인증 프로토콜

컴퓨터시스템보안

금오공과대학교 컴퓨터공학부

최태영

목차

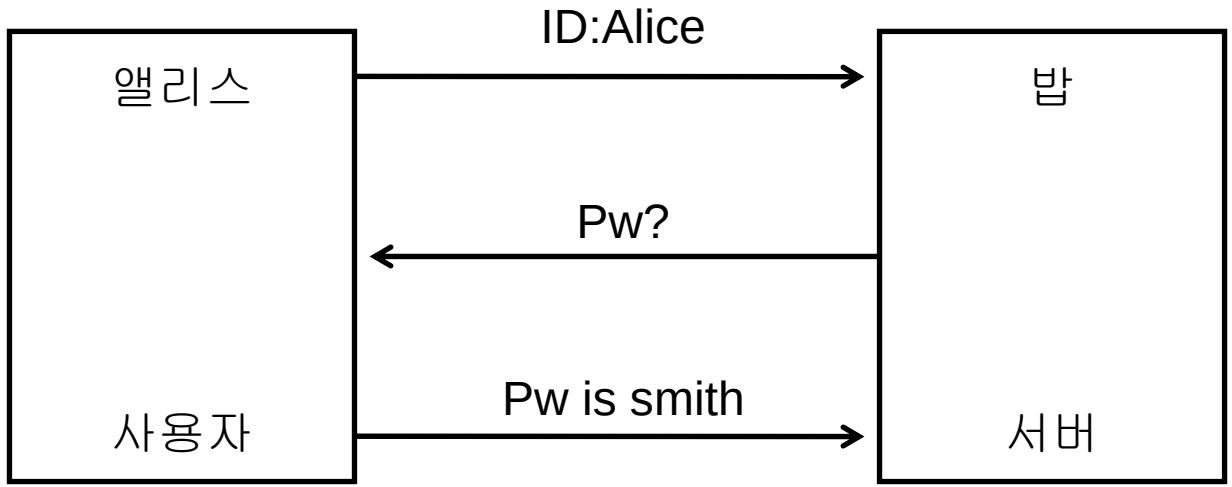
- 단순인증
- 비밀키를 사용한 인증
- 공개키를 사용한 인증
- 인증 후 통신
- 세션 키 교환이 추가된 인증
- 타임스탬프를 이용한 인증
- 영지식 증명
- OpenSSL을 이용한 타임스탬프 인증

주어진 환경

- 인증을 받으려는 기계 또는 서버가 물리적으로 떨어져 있는 상황
- 사용자의 위치에서 서버로 접속할 수 있는 터미널이나 PC가 있음
- 터미널과 서버 사이에는 인터넷과 같은 공개 채널로 연결되어 데이터가 이동함
- 공개 채널은 다른 사람이 접근하여 그 내용을 읽거나 변경할 수 있음
- 공격자가 정식 사용자의 패스워드는 모르고 있다고 가정함
- 암호, 서명 알고리즘은 안전하다고 가정함

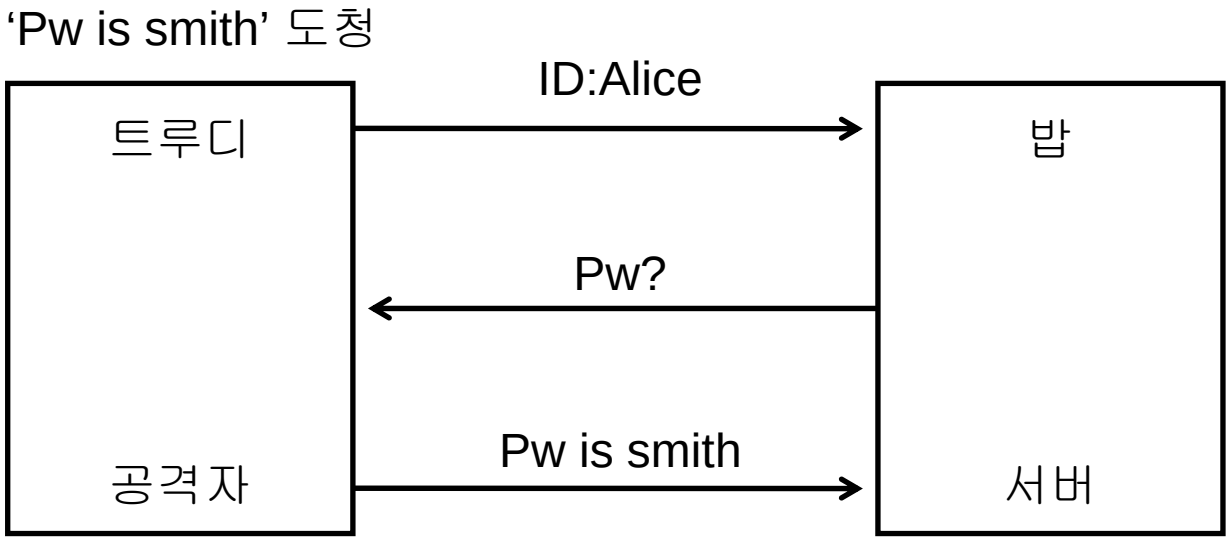
단순인증

- 인증 프로토콜 : 자신의 신원을 증명하는 메시지 교환 단계
  - 다양한 공격을 가정하며 이를 막는 것을 목적으로 함
- 단순 인증 : 서버가 사용자에게 password를 묻고 사용자는 이에 대해 자신의 password를 보내주는 방식
  - 서버는 각 사용자의 password를 보관



단순인증에 대한 공격

- 공격자 트루디는 앨리스인 것처럼 흉내 (impersonate)낼 수 있음
- 재연 공격 (replay attack)
  - 전송된 메시지를 공격자가 다시 보내는 공격
- Password가 공개 채널에 노출된 것이 문제



단순해시인증

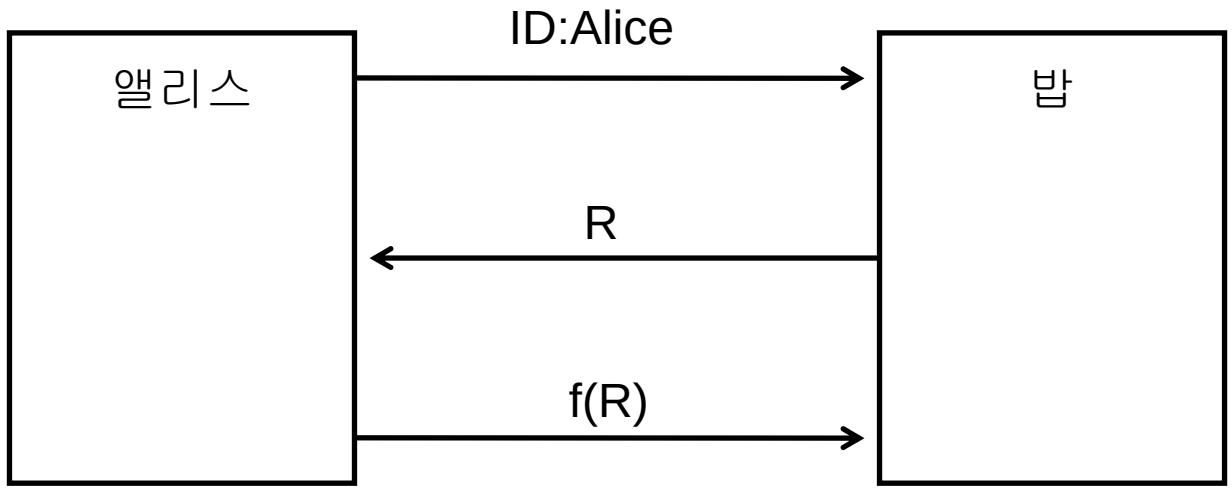
- Password 대신 password의 hash를 보내는 방법
  - h(pw) 전송
- 서버는 사용자의 id와 그 사용자 password의 hash를 저장
- Hash가 도착하면 저장된 hash와 비교
- 여전히 replay attack에 노출됨
  - 공격자는 h(pw)를 보내면 됨
- Replay attack에 당하는 프로토콜들의 특징
  - 동일한 값이 전달됨

Challenge-Response Protocol

- Replay attack을 막기위해서는 매 인증 시마다 다른 메시지가 전달되어야 함
  - 다른 메시지의 내용은 공격자가 예측할 수 없는 값이어야 함
- 인증을 하려는 서버는 random number *R*을 사용자에게 전달
  - R은 한 번만 사용되므로 nonce (number used once)라고 불림
- 사용자는 R과 자신의 비밀 pw를 혼합하여 서버에게 전달

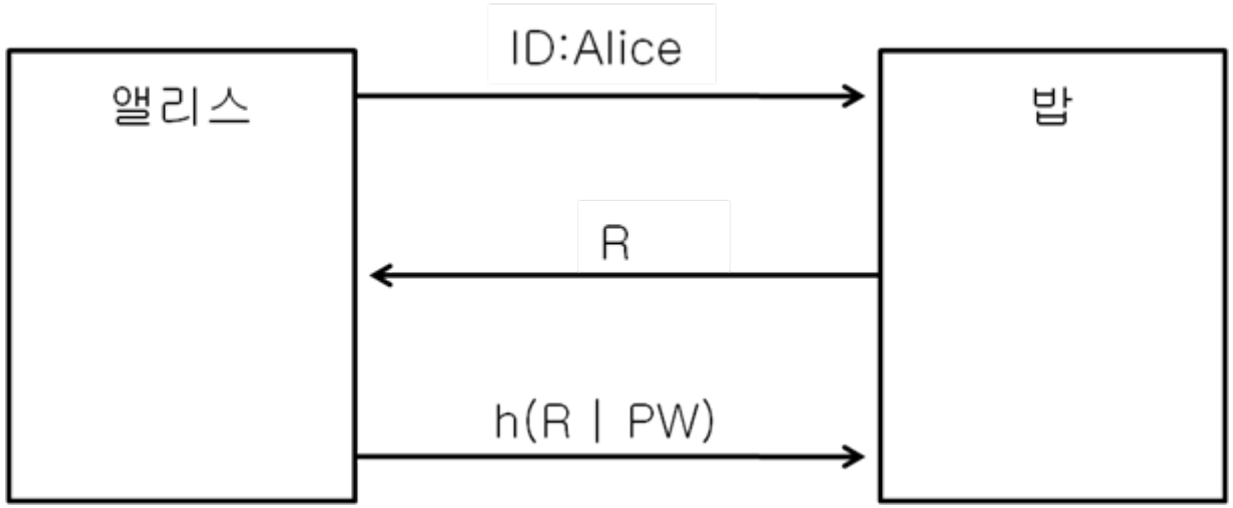
Challenge-Response Protocol

- f() : 가공처리 함수, 암호 알고리즘이나 해시 함수가 될 수 있음.
- 아래 그림의 프로토콜은 앨리스의 비밀이 포함되어 있지 않으므로 앨리스에 대한 인증이 될 수 없음



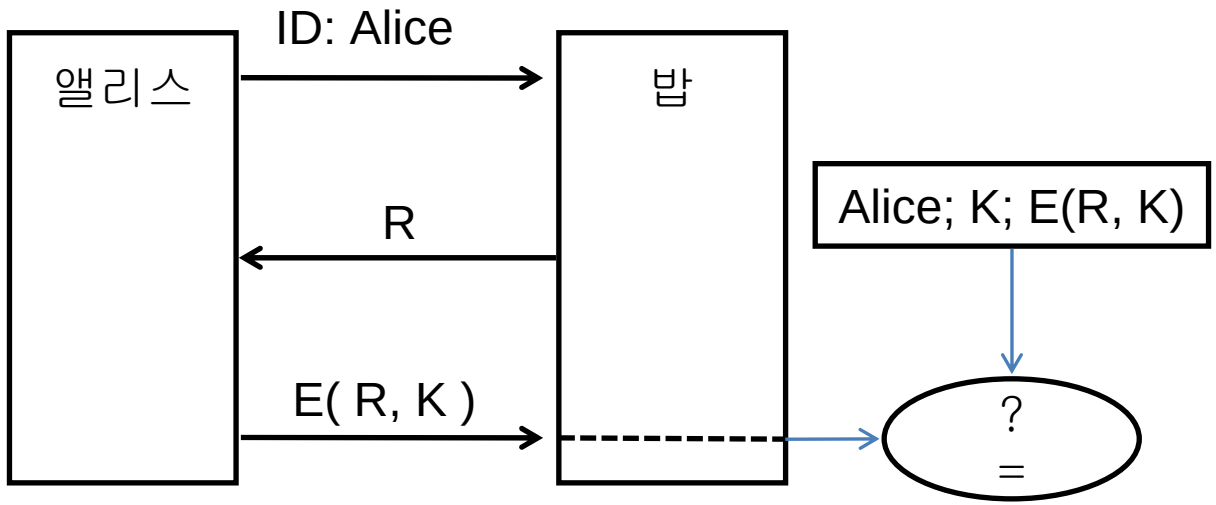
Challenge-Response Protocol

- 양쪽이 해시 함수 h()를 가지고 있는 경우에 사용
- Replay attack은 막을 수 있으나
- 공격자가 서버 밥을 침입하여 앨리스의 password를 탈취하는 공격은 대처할 수 없음



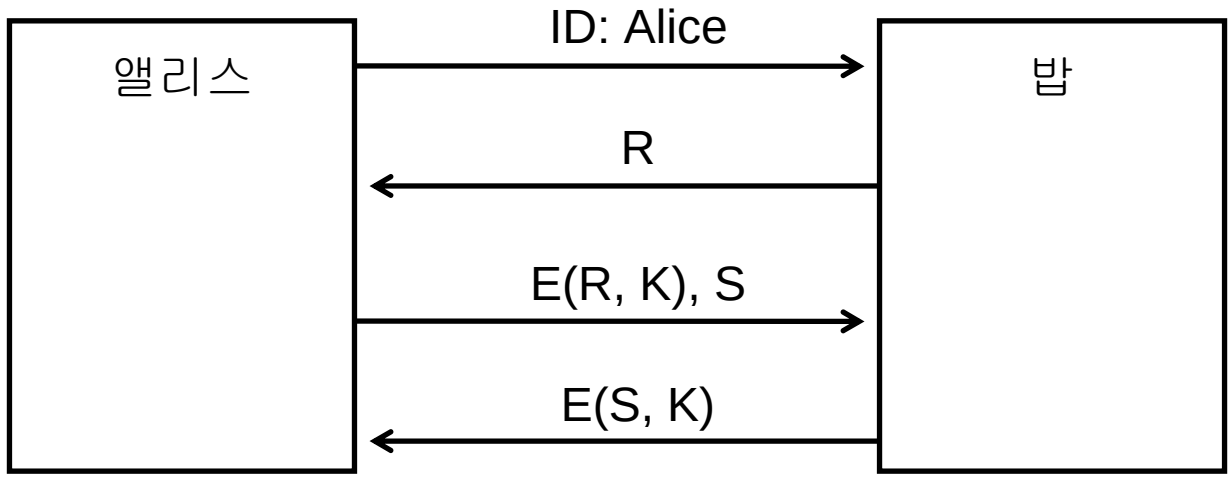
비밀키를 사용한 인증

- 비밀키 표현
  - 암호화 : C = E(P, K) ← 평문 P, 키 K에 대해
  - 복호화 : P = D(C, K)
- 서버 입장에서 사용자 신원 인증



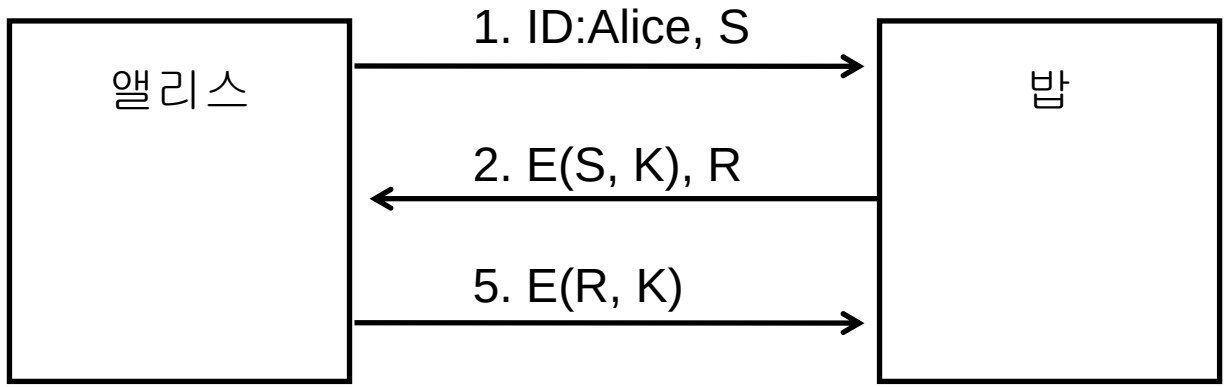
Mutual Authentication

- 서로가 상대방을 인증하는 프로토콜
- 비밀키 K를 공유하고 있다고 가정함



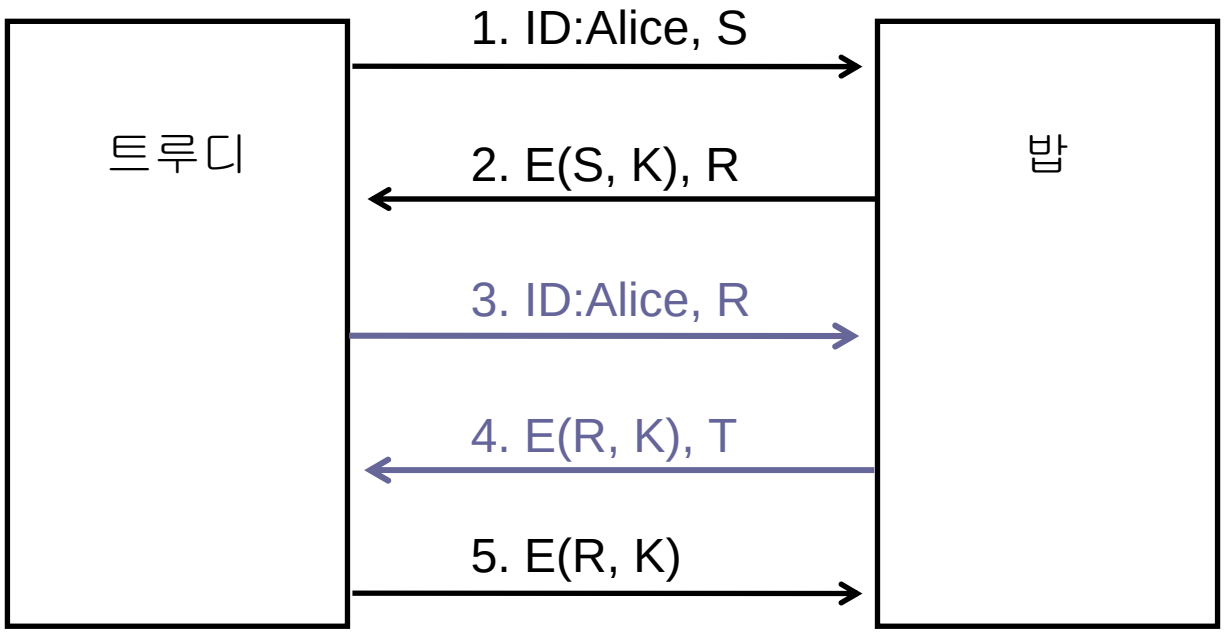
상호인증에서 메시지 감소 시도

- 사용자가 미리 nonce를 보냄으로써 메시지 전송 회수를 줄이려는 시도



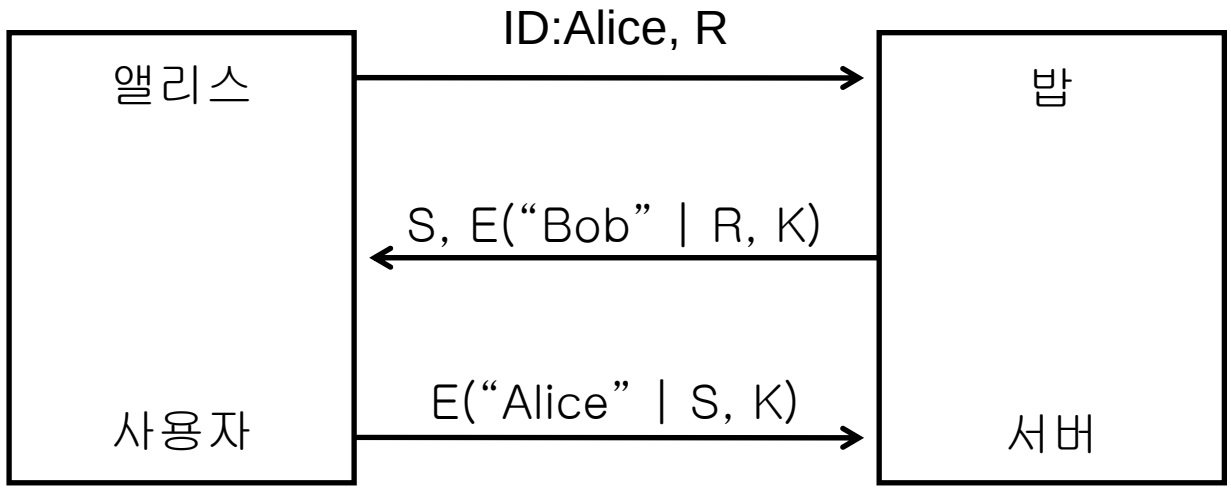
앞의 프로토콜은 공격에 노출됨

- Reflection attack
  - 다른 인증 프로토콜에서 사용된 메시지를 도용함



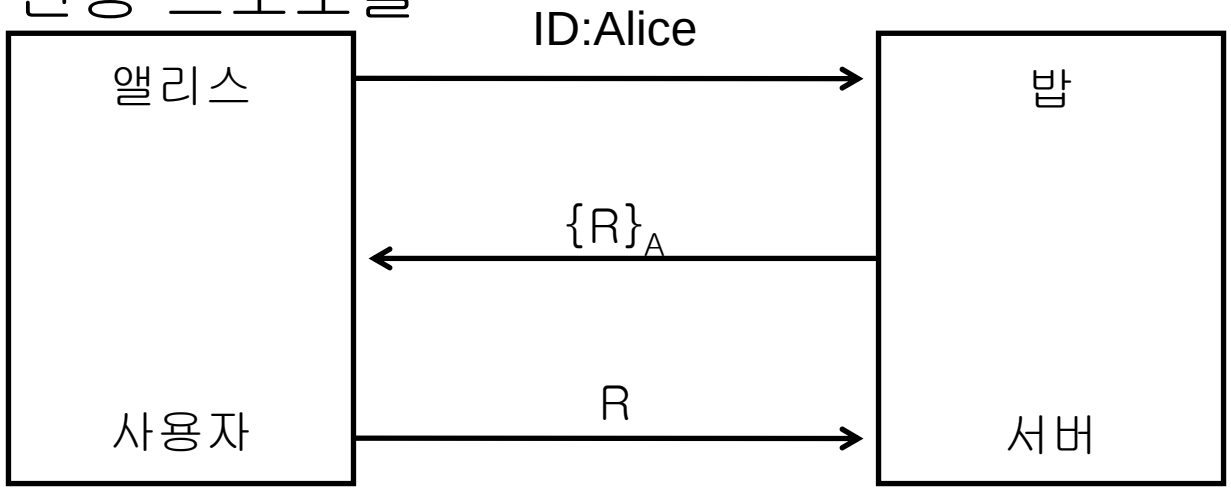
Mutual Authentication with 3 Messages

- 메시지에 받을 사람의 이름을 기입
  - reflection attack을 막기 위해
- 받은 사람은 메시지내의 이름이 상대방임을 확인



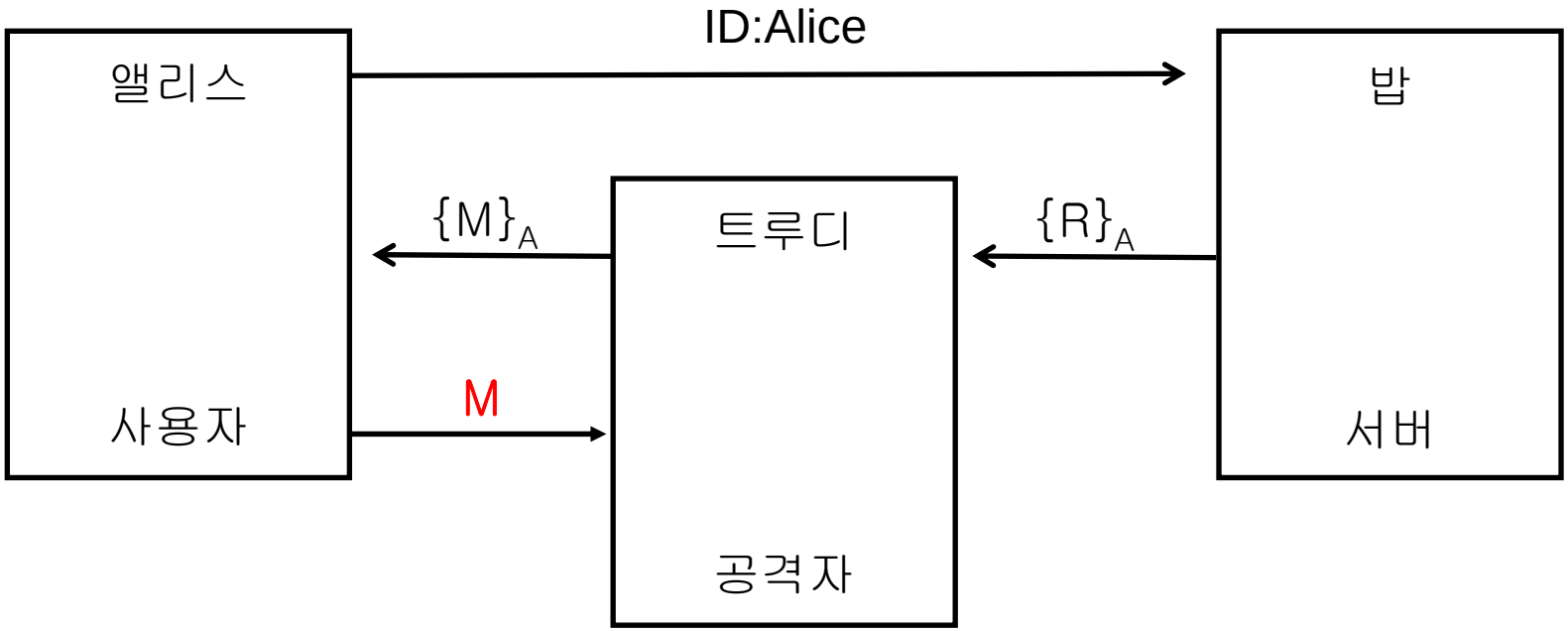
공개키를 사용한 인증

- 표기법
  - A의 공개키로 암호화 : C = {P}A
  - A의 개인키로 복호화 : P = [C]A
  - A의 개인키로 서명 : S = [P]A
  - A의 공개키로 검증 : P = {S}A
- 공개키 인증 프로토콜



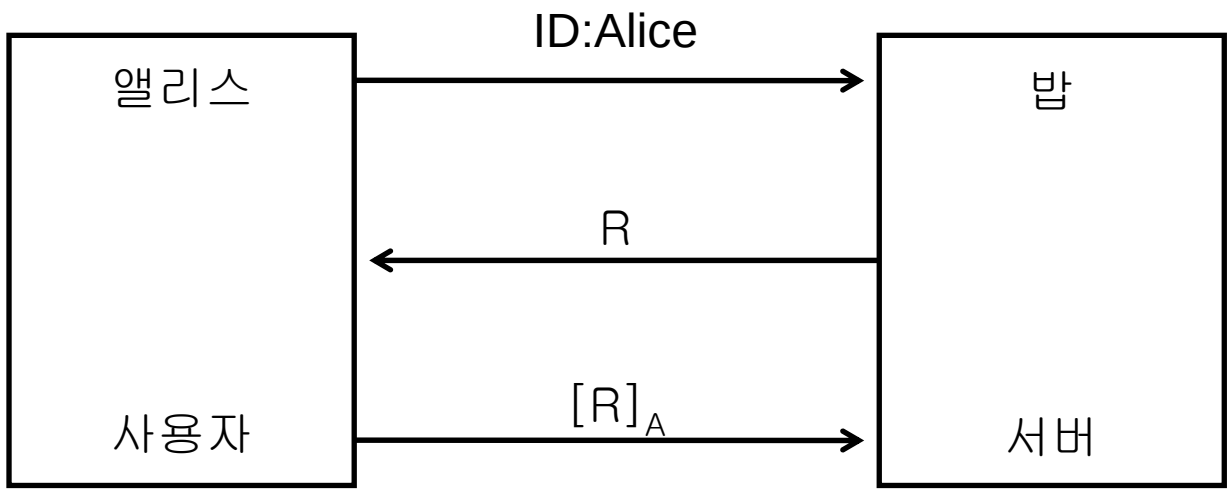
공개키를 사용한 인증 (계속)

- 암호화에 사용되는 공개키쌍을 인증에 사용하는 것은 위험함



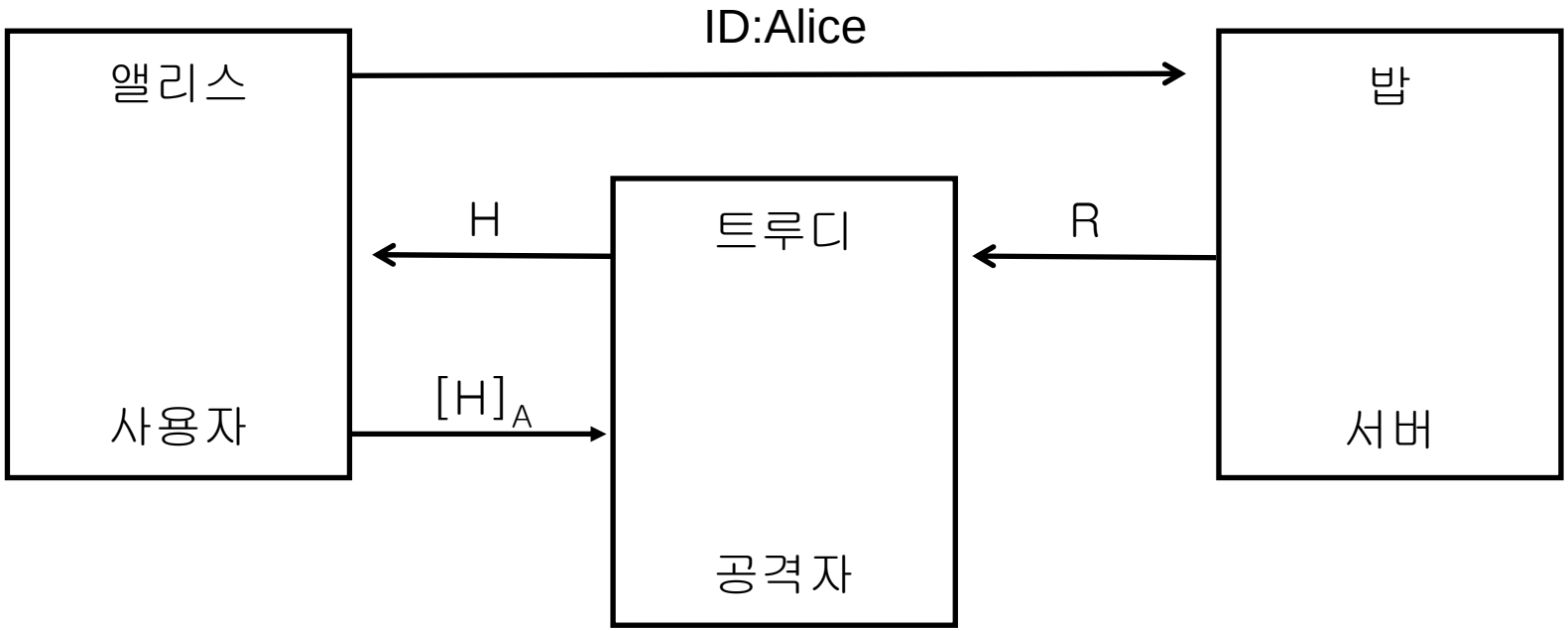
개인키를 사용한 인증

- 서버는 사용자의 공개키를 가지고 있어야 함



개인키를 사용한 인증

- 공개키의 경우와 마찬가지로 서명에 사용하는 개인키를 인증에 사용하면 위험함



인증 후 통신

- 인증 프로토콜이 완료되었다고 해서 이후 통신에 대한 안전이 보장되는 것은 아니다.
- 인증 후 보내는 메시지에 아무런 처리가 가해지지 않으면 공격에 노출된다.
- 메시지 전송을 위한 단순한 보장 시도
  - 메시지의 앞에 보내는 사람의 이름을 넣는 방법
    - ID | P
    - 공격자가 내용을 바꿀 수 있다.
  - 메시지를 암호화하여 보내는 방법
    - E(P, K)
    - 어떤 키로 암호화했는지 알 수 없다.
  - 메시지 앞에 사람 이름을 넣고 암호화하는 방법
    - ID | E(P, K)
    - 두 값을 모두 바꿀 수 있다. 받는 사람은 메시지 변경 여부를 보장하지 못한다.
  - 메시지 앞과 암호화할 메시지 앞에 사람 이름을 넣고 암호화하는 방법
    - ID | E(ID | P, K)

TCP 연결의 경우

- TCP (Transmission Control Protocol)
  - connection oriented protocol
  - telnet, FTP, Web 등의 프로그램에서 사용
- TCP 연결 과정
  - Client → Server : SYN, SEQ i
    - i : random number
  - Server → Client : SYN, ACK i+1, SEQ j
    - j : random number
  - Client → Server : ACK j+1, data
- 이후 통신 : 일련번호와 ip 주소를 통해 상대방을 확인
- 인증이 필요한 경우 : ID와 PW를 서버에 전송
  - 소극적 공격자는 packet scanning을 통해 pw를 얻음

TCP Connection Hijacking

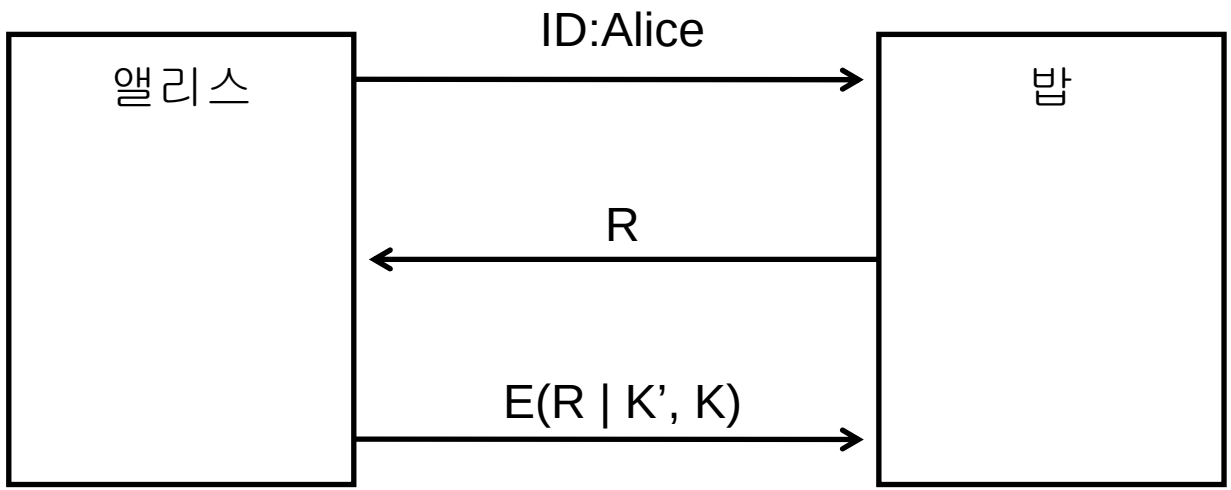
- 정상적인 client의 패킷을 server가 무시하도록 하는 것이 목적
  - server가 인식하는 일련번호를 바꿈
- client에게 연결종료 패킷을 보내고, server를 비연결 상태로 유도하는 패킷을 보냄
- server의 일련번호를 조작하는 NULL data packet을 보냄
  - server는 변경된 일련번호로 공격자와의 연결이 설정됨
- 사용자가 인증에 성공한 후 이 공격을 가하면 공격자는 password가 없어도 서버에 접속할 수 있음

세션 키 교환이 추가된 인증

- 인증이 정상적으로 유지되기 위해서는 패킷들이 암호화되고 서명이 첨부되는 것이 바람직함.
- Client와 server간의 공개키로 암호화하는 것은 속도가 느림.
- 둘 간의 master secret key를 사용하는 것은 위험함 (키가 닳게 됨)
- 한 번의 접속을 위해서만 사용되는 비밀키인 session key를 사용하는 것이 바람직함
- 인증 과정에서 session key를 교환할 필요가 있음
- 인증 후 통신은 session key로 암호화하여 진행됨

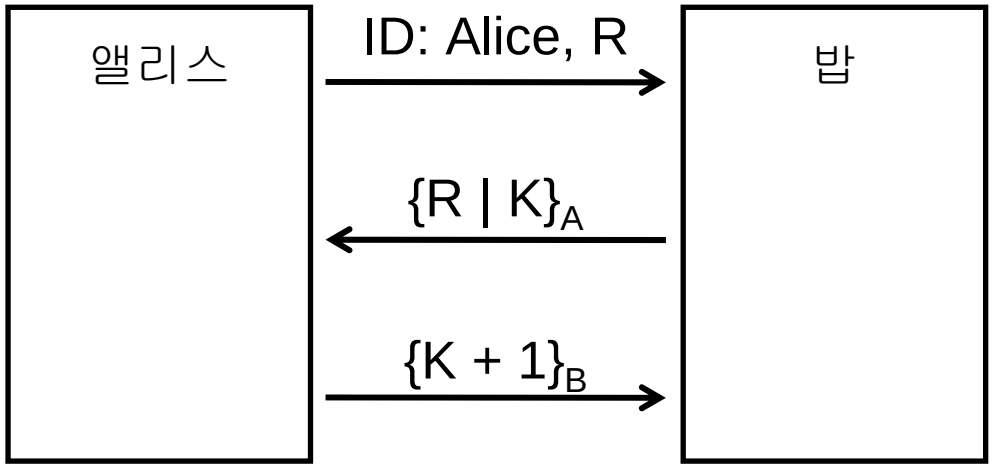
비밀키 암호화를 이용한 키 교환

- Client와 server가 비밀키 K를 공유한 경우
- K'가 session key가 됨
- 인증 과정도 포함
- 상호 인증이 됨 (어떻게?)



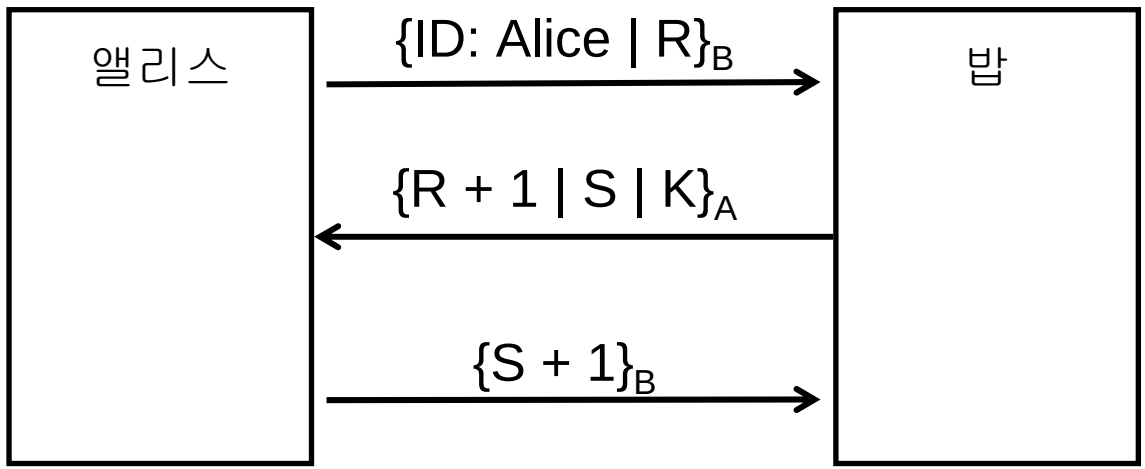
공개키 암호화를 이용한 키 교환

- Server 밥은 client 앨리스를 인증함.
  - K+1은 앨리스가 개인키로 두 번째 메시지를 복호화해야 구할 수 있으므로
- 상호인증은 안됨 (왜?)



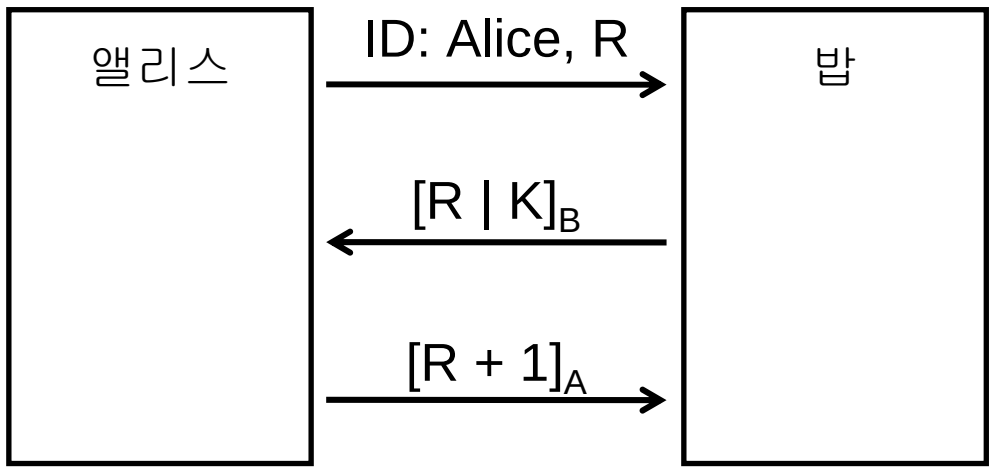
공개키를 이용한 상호인증 키 교환

- 두 번째 메시지에서 R+1은 밥이 자신의 개인키가 있어야 만들 수 있는 값 (밥 인증)
- 세 번째 메시지에서 S+1은 앨리스의 개인키 필요 (앨리스 인증)
- Reflection attack 가능성
  - 모두 nonce를 사용하므로 매번 다른 메시지가 발생



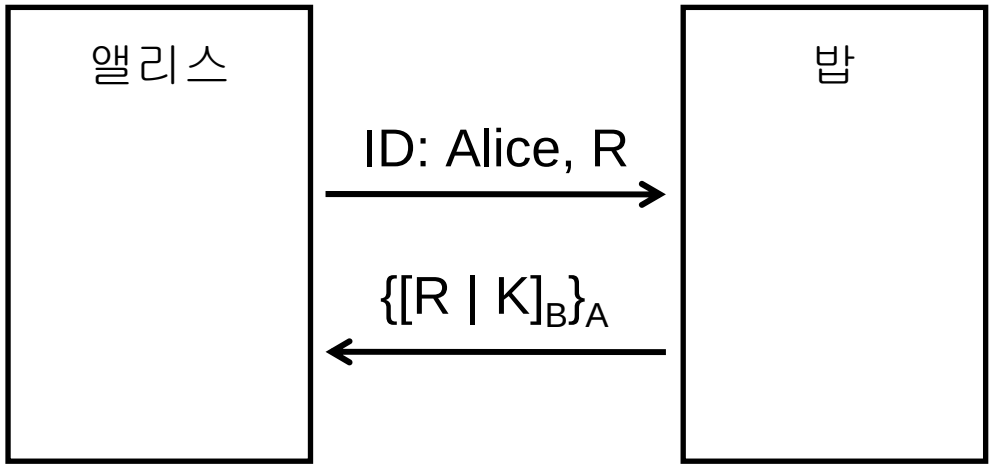
개인키를 이용한 키 교환 시도

- 상호인증은 되지만 키는 노출됨
- 키 노출을 막기 위해서 암호화가 동반되어야 함



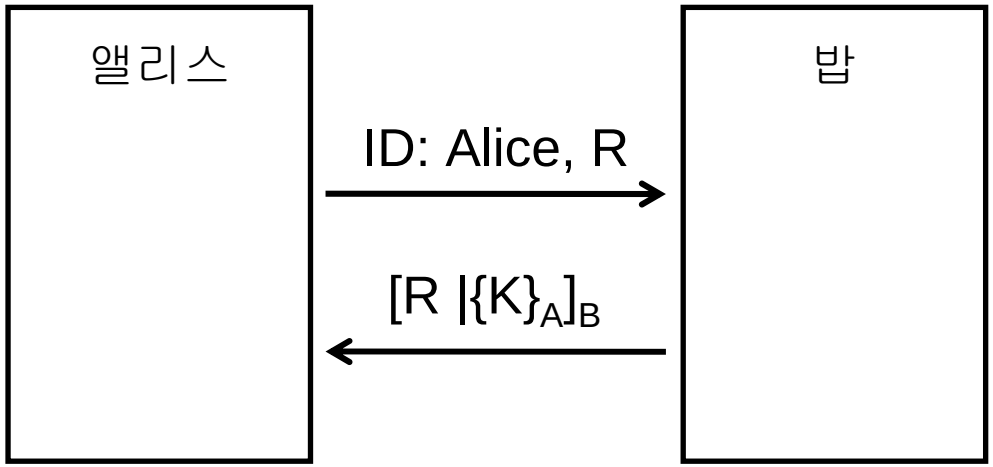
개인키를 이용한 인증 및 키 교환

- 서명 후 암호화
- 공개키와 개인키를 모두 사용함
- 명시적으로 사용자 앨리스는 서버 밥을 인증함
  - 밥이 서명한 메시지를 받으므로
- 밥에 대한 인증은 비밀키 K를 통한 암호화 통신으로 대체됨



개인키를 이용한 인증 및 키 교환

- 명시적으로 사용자 앨리스는 서버 밥을 인증함
- 밥이 앨리스를 인증하는 방법은 비밀키 K로 이루어지는 앨리스와의 통신임
  - 앨리스가 자신의 개인키를 가지지 않은 경우 K를 얻을 수 없으며, 그 결과 둘 사이의 암호화된 통신이 이루어지지 않음

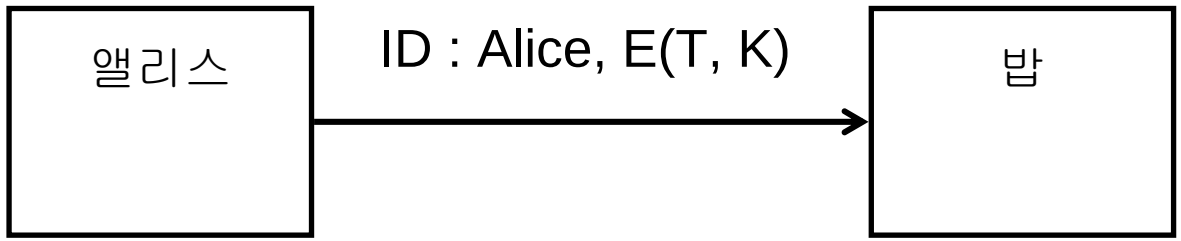


Timestamp

- 타임스탬프 : 현재 시각을 나타내는 정수
  - 통신을 주고 받는 컴퓨터들 사이에 동일한 값을 가지는 것이 이상적임
- Clock skew : 컴퓨터 timestamp 차이의 최대값으로 인정되는 값
  - Clock skew 이내의 오차를 가지는 timestamp는 동일하다고 판단
  - 너무 크면 공격 가능성 증가, 너무 작으면 정식 사용자를 인증하지 못하게 됨

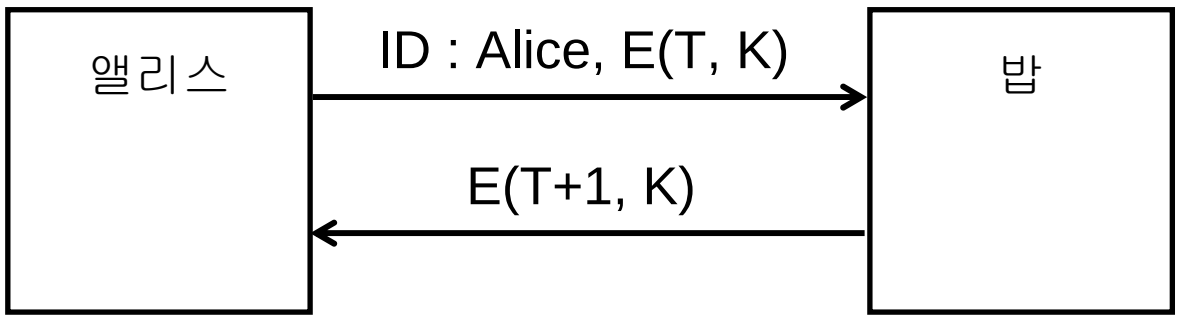
대칭키와 타임스탬프을 이용한 인증 (사용자 인증)

- 밥은 앨리스의 메시지를 키 K로 복호화하여 자신의 현재 시각 T_B와 비교
  - time skew > $|T - T_B|$ 이면 앨리스 인증
- 공격자는 앨리스가 보낸 메시지에 대해 replay attack을 가함
  - time skew가 크면 이 공격이 성공함



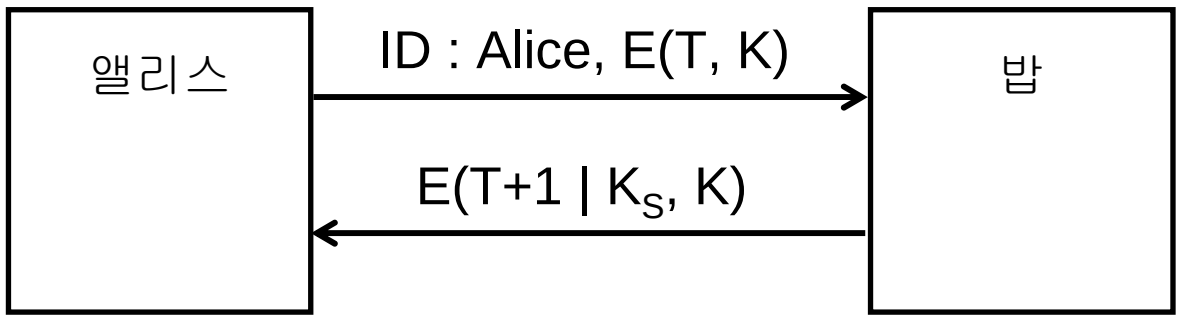
Timestamp (상호 인증)

- 여전히 replay attack 가능성이 있음
- 해결책
  - 밥은 인증한 ID와 timestamp T를 기억
  - 동일한 메시지가 인증을 요구하면 차단함
  - 문제점 : 저장된 ID와 T를 위한 공간이 증가됨
    - 해결 : time skew를 넘는 값들은 폐기함
- 공격자는 E(T+1, K)를 replay attack에 사용함
  - 해결책 : 밥은 E("Bob"|T+1, K)를 앨리스에게 보냄



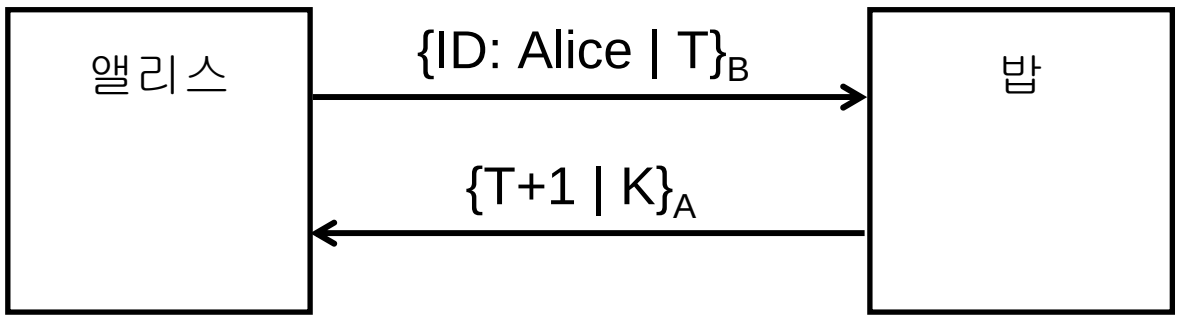
Timestamp (상호인증, 키 교환)

- Timestamp T를 비밀키로 사용하는 것은 위험 (상위 값은 바뀌지 않으므로)
- 별도의 세션키 K_S를 비밀키로 사용



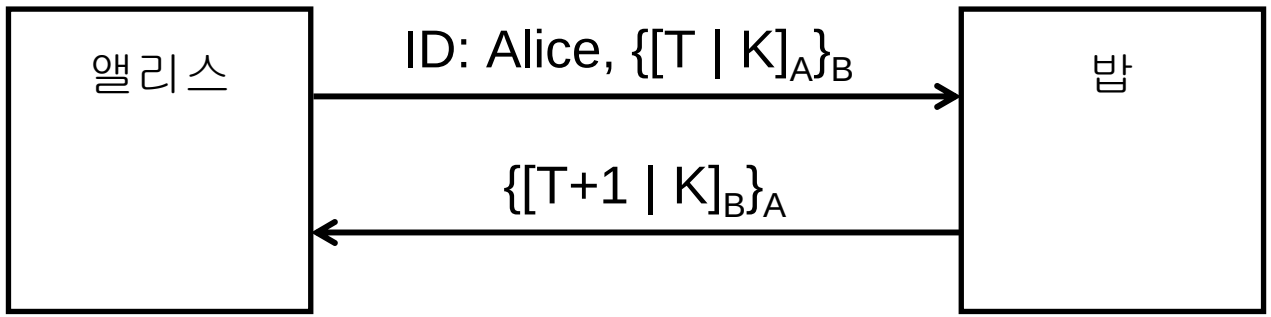
Timestamp와 공개키를 이용한 인증

- 첫 번째 메시지는 아무런 인증도 수행하지 않음
  - timestamp T를 밥에게 비밀리에 보내기 위해서만 사용
- 두 번째 메시지에서 T+1으로 앨리스는 상대방이 밥임을 인증함 (상대방이 T를 보았으므로)
- 앨리스는 명시적으로 인증되지는 않았음
  - 비밀키 K로 통신이 된다는 사실이 앨리스임을 인증하는 수단임



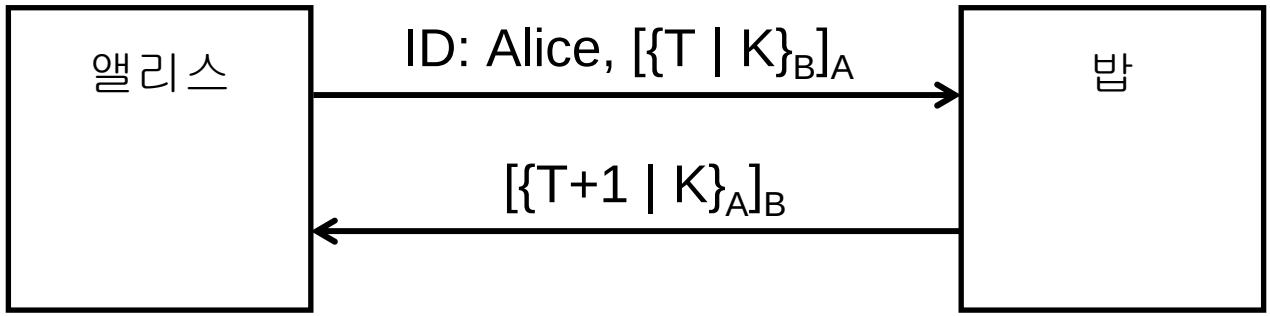
Timestamp을 통한 공개키 상호인증

- 명시적 상호인증이 필요한 경우
  - 공개키와 개인키를 모두 사용
  - 첫 번째 메시지로 앨리스 인증 (T = 현재시각)
  - 두 번째 메시지에서 키가 포함될 필요가 있는가?

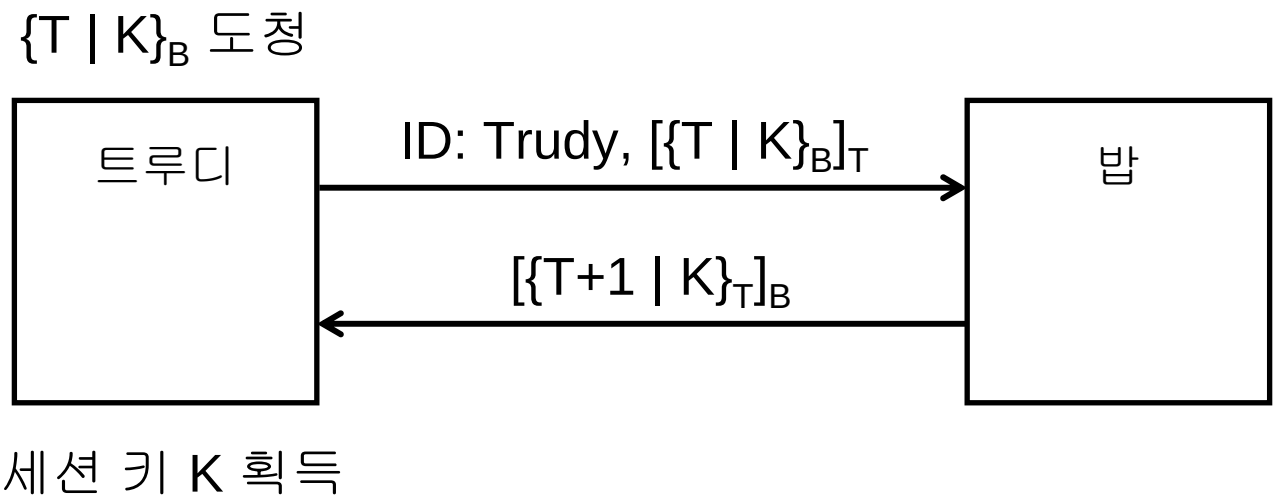


암호화 순서에 따른 결함 발생

- 밥의 공개키로 암호화되어 있는 세션키 K를 공격자가 볼 수는 없지만 이를 이용할 수 있음
- 세션키가 두 개의 메시지에 모두 있는 것은 공격의 가능성과 네트워크 오버헤드를 유발시킴



- 공격자 트루디는 $[\{T \mid K\}_B]_A$에서 앨리스의 공개키로 $\{T \mid K\}_B$를 얻음
- 아래의 프로토콜로 앨리스와 밥의 세션키 K를 얻음
- 해결책 : 둘 중의 한 메시지에 세션키를 제거함



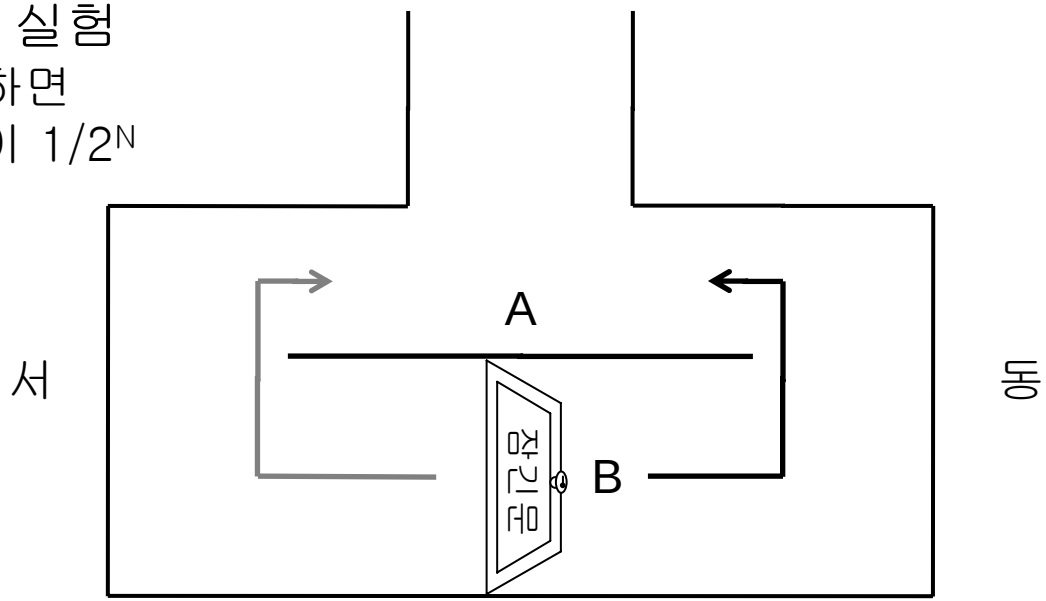
Zero Knowledge Proof

- 영지식 증명 (ZKP)은 Fiege, Fiat, 그리고 Shamir가 발표
- 피아트-샤미르 프로토콜, 또는 피에지-파이트-샤미르 프로토콜로 알려져 있음
- 목적 : 자신의 비밀에 대한 정보를 전혀 노출하지 않은 상태로 자신을 인증하는 것
- 동굴 모형이 ZKP의 설명에 사용됨
  - B는 자신이 열쇠를 가지고 있음을 A에게 입증하려고 함
  - B는 A에게 열쇠를 보여주고 싶지 않음

동굴 모형

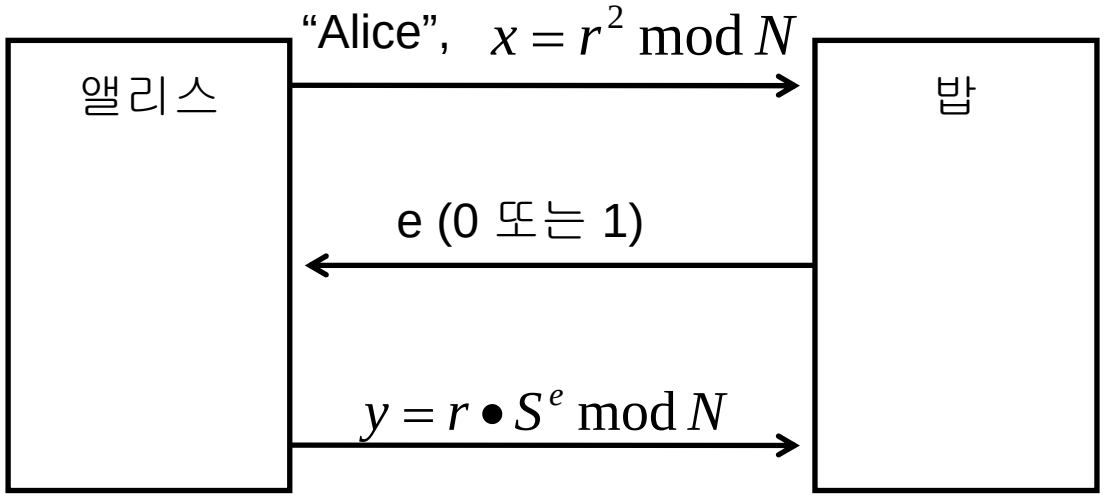
- A는 동쪽 혹은 서쪽으로 나오라고 요청
- B가 A가 요청하는 방향으로 나오면 열쇠를 가지고 있을 가능성이 있음
- 그렇지 못하면 B가 열쇠가 없다는 것이 확실함
- 가능성을 높이려면 반복 실험
  - N번 시도가 모두 성공하면
  - B가 열쇠를 가질 확률이 $1/2^N$

A가 요청할 방향을 B가 미리 예측할 수 있다면 열쇠가 없더라도 A가 말하는 방향으로 나올 수 있다. (미리 그 위치에 있으면)



Feige-Fiat-Shamir Identification Scheme

- 사용자 앨리스의 선행 작업
  - 소수 p와 q를 선택, N = p·q 계산, p와 q는 버림
  - 비밀 S를 만들고 밥에게 v = S^2 mod N 전달



FFS (Cont.)

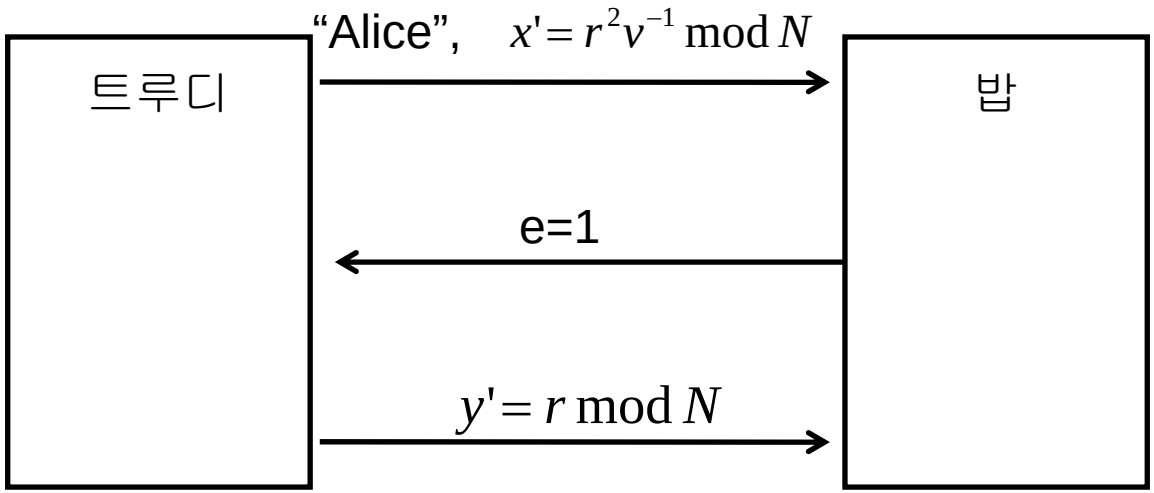
- 앨리스
  - $x = r^2 \bmod N$을 계산하여 밥에게 전달
- 밥
  - e=0을 제시하면 앨리스는 $y = r$을 전달
  - e=1을 제시하면 앨리스는 $y = r \cdot S \bmod N$을 계산하여 전달
- 밥은 y을 제곱하여 $x \cdot v^e \bmod N$과 비교
  - 다르면 앨리스가 아니라고 판단
  - 같으면 앨리스일 가능성이 높아짐
  - $y^2 = (r \cdot S^e)^2 = r^2 \cdot (S^2)^e = x \cdot v^e \pmod N$

FFS에 대한 공격자의 입장

- 밥이 e=0을 보낼 것으로 예측한다면
  - 공격자 트루디는 1번째 메시지에서 임의의 r을 선택해서 $x = r^2 \bmod N$을 밥에게 보냄
  - 3번째 메시지에서 y = r을 밥에게 보냄
  - 밥: $y^2 = r^2 = x = x \cdot 1 = x \cdot v^0$
- 밥이 e=1을 보낼 것으로 예측한다면
  - 1번째 메시지: $x = r^2 \cdot v^{-1} \bmod N$
  - 3번째 메시지: $y = r \bmod N$
  - 밥: $y^2 = r^2 = r^2 \cdot v^{-1} \cdot v = x \cdot v^1$

FFS에서 밥이 e=1을 보낼 것을 공격자가 예상했을 때

- 여전히 공격자 트루디는 비밀 S를 모르더라도 밥을 속일 수 있다.



FFS를 이용한 인증 프로토콜

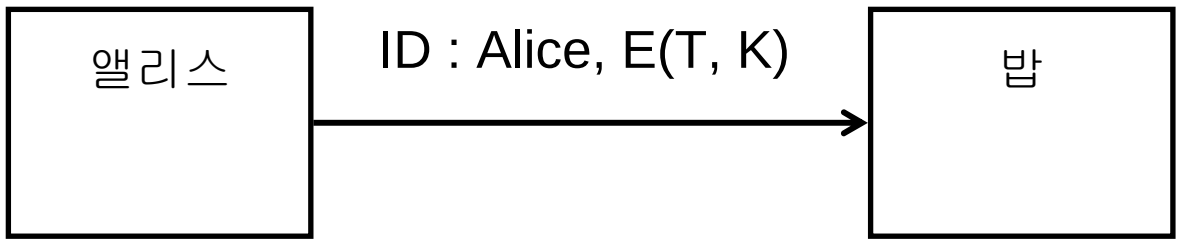
1. 오차범위 ε을 설정한다.
2. 반복회수 N은 $2^{-N} \leqq \varepsilon$ 이 되도록 설정하여 다음을 N번 수행한다.
   1. 파이지-피아트-샤미르 프로토콜을 수행한다.
   2. 수행 결과 $y^2 = x \cdot v^e \bmod N$가 만족되지 않으면 상대방을 인증하지 않는다.
3. N번 수행을 모두 통과하면 상대방을 인증한다

FFS의 안전성과 공개키와의 비교

- 안전성
  - 사용자의 비밀 S는 v=S2 mod N이나 y = r·Se mod N으로부터 얻기 어렵다 (S2 > N인 경우)
- 공개키도 개인키를 노출하지 않고 인증함
  - FFS는 모듈러 곱 연산이 매우 적음 (사용자나 서버가 각각 2N번씩의 모듈러 곱을 수행)
  - 공개키는 서명 과정에서 모듈러 곱 회수가 많음

OpenSSL을 이용한 타임스탬프 인증

- 현재시각 함수로 timestamp를 얻음
  - #include <sys/time.h>
  - int gettimeofday(struct timeval *restrict tp, void *restrict tzp);
- 한 컴퓨터에서 client와 server process를 각각 수행
  - Server process는 port no. 9999를 사용
- 단일 인증 프로토콜 구현



사용하는 암호 함수

- #include <openssl/des.h>
- void DES_ncbc_encrypt(const unsigned char *input, unsigned char *output, long length, DES_key_schedule *schedule, DES_cblock *ivec, int enc);
  - input : 평문이 저장된 버퍼의 위치
  - output : 암호문이 저장될 버퍼의 위치
  - length : 평문의 길이 (byte)
  - schedule : 비밀키 (key schedule의 형태)
  - ivec : Initial vector
  - enc : encryption / decryption
- int DES_set_key(const_DES_cblock *key, DES_key_schedule *schedule);
  - 8 bytes 난수 key를 key schedule 형태인 schedule에 저장하는 함수

코드의 구성

- Client
  1. 비밀키를 읽는 부분
  2. 현재시각 타임스탬프를 구해서 암호화하는 부분
     - 먼저 메시지 크기를 8 byte의 배수로 만들어야 함
  3. 암호화된 타임스탬프를 보내는 부분
  4. 결과를 받아 출력하는 부분
- Server
  1. 포트 설정 및 메시지 대기
  2. 메시지 수신 및 비밀키 읽어 들임
  3. 복호화 및 타임스탬프 검사
  4. 결과 메시지 회신

Client가 timestamp를 보낼 때

```
gettimeofday(&timeStamp, NULL);
fd = open("symmKey.sec", O_RDONLY); assert(fd != -1);
res = read(fd, rawkey, 8); assert(res == 8);
close(fd);
DES_set_key(&rawkey, &keySched);

bzero(iv, sizeof(DES_cblock));
DES_ncbc_encrypt((unsigned char *)&timeStamp, buff,
 sizeof(struct timeval), &keySched, (DES_cblock
 *)iv, DES_ENCRYPT);
cipherBSz = multiple8(sizeof(struct timeval));

/* 서버에게 전송 */

/* 결과를 회신 받음 */
```

Server의 인증 과정

```
 /* buff에 메시지를 받음 */

 gettimeofday(&myTime, NULL);
 fd = open("symmKey.sec", O_RDONLY); assert(fd != -1);
 res = read(fd, rawkey, 8); assert(res == 8);
 close(fd);

 DES_set_key(&rawkey, &keySched);
 bzero(iv, sizeof(DES_cblock));
 DES_ncbc_encrypt(buff, (unsigned char
*)&timeStamp, cipherBSz, &keySched, &iv, DES_DECRYPT);
 if (abs(myTime.tv_sec-timeStamp.tv_sec) < TIMESKEW){
   strcpy(buff, "yes");       send(sockfd, &buff, 5, 0);
 } else {
   strcpy(buff, "no");        send(sockfd, &buff, 5, 0);
 }
```