#14

Web/Mobile Authentication

영화 <존윅3> 에서 주인공 존 윅은 자신의 권리를 주장하기 위한 증표를 보여준다. 모든 인증은 상대로 하여금 자신을 인증하는데 사용된다.

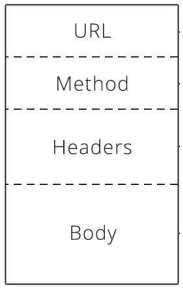
웹 애플리케이션이나 모바일 앱에서 사용자 계정 관리를 하고 있다면 사용자 인증은 반드시 필요한 기능이다. (물론, 계정이 있는 서비스라면 종류에 상관없이 모두 인증 과정이 있어야 하는 것이 맞다.)

인증에 대해 먼저 생각해보자. 인증은 사용자를 특정하여 서비스를 제공 할 때 쓰이는 사용자 확인 과정이다. 그리고 어느 서비스든 인증이 이루어지면 서비스를 받는 동안 인증이 유지될 필요가 있고 서비스 기간이 만료되면 인증을 폐기하거나 종료시켜야 한다. 흔히 이용하는 회원 서비스를 떠올리면 전혀 어렵지 않은 개념이다!

하지만 이런 특징을 바탕으로, 웹/모바일 서비스가 이뤄지는 HTTP 프로토콜에 대해 생각해보면 머리가 아파진다. 기본적으로 HTTP는 Stateless 프로토콜이다. 서버 - 클라이언트 간 상태 정보를 유지하고 있지 않는다는 의미다. 첫 번째 요청으로 사용자가 서버로부터 인증을 받았다고 하더라도 두 번째 요청에서 인증 상태를 유지할 수 없다. 따라서 HTTP 서버(웹서버)가 사용자(브라우저 또는 클라이언트 프로그램)의 정보를 식별하고 사용자의 상태를 유지하려면 별도의 방법을 마련해야 한다.

HTTP Message Structure

서버와 통신에 사용되는 HTTP 요청 메시지의 구조는 오른쪽 그림과 같이 크게 요청 주소(URL), 헤더(Method + Headers) 그리고 본문(Body)로 구분된다. 헤더 부분에 요청 정보가 들어가고, 본문에 서버로 보내고자 하는 데이터가 들어간다. 보통 웹/모바일 서비스의 인증을 위한 정보는 헤더에 포함된다.



<HTTP Structure>

Simple authentication data in headers

초기 웹 서비스는 사용자 인증을 위해 단순히 헤더에 계정 정보를 담아 보내는 방식을 사용하였다.

```
GET /async/ddljson?async=ntp:1 HTTP/1.1
Host: www.google.com
Connection: close
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/75.0.3770.142 Safari/537.36
Accept-Encoding: gzip, deflate
Accept-Language: ko-KR,ko;q=0.9,en-US;q=0.8,en;q=0.7
Cookie: userid=admin; password=s3cr3t;
```

최악의 인증방식이 아닐 수 없는데, 공격자는 마음대로 요청 정보를 가로채 사용자 정보를 알아내고 조작할 수 있었다.

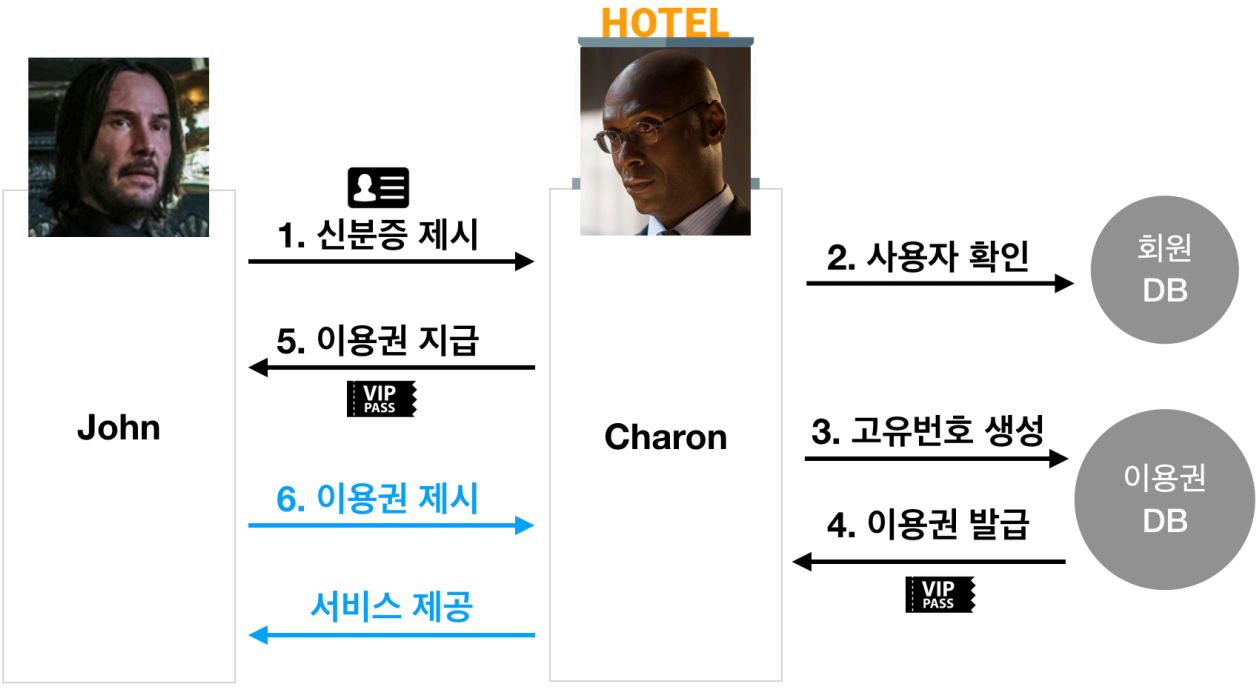
- **매우 쉽게 구현할 수 있고, 빠르게 인증을 처리할 수 있지만,[1]**
- **보안에 매우 취약하고,**
- **사용자가 요청을 보낼 때마다 계정정보(사용자 ID, PW)를 계속 보내야 하는 문제점이 있다.**

HTTPS를 사용하면 안전한 방식이라고 말할 수도 있겠다. 하지만, **엔지니어로서 할 말은 아니다!**

Memberships

그렇다면 어떻게 해야할까? 답은 이미 나와있다. 앞서 언급했듯이, 회원 서비스(이용권) 모델을 사용하면 문제를 쉽게 해결할 수 있다.

존 윅이 콘티넨탈 호텔의 서비스를 회원 자격으로 이용한다는 가정을 해보자. (영화의 설정과는 조금 다르다.)



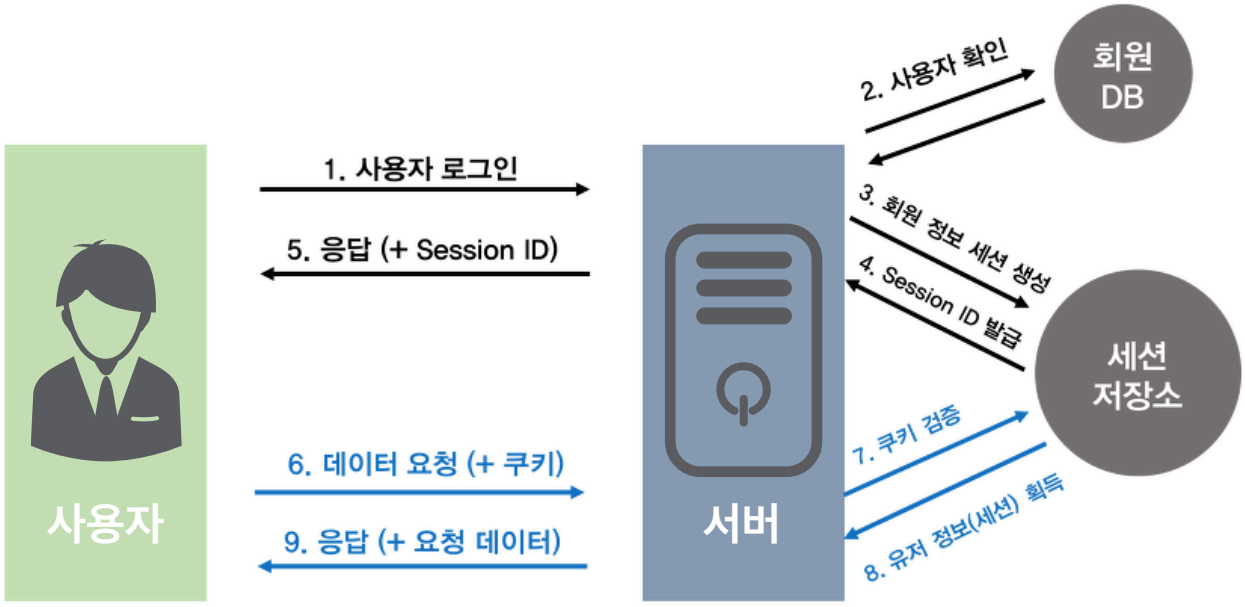
[1] 문서 전체에서 장점은 파랑색, 단점은 주황색으로 표기한다.

1. 존(사용자)이 호텔에 방문하여 신분증을 제시하면,
2. 샤론(호텔 매니저)은 신분증을 바탕으로 호텔 DB에 접속하여 회원 여부를 확인.
3. 샤론은 다시 존이 12시간(만료기간)동안 사용할 수 있는 고유 번호를 생성하고 암호화하여 보관하고,
4. 샤론은 고유 번호에 해당하는 이용권을 발급한다.
5. 샤론은 존에게 이용권 지급하고,
6. 존은 유효 시간동안 신분증 없이 이용권을 제시하면 호텔의 서비스를 자신의 권한 범위 내에서 이용 가능.

존은 이용권을 이용하여 신분증 없이도 호텔의 서비스를 간편하게 이용할 수 있게 되었다. 신분 정보가 노출될 위험이 줄어들었고, 서비스를 이용하기 위해 매번 신분증을 보여줄 필요도 없다.

Cookie and Session

같은 아이디어를 웹 통신에 적용시킨 것이 세션(고유번호)과 쿠키(이용권)다. 서버와 클라이언트간 세션과 쿠키가 어떻게 생성되고 이용되는지 그림으로 살펴보자.



1. 사용자는 서버에 인증 요청(+계정 정보(ID/PW))을 보낸다.
2. 서버는 계정 정보를 회원 DB에서 확인하고 사용자에게 고유한 값을 부여한다.
3. 부여한 고유값을 암호화하여 세션 저장소에 저장한다.
4. 세션 저장소에 저장된 고유값에 해당하는 세션ID를 발급한다.
5. 발급한 세션ID를 사용자에게 전달하고, 사용자는 세션 ID를 쿠키에 저장한다.
6. 사용자는 발급받은 세션ID를 요청 헤더(쿠키)에 담아 서비스 요청 시 본인이 누구인지 알린다.
7. 서버는 헤더(쿠키)에 담긴 세션ID를 세션 저장소에서 확인하여 사용자가 누구인지 판단한다.
8. 서버는 사용자의 권한에 맞는 서비스를 제공한다.

서버에는 세션 저장소라는 공간이 추가로 필요하고, 이 곳에는 세션ID와 서버가 관리하고자 하는 사용자의 정보를 저장하고 있다. 별도의 DB(Redis)를 설정하지 않고 메모리에 저장할 수 있지만 이중화가 불가능하고 서버를 재시작하면 세션 정보가 완전히 사라지는 문제가 있다. 마찬가지로 **사용자는 쿠키 저장소**를 자신의 하드디스크 어딘가에 마련해두고 서버로부터 전달 받은 세션ID를 쿠키 저장소에 저장하고 있다가 인증정보가 필요한 서비스를 요청할 때 세션ID를 HTTP 헤더(쿠키)에 넣어 보낸다.

최소한 이건 알고 넘어가자!

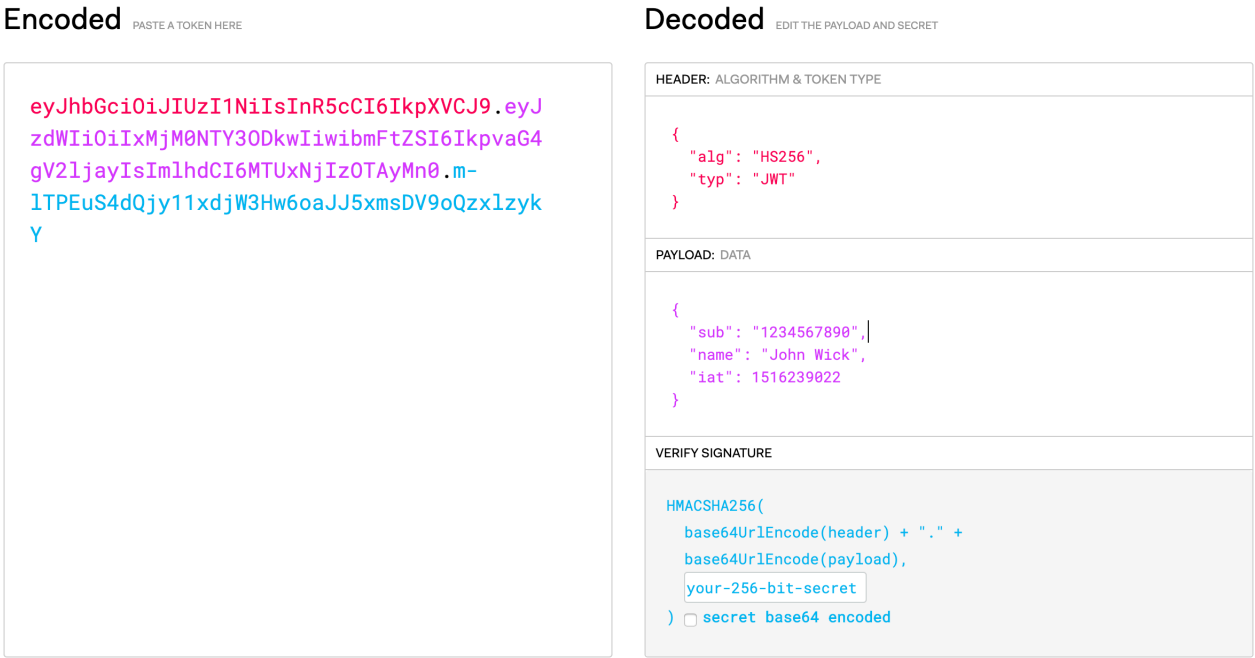
- ☑ 세션은 서버가 가지고 있는 정보 / 쿠키는 세션 정보를 확인하기 위한 열쇠(세션ID)
- ☑ 쿠키값으로만 인증을 하는 경우 암호화했다고 하더라도 앞서 설명한 Simple Authenticaion data in headers 의 방식과 동일하며, 책임은 사용자에게 있음. - 보안과 상관없는 경우에만 사용할 수 있도록 제한.
- ☑ 세션을 사용할 경우 책임은 서버에 있음. - 서버가 해킹당하는 것이 훨씬 어렵기 때문에 세션을 사용하도록 권고.

- **쿠키 정보(세션ID) 자체만으로는 노출된다 하더라도 큰 의미가 없고,**
- **서버는 세션ID만 확인하는 것으로 사용자를 식별할 수 있어 사용이 편하다.**
- **서버에 세션 저장소라는 추가 공간이 필요하고,**
- **해커가 쿠키(세션ID)를 재사용 한더라도 서버는 사용자를 정확히 식별하지 못한다.**

세션/쿠키 방식의 문제점을 해결하기 위해 HTTPS를 사용하고, 세션 유효기간을 설정하는 방법이 있다.

JWT (Json Web Token)

모바일 서비스가 급증하면서 대표적으로 사용하고 있는 인증 방식이다. 우리나라에선 카카오의 거의 모든 서비스가 JWT(토큰 기반 인증 방식)를 사용하고 있다. 앞서 설명한 세션/쿠키 방식과 비슷하게 사용자는 Access Token(JWT 토큰)을 HTTP 헤더에 담아 서버에 요청을 보낸다.



왼쪽에 Encoded에 표시된 값이 JWT 토큰 값이고 오른쪽에 표시된 값이 JWT 토큰을 해석한 값이다. JWT 토큰은 3개 영역으로 나뉘는데,

* Header : 토큰에 사용될 암호화 방식(alg)과 타입(typ),

* Payload : 서버에 보낼 데이터 (사용자 ID, 이름, 유효기간등),

* Verify Signature : Base64 방식으로 인코딩한 Header, Payload 그리고 Secret key를 더해 서명

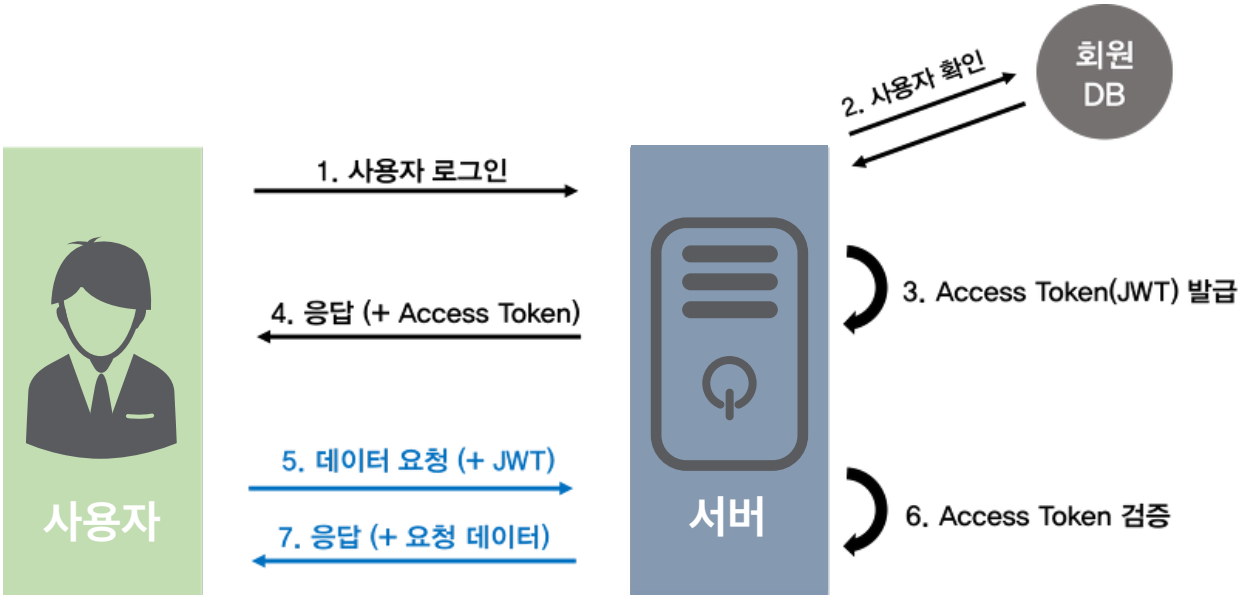
Encoded Header + “.” + Encoded Payload + “.” + Verify Signature

Header와 Payload는 16진수로 인코딩만 되기 때문에 노출될 경우 누구나 디코딩하여 확인할 수 있어 쉽게 사용자의 중요한 정보가 노출될 수 있다. 하지만 Verify Signature 는 Secret key로 암호화 되어 있기 때문에 Secret key를 알지 못하면 복호화 할 수 없다.

지금까지 알아본 내용을 바탕으로 예상되는 해킹 시나리오!!

해커 존은 자신의 JWT 토큰 값을 이용해 샤론의 정보를 훔쳐보고 싶어졌다. Payload에 있던 자신의 name을 샤론의 이름으로 바꿔 다시 인코딩하고 JWT 토큰을 서버로 보냈다. 성공했을까?

실패다. 서버는 암호화된 Verify Signature를 검사하는데 최초에 John Wick이라는 이름이 담긴 Payload로 암호화를 했기 때문에 이름을 Charon으로 바꾼 JWT 토큰은 유효성 검사에 실패하게 되어 유효하지 않은 토큰으로 간주된다. JWT가 어떻게 인증에 사용되는지 알아보자.



1. 사용자는 서버에 인증 요청(+계정 정보(ID/PW))을 보낸다.
2. 서버는 계정 정보를 회원 DB에서 확인하고 고유한 값(ID)과 사용자 정보를 Payload에 담는다.
3. 토큰의 유효기간을 설정해 Secret key로 암호화 하여 Access Token(JWT) 발급
4. 발급한 Access Token(JWT)을 사용자에게 전달하고, 사용자는 JWT를 쿠키에 저장한다.
5. 사용자는 발급받은 JWT를 요청 헤더(쿠키)에 담아 서비스 요청 시 본인이 누구인지 알린다.

6. 서버는 JWT를 복호화하여 조작 여부, 유효기간을 확인한다.
7. 검증이 완료되면 Payload를 디코딩하여 사용자의 권한에 맞는 서비스를 제공한다.

JWT는 세션/쿠키 방식과 마찬가지로 HTTP 헤더에 인증정보(세션ID/토큰)를 담아 요청한다는 점에서 동일하지만, 세션 저장소를 필요로 하지 않는다는 점에서 차이가 있다.

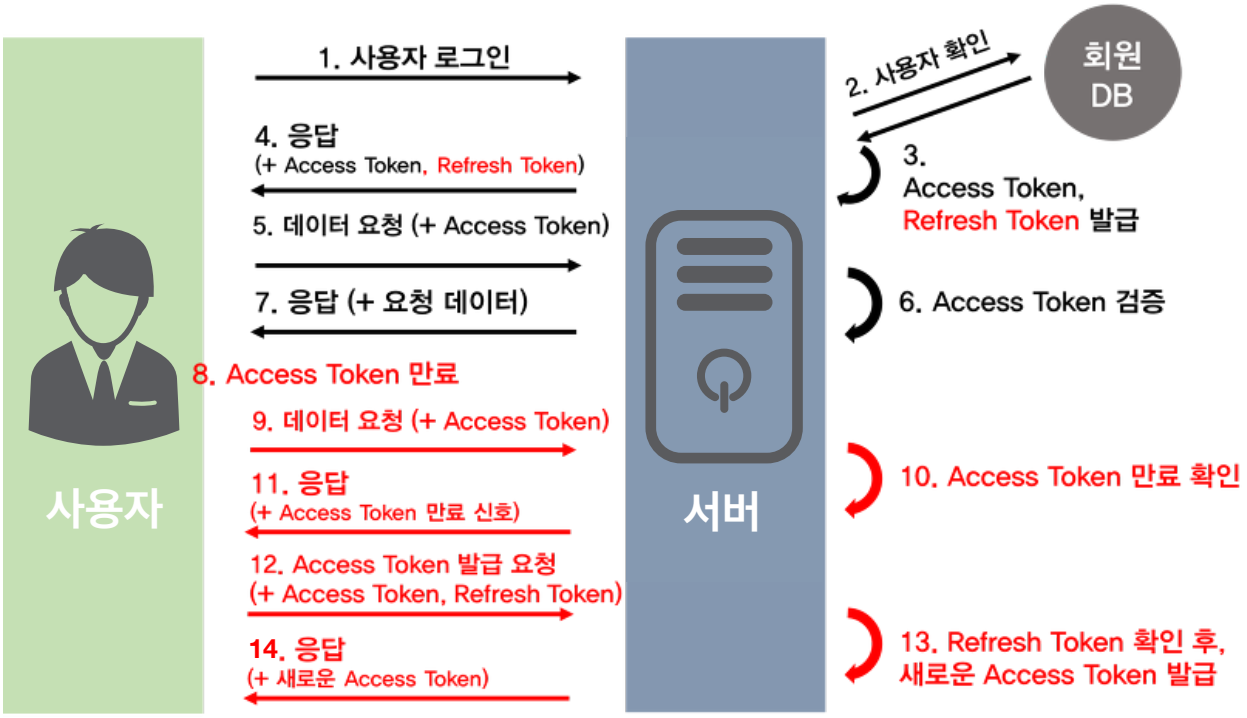
- **JWT는 토큰 발급 후 검증만 하면 되기 때문에 추가 저장소가 필요없다.**
- **추가 저장소가 없다는 것은 Stateless 서버를 만드는데 큰 장점을 갖는다.**
- **세션과 같이 상태 정보를 저장하고 있지 않기 때문에 서버를 확장하거나 유지/보수하는데 유리하다.**
- **JWT는 한 번 발급되면 유효기간까지 계속 사용이 가능하다.**
- **쿠키가 노출되어 다른 사람에 의해 악의적으로 재사용된다면 막을 수 있는 방법이 없다.**
- **Payload는 따로 암호화되지 않기 때문에 저장할 수 있는 정보가 제한적이다.**
- **JWT는 길이가 길어 요청이 많아질수록 서버의 자원낭비가 발생한다.**

Access Token + Refresh Token

JWT의 문제점을 보완한 Access Token + Refresh Token 방식에 대해 알아보자.

Access Token(JWT)방식의 가장 큰 문제점은 노출되었을 경우 유효기간이 종료될 때까지 막을 수 있는 방법이 없다는 것이었다. 유효기간을 짧게 하면 사용자가 로그인을 자주해야하는 불편함이 있고, 늘리면 늘릴수록 보안에 취약하게 된다.

Refresh Token은 Access Token의 유효기간을 짧게 하면서 사용자 편의성과 보안성을 동시에 높인 방법이다. Refresh Token 역시 Access Token 과 마찬가지로 유효기간이 있는 하나의 토큰이다. Access Token과 Refresh Token이 어떻게 상호작용하는지 간단한 예를 들어보자.



Refresh Token의 유효기간은 2주, Access Token의 유효기간은 30분이라고 해보자. 사용자는 로그인 후 30분동안 토큰의 갱신 없이 자유롭게 서비스를 이용한다. 30분이 지나면 Access Token은 만료된다. 그리고 Refresh Token이 새로운 Access Token을 발급해준다.

1. 사용자는 서버에 인증 요청(+계정 정보(ID/PW))을 보낸다.
2. 서버는 계정 정보를 회원 DB에서 확인한다.
3. 서버는 Access Token과 Refresh Token을 발급한다. 서버는 회원DB에 Refresh Token 저장.
4. 사용자에게 Access Token과 Refresh Token을 전달한다. 사용자도 RefreshToken을 저장.
5. Access Token(JWT)을 요청 헤더(쿠키)에 담아 서비스 요청.
6. 서버가 Access Token을 검증.
7. 서버는 요청한 정보에 맞는 서비스 제공.
8. Access Token이 만료됨.
9. 이전과 동일하게 Access Token을 헤더(쿠키)에 담아 서비스 요청.
10. 서버는 Access Token이 만료됨을 확인.
11. 서버는 사용자에게 Access Token이 만료되어 권한이 없음을 알림.
12. 사용자는 저장하고 있던 Refresh Token과 만료된 Access Token을 서버에 전달.
13. 서버는 Access Token 검증, Refresh Token 검증 후 새로운 Access Token 발급.
14. 새로운 Access Token을 전달.

- **Access Token(JWT)만 사용했을 때보다 훨씬 안전하지만,**
- **구현이 복잡하고 Access Token이 만료될 때마다 요청이 많아 서버의 자원 낭비가 심하다.**

OAuth 2.0

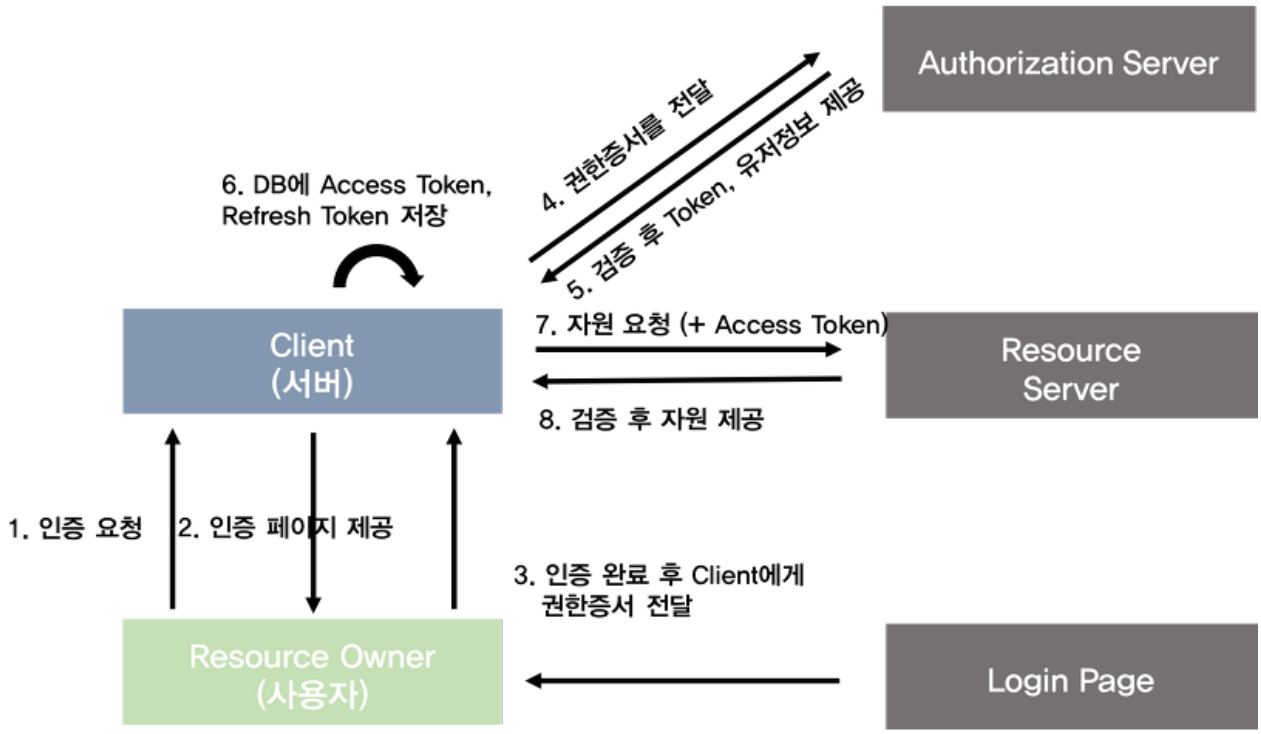
최근 거의 모든 웹/모바일 서비스가 제공하고 있는 인증 방법이다. 2007년 OAuth 1.0이 발표되고, 취약점이 나타난 이후 2012년부터 OAuth 2.0을 사용하고 있다.

OAuth 2.0[2] 의 특징은 Access Token의 유효기간이 생겼고, 모바일에서 사용 가능하며, HTTPS를 사용한다.

용어 정리!

- ☑ **Resource Owner** : 일반 사용자, 사람이 사용하는 브라우저나 모바일 기기로 보면 편하다.
- ☑ **Client** : 웹 애플리케이션 서버
- ☑ **Authorization Server** : 권한 관리 서버, Access Token과 Refresh Token을 (재)발급
- ☑ **Resource Server** : OAuth 2.0 서버, 서비스를 제공하는 회사의 자원을 관리하는 서버

[2] OAuth 는 용어가 익숙하지 않다. 용어가 익숙해지면 좀 더 쉽게 이해할 수 있다.



1. Resource Owner는 Client에 인증 요청.
2. Client는 Resource Owner가 인증할 수 있는 수단(e.g. Google 로그인 페이지)을 제공.
3. Resource Owner는 인증을 마치고 Authorization Grant(권한증서)를 url에 담아 Client에게 전달.
4. Client는 Authorization Grant를 Authorization Server에 전달.
5. Authorization Server는 Authorization Grant를 확인 후 Access Token, Refresh Token, 사용자 정보(ID 등)를 발급.
6. Client는 Access Token과 Refresh Token을 DB에 저장하고, Access Token과 Refresh Token을 Resource Owner에게 전달.
7. Resource Owner는 Access Token을 헤더에 담아 Resource Server에 자원 요청.
8. Resource Server는 Access Token이 유요한지 확인 후 Client에게 자원 전달.

Token 검증 과정

1. Access Token이 만료되면, Client는 Authorization Server에 Refresh Token을 보내 Access Token을 재발급.
2. Authorization Server는 Resource Server에 자원 요청.
3. Refresh Token이 만료되면 Resource Owner는 다시 로그인.

물론, OAuth 2.0에도 취약점이 존재한다. CSRF 공격이나 Covert Redirect 공격에 취약할 수 있다. 그리고 구현이 매우 복잡한데, 제대로 구현한다면 이 방법만큼 안전하고 편리한 인증방법이 또 있을까?