

Seguridad y Autenticación en Microservicios con C++

Antonio Espín Herranz

Contenidos

- Seguridad y Autenticación en Microservicios con C/C++
- Autenticación con JWT (JSON Web Tokens):
 - Generación y validación de tokens JWT en C++.
 - Uso de bibliotecas criptográficas como OpenSSL para manejar tokens seguros.
- Protección de la comunicación entre microservicios:
 - Implementación de encriptación con SSL/TLS.
 - Configuración de políticas de seguridad para prevenir ataques como CSRF y XSS.

Autenticación con JWT

JSON Web Tokens

JWT

- JSON Web Tokens
 - Formato compacto y seguro
 - Sirve para transmitir información entre partes como un objeto JSON
 - Se utiliza para autenticación y autorización
- Tiene 3 partes codificadas en Base64
 - **HEADER.PAYLOAD.SIGNATURE**

jwt Header

- Contiene el tipo de token y el algoritmo de firma:

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

jwt - PayLoad

- Contiene los datos (claims) que quieres transmitir, como el usuario, roles, expiración, etc.

```
{  
  "sub": "1234567890",  
  "name": "Antonio",  
  "admin": true,  
  "exp": 1699999999  
}
```

jwt - Signature

- Es una firma digital generada con el algoritmo especificado (como HMAC o RSA), que garantiza que el contenido no ha sido alterado.

JWT se utiliza

- **Autenticación de usuarios:** El servidor genera un token tras el login y el cliente lo usa en cada petición.
- **Autorización:** El token puede incluir roles o permisos.
- **Intercambio seguro de datos:** Entre servicios o microservicios.

Tipo de Seguridad que ofrece

- JWT **no cifra** el contenido, pero **sí lo firma**. Proporciona:
 - **Integridad**: Nadie puede modificar el contenido sin invalidar la firma.
 - **Confidencialidad**: Cualquiera puede leer el contenido si intercepta el token (por eso se recomienda usar HTTPS).
 - **Autenticidad**: El receptor puede verificar que el token fue emitido por una fuente confiable.

Tipos de JWT

Tipo	Seguridad	Uso común
JWS	Firmado	Autenticación y autorización
JWE	Encriptado	Transmisión de datos sensibles

Tipos de proyectos con JWT y C++

- Servidores REST en C++ (con Boost.Beast, Crow, etc.)
- Microservicios en entornos embebidos
- Aplicaciones cliente que consumen APIs protegidas
- Sistemas distribuidos que necesitan autenticación sin sesiones
- Integración con sistemas web donde C++ actúa como backend o middleware

Trabajar en jwt con C++

- La biblioteca más popular en **jwt-cpp**. Es ligera moderna y compatible con C++11.
- Se puede crear, firmar y verificar tokens fácilmente y soporta algoritmos como HS256, RS256 y ES256.

HS256, RS256 y ES256

Algoritmo	Tipo de Criptografía	Clave	Seguridad	Velocidad
HS256	Simétrica (HMAC + SHA-256)	Una sola clave compartida	Menor (clave compartida)	Muy rápida
RS256	Asimétrica (RSA + SHA-256)	Par de claves pública/privada	Alta	Más lenta que HS256
ES256	Asimétrica (ECDSA + SHA-256)	Par de claves pública/privada	Muy alta	Más eficiente que RS256

HS256, RS256 y ES256

- **HS256 (HMAC-SHA256)** Usa una **clave secreta compartida** para firmar y verificar. Es simple y rápido, pero **ambas partes deben tener la misma clave**, lo que puede ser un riesgo si se comparte mal.
- **RS256 (RSA-SHA256)** Utiliza **criptografía asimétrica**: una clave privada para firmar y una clave pública para verificar. Esto permite que el emisor mantenga su clave privada segura, mientras que cualquiera con la clave pública puede verificar la firma.
- **ES256 (ECDSA-SHA256)** También es asimétrico, pero usa **curvas elípticas**, lo que lo hace más eficiente y seguro con claves más pequeñas. Es ideal para dispositivos con recursos limitados o sistemas que requieren alta seguridad.

HS256, RS256 y ES256

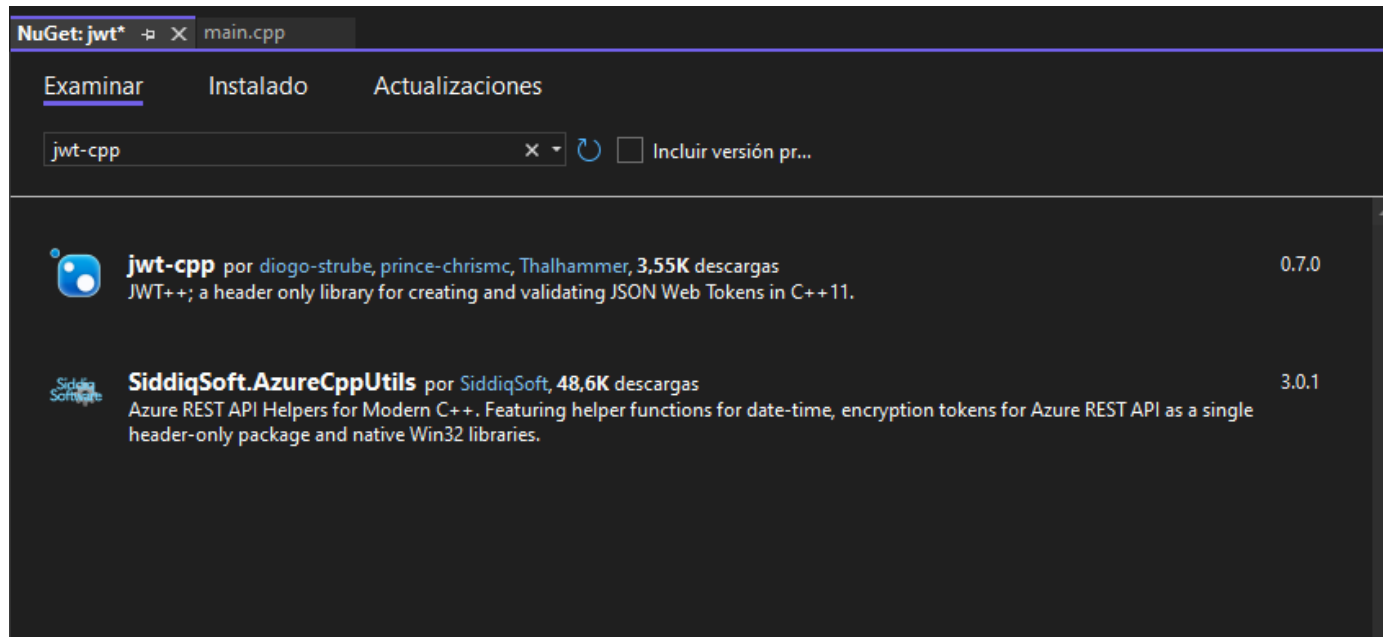
- **HS256**: útil si controlas ambos extremos (emisor y receptor) y necesitas velocidad.
- **RS256**: preferido en sistemas distribuidos donde el receptor no debe conocer la clave privada.
- **ES256**: recomendado para aplicaciones modernas que priorizan seguridad y eficiencia.

Flujo jwt

- **Inicio de sesión:**
 - El cliente envía las credenciales al servidor
- **Creación del JWT:**
 - El servidor valida las credenciales y genera un token firmado
- **Validación del JWT:**
 - El cliente guarda el token y lo envía al servidor en futuras peticiones, el servidor lo verifica
- **Solicitud y respuesta:**
 - Si el token es válido, el servidor procesa la solicitud y responde.

Instalar jwt-cpp en Visual Studio

- Con el botón derecho sobre el proyecto:
- Administrar paquetes **NuGet**
- **Examinar: jwt-cpp**



Instalar jwt-cpp en Visual Studio

- Se instalan estas dependencias (se puede crear una plantilla y configurarla para nuevas soluciones).

Paquete	¿Para qué sirve?
jwt-cpp	La librería principal para crear y verificar JSON Web Tokens en C++
OpenSSL	Necesaria para algoritmos de firma asimétrica como RS256 y ES256
picojson	Una alternativa ligera a nlohmann-json para manejar JSON (usada internamente por jwt-cpp si no usas otra)

Crear un token

- usuario se ha recogido de la request. Se habrán enviado datos por POST → auto
datos = crow::json::load(req.body);

- Primero habrá que validar, si viene el campo usuario:

```
if (datos.has("usuario"))  
    std::string usuario = datos["usuario"].s()
```

```
auto token = jwt::create()  
    .set_issuer("Antonio")  
    .set_payload_claim("usuario", jwt::claim(usuario))  
    .set_expires_at(std::chrono::system_clock::now() + std::chrono::minutes{ 30 })  
    .sign(jwt::algorithm::hs256{ jwt_secret });
```

Crear el token

- Token JWT:
 - **Issuer:** quién emite el token (en este caso, "Antonio").
 - Si hay varios emisores de tokens este valor puede variar.
 - Incluso, se puede validar
 - **Claim** personalizado: se añade "usuario" como dato dentro del token.
 - **Expiración:** el token caduca en 30 minutos.
 - **Firma:** se firma con el algoritmo HS256 usando la clave secreta `jwt_secret`.

payload

- Son los datos que se quieren transmitir dentro del token.

Claim	Descripción
<code>sub</code>	Subject: el identificador del usuario (ID único, email, etc.)
<code>iss</code>	Issuer: quién emitió el token (como ya vimos)
<code>aud</code>	Audience: quién debe aceptar el token (por ejemplo, tu API)
<code>exp</code>	Expiration: fecha/hora en que el token expira
<code>iat</code>	Issued At: cuándo se emitió el token
<code>nbf</code>	Not Before: cuándo empieza a ser válido
<code>role</code> / <code>scope</code>	Permisos o roles del usuario (admin, user, etc.)

Ejemplo

```
{  
  "sub": "user-12345",  
  "iss": "https://auth.miempresa.com",  
  "aud": "https://api.miempresa.com",  
  "exp": 1694352000,  
  "iat": 1694348400,  
  "role": "admin"  
}
```

Implementación

jwt::create()

```
.set_issuer("https://auth.miempresa.com")  
.set_subject("user-12345")  
.set_audience("https://api.miempresa.com")  
.set_issued_at(std::chrono::system_clock::now())  
.set_expires_at(std::chrono::system_clock::now() +  
std::chrono::minutes{30})  
.set_payload_claim("role", jwt::claim(std::string("admin")))  
.sign(jwt::algorithm::hs256{"clave-secreta"});
```

Respuesta al cliente

- `crow::json::wvalue respuesta;`
- `respuesta["token"] = token;`
- `return crow::response(respuesta);`

Token generado

```
{  
  "token":  
  "eyJhbGciOiJIUzI1NiJ9.eyJleHAiOjE3NTc0MTg1ODIsImZlcyI6IkJudG9uaW8iLCJ1c3VhcmlvIjoieW50b25pbyJ9.uBwFpR2cKO05SYIYun9UBtTH2hMnxhITxGiNLheJrjc"  
}
```

- El token se almacena en un lugar seguro y luego hay que enviarlo.

Enviar el token

- Desde javascript:

```
fetch("https://tu-api.com/protegido", {  
  method: "GET",  
  headers: {  
    "Authorization": "Bearer TU_TOKEN_AQUI",  
    "Content-Type": "application/json"  
  }  
})  
.then(res => res.json())  
.then(data => console.log(data));
```

Validar el token

```
CROW_ROUTE(app, "/protegido")
.methods("GET"_method)
([jwt_secret](const crow::request& req) {
    // Extraer el encabezado Authorization
    auto auth_header = req.get_header_value("Authorization");

    // Verificar que el encabezado exista y comience con "Bearer "
    if (auth_header.substr(0, 7) != "Bearer ")
        return crow::response(401, "Token no proporcionado o mal formado");

    // Extraer el token
    std::string token = auth_header.substr(7);

    try {
        // Verificar y decodificar el token
        auto decoded = jwt::decode(token);
```

```
// Verificar la firma y la validez
        auto verifier = jwt::verify()
            .allow_algorithm(jwt::algorithm::hs256{ jwt_secret })
            .with_issuer("Antonio");
        verifier.verify(decoded);
        // Extraer el claim "usuario"
        std::string usuario =
            decoded.get_payload_claim("usuario").as_string();

        // Respuesta exitosa
        crow::json::wvalue respuesta;
        respuesta["mensaje"] = "Acceso concedido";
        respuesta["usuario"] = usuario;
        return crow::response(respuesta);
    }
    catch (const std::exception& e) {
        // Token inválido o expirado
        return crow::response(401, "Token inválido: " + std::string(e.what()));
    }
});
```

Validar el token

- Verifica que el encabezado Authorization esté presente y tenga el formato correcto.
- Extrae el token JWT.
- Decodifica el token y verifica su firma con la clave secreta `jwt_secret`.
- Comprueba que el emisor (issuer) sea "Antonio".
- Extrae el claim "usuario" si el token es válido.
- Devuelve una respuesta JSON con el nombre del usuario si todo está correcto.


OpenSSL

Bibliotecas criptográficas

Protección de la comunicación entre microservicios

Implementación de encriptación SSL/TLS

- Diferencias entre SSL / TLS

Característica	SSL (Secure Sockets Layer)	TLS (Transport Layer Security)
 Origen	Creado por Netscape en los 90s	Evolución de SSL, estandarizado en 1999
 Seguridad	Menos segura, ya obsoleta	Más segura, moderna y mantenida
 Versiones	SSL 2.0, SSL 3.0 (ambas obsoletas)	TLS 1.0 → 1.3 (la 1.3 es la actual)
 Algoritmos	Vulnerable a ataques conocidos	Mejores algoritmos y cifrado fuerte
 Compatibilidad	Aún se menciona por costumbre	Usado realmente en casi todos los sistemas
 Uso actual	Prácticamente desactivado	Estándar en HTTPS, gRPC, VPNs, etc.

Configuración para evitar ataques CSRF / XSS

CSRF

- **Cross-Site Request Forgery**

- **CSRF** es un tipo de ataque en el que un usuario autenticado en un sitio web es engañado para ejecutar una acción no deseada en ese mismo sitio, sin saberlo.
- Nos podemos proteger de estos ataques utilizando las librerías: Crow, Boost.Beast o Pistache.

XSS

- **XSS (Cross-Site Scripting)** es una de las vulnerabilidades más comunes y peligrosas en aplicaciones web.
- Afecta principalmente a lenguajes como JavaScript, **los servicios web escritos en C++ también pueden ser vulnerables** si no se validan correctamente los datos que se envían al navegador.
- **XSS** es un tipo de ataque que permite a un atacante **inyectar código malicioso (generalmente JavaScript)** en páginas web que otros usuarios visitan.
 - El código se ejecuta en el navegador de la víctima, no en el servidor, lo que lo hace difícil de detectar.

¿Qué puede hacer un ataque XSS?

- Robar cookies o tokens de sesión
- Suplantar identidad del usuario
- Redirigir a sitios maliciosos
- Mostrar contenido falso o engañoso
- Ejecutar acciones en nombre del usuario

Tipos de XSS

Tipo de XSS	Descripción
Reflejado	El script se envía en la URL y se refleja directamente en la respuesta
Almacenado	El script se guarda en la base de datos y se muestra a otros usuarios
Basado en DOM	El script se ejecuta por el navegador al manipular el DOM con datos maliciosos

SQL Injection

- **SQL Injection (SQLi)** es un ataque que consiste en **insertar código SQL malicioso** en campos de entrada (como formularios o URLs) para manipular las consultas que tu aplicación envía a la base de datos.
- ¿Qué se puede hacer con SQLi?
 - Ver o robar datos confidenciales (usuarios, contraseñas, tarjetas)
 - Borrar o modificar registros
 - Eludir autenticaciones
 - Tomar control del servidor de base de datos
 - Es como si el atacante escribiera comandos directamente en tu consola SQL, aprovechando que tu aplicación confía ciegamente en lo que el usuario escribe

Ejemplo

- `std::string query = "SELECT * FROM usuarios WHERE nombre = '"`
`+ nombre + "'";`
- Se añade `OR '1' = '1'`

