extendemos un sincero reconocimiento a los lectores de este documento. Su interés y dedicación al explorar cada sección contribuyen a fortalecer la comprensión colectiva sobre la aplicación práctica de los protocolos criptográficos y la construcción de conexiones seguras mediante HTTPS. Esperamos que el contenido aquí presentado sea de utilidad, inspire nuevas investigaciones y aporte a la consolidación de buenas prácticas en el ámbito de la seguridad informática.

Atentamente,
Grupo 6