UNIVERSIDAD AUTÓNOMA DE NUEVO LEÓN

FACULTAD DE CIENCIAS FISICO MATEMÁTICAS

PROGRAMACIÓN AVANZADA PARA CIBERSEGURIDAD

PRODUCTO INTEGRADOR DE APRENDIZAJE

Grupo: 063

Mtro. Perla Marlene Viera González

Ángel Adrián Alvares Flores	2050690
Abraham Alejandro Carreón Soriano	1953829
Jesús Kenneth Maurizio Martínez Vázquez	1875474
Raúl Alejandro Ríos Turrubiates	2132823

Monterrey, N.L. a 20 de noviembre de 2025

Resumen Ejecutivo

El análisis completo del payload evaluado confirma que se trata de un cliente HTTP benigno diseñado exclusivamente para realizar verificaciones de estado mediante una solicitud GET a la ruta /status en un servidor local. El examen estático mostró un binario simple, sin funciones ocultas ni capacidades ofensivas, mientras que las pruebas dinámicas validaron un comportamiento transparente, limitado y sin exfiltración de datos. La arquitectura del código prioriza simplicidad, uso controlado de recursos y validación de parámetros. Los riesgos asociados (principalmente fallos de conexión, validación de entrada y bloqueo por operaciones de red) fueron mitigados de forma proporcional al propósito del laboratorio. En conjunto, la evidencia técnica respalda que el payload es seguro, ético y adecuado para entornos controlados de prueba, cumpliendo con los requisitos del proyecto y sirviendo como base para la entrega final

Contenido

1 Descripción del payload y Límites éticos

El payload realiza una petición HTTP GET /status de un servidor local por defecto en 127.0.0.1:8080 (la dirección IP puede modificarse mediante un parámetro en el cliente), con el objetivo de comprobar el estado/latido del servicio de manera segura y no intrusiva; su comportamiento está diseñado para ser benigno (solo solicita información de estado), maneja respuestas esperadas y manejo básico de errores para evitar bloqueos, y no envía ni extrae datos sensibles ni ejecuta operaciones remotas en el servidor.

2 Diseño e implementación técnica

2.1 Manejo de memoria

- Locación en pila: El programa evita la asignación dinámica explícita (no hay new, malloc ni contenedores que hagan heap internamente en el fragmento dado), por lo que toda la memoria temporal clave (estructuras y buffer) está en la pila y se libera automáticamente al salir de cada función.
- Recursos del sistema: El recurso principal que requiere liberación explícita es el descriptor de socket sock. El código llama a close(sock) al final de runClient para liberar el descriptor; sin embargo, hay casos en los que los errores se imprimen pero no se hace return o close inmediatamente (por ejemplo, si socket() falla el valor de sock es negativo y no debe cerrarse; si connect() falla, el descriptor sí existe y debería cerrarse antes de salir).
- Recepción de datos: recv escribe hasta sizeof(buffer) - 1 bytes y el código añade '\0' en buffer[bytesReceived], asegurando nul-terminación para impresión segura. Se debe controlar el caso de recv que devuelve -1 (error) para evitar usar un índice negativo.
- Ausencia de fugas de heap: Dado que no hay heap usage en el código actual, no hay fugas de heap; sin embargo, el manejo de recursos del sistema (sockets) debe ser robusto ante errores para evitar descriptores abiertos.

2.2 Decisiones relevantes

- Uso de BSD sockets (AF_INET, SOCK_STREAM): decisión simple y portátil para una conexión TCP IPv4. Trade-off: no hay soporte directo para IPv6; si se requiere compatibilidad IPv6, convendría usar getaddrinfo en lugar de inet_pton y sockaddr_in.
- Validación con std::regex: se eligió una expresión regular para validar la sintaxis de la IPv4 antes de intentar conectar. Ventaja: evita intentos de conexión con entradas mal formadas; contrapartida: std::regex puede ser más pesado en rendimiento y tamaño binario. Alternativa ligera: validar con inet_pton y revisar su resultado.

- Bloqueo y E/S sincrónica: el cliente usa llamadas bloqueantes (connect, send, recv) en un solo hilo (sencillo y predictible) pero en redes lentas puede bloquear la ejecución; si se necesita mayor robustez o concurrencia, considerar timeouts, select/poll o sockets no bloqueantes.
- Buffer fijo de 4096 bytes: elección práctica para la mayoría de las respuestas de /status. Ventaja: simplicidad; riesgo: respuestas extremadamente grandes requerirían múltiples iteraciones (ya soportadas por el loop) pero si la respuesta excede repetidamente este tamaño puede impactar memoria/latencia. Tamaño configurable sería una mejora.
- Construcción manual de la petición HTTP: se arma una petición HTTP/1.1 simple con Connection: close para forzar cierre del socket tras la respuesta (facilita limpieza de recursos). Trade-off: no hay parsing HTTP avanzado ni manejo de chunked transfer-encoding; para interacciones más complejas convendría usar una librería HTTP o implementar parsing más completo.
- Conversión de puerto con std::atoi: método sencillo pero sin comprobación de errores detallada (si argv[2] no es numérico atoi devuelve 0). Se decidió validar el rango con validarPuerto, pero std::stoi con manejo de excepciones daría validación más estricta.
- Manejo de errores básico y salida inmediata: el programa imprime errores y usa exit(1) en validaciones críticas (IP/puerto). Decisión orientada a simplicidad en herramientas de línea de comandos; para una biblioteca o servicio se preferiría propagar errores con códigos y mensajes detallados.
- Seguridad y benignidad: el payload está diseñado como health-check benigno (solo GET /status) sin envío de datos sensibles ni ejecución remota. No obstante, por seguridad operativa se asume uso en entornos controlados y se recomienda validar explícitamente los endpoints y emplear TLS si se comunican secretos o en redes inseguras.

3 Metodología de pruebas

3.1 Entorno

- Nombre de la Máquina Virtual: kali-linux-2024.2-virtualbox-amd64
- Arquitectura: x86_64 (64-bits)
- Compilador: g++ 14.2.0
- Sistema operativo: Debian (Kali Linux 2024.
- IP: 10.0.2.15
- Gateway: 10.0.2.1
- Mascara de red: 255.255.255.0
- Adaptador de red: Red Interna

3.2 Pasos para las pruebas

Antes de cada prueba se crea (o se vuelve a poner) una snapshot de la máquina virtual que se usará para evitar estados residuales. Verificar que la VM esté configurada en red interna (NAT/host-only/solo-interna según tu entorno) para asegurar que las pruebas se ejecutan en un entorno controlado.

El flujo de cada prueba es: (1) compilar el cliente con la modificación a probar, (2) poner en ejecución un servidor local que responda a GET /status, (3) ejecutar el binario cliente con los parámetros de la prueba, (4) recolectar evidencias (logs, pcap, capturas) y (5) comparar el comportamiento obtenido con el esperado. Al final se documentan resultados y se restaura la snapshot si procede.

4 Análisis estático

El binario fue cargado en Ghidra y se identificó que está compilado en C++ con dependencias estándar, sin símbolos externos ni componentes ocultos. La vista de strings muestra principalmente rutas del sistema y funciones de la STL, sin presencia de cadenas sospechosas ni indicios de persistencia o capacidades ofensivas.

El grafo de funciones refleja un flujo simple y lineal: lectura de argumentos, creación del socket, conexión al servidor, envío de una solicitud HTTP GET a /status, recepción de la respuesta y cierre del proceso.

No se observaron llamadas a API inusuales, cifrado, manipulación de archivos ni lógica no documentada. En conjunto, los hallazgos estáticos indican que el binario implementa únicamente la funcionalidad esperada de cliente HTTP y no presenta comportamientos adicionales.

5 Análisis dinámico

Durante la ejecución del binario ./bin/client utilizando el comando ./bin/client 127.0.0.1 8080 y un servidor de prueba en Python, se observó que el comportamiento del payload fue consistente con lo esperado. El cliente estableció una conexión TCP al puerto 8080 y realizó una única solicitud HTTP GET hacia la ruta /status. El servidor respondió correctamente con un código 200 OK y el mensaje "Servicio en línea", tras lo cual el payload procesó la respuesta y finalizó su ejecución sin intentar acciones adicionales.

No se detectaron comportamientos anómalos como persistencia, comandos remotos o exfiltración de datos. La comunicación se mantuvo limpia y estrictamente limitada a la petición HTTP prevista en el diseño del proyecto. Las evidencias incluyen capturas de tráfico y logs asociados al análisis.

6 Ingeniería inversa

6.1 Funciones y flujo relevante

El flujo interno del binario /bin/client es simple y lineal, coherente con la función de un cliente HTTP básico. Al observar el binario y su descompilación, se aprecia que el programa inicia leyendo los argumentos recibidos por consola (IP y puerto). Después valida que la cantidad de parámetros sea correcta y procede a crear un socket TCP utilizando las funciones estándar de la biblioteca de red. Una vez creado el socket, construye la estructura de conexión con la dirección del servidor y realiza el intento de conexión.

Cuando la conexión se establece, el binario prepara la cadena correspondiente a la solicitud HTTP GET dirigida a la ruta /status. Esta construcción suele realizarse mediante funciones de la STL o sprintf/stringstream, según la implementación observada. Luego envía la solicitud a través del socket y pasa a la fase de lectura. El cliente utiliza funciones como recv() o equivalentes C++ para recibir la respuesta del servidor, la almacena en un buffer y la muestra o procesa antes de cerrar la conexión y finalizar el programa.

Entre las funciones relevantes identificadas durante la ingeniería inversa destacan las asociadas al manejo de sockets (creación, conexión, envío y recepción), las rutinas de manejo de cadenas para construir la solicitud HTTP, y una función principal que orquesta todo el flujo desde la validación de argumentos hasta la finalización. No se encontraron funciones ocultas, módulos auxiliares externos ni bifurcaciones lógicas complejas; toda la estructura se reduce a inicializar, conectar, enviar, recibir y terminar, sin operaciones adicionales.

7 Riesgos y mitigaciones

7.1 Falla en la creación o conexión del socket:

Existe la posibilidad de que el socket no se cree correctamente (socket() devuelve < 0) o que la conexión al servidor falle (connect() < 0), por causas como puerto ocupado, servidor no activo o IP incorrecta.

- Impacto: el cliente no podrá comunicarse con el servidor y la ejecución puede terminar sin liberar el descriptor o sin reportar claramente la causa.
- Mitigación aplicada: se implementan verificaciones de error con mensajes en std::cerr que informan al usuario sobre el fallo. El código también cierra el socket al final de la ejecución, reduciendo el riesgo de fugas de recursos. Para robustez adicional, se podría mejorar retornando en cada error crítico o integrando manejo RAII.

7.2 Validación insuficiente de entrada (IP y puerto):

Los parámetros del usuario (ip, port) pueden ser inválidos o estar fuera de rango. Si no se validaran, podría producirse un intento de conexión fallido o comportamiento indefinido.

- Impacto: posibles errores en tiempo de ejecución o conexiones a direcciones no deseadas.
- Mitigación aplicada: se implementaron funciones validarIP() y validarPuerto() para garantizar que las direcciones IP cumplan con el formato IPv4 y que los puertos estén en el rango permitido (1–65535). Si las validaciones fallan, el programa termina de forma segura (exit(1)).

7.3 Bloqueo por operaciones de red (socket bloqueante):

Las llamadas a connect(), send() y recv() son bloqueantes. Si el servidor no responde o la red tiene problemas, el cliente puede quedar esperando indefinidamente.

- Impacto: congelamiento del proceso o tiempos de espera excesivos.
- Mitigación aplicada: se considera un riesgo aceptable para el entorno de laboratorio (localhost), donde la latencia es mínima. Para entornos reales, se recomienda agregar setsockopt() con SO_RCVTIMEO y SO_SNDTIMEO para limitar el tiempo de espera.

7.4 Riesgo de lectura incompleta o corrupción de buffer:

El uso de un buffer de tamaño fijo (char buffer[4096]) implica que si el servidor envía más datos, estos se recibirán en varias iteraciones de recv(). Si no se agregara el terminador '\0', podrían imprimirse caracteres no válidos.

- Impacto: salida corrupta o comportamiento indefinido en impresión.
- Mitigación aplicada: el código añade buffer[bytesReceived] = '\0' en cada iteración, asegurando que la cadena esté correctamente terminada y evitando lecturas fuera de rango.

7.5 Falta de soporte para IPv6 o DNS:

La conexión está limitada a direcciones IPv4 y requiere que el usuario proporcione la IP explícita.

- Impacto: el cliente no podría probar servidores con nombres de dominio o direcciones IPv6.
- Mitigación aplicada: riesgo documentado y aceptado dado que el objetivo del proyecto es realizar pruebas locales controladas en 127.0.0.1.

7.6 Ausencia de cifrado o autenticación:

El payload usa HTTP sin TLS, lo que en un entorno abierto podría exponer información de conexión.

- Impacto: potencial exposición del tráfico si se ejecutara fuera del entorno local.
- Mitigación aplicada: se define que el uso es benigno y local, con comunicación únicamente sobre 127.0.0.1, sin transmisión de datos sensibles, por lo que el riesgo se considera nulo en el contexto de pruebas.

8 Conclusiones y trabajo futuro

La evaluación completa del payload confirma que su diseño, funcionamiento y riesgos asociados están plenamente alineados con el propósito declarado del proyecto: operar como un cliente HTTP simple, seguro y limitado a realizar verificaciones de estado por medio de una petición GET al endpoint `/status`. Tanto el análisis estático como el dinámico evidencian una arquitectura transparente, sin funciones ocultas ni comportamiento sospechoso; el binario se ejecuta de forma lineal, maneja recursos de manera correcta y no realiza acciones ajenas a su objetivo. El análisis de ingeniería inversa confirmó que las funciones relevantes se centran exclusivamente en la creación del socket, la conexión, el envío de la solicitud y la recepción de la respuesta, sin lógica adicional o bloques ofuscados.

Los riesgos identificados (fallos de conexión, validación de entrada, operaciones bloqueantes y manejo de buffers) fueron atendidos de forma proporcional al alcance del laboratorio, garantizando un uso seguro en entornos controlados. En conjunto, la evidencia técnica respalda que el payload es benigno, ético y adecuado para tareas de monitoreo básico, constituyendo una base sólida para mejoras futuras, como soporte ampliado de protocolos, validaciones más robustas y funcionalidades extendidas según las necesidades del proyecto.

Anexo A – Tabla de IOCs

Indicador de Compromiso (IoC)	*Descripción*	*Por qué es sospechoso*
Conexiones a IPs externas	El cliente establece comunicación con direcciones distintas de 127.0.0.1.	El cliente está diseñado para operar solo en localhost; conexiones externas indican manipulación o abuso.
Solicitudes a rutas distintas de /status	Se observan solicitudes HTTP hacia otros endpoints.	Sugiere alteración del binario para realizar acciones no previstas o recopilación de información.
Múltiples solicitudes GET consecutivas	El cliente envía más de una petición por ejecución.	El diseño contempla una sola petición; múltiples llamados pueden indicar bucles maliciosos o escaneo.
Tráfico saliente no HTTP	Se detectan protocolos o payloads no relacionados con HTTP.	El binario solo debe enviar una petición GET; otro tipo de tráfico implica comportamiento malicioso.
Escritura o lectura de archivos locales	El programa interactúa con el sistema de archivos.	No está contemplado en su funcionalidad; podría significar exfiltración o persistencia.
Creación de procesos secundarios	El cliente ejecuta comandos o abre subprocesos.	La aplicación no debe ejecutar nada más que la conexión TCP.
Persistencia o reinicios automáticos	El proceso se mantiene activo o se relanza a sí mismo.	Indica intención de mantenerse residente, comportamiento no previsto.
Uso de cifrado inesperado o datos ofuscados	El tráfico contiene datos cifrados sin motivo.	El cliente solo envía texto plano HTTP; ofuscación podría esconder exfiltración.
Modificación del binario o tamaño alterado	El ejecutable cambia sin motivo aparente.	Señal de manipulación, inyección o infección del archivo.
Consumo de CPU o red inusual	Uso excesivo de recursos.	La operación del cliente es mínima; cualquier aumento sostenido es anómalo.

Anexo B – Lista de archivos

Archivos en /análisis

Archivo	*Descripción*
Analisis_Ghidra.zip	Archivo comprimido con los resultados del análisis con Ghidra.
dynamic_log.txt	Archivo de registro del análisis dinámico.
dynamic_notes.md	Archivo de notas del análisis dinámico.
static_notes.md	Archivo de notas del análisis estático.

Archivos en /evidence

Archivo	*Descripción*
/dynamic_analysis/dynamic_client.png	Captura de pantalla de captura de tráfico del cliente.
/dynamic_analysis/dynamic_server.png	Captura de pantalla de captura de tráfico del servidor.
/dynamic_analysis/dynamic_traffic.pcapng	Archivo de captura de paquetes de Wireshark
/static_analysis/Decompiler.png	Evidencia del decompilador para análisis estático.
/static_analysis/Functions.png	Evidencia de funciones obtenidas en el análisis estático.
/static_analysis/Results_Summary.png	Evidencia del resumen de resultados obtenidos del análisis estático.
/static_analysis/Strings.png	Evidencia de las cadenas obtenidas en el análisis estático.
/static_analysis/strings.txt	Archivo de texto con las cadenas obtenidas en el análisis estático.
/test_n1/evidence_20261106_compilacion.png	Evidencia de compilación de cliente en prueba test n1.
/test_n1/evidence_20251106_ejecucion.png	Evidencia de ejecución del cliente en prueba test n1.
/test_n1/evidence_20251106_servidor.png	Evidencia de ejecución del servidor en prueba test n1.
/test_n2/evidence_20251112_cliente.png	Evidencia de ejecución del cliente en prueba test n2.
/test_n2/evidence_20251112_servidor.png	Evidencia de ejecución del servidor en prueba test n2.
/test_n2/evidence_20251112_vm_snapshot.png	Evidencia de creación de Snapshot para la prueba test n2.