비밀번호가 필요 없는 차세대 로그인 기법, 패스키

이규연 - 청강문화산업대학교

요약

네트워크 환경에서 본인임을 인증하는 것은 매우 중요하다.
네트워크 기술이 발전함에 따라 네트워크 환경에서 동작하는 서비스의 수가 점차 늘어났고,
이에 따라 네트워크 환경에서 이용되는 데이터에 이용자의 민감한 개인 정보까지 포함되는 경우가 빈번하게 발생하게 되었다.
이런 상황들이 점차 잦아지며 이용자가 본인임을 인증하는 수단에 대해 필요성이 증가하였고,
JWT와 같이 인증 정보를 보존하는 수단 역시 꾸준히 발전하게 되었다.
하지만 "아이디와 비밀번호를 통해 본인임을 인증한다"는 기초적인 개념은 몇 십년간 변화하지 못했는데,
이 방식은 비밀번호만 유출된다면 아무리 보안 환경이 잘 갖추어져 있는 서비스일지라도
해킹에 매우 취약해지며, 비밀번호를 통일해 사용하는 이용자의 경우 보안이 취약한 사이트에 가입만 하더라도
해당 비밀번호를 사용하는 모든 서비스에 대해 보안 취약점이 발생하게 된다.
본 보고서에서는 이를 대체하기 위해 탄생한 기술인 패스키에 대해 다루며,
패스키가 탄생한 배경, 패스키의 원리, 패스키의 장단점 등을 분석한다.

Abstract

Authenticating oneself in a network environment is critically important. As network technology has advanced, the number of services operating online has steadily increased. Consequently, it has become common for data used in these environments to include users' sensitive personal information. The growing frequency of these situations has heightened the need for user authentication methods, leading to the continuous development of means to preserve authentication information, such as JWT (JSON Web Tokens).
However, the fundamental concept of "authenticating with an ID and password" has remained unchanged for decades. This method is highly vulnerable to hacking if the password is leaked, regardless of how secure the service's environment is. Furthermore, for users who reuse the same password across multiple sites, signing up for even one insecure site creates a security vulnerability for all services where that password is used.
This report addresses passkeys, the technology created to replace this system, and analyzes the background of their creation, their underlying principles, and their advantages and disadvantages.

I. 서론

기존의 아이디와 비밀번호 기반 로그인 방식은 고질적인 문제점을 안고 있습니다. 이는 보안성과 편의성 사이의 딜레마에서 비롯된다. 모든 서비스의 비밀번호를 동일하게 설정하면, 한 곳의 정보만 유출되어도 모든 계정이 연쇄적으로 탈취될 위험에 노출된다. 반대로 서비스마다 다른 비밀번호를 사용하면 사용자가 이를 기억하고 관리하기 매우 번거롭다는 단점이 있다.

이러한 문제를 해결하기 위해 업계에서는 비밀번호 관리 프로그램을 개발하거나, 궁극적으로 비밀번호를 없애려는 '비밀번호 없는 인증(Passwordless Authentication)' 기술을 연구해왔다. 이러한 흐름 속에서 구글, 애플, 삼성 등 글로벌 IT 기업들은 표준화된 인증 기술을 개발하기 위해 FIDO 얼라이언스(FIDO Alliance)를 결성했다.

FIDO 얼라이언스는 비밀번호를 대체할 안전하고 편리한 기술 표준을 제시했으며, 그 핵심 결과물 중 하나가 바로 본 논문에서 다룰 '패스키(Passkey)'이다. 본 보고서에서는 이 패스키의 개념과 원리, 사용하는 방법 등을 서술한다.

II. 패스키의 개념

개발자의 입장에서, 사용자가 누구인지 알아내는 것은 매우 중요하다. 최근 은행 업무와 같은 민감한 개인정보가 요구되는 서비스들도 점차 온라인 서비스로 전환되며 이는 더더욱 중요해지는 추세이다.

그동안 개발자들은 사용자가 누구인지 알아내기 위해 "비밀번호"라는 수단을 사용해 왔는데, 이는 사용자 본인만이 알고 있는 문자열을 이용해 사용자가 누구인지 확인하는 방식이다.

하지만 비밀번호를 통한 로그인은 생각보다 안전하지 않다. 대부분의 사람들은 하나의 비밀번호를 수많은 사이트들에 반복해서 사용하며, 하나의 서비스라도 해킹 당하게 된다면 해당 비밀번호를 사용하는 모든 서비스도 위험에 빠지게 된다.

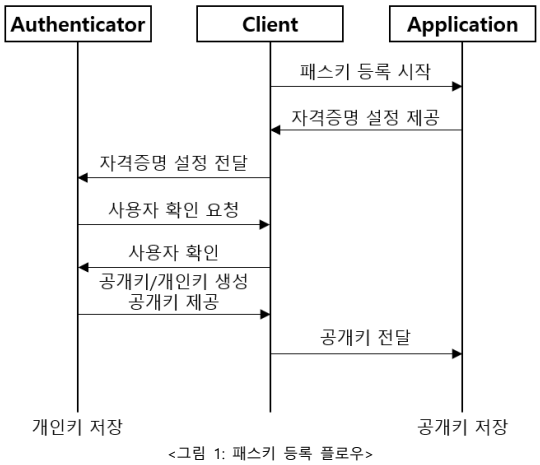
이 비밀번호를 훔치는 방법은 여러 가지가 있는데, "키로거를 통한 키보드 입력 탈취", "피싱 사이트를 통한 비밀번호 수집", "무차별 대입 공격(Brute Force Attack)", "레인보우 테이블을 이용한 해시 값 역추적", "사회공학적 공격을 통한 비밀번호 획득"", "데이터베이스 유출 시 해시된 비밀번호 복구" 등 수많은 방법들을 동원해 사용자의 비밀번호를 탈취하고 있다.

특히 사회공학 기법의 경우, AI 기술의 발전으로 인해 더 더욱 정교화되고 있으며, 딥 페이크 기술을 이용한 화상 통화나 AI가 생성한 피싱 이메일 등은 일반 사용자가 진위를 구분하기 매우 어렵다.

패스키는 이러한 문제들을 해결하기 위해 개발되었는데, 사용자의 기억에 의존하는 대신 사용자의 생체 정보와 기기 정보에 의존한다. 지문인식, Face ID, Windows Hello 등 다양한 생체 인식 방식을 사용한다. 이 정보들은 기기 내부에 안전하게 저장되며, 네트워크를 통해 전송되지 않아 해킹의 위험이 매우 낮다.

패스키의 사용자는 휴대폰, 노트북 등 생체 인증이 가능한 기기들을 가지고 있기만 한다면 간단한 생체 인증만으로도 인증을 진행할 수 있다. 특히 모바일과 같은 환경에선 비밀번호를 따로 입력할 필요가 없다는 크나큰 장점도 가지고 있으며, 기기의 정보와 사용자의 생체 정보 모두가 필요하기에 제 3 자가 인증 정보를 탈취하기 매우 어려워진다. 실제로 피싱이나 중간자 공격(Man-in-the-Middle-Attack) 공격에 매우 안전하게 설계되어 있으며, 각 인증 시도마다 새로운 암호화 키가 생성되기에 이전 인증 정보를 재사용하는 것도 불가능하다.

III. 패스키의 동작 방식



<그림 1: 패스키 등록 플로우>

패스키를 등록하기 위해선 "공개 키(Public Key)"와 "프라이빗 키(Private Key)" 두 가지를 생성해야 하며, 디바이스는 이들 중 프라이빗 키를 일반적인 저장 공간이 아닌 iCloud 키체인, Google 비밀번호 관리자, 삼성 패스 등 하드웨어 제조사에서 제공하는 별도의 공간에 저장한다.

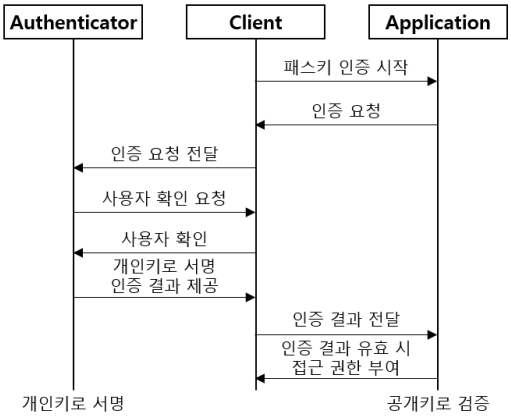
여기서 서버는 WebAuthn이라 불리는 W3C와 FIDO Aliance가 제공하는 표준 규약을 준수해 패스키 등록을 위한 정보들을 클라이언트 측에 전달해야 하며, 여기엔 유저 식별자 정보 등이 포함된다.

```
async generateRegistrationOptions(
  generateRegistrationOptionsDto: GenerateRegistrationOptionsDto,
  userId: number,
) {
  const userRepository = this.dataSource.getRepository(UserEntity);
  const challengeRepository = this.dataSource.getRepository(
    PasskeyChallengeEntity,
  );

  const user = await userRepository.findOne({
    where: { id: userId },
    relations: { passkeys: true },
  });
  if (!user) throw new UnauthorizedException();

  const isPwCorrect = await bcrypt.compare(
    generateRegistrationOptionsDto.user_password,
    user.password,
  );
  if (!isPwCorrect) throw new BadRequestException(errors,INVALID_PASSWORD);

  // BUG FIX 1: userID를 문자열 변환 후 Buffer로 생성
  const userIDBuffer = Buffer.from(user.id.toString());

  const options = await generateRegistrationOptions({
    rpID: this.configService.get('PASSKEY_RP_ID'),
    rpName: this.configService.get('PASSKEY_RP_NAME'),
    userID: userIDBuffer,
    userName: user.email,
    attestationType: 'none',
    excludeCredentials:
      this.configService.get<string>('NODE_ENV') === 'PROD'
        ? user.passkeys.map((passkey) => ({
            id: passkey.id,
            type: 'public-key',
            transports: passkey.transports
              ? (passkey.transports.split(
                  ',',
                ) as AuthenticatorTransportFuture[])
              : undefined,
          }))
        : undefined,
    authenticatorSelection: {
      residentKey: 'required',
      userVerification: 'required',
    },
    timeout: 60000, // 1분으로 타임아웃 조정 (권장)
  });

  const expires_at = new Date(Date.now() + 60000); // 타임아웃과 일치

  // 기존 챌린지 삭제
  const oldChallenge = await challengeRepository.findOneBy({
    user_id: user.id,
    type: ChallengeType.REGISTRATION,
  });
  if (oldChallenge) await challengeRepository.remove(oldChallenge);

  const challenge = challengeRepository.create({
    challenge: options.challenge,
    expires_at,
    type: ChallengeType.REGISTRATION,
    user_id: user.id,
  });
  await challengeRepository.save(challenge);

  return options;
}
```

<그림 2: 패스키 등록 옵션 생성 예시>

이후, 클라이언트가 패스키를 성공적으로 생성했다면, 클라이언트 측에서 전달받은 프라이빗 키와 디바이스 타입, 생성 시각, 유저 식별을 위한 일부 정보 등 패스키에 대한 정보들을 함께 데이터베이스에 저장한다.

이 과정을 통해 패스키를 생성했다면, 해당 정보를 통해 로그인을 진행해야 하는데, 이 과정은 패스키 생성 과정과 매우 비슷하다.

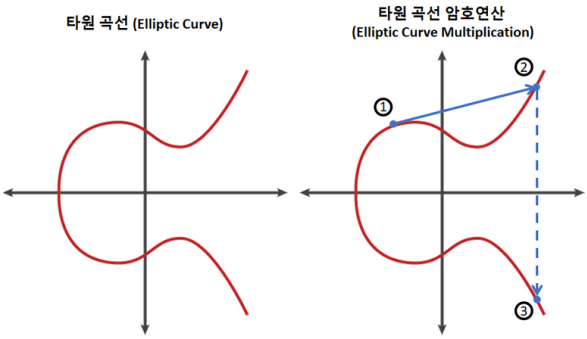


<그림 3: 패스키 인증 플로우>

사용자가 로그인을 시도하면, 클라이언트는 서버에게 패스키 로그인을 요청하게 되는데, 서버는 로그인을 위해 챌린지라 불리는 랜덤한 문자열을 생성하고 클라이언트로 전달하게 된다.

클라이언트는 앞서 생성한 프라이빗 키를 이용해 챌린지 문자열을 암호화(서명)하고 암호화된 문자열을 서버에 전송하게 되는데, 서버는 저장되어 있는 공개 키로 암호화된 문자열을 검증한 이후, JWT와 Refresh 토큰 등 해당 서비스가 이용하고 있는 일반적인 아이디 + 비밀번호 로직으로 로그인했을 때와 같은 인증 정보를 클라이언트에게 전달하게 된다.

III. 패스키가 동작하는 원리



<그림 4: ECDSA 타원곡선 예시>

지금까지 서술한 모든 과정들 일반적으로 ECDSA 알고리즘이라는 서명 알고리즘을 통해 진행되는데, ECDSA 알고리즘을 예시로 들자면, 이 과정은 모두 $y^2 = x^3 + ax + b \pmod p$인 타원곡선이라는 특별한 수학적 구조 위에서 이루어지게 된다. a와 b는 각각 곡선 모양을 결정하는 상수이며, p는 매우 큰 소수로, 모든 계산이 이 숫자로 나눈 나머지에서 이루어짐을 의미한다. 이로 인해 곡선 위의 점들은 연속적이지 않고, 유한한 개수의 점들로 흩어져 있는 형태가 된다.

이 곡선엔 특별한 연산 규칙이 하나 있는데, 바로 "스칼라 곱셈"이며 이는 점 P에 대해 k * P = P + P + ... + P로 표현할 수 있다. 일반 곱셈과의 결정적인 차이는, "연산 공간이 유한체 위의 휘어진 타원 곡선"이라는 것에 있다. 예시로 P + P는 점 P에 대한 접선을 구하고, 이 접선이 곡선과 만나는 점 -R(반드시 한 점에서 더 만나게 된다) 이 -R을 x축 기준으로 대칭시킨 R이 P + P의 결과가 된다. 이 덧셈의 결과는 매우 불규칙하기에, P * k를 계산하기 위해선 반드시 P를 k번 더해보는 방법밖에 없다. 여기서 k * P = Q 일 때, k와 Q값을 통해 P를 알아내는 것은 앞서 말한 연산 과정의 반복이기에 쉽게 알아낼 수 있지만, 반대로 P와 Q값을 통해 k를 알아내는 것은 k가 천문학적으로 큰 숫자이기에 현재의 컴퓨터 기술로는 사실상

불가능하다.

이 스칼라 곱셈이 ECDSA의 핵심으로, k는 프라이빗 키, Q는 공개 키, P는 하나의 기준점이 된다.

모든 참여자가 동일한 환경에서 암호를 사용하기 위해, 미리 약속된 값들을 공유하는데, 이를 "도메인 매개변수"라고 하며, 이는 다음 값을을 포함한다.

<표 1: 도메인 매개변수 >

a, b	타원곡선 방정식을 정의
p	모든 계산이 이루어지는 유한체의 크기 (매우 큰 소수)
G	생성점 또는 기준점, 곡선 위의 한 점이다.
n	G를 n번 더했을때 무한원점이 되는 값, 즉 n * G는 0이 되는 값.
h	보조 인자 (cofactor)

위와 같은 도매인 매게변수가 존재하는데, 여기서 이상한 점이 하나 있다. 스칼라 곱셈의 기본 원칙에 의해, k * P가 Q가 되는 k의 값은 분명 구할 수 없다고 했는데, n * G가 0이 되는 n은 어떻게 구한 걸까?

실제로 G를 계속 더해보며 0이 되는 n을 찾는 것은 우주가 끝날 때까지 계산해도 찾아낼 수 없다. 하지만 우리는 "$y^2 = x^3 + ax + b \pmod p$인 타원곡선이라는 특별한 수학적 구조 위에서 이루어지게 되는데, p는 매우 큰 소수로, 모든 계산이 이 숫자로 나눈 나머지에서 이루어짐을 의미한다."라는 것을 기억해야 한다.

12시간짜리 아날로그 시계로 비유해 보면, 숫자 1이 기준점 G라고 생각해 보자. 1 * G = 1시, 2 * G = 2시 ... 12 * G = 12시 (0시라고 생각해 보자), 13 * G = G시 (다시 돌아옴)이 될 텐데, 여기서 n은 12이다. 1을 12번 더하면 처음으로 0이 되기 때문인데, 시계의 전체 숫자가 12개라는 것을 알면, G가 무슨 값이 되든 n이 12의 약수 (1, 2, 3, 4, 6, 12) 중 하나일 것이라는 것이라고 예측할 수 있다. 도메인 매개변수의 n도 이와 같은 방식으로 구할 수 있다. 다만, 타원곡선 위의 점의 개수는 무수하게 많기 때문에, 점의 총 개수 N을 세기 위해선 Schoof's Algorithm과 같은 매우 발전된 알고리즘을 사용한다.

N을 구했다고 가정하고, 이를 이용해 n을 구하는 방법을 알아보자. 여기선 "라그라주 정리"가 사용되는데, 아까 말한 간단히 말하면 "부분그룹의 크기 n은 반드시 전체 그룹 크기 N의 약수여야 한다"는 정리이다. 즉, n은 N을 나누어 떨어지게 하는 숫자일 것이다. 위에 서술한 시계 비유를 생각하면 간단하게 이해할 수 있을 것이다. G가 어떤 값이든 다시 0시로 돌아오기 위한 n의 값은 반드시 12의 약수가 되는 것처럼 말이다. 따라서, N을 소인수분해한 값들 중 하나가 n이 되는 것이고, 이 값들을 하나하나 G에 곱해 보며 n의 값을 찾게 된다.

이제 이 도매인 매개변수를 활용해 공개 키를 생성하는데, 프라이빗 키는 $1 <= d_A <= n-1$인 무작위 수, 공개 키는 $Q_A = d_A * G$인 점 Q_A 이다. 앞서 서술한 원리에 의해, 공개 키와 G를 알아도 프라이빗 키를 알아내는 것은 불가능하다.

클라이언트가 서버에서 챌린지를 전달받았다면, 이제 해당 키를 가지고 서명을 생성하는데, 서명은 (r, s)라는 한 쌍의 숫자로 구성된다. 클라이언트는 챌린지를 SHA-256 같은 해시 함수를 사용해 고정된 길이의 숫자 e로 변환한다. 보안을 위해 e의 비트 길이를 n의 비트 길이에 맞게 잘라내는데, 이 값을 z라고 하고, 다시 1과 n-1 사이의 무작위 정수 k를 선택한다. 이 k와 기준점 G를 곱해 점 (x_1, y_1)을 계산하는데, 여기서 k는 매 서명마다 반드시 새로 생성되며, 절대로 재사용되지 않는다. 이 값이 반복해서 재사용된다면 프라이빗 키가 즉시 노출되게 된다. 임시 키 k와 기준점 G를 곱해 하나의 점을 찾고, 해당 점의 x 좌표를 n으로 나눈 나머지를 서명의 값들 중 하나인 r로 정한다. 여기서 r이 0이라면 다른 k를 선택해 위 과정을 다시 진행해야 한다.

이제 여기서 $s = k^{-1}(z + r * d_A) \pmod n$ 라는 공식을 사용해 서명의 나머지 값인 s를 구하는데, 여기서 k^{-1}은 $(k * k^{-1}) \pmod n = 1$을 만족하는 값이다. 여기서 사용하는 d_A가 바로 프라이빗 키이며, s가 0이면 r을 구하는 과정과 마찬가지로 k를 다시 생성하여 처음부터 다시 반복하게 된다.

이렇게 서명을 완료했다면, 서명된 값을 서버에게 전달해 검증을 해야 하는데, 먼저 r과 s가 1과 n-1 사이의 값인지 확인한다. 이후, 클라이언트가 했던 것과 같이 e와 z를 구하고 두 개의 중간값인 u_1과 u_2를 계산하는데, 각각 $u_1 = z * s^{-1}$와 $u_2 = r * s^{-1}$라는 공식을 통해 계산한다. 이 값들과 $(x_0, y_0) = u_1 * G + u_2 * Q_A$(공개 키)을 이용해 곡선 위의 점 (x_0, y_0)을 계산하고, 이 점의 x좌표인 x_0과 서명된 값 r을 비교한다. 만약 두 값이 같다면 서명이 유효한 것이며, 다르다면 서명이 위조되거나 챌린지가 변경된 것이다.

이런 방법이 어떻게 작동하는 것일까? 여기엔 간단한 수학적 원리가 숨어 있는데, 검증 과정에서 (x_0, y_0)을 계산한 식에서 시작해 보자. $(x_0, y_0) = u_1 * G + u_2 * Q_A$에 $Q_A = d_A$(프라이빗 키) $* G = z * s^{-1}$, $u2 = r * s^{-1}$ 두 식을 대입해 보자. 이들은 각각 앞서 서술한 공개 키를 구하는 식, 서

명 과정에서 r, s를 구하는 데 사용되었던 식들이다. 이들을 대입한다면

$$(x_0, y_0) = u_1 * G + u_2 * Q_A$$
$$= (z * s^{-1}) * G + (r * s^{-1}) * (d_A * G)$$
$$= (z * s^{-1} + r * s^{-1} * d_A * G)$$
$$= s^{-1}(z+r * d_A) *G$$

가 된다. 여기에서 서명 과정 중에 $s = k^{-1}(z+r*d_A)$라고 정의했는데, 이 식의 양변에 k를 곱하고, s^{-1}을 곱하면, $k = s^{-1}(z + r * d_A)$가 된다. 따라서, 위 식은 $(x_0, y_0) = k * G$가 된다. 이는 서명을 생성할 때 계산했던 $(x_1, y_1) = k * G$와 완전히 동일한데, 따라서 이 점의 x좌표인 x_0은 x_1과 완전ㅌ히 같아야 하며, 따라서 $x_0 \pmod n = x_1 \pmod n = r$이 성립하게 된다.

결론적으로 서명 검증은 서명자가 개인 키인 d_A를 알고 있다는 것을 d_A를 노출하지 않고도 검증할 수 있는 수학적인 증명 과정이 된다.

IV. 패스키의 단점

패스키는 비밀번호와 같은 개인정보 교환 없이 본인임을 증명할 수 있는 매우 안전한 방법임에도 불구하고 치명적인 단점들이 여럿 존재한다.

먼저, 기기를 잃어버렸다면 패스키를 다시 발급해야 한다. 앞서 서술했듯 패스키의 인증 과정은 "프라이빗 키를 알고 있음"을 증명하는 과정이기에 프라이빗 키를 저장하고 있는 기기를 잃어버린다면 해당 패스키를 통한 인증이 불가능하게 된다. 이를 해결하기 위해 대부분의 서비스들은 백업 인증 수단(이메일, SMS 등)을 설정해 두는 것을 강력하게 권장하며, 애플과 구글 같은 기업들은 클라우드를 통해 패스키를 백업해 두는 기능을 제공하기도 한다.

두 번째로, 생체 인증이 가능한 기기가 반드시 필요하다는 것이다. 해당 기기에 프라이빗 키가 저장되어 있다고 한들, 해당 기기의 사용자가 반드시 해당 기기의 소유주라고 보장할 수 있는 것이 아니다. 기기 도난과 같은 상황에 패스키를 아무런 보안 장치 없이 사용한다면 오히려 비밀번호보다 취약해지기에, FIDO에선 패스키를 사용하기 위해 반드시 "생체 인식"을 사용하도록 하고 있다. 하지만 일반적인 윈도우 데스크탑 환경에 지문 인식, 얼굴 인식 등 생체 인식을 위한 장비를 가지고 있는 사람은 많지 않기에, 결국 비밀번호와 병행해서 사용해야 한다는 불편함이 있다.

세 번째로 호환성 문제가 있는데, 일반적으로 패스키를 대다수의 OS에서 지원하지만, 리눅스의 일부 배포판과 같은 일부 마이너한 OS들은 이들을 지원하지 않는 경우가 많지 않고, 오래된 기기에서 이들을 지원하기엔 상당한 기술 문제가 많이 발생할 것이다.

마지막으로, 패스키는 기존의 인증 방식과는 완전히 다른 개념이기 때문에, 사용자에게 새로운 인증 방식에 적응할 시간이 필요한데, 이는 서비스 제공자 입장에선 추가적인 교육과 지원이 필요함을 의미한다.

V. 결론

패스키는 기존의 비밀번호라는 인증 시스템을 대체하기 위해 만들어진 인증 수단이고, Mac과 같은 기기에서 사용했을 때 버튼 한 번으로 로그인할 수 있어 굉장히 편리함을 느낄 수 있었다. 공개 키와 프라이빗 키 서명 방식으로 기존의 비밀번호 시스템보다 훨씬 안전하며, 구글, 애플 등 해외의 대기업들 뿐 아닌 카카오, 네이버와 같은 국내의 기업들도 천천히 도입하고 있는 추세이다.

일부 단점들이 존재하긴 하지만, 보안성과 사용자 경험 측면에서의 장점이 더 크기에 앞으로의 패스키 사용자와 패스키를 제공하는 서비스들은 계속 증가할 것으로 예상된다. 특히 거대 기술 기업들의 적극적인 도입과 지원으로, 패스키는 머지않아 우리의 일상적인 인증 수단이 될 것으로 기대한다.

참고 자료

[1] 비밀번호를 대체할 새로운 기술, 패스키 / Gyuyeon.DEV, March 2025

[2] Passkeys and Verifiable Digital Credentials: A Harmonized Path to Secure Digital Identity / FIDO Alliance, September 2025

[3] The Elliptic Curve Digital Signature Algorithm (ECDSA) / Don Johnson, Alfred Menezes & Scott Vanstone, January 2014

[4] A method for obtaining digital signatures and public-key cryptosystems / R. L. Rivest, A. Shamir, L. Adleman, February 1978

[5] Web Authentication: An API for accessing Public Key Credentials Level 3 / W3C (Tim Cappalli (Okta), Michael B. Jones(Self-Issued Consulting), Akshay Kumar (Microsoft), Emil Lundberg (Yubico), Matthew Miller (Cisco)) , January 2025