Lecture 15: Tokens

CS 181S

Spring 2024

Review: Authentication of humans

- Something you are

  fingerprint, retinal scan, hand silhouette, a pulse
- Something you know

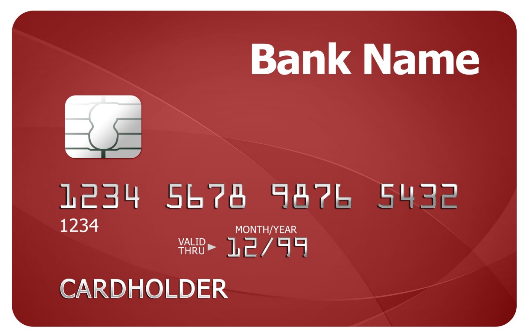
  password, passphrase, PIN, answers to security questions
- Something you have

  physical key, ticket, {ATM, prox, credit} card, token

Authentication Tokens

- What hardware authentication tokens and/or phone apps have you used in real life?

Authentication tokens



Threat Model: Eavesdropper

- Adversary can read read and replay messages
- Adversary cannot change messages during protocol execution (not full Dolev-Yao)

Fixed codes (Keyless Entry)

- Token stores a secret value id_T (e.g., key, id, password)
- Reader stores list of authorized ids
- To enter: **`T->B: id_T`**

- **Attack:** replay: thief sits in car nearby, records serial number, programs another token with same number, steals car
- **Attack:** brute force: serial numbers were 16 bits, devices could search through that space in under an hour for a single car (and in a whole parking lot, could unlock some car in under a minute)
- **Attack:** insider: serial numbers typically show up on many forms related to car, so mechanic, DMV, dealer's business office, etc. must be trusted

Fixed codes (RFIDs)

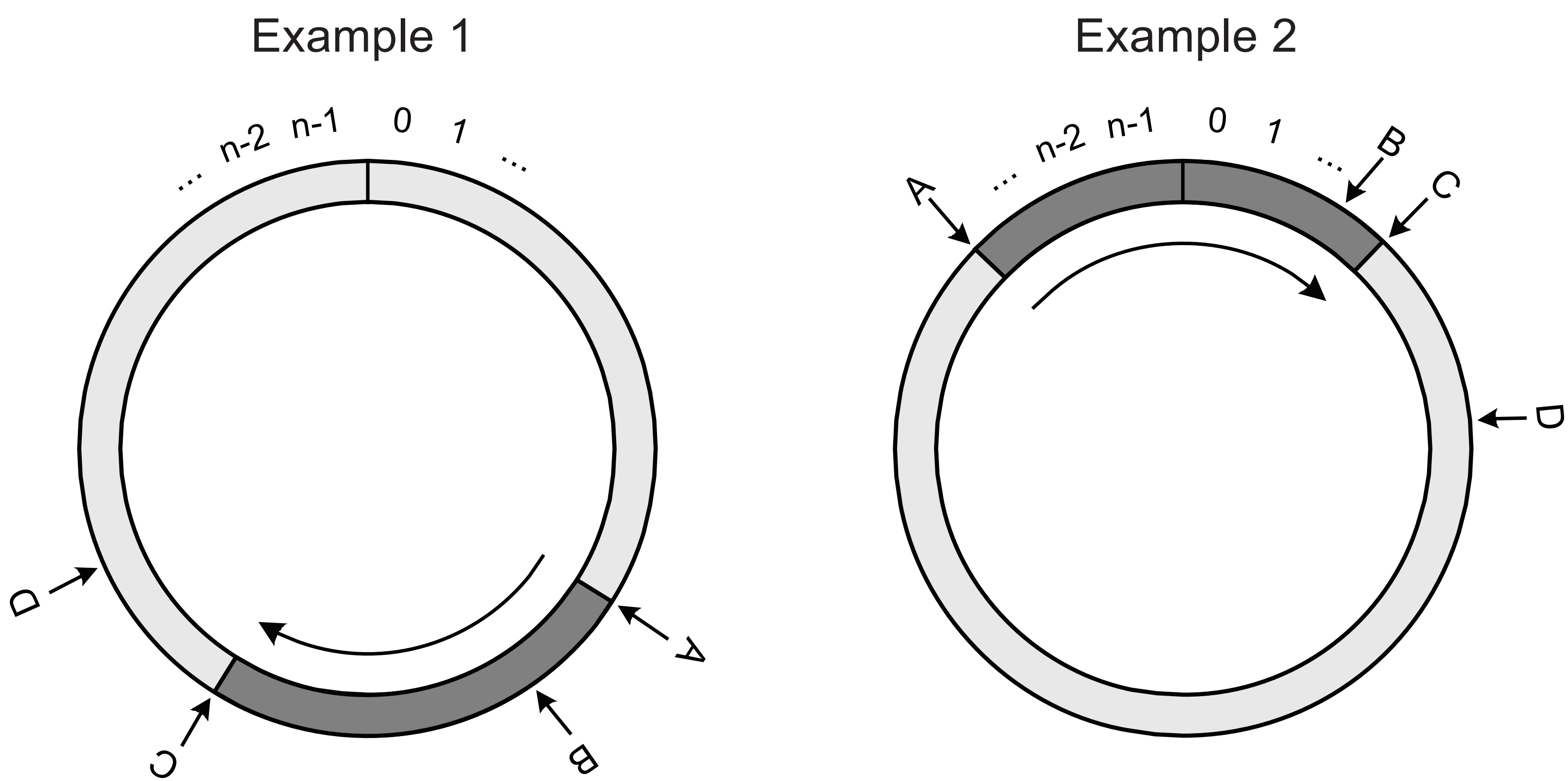
- Token stores a secret value id_T (e.g., key, id, password)
- Reader stores list of authorized ids
- To enter: **`T->B: id_T`**

- **Attack:** replay: thief sits nearby, records serial number, programs another token with same number, authenticates
- **Attack:** privacy: adversary tracks token usage across system and learns user attributes and/or behaviors

“Rolling” codes

- There is a root key, rk, for the barrier
- Token stores:
  - serial number T
  - shared key k, which is H(rk, T)
  - nonce N, which is a sequence counter
- Barrier stores:
  - serial numbers and current nonces for all authorized tokens
  - as well as root key rk
- To enter: **`T->B:  T, MAC(T, N; k)`**
  - And T increments N
  - So does B if MAC tag verifies
- **Problem:** desynchronization of nonce

Rolling window



A - Value from last valid message

B - Accepted counter values

C - End of window

D - Rejected counter values

Image source: [Atmel]

One-Time Passwords

- OTP may be deemed valid only once (the first time)
- Adversary cannot predict future OTPs, even with complete knowledge of what passwords have already been used

Unique challenge: MACs

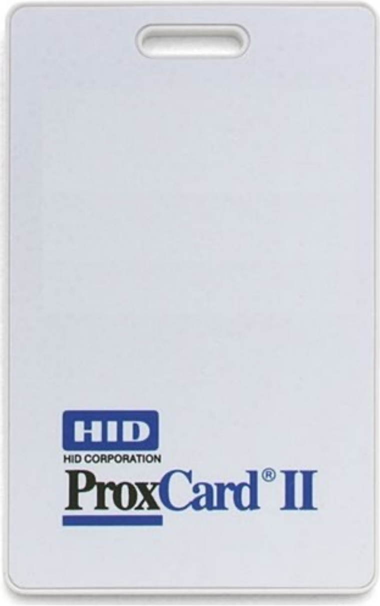
Assume: B stores a **MAC** key for each token,
i.e., a set of tuples (id_T, uid, k_T), and T stores k_T

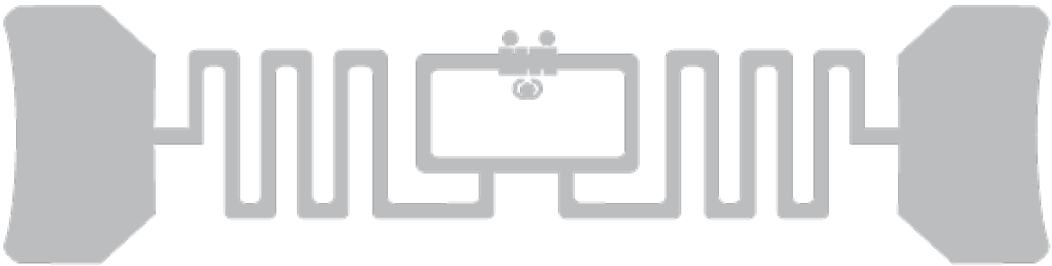
```
1. U->B: I want to authenticate with T
2. B: invent unique nonce N
3. B->T: N
4. T: t=MAC(N; k_T)
5. T->B: id_T, t
6. B: lookup (uid, kT) for id_T;
      U is authenticated as uid if t=MAC(N; k_T)
```

Non-problem: key distribution: already have to physically distribute tokens

Problem: key storage at B: what if key is stolen?

EPC Gen2v2 RFID Cards





Exercise 2: Digital Signatures

Assume: B stores a **MAC** key for each token and T stores k_T

Assume: B stores a verification key for each token and T stores signing key k_T

```
1. U->B: U,T
2. B: invent nonce N
3. B->T: N
4. T: t=MAC(N; k_T)
5. T->B: id_T, t
6. B: U is auth as uid
if t=MAC(N; k_T)
```

U2F

Remote Authentication

- (Usually) No communication from server to token
- Usability considerations render challenge-response impractical

Hypothetical protocol

Assume: S stores a set of tuples (id_T, uid, kT, pin), and T stores kT

```
1. U->L: I want to authenticate as uid to S
2. L and S: establish secure channel
3. L->U: Enter PIN and code on my keyboard
4. T->U: code = MAC(time@T, id_T; kT)
5. U->L: pin, code
6. L: compute h = H(pin, code)
7. L->S: uid, h
8. S: lookup (pin, id_T, kT) for uid;
      id_Hu is authenticated
        if h=H(pin, MAC(time@S, id_T; kT))
```

Engineering challenge: clock synchronization

Exercise 3: Clock Synchronization

- Assume that timestamps have a granularity of 1 second
- Assume that T and S last synchronized their clocks 24 hours ago (at noon the previous day)
- Assume that the network latency is 1-10 seconds
- Assume that the clock drift between the two clocks is at most .01 seconds per second

- If S receives a message at noon, what is the maximum and minimum timestamp it should accept?

SecurID

- Token: displays code that changes every minute
  - LCD display
  - Internal clock (1 minute granularity)
  - No input channel
  - Can compute hashes, MACs
  - Stores a secret

- Ideas used:
  - replace random value with current time
  - use L to input PIN
  - server checks $\pm$10 minutes to allow for clock drift

Hash chains

- Let $H^i(x)$ be i iterations of H applied to x
  - $H^0(x) = x$
  - $H^1(x) = H(x)$
  - $H^2(x) = H(H(x))$

  …
  - $H^{i+1}(x) = H(H^i(x))$

- **Hash chain:** $H^1(x), H^2(x), H^3(x), \ldots, H^n(x)$

OTPs from hash chains

- Given a randomly chosen, large, secret seed s...
- **Bad idea:** generate a sequence of OTPs as a hash chain: $H^1(s)$, $H^2(s)$, ..., $H^n(s)$
  - Suppose untrusted public machine learns $H^i(s)$
  - From then on can compute next OTP $H^{i+1}(s)$ by applying H, because hashes are easy to compute in forward direction
  - But hashes are hard to invert...
- **Good idea [Lamport 1981]:** generate a sequence of OTPs as a reverse hash chain: $H^n(s)$, ..., $H^1(s)$
  - Suppose untrusted public machine learns $H^i(s)$
  - Next password is $H^{i-1}(s)$
  - Computing that is hard!

Exercise: Reverse Hash Chains

- How could we use a reverse Hash Chain to authenticate users with tokens?

Solution 1

Assume: S stores a set of tuples (uid, n_u, s_u)

```
1. U->L->S: uid
2. S: lookup (n_u, s_u) for uid;
      let n = n_u;
      let otp = H^n(s_u);
      decrement stored n_u
3. S->L->U: n
4. U: p = H^n(s_u)
5. U->L->S: p
6. S: uid is authenticated if p = otp
```

Problem: S has to compute a lot of hashes if authentication is frequent

Solution 2

- S stores **`last`**: last successful OTP for id_Hu, where **`last`** = $H^{n+1}(s)$
- S receives **`next`**: next attempted OTP, where if all is well **`next`** = $H^n(s)$
- S checks its correctness with a single hash:

  H(**`next`**) = $H(H^n(s)) = H^{n+1}(s)$ = **`last`**
- And if correct S updates last successful OTP: **`last := next`**

Next problem: what if Hu and S don't agree on what password should be used next? i.e., become *desynchronized*

- network drops a message
- attacker does some online guessing (impersonating Hu) or spoofing (impersonating S)

Solution 3

- Hu and S independently store index of last used password from their own perspective, call them m_Hu and m_S
  - Neither is willing to reuse old passwords (i.e., higher indexes)
  - But both are willing to skip ahead to newer passwords (i.e., lower indexes)
- To authenticate:
  - S requests index m_S
  - Hu computes min(m_S, m_Hu), sends that along with OTP for it
  - S and Hu adjust their stored index

Next problem: running out of passwords: have to bother sysadmin periodically

Solution 4

- Compute OTP as H^n(pass,salt)
- Whenever Hu wants to generate new set of OTPs:
  - find a local machine Hu trusts (could be offline, phone, ...)
  - request new salt from S
  - enter pass
  - generate as many new OTPs as Hu likes by running hash forward
  - let S know how many were generated and what the last one was

S/KEY

[RFC 1760]:

- Instantiation of that protocol for particular hash algorithms and sizes
- But same idea works for newer hashes and larger sizes