암호프로그래밍
CRYPTOGRAPHY PROGRAMMING

11. 토큰인증

정보보호학과
이병천 교수

차례

11. 토큰인증

1. 토큰인증 소개
2. JWT (JSON Web Token) 프로그래밍

1. 토큰인증 기술

- 초기인증 vs. 인증유지
- 쿠키, 세션, 토큰
- 암호화 통신
- OAuth 2.0
- JSON Web Token (JWT)

초기인증 vs. 인증유지

처음 접속시의 사용자 신분 인증
ID/pass, 인증서, Biometrics, 멀티팩터 인증 등 다양한 인증 사용
서버측에서는 사용자 계정 DB 확인 등 엄밀한 검증 절차 필요

클라이언트

초기인증기술 (상태형)

인증유지기술 (무상태형)

서버

초기 인증 완료된 사용자의 인증된 상태를 오랜기간 유지하는 기술
서버가 사용자 정보를 관리할 필요 없는 무상태 서비스(stateless service) 요구
쿠키, 세션, 토큰 등의 기술 사용

* 상태형(stateful) 인증 : 서버가 사용자의 정보를 유지해야 하는 인증기술

* 무상태형(stateless) 인증 : 서버가 사용자의 정보를 유지하지 않아도 되는 인증기술

인증유지 기술

공격위협

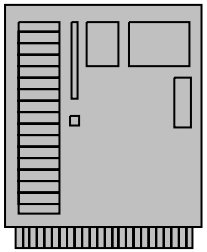
1. 도청공격으로 쿠키 탈취하여 ID 도용
2. 세션ID탈취 탈취하여 로그인세션 가로채기
3. 도청으로 토큰을 탈취하여 ID 도용

도청을 방지하기 위해
보안통신환경 필요

일반적인 패스워드 로그인

ID, h1 저장

서버



클라이언트

1. 사용자 등록

h1=H(pass, salt, It)

<ID, pass>

ID/pass 입력

2. 로그인

h1′=H(pass, salt, It) 계산
h1′=?h1 검사

<ID, pass>

ID/pass 입력

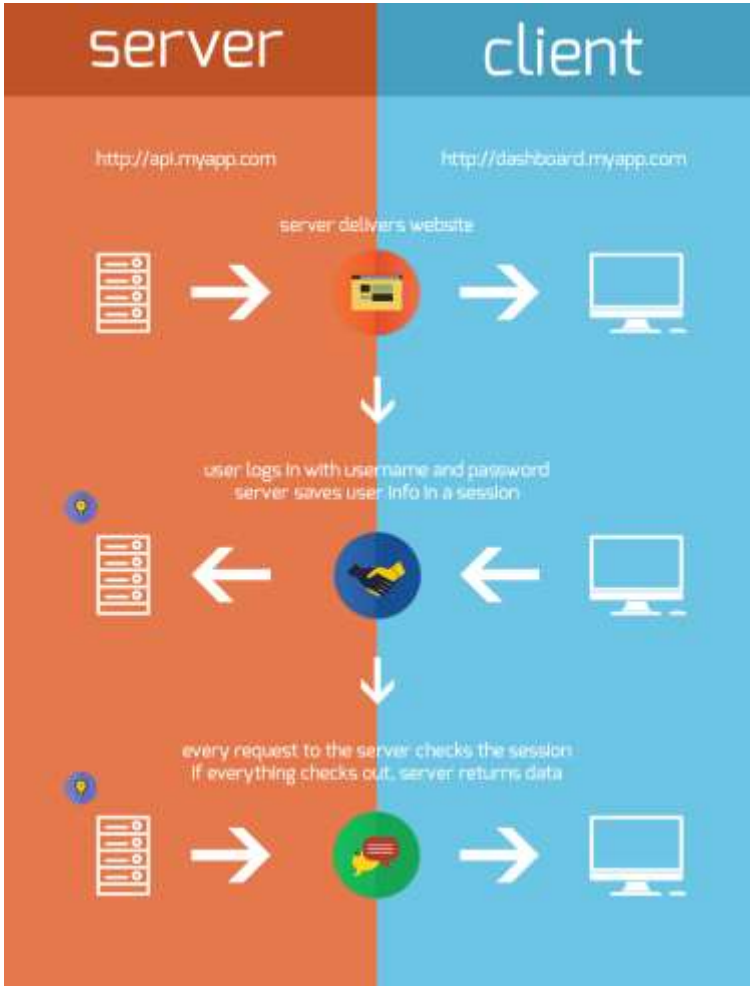
장점: 손쉽게 적용 가능
단점: 도청 공격, 재전송 공격

pass를 평문으로 전송
암호화 통신채널(https) 적용 필수

통신채널 암호화

- SSL/TLS (전송계층 보안 프로토콜)
  - 통신보안 제공 (합의된 비밀키 이용)
  - 서버 신분 인증 (서버의 인증서 검증)
  - 사용자 신분 인증 (사용자의 인증서 검증, 아직 널리 사용 안됨)
- 웹 통신 프로토콜
  - HTTP (port 80) – 통신내용이 평문으로 전송
  - HTTPS (port 443) – SSL/TLS 적용된 웹 프로토콜
- SSL Strip 공격
  - HTTPS 프로토콜 적용시 서버의 신분 인증 반드시 필요
  - 서버의 신분인증이 없는 경우 SSL 프록시를 이용한 중간자 공격으로 도청 및 변조 가능

인증 유지 기술 - 세션 인증

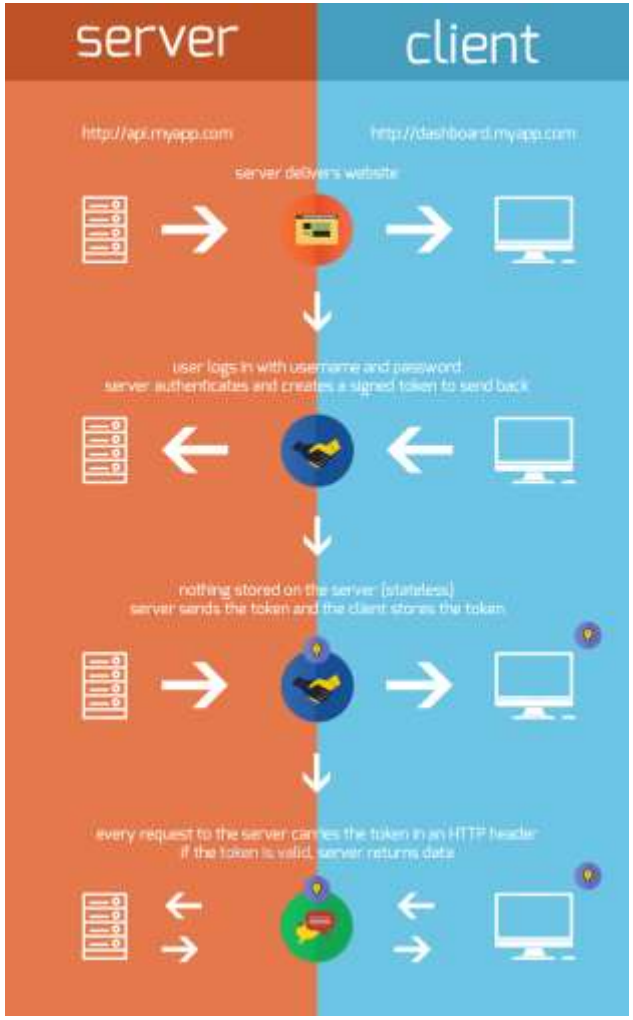


필요성
- 한 서버에서 서로 다른 페이지마다 인증을 요구하는 것은 어려움

방법
- 한번 인증되면 세션이 유지되는 동안은 인증 요구를 하지 않음
- 인증된 세션정보를 서버에 저장

단점
- 서버가 로그인된 사용자 정보를 유지해야 함
- 대규모 서버 운영시 확장성 부족
- CORS(cross-origin resource sharing) 방식의 웹서비스에 적용이 어려움

인증 유지 기술 - 토큰 인증 (자동 로그인)



필요성
- 서버에서 세션 정보를 유지하는 것은 확장성 측면에서 불리
- 사용자가 전송하는 정보만으로 인증을 완료하고 싶음

방법
- ID/pass로 한번 인증되면 서버는 토큰을 생성하여 클라이언트에 전송
- 클라이언트는 브라우저의 로컬스토리지에 토큰을 저장
- 클라이언트는 다음 요청시 토큰을 자동 첨부, 자동 로그인

단점
- 동일한 토큰이 반복 전송됨
- 도청 및 재전송 공격 가능성
- 암호화 통신채널(https) 적용 필수

OAuth

- OAuth=Authentication+Authorization
- 인증 및 권한관리 프레임워크

OAuth

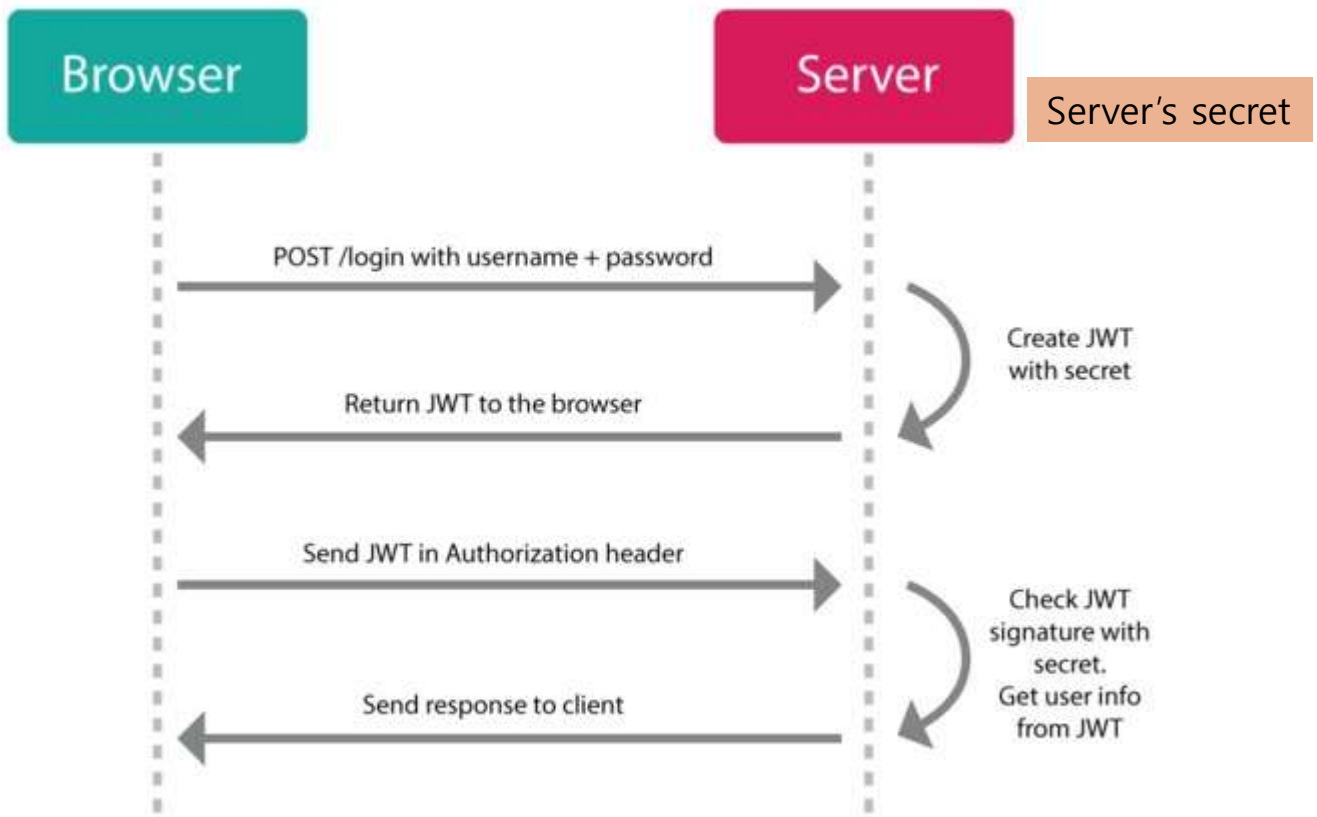
- 인증토큰(access token)
  - 서비스에 패스워드 전달 없이 인증
  - 인증토큰을 저장, 인증토큰으로 API 사용
  - 필요한 권한만 부여
  - 언제나 다시 권한 취소 가능
  - 유효기간이 있음
  - 패스워드 변경시에도 유효

OAuth 표준

- OAuth 2.0
  - OAuth 2.0 Framework - RFC 6749
  - Bearer Token Usage - RFC 6750
  - Threat Model and Security Considerations - RFC 6819
  - JSON Web Token (JWT) – RFC 7519
  - OAuth 2.0 Message Authentication Code (MAC) Tokens – draft
  - OAuth 2.0 Proof-of-Possession (PoP) – draft
- JWT가 대세
  - 고정된 bearer token + https 필수 사용

JSON Web Token

- https://jwt.io/



JSON Web Token

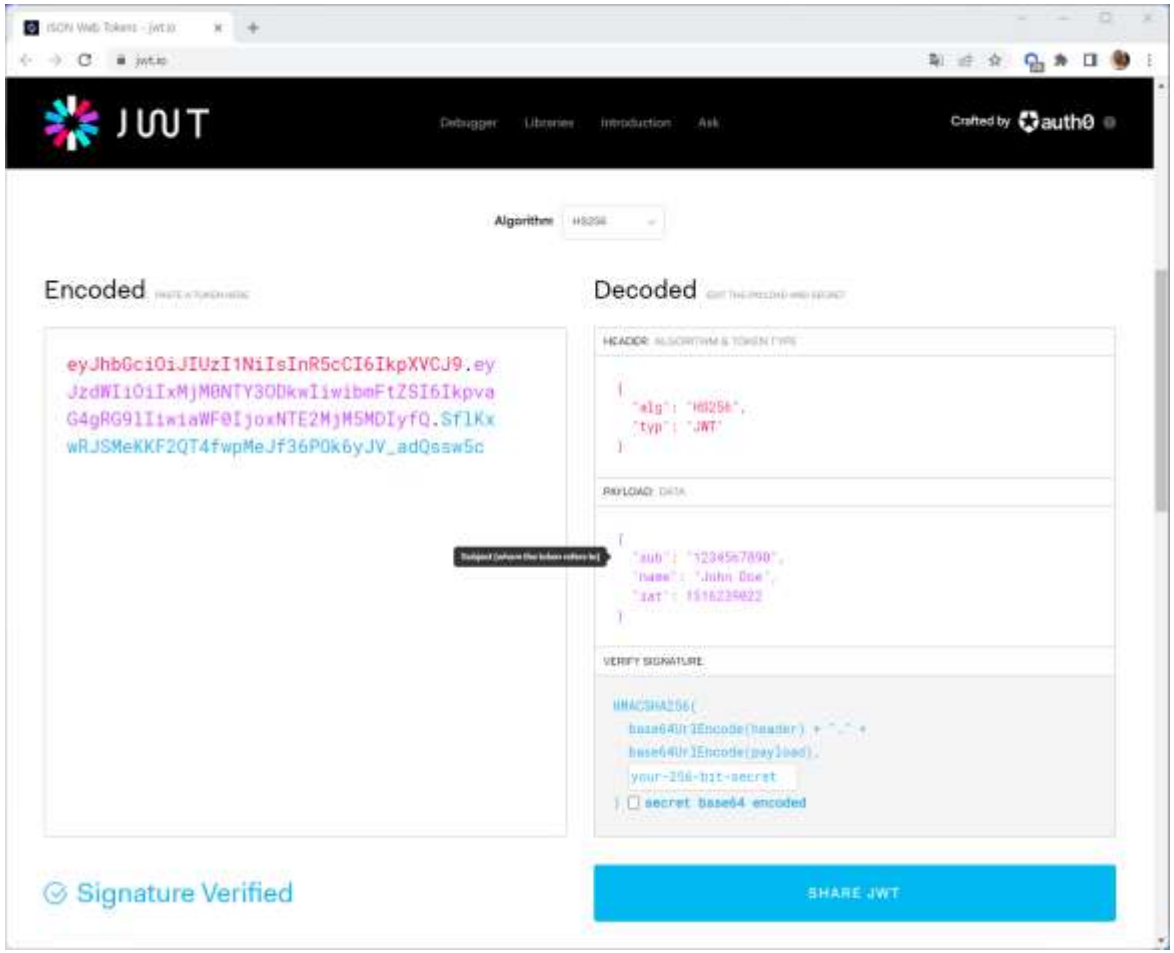
- Payload에 사용자 정보를 입력하고 Signature에 HMAC 서명 부가
- 서명은 서버 비밀값을 이용한 HMAC 계산으로 서버만 계산 및 검증 가능. 제3자의 위조 불가.
- 유효기간을 가진 토큰정보

2. JWT 프로그래밍

- jsonwebtoken 패키지 이용
  - > npm install jsonwebtoken
  - https://www.npmjs.com/package/jsonwebtoken 참조
- 서버가 클라이언트에게 토큰 발급
  - jwt.sign(payload, secretOrPrivateKey, [options, callback])
  - 서버가 클라이언트에게 토큰 전송
- 서버가 토큰의 유효성 검증
  - 클라이언트가 서버에게 토큰 전송
  - jwt.verify(token, secretOrPublicKey, [options, callback])

JWT Debugger

- https://jwt.io/



옵션

- Algorithm: 서명 알고리즘
- expiresIn: 유효기간
- notBefore: 유효기간
- Audience: 청중
- Issuer: 발급자
- Keyid
- Subject: 사용자
- ....

서명 알고리즘

- HMAC기반 서명 (default)
  - Algorithm: HS256, HS384, HS512
  - 256, 384, 512는 사용되는 해시함수의 출력 길이
  - 마스터비밀키를 이용한 HMAC 서명 및 HMAC 검증
- RSA 전자서명
  - Algorithm: RS256, RS384, RS512
- ECDSA 전자서명
  - Algorithm: ES256, ES384, ES512
- RSA/PSS 전자서명
  - Algorithm: PS256, PS384, PS512

secretOrPrivateKey
- 마스터비밀키
- 개인키
  - RSA 개인키
  - ECDSA 개인키

유효기간

- expiresIn 속성
  - “2 days” : 2일
  - “10h” : 10시간
  - “7d” : 7일
  - “120” : 120 ms (기본 ms 단위)

HS토큰 (HMAC 기반 서명)

jwt-hs.js

```
const jwt = require("jsonwebtoken");

const jsonData = { foo: "bar", id: "hello" };
const masterKey = "hjkshdfkjshf";

// 1. 토큰 발급
let token = jwt.sign(jsonData, masterKey, {
  algorithm: "HS512", // HS256, HS384, HS512
  expiresIn: "2h",
  audience: "people",
  issuer: "jbu",
  subject: "bclee",
});
console.log("1. 토큰 발급: " + token);

// 2. 토큰 검증
var decoded = jwt.verify(token, masterKey);
console.log("2. 토큰 검증: " + decoded.id);

// 3. 토큰 검증 - 에러 발생시 처리
try {
  var decoded = jwt.verify(token, "wrong-secret");
  console.log("3. 토큰검증: " + decoded.id);
} catch (err) {
  // err
}

// 4. 토큰 검증 없이 페이로드 출력 decode
var decoded = jwt.decode(token);
console.log("4. 페이로드 출력: " + decoded.id);
```

```
>node jwt-hs.js
1. 토큰 발급:
eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJmb2
8iOiJiYXIiLCJpZCI6ImhlbGxvIiwiaWF0IjoxNjUxNjY
0NDkwLCJleHAiOjE2NTE2NzE2OTAsImF1ZCI6InB
lb3BsZSIsImlzcyI6ImpidSIsInN1YiI6ImJjbGVlIn0.A
9M9Xo2wGKU5uYu7ldYNlyFJ_k8ysPVOJdYKvKiL5
22wrxqfTA7ZQx4jdCAnioUJ7BnCcvwaqtIjUGZbU
1413Q
2. 토큰 검증: hello
4. 페이로드 출력: hello
```

RS토큰 (RSA 기반 전자서명)

jwt-rs.js

```
const jwt = require("jsonwebtoken");
const forge = require("node-forge");

const pki = forge.pki;
const rsa = forge.pki.rsa;

// 1. 서버 키쌍 생성
var keypair = rsa.generateKeyPair({ bits: 1024, e: 0x10001 });
var publicKey = keypair.publicKey;
var privateKey = keypair.privateKey;

var publicKeyPem = forge.pki.publicKeyToPem(publicKey);
var privateKeyPem = forge.pki.privateKeyToPem(privateKey);
// console.log(publicKeyPem);
// console.log(privateKeyPem);

// 2. 토큰 발급
var token = jwt.sign({ foo: "bar" }, privateKeyPem, {
  algorithm: "RS512", // RS256, RS384, RS512, PS256, PS384, PS512
  expiresIn: "3d",
});
console.log(token);

// 3. 토큰 검증
jwt.verify(token, publicKeyPem, function (err, decoded) {
  console.log(decoded);
});

// 4. 페이로드 출력 (서명 미검증)
let decoded = jwt.decode(token);
console.log(decoded);
```

```
>node jwt-rs.js
eyJhbGciOiJSUzUxMiIsInR5cCI6IkpXVCJ9.eyJmb28iOiJiYXIiLCJpYXQiOjE2NTE2NjQ4MDMsImV4cCI6MTY1MTkyNDAwM30.R31oV8TUPyRNI-T2p6-HeEah0ZI4evDhTkWMPyFo2wTFltoeL2KZGMyvnQIIHENAC9el0-ub53fBBjiZroYs0G3U1PONlHV9eZVsg5yb71gto-Nsuii3t1iY3AbVqppIbUf5AtkBqoiAHLl1UZIjKLe64RIt8vXOStUyNcfjtAA
{ foo: 'bar', iat: 1651664803, exp: 1651924003 }
{ foo: 'bar', iat: 1651664803, exp: 1651924003 }
```

클라이언트 공개키를 포함하는 토큰

jwt-pk.js

```
const jwt = require("jsonwebtoken");
const forge = require("node-forge");

const pki = forge.pki;
const rsa = forge.pki.rsa;

// 1. 서버 키쌍 생성
let K = "masterKeysdlkdjflkasdfjlkj"; // Server's master key
var kpS = rsa.generateKeyPair({ bits: 1024, e: 0x10001 });
var pkS = kpS.publicKey;
var skS = kpS.privateKey;

var pkSPem = forge.pki.publicKeyToPem(pkS);
var skSPem = forge.pki.privateKeyToPem(skS);
// console.log("Server Public Key: ₩n" + pkSPem);
// console.log("Server Private Key : ₩n" + skSPem);

// 2. 클라이언트 키쌍 생성
var kpC = rsa.generateKeyPair({ bits: 1024, e: 0x10001 });
var pkC = kpC.publicKey;
var skC = kpC.privateKey;

var pkCPem = forge.pki.publicKeyToPem(pkC);
var skCPem = forge.pki.privateKeyToPem(skC);
//console.log("Client Public Key: ₩n" + pkCPem);
//console.log("Client Private Key : ₩n" + skCPem);
```

```
// 3.1 서버의 자체서명토큰 생성
let idS = "server";
let payloadDataS = { idS, pkSPem };
let jwtS = jwt.sign(payloadDataS, skSPem, {
  expiresIn: "1y",
  algorithm: "RS256",
});
console.log("3.1 Server self-signed token - RS256: ");
console.log(jwtS);

// 3.2 서버의 자체서명토큰으로부터 공개키 추출 (토큰 미검증)
let payloadS = jwt.decode(jwtS);
let pkSPem1 = payloadDataS.pkSPem;
console.log("3.2 Server public key: " + pkSPem1);

// 3.2 서버의 자체서명토큰 검증 ()
jwt.verify(jwtS, pkSPem1, { algorithm: "RS256" }, function (err, decoded) {
  console.log("3.3 Verified - RS256: " + decoded.idS); // bar
  console.log("3.3 Verified - RS256: " + decoded.pkSPem); // bar
});
```

계속 →

클라이언트 공개키를 포함하는 토큰


```
// 4.1 클라이언트 토큰 발급 - HS256 - 해당 서버에서만 사용
let id = "test";
let payloadData = { id, pkCPem };
let jwtC1 = jwt.sign(payloadData, K, { expiresIn: "1d", algorith
console.log("4.1 Client token - HS256: " + jwtC1);

// verify a token symmetric - synchronous
var decoded = jwt.verify(jwtC1, K, { algorithm: "HS512" });
console.log("4.1 Verified - HS256: " + decoded.id); //
console.log("4.1 Verified - HS256: " + decoded.pkCPem); //

// 4.2 클라이언트 토큰 발급 - RS256 - 공개적으로 사용
let payloadData1 = { id, pkCPem };
let jwtC2 = jwt.sign(payloadData1, skSPem, {
  expiresIn: "1d",
  algorithm: "RS256",
});
console.log("4.2 Client token - RS256: ");
console.log(jwtC2);

// verify a token asymmetric
jwt.verify(jwtC2, pkSPem, { algorithm: "RS256" }, function (err,
  console.log("4.2 Verified - RS256: " + decoded.id); // bar
  console.log("4.2 Verified - RS256: " + decoded.pkCPem); //
});
```

```
// 5.1 클라이언트 전자서명 생성
var plaintext = "client's signature test";
var md = forge.md.sha1.create();
md.update(plaintext, "utf8");
var signature = skC.sign(md);
console.log("5.1 Client Signature: " + forge.util.bytesToHex(signature));

// 5.2 클라이언트 토큰으로부터 공개키 추출
let payloadC1 = jwt.decode(jwtC1); // 토큰 미검증
let pkCPem1 = payloadC1.pkCPem;
console.log("5.2 Client public key: " + pkCPem1);
let pkC1 = forge.pki.publicKeyFromPem(pkCPem1);

let pkCPem2; // 토큰 검증 후 출력
let pkC2;
jwt.verify(jwtC2, pkSPem1, { algorithm: "RS256" }, function (err, decoded) {
  pkCPem2 = decoded.pkCPem;
  console.log("5.2 Client public key: " + pkCPem2); // bar
  pkC2 = forge.pki.publicKeyFromPem(pkCPem2);
});

// 5.2 클라이언트 전자서명 검증
// (defaults to RSASSA PKCS#1 v1.5)
var verified1 = pkC1.verify(md.digest().bytes(), signature);
console.log("5.2 Verified: " + verified1);
var verified2 = pkC2.verify(md.digest().bytes(), signature);
console.log("5.2 Verified: " + verified2);
```