정보보호개론(CSE441)
www.facebook.com/KoreaTech.Crypto

해시함수의 활용

NOTE 11

한국기술교육대학교 컴퓨터공학부 김상진

sangjin@koreatech.ac.kr
www.facebook.com/sangjin.kim.koreatech

교육목표

- 패스워드
  - 패스워드 해싱
  - 패스워드 기반 프로토콜
  - 일회용 패스워드
- 해시체인
- 해시퍼즐
  - 작업증명과 블록체인
- 해시트리
- 해시 기반 서명

해시함수

- 해시함수(hash function): 임의의 길이에 이진 문자열을 고정된 짧은 길이의 이진 문자열로 매핑하여 주는 함수
- 요구사항
  - 압축
  - 계산의 용이성: x가 주어지면 $H(x)$는 계산하는 것이 효율적이어야 함
  - 일방향성(one-wayness): 입력을 모르는 해시값 y가 주어졌을 때, $H(x) = y$를 만족하는 x를 찾는 것은 계산적으로 어려워야 함 (OWHF, One-Way Hash Function)
    - $H(x) = y$를 만족하는 x는 여러 개 존재함. 이 중 어떤 것도 찾을 수 없어야 함
  - 약한 충돌회피성(weak collision-resistance): x가 주어졌을 때 $H(x') = H(x)$인 $x'(\neq x)$을 찾는 것은 계산적으로 어려워야 함
  - 강한 충돌회피성(strong collision-resistance): $H(x') = H(x)$인 서로 다른 임의의 두 입력 x와 x'을 찾는 것은 계산적으로 어려워야 함 (CRHF, Collision-Resistant Hash Function)

해시함수의 활용

- 전자서명 (다음 슬라이드)
  - 전자서명의 비용을 줄이기 위함
    - 메시지 대신에 메시지 해시값에 전자서명을 함 (계산 속도⇑, 서명값 크기⇓)
  - 충돌회피 해시함수이어야 함
- 무결성 서비스: 예) 파일과 파일의 해시값 보관
  - 해시함수보다는 MAC을 사용하는 것이 안전함
- 비트 약속
  - 값을 전달할 때 먼저 값의 해시값을 전달하고, 나중에 필요한 시점에 공개함
    - (값 은닉) 일방향성 특성 때문에 해시값을 받은 참여자는 어떤 값을 받았는지 알 수 없음
    - (바인딩) 충돌회피 특성 때문에 최초 해시값을 계산할 때 사용한 입력이 아닌 다른 값을 나중에 공개할 수 없음

해시함수와 전자서명

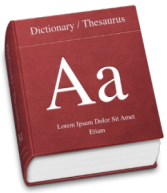
- 서명 크기와 비용을 줄이기 위해 전체 메시지 대신에 메시지의 해시값에 서명함
  - 요구사항. 충돌회피
- RSA의 경우 값에 직접 서명하면 위조가 가능하기 때문에 반드시 해시함수를 활용하여야 하며, 안전성을 높이기 위해서는 서명할 내용이 RSA 법의 크기와 유사하게 되도록 전체 영역 해시를 해야함
  - 가장 널리 사용하는 방법: RSA-PSS
  - 전자서명뿐만 아니라 RSA의 경우 암호화할 때에는 작은 값을 암호화하거나 효율성을 위해 3, 65537을 공개키로 사용할 수 있는데, 이 경우 메시지를 직접 암호화하지 않고 해시함수 등을 이용하여 RSA 법으로 크기를 확장하여 암호화함 (RSA-OAEP)
  - 둘 다 결정적 RSA 기법을 확률적 알고리즘을 바꾸어 주는 효과도 있음

패스워드 인증

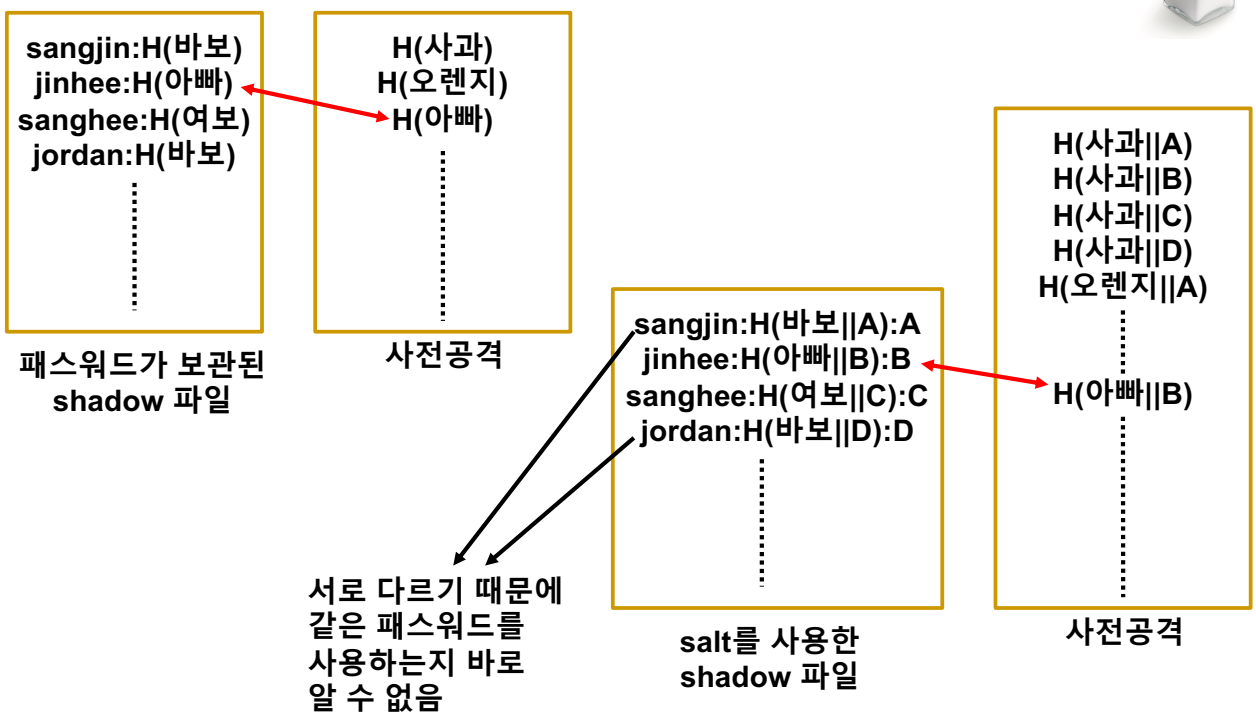
- 현재 사용자 인증을 위해 가장 널리 사용하는 것이 패스워드임
- 패스워드를 보호하지 않은 상태로 전달하면 쉽게 노출될 수 있음
- 현재 인터넷에서 패스워드를 보호하기 위해 가장 널리 사용하는 방법은 TLS임
  - 브라우저와 웹 서버 간의 세션키를 확립한 후에 이를 이용하여 보호된 상태로 서버에 전달함
- 장기간 키를 안전하게 유지하기 위해 가장 널리 사용하는 메커니즘도 패스워드임
  - 예) 공동인증서 개인키
- 패스워드의 가장 큰 문제는 쉽게 예측이 가능하다는 것임
- 사용자들은 7, 8자의 짧은 패스워드를 기억하는 것조차 어려워하는 것으로 조사되었음
- 7, 8자의 영문자와 숫자만으로 구성된 패스워드는 컴퓨터를 사용하면 쉽게 전사 공격하여 알아낼 수 있음
  - 횟수 제한을 통하여 어느 정도 이 문제를 완화시킬 수 있음
  - https://howsecureismypassword.net/

패스워드 해싱

- 유닉스, 웹 서비스 등에서 패스워드 관리
  - 패스워드를 그대로 저장: 패스워드가 저장된 파일에 대한 비밀성 유지가 필수
  - 패스워드의 해시값 보관
    - 사전공격(dictionary attack): 흔하게 사용하는 패스워드에 대한 해시값을 모두 구한 후에 저장되어 있는 해시값과 비교
  - Salt의 사용: 해시값을 구할 때 임의의 랜덤값인 salt를 포함하여 계산
    - 사전공격 비용이 증가함
      - salt는 어디에? 보통 해시값과 함께 저장함
    - 같은 패스워드를 선택한 여러 사람의 해시값도 서로 다름
  - 패스워드를 보관하고 있는 서버의 보안 방법은?
  - 통신 채널로 전달되는 패스워드는?



Salt의 사용



Bcrypt

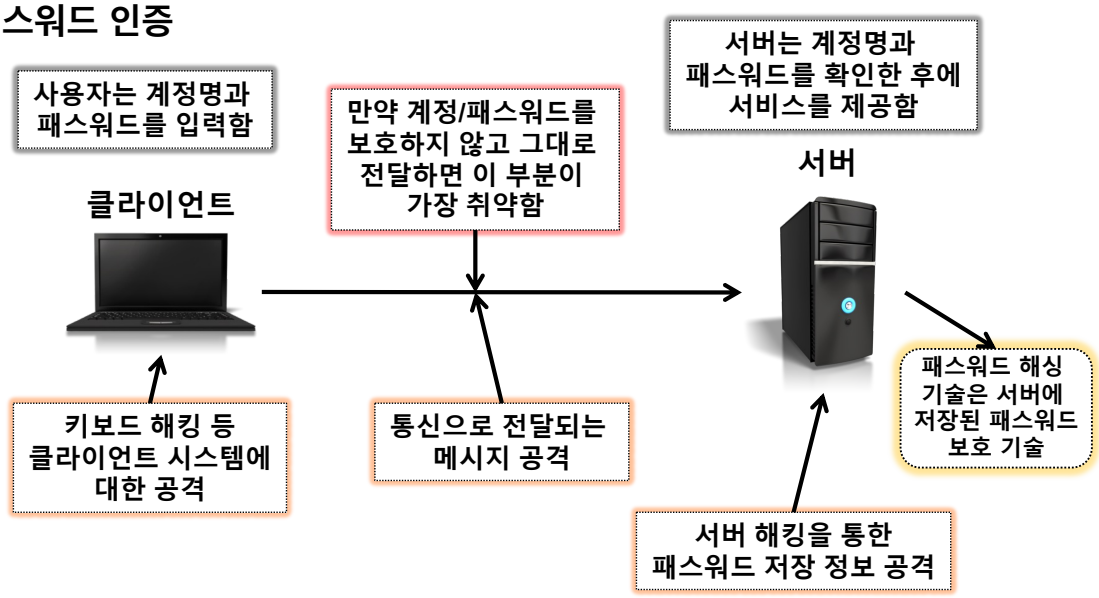
- N. Provos와 D. Mazieres가 제안한 패스워드 해시함수
  - Blowfish 암호알고리즘에 기반함
  - 패스워드 해싱할 때 salt를 사용하는 것은 물론 기존 다른 암호알고리즘과 달리 알고리즘 수행 속도를 느리게 만들 수 있음
    - 수행 속도를 느리게 한다는 것은?
  - Shadow 파일에 $2a$, $2b$로 시작되면 Bcrypt로 해시된 값임
  - Salt의 길이는 128비트이고 결과 해시값은 184비트임
- NIST는 2013년에 패스워드를 해시할 때 사용할 함수에 대한 공개 대회를 개최하였으며, 2015년에 Biryukov 등이 제안한 Argon2가 선정됨

패스워드의 안전성 강화 방법

- 짧은 패스워드 대신에 패스문장(pass phrase)을 사용하는 방법
  - 단점. 입력해야 하는 것이 길어 불편함
- Abadi 등이 제안한 방법
  - 패스워드 π를 전달할 때 20비트 정도의 랜덤 비트 α를 선택하여, $H(\pi||\alpha)$를 전달함
    - 문제점. 서버는 패스워드 자체를 유지해야 함
      - $H(H(\pi)||\alpha)$ 형태로 사용할 수 있음
  - 서버는 2^{20} 가능한 경우를 모두 검증하여 패스워드를 확인함
  - 외부 공격자는 α 때문에 $H(\pi||\alpha)$에 대한 패스워드 추측 공격을 하기가 매우 어려움
  - 서버의 부담이 크지만 환경에 따라서는 수용할 수 있는 정도의 연산량임
- SSO(Single-Sign On)
- Password 관리자. 최근 브라우저는 이 기능을 다 지원함
- FIDO: Note 02

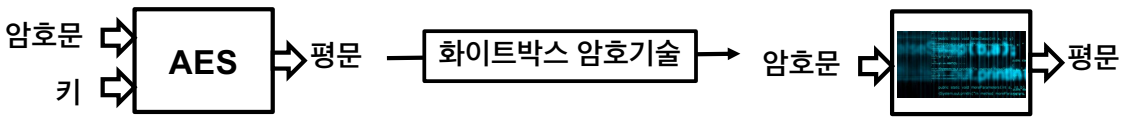
Weakest Link Principle

- 가장 취약 요소 원리.
  시스템의 전체 안전성은 가장 취약한 부분의 안전성과 같음
- 가장 취약한 부분: 보통 사람 (social engineering)
- 예) 패스워드 인증



장기간 키의 보호

- 장기간 키는 비휘발성 메모리에 유지되어야 하며, 불법적인 접근이 있더라도 노출되지 않도록 보호되어야 함
  - 사용자가 암호키 자체를 기억하기는 어렵고, 기억할 수 있다하더라도 그것을 사용할 때 입력하는 것은 번거로움
- 보통 패스워드로부터 대칭키를 생성하고 이 키로 장기간 키를 암호화하여 유지함
  - 예) 공동인증서에서 개인키의 보호
- 저장된 파일이 노출되면 키 자체가 노출될 가능성이 높음
- 이 문제를 해결하기 위해 화이트박스 암호기술을 사용할 수 있음
  - 변하지 않는 암호키를 별도 파일에 저장하는 것이 아니라 암호화 알고리즘 소프트웨어 코드 자체에 포함한 후 해당 코드를 난독화(obfuscation)하여 코드로부터 키를 추출할 수 없도록 만드는 기술
    - 심지어 실행 중 프로그램의 메모리에서 키를 추출할 수 없음

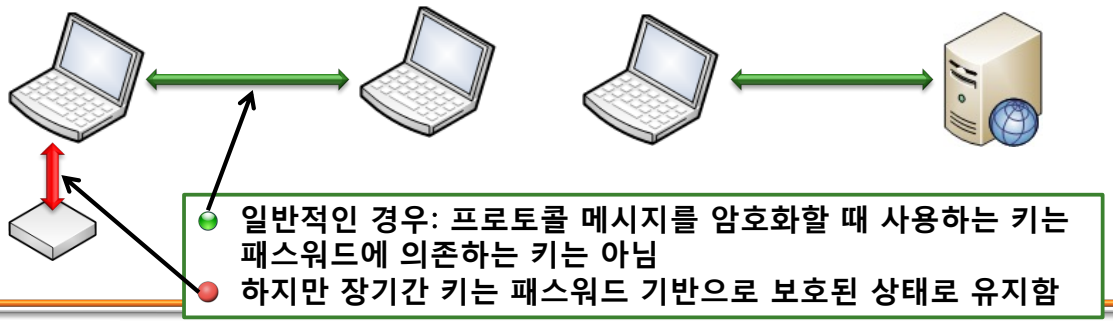


키 관리와 하드웨어

- 장기간 키의 탈취 방지를 위해 키를 특수 하드웨어에 유지할 수 있음
  - 예) 핸드폰의 USIM, 구 셋톱박스의 스마트카드
- 하드웨어를 이용한 키 관리
  - 기본적으로 2-factor 인증: 기기 + (PIN 또는 password)
  - 종류에 따라 기기 내에서 암호 연산 수행
    - HSM(Hardware Security Module)
    - 키를 사용하기 위해 하드웨어 밖으로 복사해야 하는 경우가 없음
  - 조작 불가능 기능 탑재
    - tamper-detection, tamper-resistant, tamper-proof
    - 보통 조작이 발견되면 보관된 데이터를 모두 자동 삭제함
      - 백업 필요
  - 추가 비용이 발생하고 사용 편리성이 저하될 수 있음
- aPAKE: 서버에 개인키 보관 (슬라이드 18 vs. FIDO)

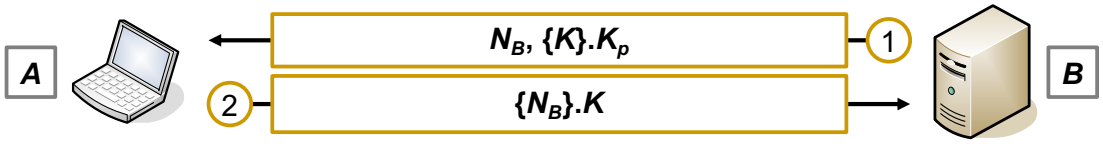
패스워드 기반 암호키

- 보통 암호키는 안전하게 생성한 후 패스워드로 보호하지만 패스워드로부터 직접 암호키(보통 대칭키)를 생성할 수 있음
- 패스워드로부터 생성된 대칭키는 사전 공격에 취약하기 때문에 이와 같은 키들은 보통 로컬에서만 사용하고 원격에 있는 사용자 간에는 사용하지 않음
- Bellovin과 Merritt가 제안한 EKE(Encrypted Key Exchange)와 같은 프로토콜을 사용하면 패스워드 기반 대칭키를 이용하더라도 원격에 있는 사용자 간에 사용할 수 있음
  - 기본 생각. 랜덤 값만 패스워드 기반 대칭키로 암호화함
  - 실제 WPA3에서 SAE 프로토콜을 사용하고 있음



패스워드 기반 프로토콜 (1/2)

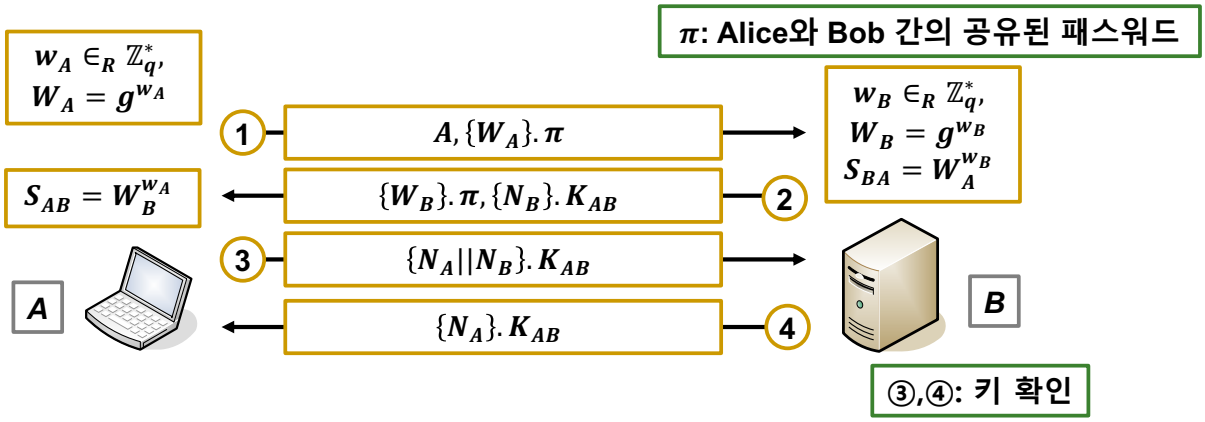
- 패스워드 기반 키를 이용하여 랜덤 값을 암호화한다고 무조건적으로 패스워드 추측 공격에 강건하지 않음
  - 암호화된 랜덤 값을 향후 어떻게 사용하는지에 따라 강건하지 않을 수 있음
  - 암호화된 랜덤 값이 패스워드 추측 공격에 사용 가능한 특성을 가지고 있으면 강건하지 않음
- 예)



- 기본생각. 패스워드를 모르는 사용자는 K를 얻을 수 없으므로 두 번째 메시지를 만들 수 없음
- 문제점. 공격자는 패스워드를 추측한 다음에 그것을 이용하여 첫 번째 메시지에 포함된 암호문을 복호화하여 키를 얻고, 이 키로 두 번째 메시지를 복호화한 값을 N_B와 비교하여 패스워드 추측 공격을 할 수 있음

Original EKE, 1992

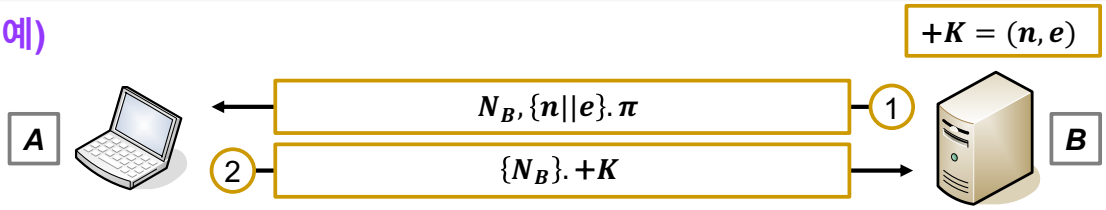
- EKE(Encrypted Key Exchange)는 Bellovin과 Merritt가 제안



- 제3자가 $\{W_A\}.\pi$에 대한 패스워드 사전공격을 하여도 W_A는 랜덤 값이므로 성공을 하였는지 알 수 없음
- 사용하는 군의 특성 때문에 추측 공격의 성공 또는 실패 여부를 알 수 있으면 여전히 패스워드 추측 공격에 취약한 프로토콜임 (다음 슬라이드 참고)

패스워드 기반 프로토콜 (2/2)

- 예)



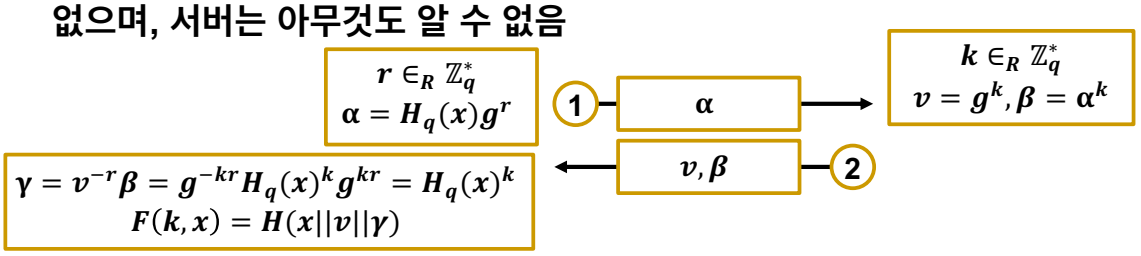
- 랜덤 대칭키 대신에 RSA 공개키를 생성하여 그것을 암호화하여 전달함
- RSA 공개키 (n, e)는 그 자체가 랜덤 값임
- 패스워드 추측 공격을 하여 얻은 임의의 값에 대응되는 개인키는 계산할 수 없음
- RSA 공개키에서 n의 특징:
  1) 특정 비트 크기의 수, 2) 홀수, 3) 소인수분해가 어려운 수
- 공격자는 메시지 1의 암호문을 복호화하여 n이 위 조건을 만족하지 않으면 자신의 추측이 틀렸다는 것을 알 수 있음
- 추측한 패스워드를 후보에서 배제할 수 있음
  ⇒ 가능한 패스워드의 범위를 제한할 수 있음
  ⇒ 패스워드 추측 공격에 강건한 것이 아님

aPAKE (1/2)

- aPAKE(Asymmetric Password Authenticated Key Exchange): 패스워드를 서버와 공유하지 않은 상태에서 패스워드 인증과 상호 인증된 키 교환을 할 수 있게 해주는 프로토콜
- 가정. 사용자는 여러 장치를 사용하여 서비스 이용
  - 장기간 공개키 쌍을 여러 장치에 복사하여 관리하는 것은 별로
- FIDO와 비교
  - FIDO
    - 각 서비스마다 사용자는 다른 공개키 쌍 사용
    - 사용자는 서버에 등록된 공개키에 대응되는 개인키를 소유하고 있음을 증명함
  - aPAKE
    - 서버에 공개키 뿐만 아니라 개인키도 유지함. 하지만 개인키는 암호화되어 유지함
    - 사용자는 패스워드를 통해 개인키를 안전하게 받아 사용함

OPAQUE (1/2)

- 내부적으로 OPRF(Oblivious Pseudo Random Function)을 사용
  - 의사난수 함수 F, 서버 입력 k, 사용자 입력 x
  - OPRF 프로토콜 수행 후 사용자는 $F(k,x)$를 획득하지만 k는 알 수 없으며, 서버는 아무것도 알 수 없음



- 패스워드 등록
  - 사용자 A는 패스워드 π_A와 장기간 공개키 쌍 $(+K_A, -K_A)$을 준비
  - 서버는 OPRF 입력 키 k_A를 준비 (모든 사용자마다 다른 키 사용)
  - 서버와 A는 OPRF를 수행하여 A는 $K = F(k_A, \pi_A)$를 확보
  - K를 이용하여 A는 자신이 장기간 공개키 쌍을 인증 암호화함. Env_A
  - A는 $+K_A$, Env_A를 서버에 전달

- 사용자는 K를 유지하지 않고 매번 서버와 프로토콜을 수행하여 K를 확보하지만 서버는 K를 얻을 수 없음

OPAQUE (2/2)

- 인증 프로토콜
  - 사용자 A가 접속 시도
  - 서버는 Env_A를 전달
  - 서버와 A는 OPRF를 수행하여 A는 $K = F(k_A, \pi_A)$를 확보
  - K를 이용하여 Env_A를 복호화하여 장기간 공개키 쌍 확보
  - 장기간 공개키 쌍을 이용하여 키 확립 프로토콜 수행

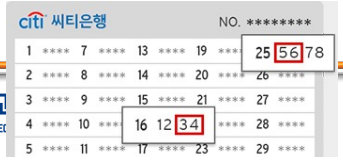
- OPRF는 패스워드 추측 공격에 강건한 프로토콜임
- 올바른 패스워드를 모르면 Env_A를 복호화하기 위한 키를 얻을 수 없음

일회용 패스워드 (1/3)

- OTP(One-Time Password)는 기존 계정명/패스워드 방식보다 안전한 인증 메커니즘을 제공하기 위해 매번 새로운 패스워드를 사용하는 방식을 말함
  - 기본생각. 매번 패스워드를 변경하면 패스워드가 노출되어도 안전성에 문제가 없음
- 클라이언트와 서버는 매번 새롭지만 같은 패스워드를 생성하기 위해 둘 간의 대칭키를 공유하고 있음
  - 대칭키를 사용하기 때문에 부인방지는 약함
  - 클라이언트와 서버가 대칭키를 이용하여 매번 다르지만 동일한 패스워드를 생성하여야 함
  - 대칭키와 매번 바뀌는 값을 이용하여 패스워드를 생성해야 함
  - 매번 바뀌는 값: 시간, 카운터, 랜덤값
    - 시간과 카운터는 공유할 수 있는 것 ⟹ 동기화 방식
    - 랜덤값은 한 쪽이 다른 쪽에 전달해야 함 ⟹ 비동기화 방식
  - 참고. 대칭키를 이용하지 않는 방식도 있음. 예) 해시체인, 공개키 기반 OTP

일회용 패스워드 (2/3)

- 초창기에는 주로 하드웨어 기반 OTP를 사용하였지만 최근에는 소프트웨어 기반 OTP(예: 모바일 뱅킹)를 더 많이 사용하고 있음
  - 하드웨어 OTP의 이점: 별도의 소프트웨어 설치 없이 사용할 수 있음
- 왜 초창기 OTP는 하드웨어 기반 OTP를 사용했나?
  - 초기 사용 방식 때문에
    - 주로 PC에서 브라우저를 이용하여 인터넷 뱅킹을 하였음
    - 어느 PC에서나 인터넷 뱅킹을 할 수 있어야 했음
      - 공동인증서의 USB 저장
  - 소프트웨어 방식이면 PC마다 추가 소프트웨어의 설치가 필요함
    - 더 중요한 것은 사용자의 비밀키가 필요함
  - 저렴한 기기에 등장하는 6자리 숫자만 입력하는 것이 이 방식에 더 유리함
- 지금은? PC보다는 모바일(스마트폰) 앱을 더 많이 사용함
  - 사용자가 일회용 패스워드를 직접 입력할 필요가 없음



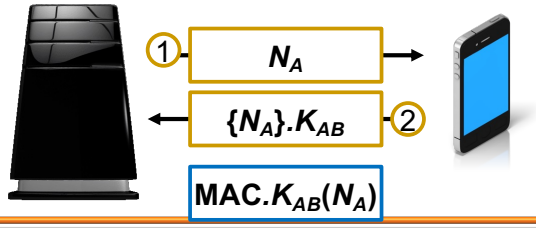
일회용 패스워드 (3/3)

- 하드웨어 기반 OTP의 동작 방식
  - 기기에 나타난 6자리 숫자를 입력함
  - 기기 특성 때문에 시간, 사건 기반 동기화 방식을 사용할 수밖에 없음
- 하드웨어 기반 OTP의 요구사항
  - R1. [OTP 불위조성] 특정 사용자에게 주어진 기기가 생성하는 패스워드는 그 기기와 서버 외에는 생성하는 것이 계산적으로 어려워야 함
    - 공유된 대칭키 기반으로 생성
  - R2. [OTP 독립성] 이전에 사용된 패스워드로부터 현재 사용할 패스워드를 제3자가 계산하는 것이 계산적으로 어려워야 함
  - R3. [OTP 간결성] 제시된 값은 사용자가 쉽게 읽을 수 있어야 하며, 사용자가 화면에 쉽게 입력할 수 있어야 함
    - 값의 길이가 적당히 짧아야 함
    - 소프트웨어 방식은 간결성을 요구하지 않을 수 있음
  - R4. [OTP 경제성] 하드웨어적으로 구현이 경제적이어야 함

OTP의 종류 (1/2)

- 비동기화 vs. 동기화 방식
  - 서버와 클라이언트가 정보의 동기화가 필요한 방식과 필요 없는 방식
- 비동기화 방식: 계산할 때 사용할 랜덤 값을 서버가 제공하는 방식
  - 시도응답 방식: 5장에서 살펴본 난스 기법과 동일함
  - 독립적인 하드웨어 장치를 사용하기 힘듦
    - 장치와 클라이언트 기기 간 통신이 가능하지 않으면 사용자가 직접 시도 값을 장치에 입력해야 함
      - 번거로울 뿐만 아니라 기기가 키보드와 같은 입력 수단을 갖추고 있어야 함
  - 시도 값이 절대적으로 매번 달라야 함

- 2에서 암호문 전체를 보내면 OTP라 하기 힘듦
  ⇒ 일반 인증 프로토콜
- 2에서 대칭키 대신 전자서명 사용하면
  공개키 기반 OTP



OTP의 종류 (2/2)

- 동기화 방식: 클라이언트와 서버가 약속된 값을 이용하여 계산하는 방식
  - 시간동기화 방식
    - 특정 시간 간격(예: 1분마다)마다 패스워드를 자동 생성함
      - 간격 사이에 중복 검사 필요
    - 서버는 현재 시각을 기준으로 $\pm n$개의 패스워드를 생성하여 비교함
  - 사건동기화 방식
    - 사용자가 요청할 때마다 카운터를 이용하여 생성하는 방식
      - 서버는 클라이언트마다 별도의 카운터를 유지해야 함
      - 서버는 자신의 카운터 값을 기준에 따라 $+n$개의 패스워드를 생성하여 비교함. 일치하면 카운터를 동기화함
      - 사용자가 카운터를 너무 많이 증가하면 동기화가 힘들 수 있음
  - 조합방식: 시간과 사건 방식을 혼합하여 사용하는 방식
    - 시간 간격마다 새롭게 생성하며, 시간 간격 내에서는 카운터 값을 활용함

시간 동기화 vs. 사건 동기화

	시간 동기화 방식	사건 동기화 방식
생성	매 시간 간격마다 자동 생성 f(K, T)	필요할 때마다 요구에 의해 생성 (카운터) f(K, C)
동기화	시간에 의존함 - 시스템간 클럭동기화 필요	look-ahead 파라미터 s 사용 즉, 다음 s개를 생성하여 비교함
미래값	확보할 수 없음	가능
편리성	(문제점)입력도중에 변경 가능	시간 동기화가 필요 없음 (문제점) 각 기기마다 별도 카운터 필요
공격가능성	유효기간 내 재사용 가능	미래값 확보 가능 (유효기간 없음)

HOTP

- HOTP(HMAC based OTP): RFC 4226
- 사건 동기화 방식
- 클라이언트와 서버는 카운터 C와 공유키 K를 공유하고 있음
- HOTP 생성 알고리즘
  - $HS = \text{HMAC}.K(C)$
    - SHA-1 알고리즘 사용, 160 비트(20바이트): byte[0]…byte[19]
  - $Sbits = DT(HS)$ // 4 바이트 비트 문자열
    - 마지막 바이트의 하위 4비트 값이 offset이라 하면 $Sbits$ = byte[offset]||...||byte[offset+3]이 된다.
    - 예)

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
1F	86	98	69	0E	02	CA	16	61	85	50	EF	7F	19	DA	8E	94	5B	55	5A

  - $Snum = \text{StToNum}(Sbits)$
  - $D = Snum \bmod 10^{\text{Digit}}$

KDF(Key Derivation Function)

- 대칭키를 랜덤하게 생성하여 사용하는 경우도 있지만 패스워드나 키 동의 프로토콜을 수행한 후에 확보한 비밀값을 이용하여 대칭키를 생성하는 경우도 많음
  - 이때 사용하는 함수를 KDF라 함
  - 패스워드로부터 대칭키를 생성할 경우에는 패스워드의 추측 공격 때문에 salt와 반복횟수라는 기법을 사용함
    - 패스워드로부터 대칭키를 생성하는 함수를 PBKDF라 함
- Diffie-Hellman 키 동의 후 공유된 비밀 정보 g^{ab}를 대칭키로 사용하는 것이 아니라 이 값을 KDF에 입력하여 필요한 개수의 필요한 길이의 키를 계산함
  - KDF를 통해 여러 개의 키를 생성하더라도 각 키는 서로 독립적인 키임

HKDF

- HMAC 기반 KDF을 말하며, 보통 키 동의 프로토콜 수행을 통해 얻은 랜덤 비밀 정보로부터 대칭키를 생성할 때 사용함
  - 단계 1. $K = \mathrm{MAC.salt}(\mathrm{master})$
    - DH의 경우 $\mathrm{master} = g^{ab}$
  - 단계 2. (확장 단계): T_0는 0비트 문자열

$$\begin{aligned} T_1 &= \mathrm{MAC}.K(T_0||\mathrm{info}||0x01) \\ T_2 &= \mathrm{MAC}.K(T_1||\mathrm{info}||0x02) \\ &\vdots \qquad\qquad\qquad \vdots \\ T_N &= \mathrm{MAC}.K(T_{N-1}||\mathrm{info}||N) \end{aligned}$$

$$T = T_1||\,T_2||\ldots||\,T_N$$

    - info는 프로토콜에서 포함하고 싶은 정보
      예) 참여자 쌍방의 식별자
- T에서 필요한 만큼이 비트를 이용하여 필요한 개수의 대칭키를 생성함

PBKDF

- RFC 2898. PBKDF1: 해시 기반, PBKDF2: MAC 기반
- PBKDF1: c는 반복횟수 (속도를 느리게 하는 요소)
  - $T_1 = H(\pi||salt), T_2 = H(T_1), \ldots, T_c = H(T_{c-1})$
  - T_c에서 필요한 비트 길이만큼 사용
- PBKDF2
  - 단계 1. $T_i = F(\pi, salt, c, i)$, c는 반복횟수
    - $T_i = U_1 \oplus U_2 \oplus \ldots \oplus U_c$

$$\begin{aligned} U_1 &= \mathrm{MAC}.\pi(salt||i) \\ U_2 &= \mathrm{MAC}.\pi(U_1) \\ &\vdots \qquad\qquad \vdots \\ U_c &= \mathrm{MAC}.\pi(U_{c-1}) \end{aligned}$$

    - 2000년: $c = 1,000$
    - 2017년: $c = 10,000$
  - 단계 2. (확장 단계): $T = T_1||\,T_2||\ldots||\,T_N$
  - T에서 필요한 만큼이 비트를 이용하여 필요한 개수의 대칭키를 생성함

SHAKE

- 주어진 입력으로부터 필요한 길이의 랜덤 출력을 얻는 함수를 XOF(eXtendable-Output Function)라 함
- SHA-3(KECCAK)는 기존 SHA-1, SHA-2와 달리 스펀지 구조를 사용함
  - 스펀지 구조는 출력 길이에 제한이 없음
    - 표준 SHA-3는 정해진 크기의 출력만 사용하지만 SHAKE는 필요한 크기의 출력을 뽑아 사용함 (FIPS 202)
      - HKDF로 활용할 수 있음
- cSHAKE: customizable SHAKE
  - 사용할 입력 외에 필요한 길이와 추가 정보를 받아 해당 길이만큼 랜덤 비트 문자열을 출력해 줌
  - (FIPS 800-185)

예지력 증명

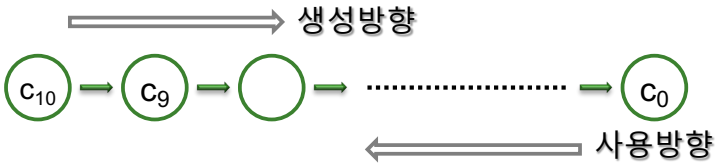
- 예지력 증명(Proof of clairvoyance)
  - 예) 축구시합 전에 결과를 미리 알고 있음을 결과를 보여주지 않고 증명하고 싶음
    - *H*("대한민국 3:브라질 0")
      - 해시함수의 어떤 특성을 활용하고 있는 것인가?
      - 이와 같은 방법으로 예지력 증명하는 것의 문제점은?
    - 2014년 브라질 월드컵 결승을 트위터에 예측한 경우가 있었음
      - 트위터에 게시가 증명 문제 관련 어떤 특성을 가지고 있나?

Lamport의 해시체인 기법

- 랜덤한 seed를 이용하여 연속적으로 해시값을 계산하여 생성함
- seed가 s이면 길이가 n인 해시체인을 생성하기 위해서는 다음을 차례로 계산함

$$c_n = H(s), c_{n-1} = H\big(H(s)\big), \dots, c_1 = H^n(s), c_0 = H^{n+1}(s)$$

  - 여기서 $c_{n-i+1} = H^i(s)$는 s를 i번 해시한 값을 말함
  - $c_0 = H^{n+1}(s)$는 이 해시체인의 루트 값이라 함
  - 해시함수의 일방향 특성 때문에 $H^i(s)$를 알고 있어도 $H^{i-1}(s)$는 계산할 수 없음
  - 하지만 $H^i(s)$를 알고 있으면 $H^{i-1}(s)$의 유효성은 한번의 해시연산을 통해 확인할 수 있음
- 값의 사용 순서는 생성한 순서와 반대임. 즉, c_1을 가장 먼저 사용함



Lamport의 일회용 패스워드

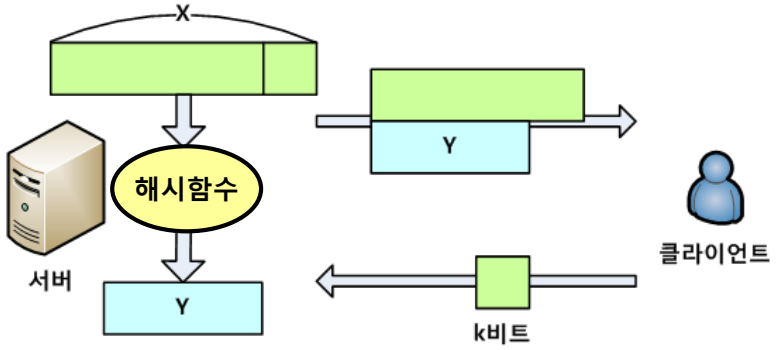
- 시스템 설정
  - 클라이언트는 길이가 n인 해시체인을 생성하여, 해시체인 루트를 서버에 등록함
    - 이 과정은 안전하게 이루어져야 함. 예를 들어 클라이언트는 루트 값을 전자서명하여 서버에 전달할 수 있음
  - 서버는 클라이언트 루트 값과 클라이언트가 마지막으로 사용한 체인값을 유지함
- 일회용 패스워드 프로토콜
  - 클라이언트는 c_i를 전달함. 이때 서버는 c_{i-1}값을 가지고 있으므로 $H(c_i) = c_{i-1}$인지 검사하여 클라이언트를 인증함
- 문제점
  - 해시체인의 길이에 의해 인증 횟수가 제한됨
  - 클라이언트는 해시체인의 모든 값을 유지하고 있지 않으면 매번 다시 많은 해시연산을 수행해야 함 (중간 값 유지)

S/KEY

- Bellcore 회사에서 개발한 것이며, 인터넷 표준(RFC 1760)임
  - 그 이후 몇 번 개선되어 현재는 RFC 2289에 정의되어 있음
- 초기 seed 값, 각 체인 값의 표현방식 등이 특수하지만 기본 생각은 Lamport의 해시체인 기법임

Hash Puzzle

- k비트 퍼즐



  - 해시함수의 일방향성 때문에 해시값으로부터 그것의 입력을 찾는 것은 계산적으로 힘듦
  - 하지만 입력의 일부(k비트)만 제외하면 일부에 대한 전수조사를 통해 입력을 찾을 수 있음
    - 답을 찾기 위한 연산 수가 예측됨

Hash Puzzle (1/2)

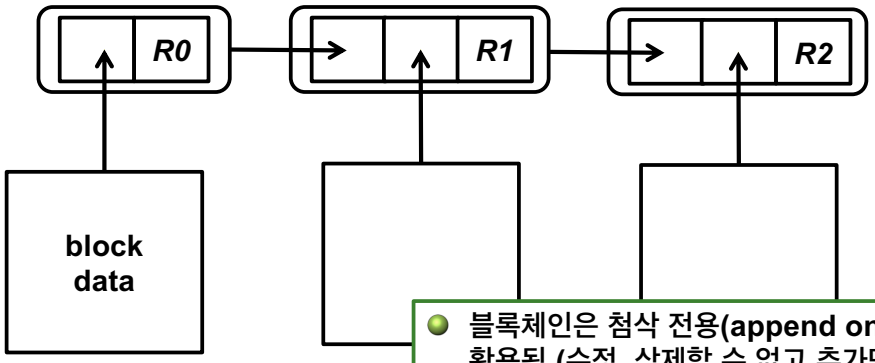
- 특정 수 이하 해시값 찾기
  - 비트코인에서 사용
  - 해시값의 최상위 k비트가 0이 되도록 입력 중 랜덤값을 바꾸어 가면서 계산함
    - 이 퍼즐을 해결하는데 소요되는 시간을 조절할 수 있음
    - 0이 되어야 하는 최상위 비트의 수에 비례하여 시간이 필요함
      - 첫 1비트가 0: 2번의 해시, 첫 2비트가 0: 4번의 해시, …, 첫 n비트가 0: 2^n번의 해시
        - 한 번의 해시에 필요한 시간을 알면, 첫 n비트가 0인 해시값을 얻기 위한 시간을 예측할 수 있음
    - 이 때문에 이를 작업 증명(proof-of-work)이라 함
  - 해시 퍼즐은 비대칭성 특징을 가지고 있음
    - 문제 해결하는데 일정 시간이 소요되지만 답은 한 번의 해시를 통해 확인할 수 있음

Hash Puzzle (2/2)

- 응용) SPAM 메일 방지
  - 해시 퍼즐의 해가 포함된 메일만 유효한 메일로 처리함
    - *H*(수신자, *H*(메일 내용), 난스)를 계산하여 주어진 조건이 충족하는 해시값과 난스를 메일과 함께 전달해야 함
  - 요구되는 작업 증명의 비용을 조절하여 하나의 메일을 보내는 비용은 저렴하지만 많은 수의 메일을 보내기 위한 비용은 스팸 메일의 전송을 포기할 정도가 되어야 함

블록 체인

- 비트코인에서 사용한 블록 체인은 블록 단위로 데이터를 기록하고 이전 블록의 해시값을 포함하여 작업 증명하는 형태 (자세한 설명은 Note 15)
  - 중간 블록에 있는 내용을 수정하고자 하면 그 블록을 포함하여 그 이후 모든 블록을 다시 계산해야 함
  - 작업증명, 응용에서 사용하는 규칙과 특성 때문에 블록체인에 저장된 값을 변경(수정 및 삭제)할 수 없음



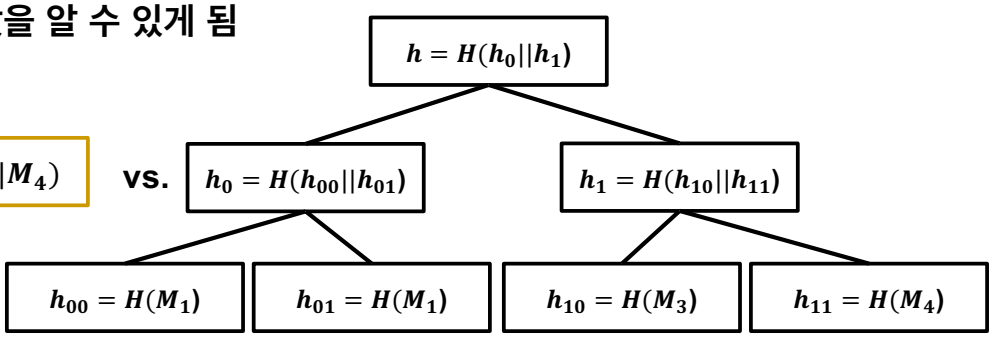
- 블록체인은 첨삭 전용(append only) 데이터베이스로 활용됨 (수정, 삭제할 수 없고 추가만 가능)

Merkle Tree

비트코인은 이전 슬라이드의 블록 데이터를 머클 트리로 표현함

- 다른 말로 해시 트리라고 함
- 단말노드에 있는 데이터가 이 트리에 포함되어 있다고 확장성 있게 증명할 수 있는 기법 (제시해야 하는 값이 트리(이진 트리) 높이에 비례하며, 이 값을 이용하여 만든 루트 값이 트리의 루트 값과 같은지 비교함)
- 이와 같은 트리를 만들어 결합서명할 수 있음. 루트값 h에 서명하면 4개 메시지 모두에 서명하는 것과 같음
- 머클 트리와 정반대로 계산하는 트리도 있으며, 이 경우에는 중간 값을 알면 모든 하위 노드의 값을 알 수 있게 됨

$h = H(M_1||M_2||M_3||M_4)$ vs.



h에 M_1이 포함되어 있는지 알기 위해 제시해야 하는 정보
방법 1. M_2, M_3, M_4
방법 2. h_{01}, h_1

Hash 기반 서명 (1/3)

- 최근 양자 내성 암호 기술로 고려하고 있는 대안 중 하나임
- Lamport 일회용 서명 (한 번만 서명할 수 있는 서명)
  - 1비트 서명
    - 공개키: $h(x)||h(y)$
    - 개인키: x, y
    - 0을 서명하고 싶으면 x를 공개, 1를 서명하고 싶으면 y 공개
  - n비트 서명
    - 공개키: $h(x_0)||h(y_0)||\ldots||h(x_{n-1})||h(y_{n-1})$
    - 개인키: $x_0, y_0, \ldots, x_{n-1}, y_{n-1},$
    - 0101을 서명하고 싶으면 x_0, y_1, x_2, y_3을 공개
- 여러 메시지를 서명하고 싶으면? 해시 트리?
- 다중 서명으로 확장은? 해시 트리?
- 공개키 인증은? 블록체인?

Hash 기반 서명 (2/3)

- Winternitz OTS
  - w bit($0 \sim w-1$) 서명
    - 공개키: $h^w(x)$
    - 개인키: x
    - 예) $w = 16, m = 9$의 서명: $h^9(x)$
    - 문제점: ?
  - $w = 256$, SHA-256 메시지에 서명 (32 바이트)
    - 메시지 $m = m_0 m_1 \ldots m_{31}$
    - 공개키: $h^{256}(x_0)h^{256}(x_1) \ldots h^{256}(x_{31})$
    - 개인키: $x_0, x_1, \ldots, x_{31}$
    - 서명값: $h^{m_0}(x_0)h^{m_1}(x_1)\ldots h^{m_{31}}(x_{31})$

- Lamport에 비해 공개키의 길이를 대폭 줄였음
- 여전히 일회용 서명

- $h^9(x)$가 공개되면 $h^8(x)$는 계산할 수 없지만 $h^{10}(x)$는 계산할 수 있음

Hash 기반 서명 (3/3)

- 여러 개의 공개키를 해시 트리의 단말로 설정하여 해시 트리를 구성하고, 이 트리의 루트 값을 공개키로 사용하면 하나의 공개키로 여러 메시지에 대해 서명할 수 있음 (공개키 크기 축소)
  - 사용자는 여전히 많은 수의 개인키를 유지해야 함
    - 개인키를 PRNG를 이용하여 생성하면 seed만 유지하면 됨
- 한 해시 트리에 유지할 수 있는 공개키의 수는 제한적임
  - 많을수록 트리의 높이는 증가함
  - 여러 개의 작은 해시 트리를 사용하고, 이 트리의 루트가 단말이 되는 해시 트리의 루트를 공개키로 사용할 수 있음 (하이퍼트리)
- SPHINCS+가 NIST 양자내성암호 대안 표준으로 3라운드 후보
  - 사용자가 직접 사용한 개인키 정보를 관리하지 않아도 됨
    - 일회용 서명이므로 한번 사용한 것을 절대 재사용하면 안 됨