CS 5430

Tokens

Prof. Clarkson
Spring 2016

Review: Authentication of humans

Categories:

- Something you know

  password, passphrase, PIN, answers to security questions

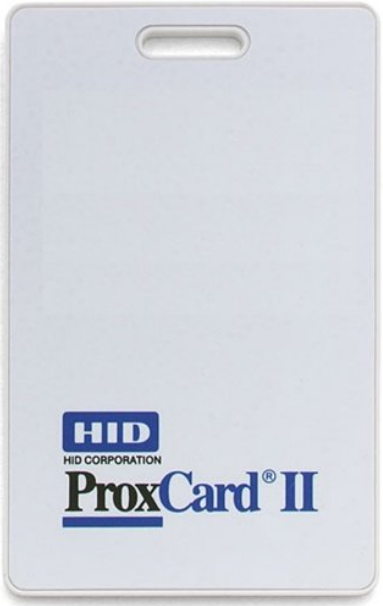
- Something you have

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

- Something you are

  fingerprint, retinal scan, hand silhouette, a pulse

Authentication tokens



Humans vs. machines

- At enrollment, human is issued a token
  - Ranges from dumb (a physical key, a piece of paper) to a smart machine (a cryptographic processor)
  - Token becomes attribute of human's identity
- Authentication of human reduces to authentication of token
  - So we're halfway to authentication of machines

Engineering goals

Convenience of token matters...

- Recall criteria:
  - **Usability:** memoryless, scalable for users, nothing to carry, physically effortless, easy to learn, efficient, infrequent errors, easy recovery from loss
  - **Deployability:** accessible, low cost, server compatible, browser compatible, mature, non-proprietary
- What tokens usually achieve: easy to carry, maintenance-free, low cost

Authentication with tokens

- **Goal:** authenticate human Hu to local system L using token T

- **Threat model:** eavesdropper
  - may read and replay messages
  - cannot change messages during protocol execution
  - not full Dolev-Yao adversary
  - motivation: wired keyboards, short-range radios (e.g., RFID)

- **Enrollment:** associate identifier id_T with identifier id_Hu

Challenge-response

Assume: L stores a fixed challenge and response for each token, i.e., a set of tuples (id_T, id_Hu, c_T, r_T), and T stores r_T

```
1. Hu->T: I want to authenticate to L
2. T->L: id_T
3. L: look up (id_Hu, c_T, r_T) for id_T
4. L->T: c_T
5. T->L: r
6. L: id_Hu is authenticated if r=r_T
```

Note: human never declares its identity

Vulnerability: replay

Case study 1: Keyless entry

- Into car or garage
- Activated typically with some physical action (button press, handle pull)
- Provide entry past some barrier (gate, door)

Keyless entry: fixed codes

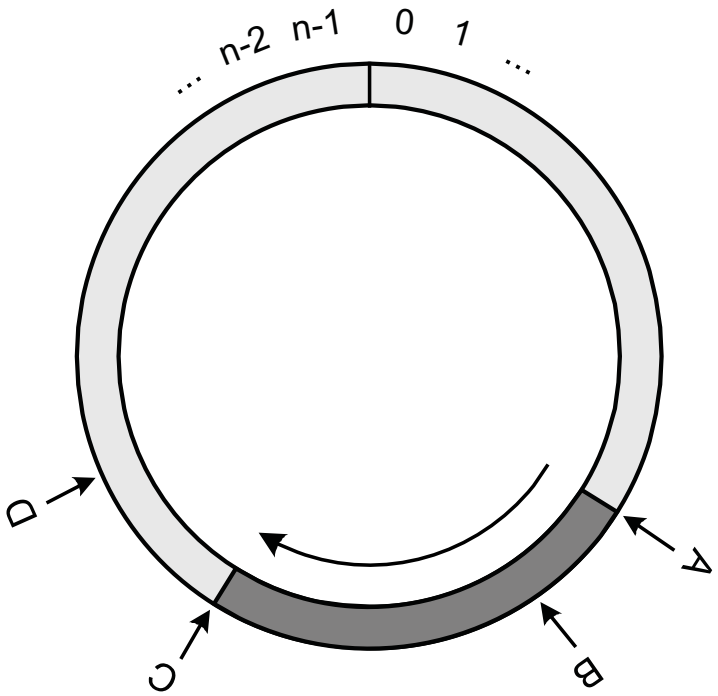
- Token stores its serial number, call it T
- Barrier stores all serial numbers for authorized tokens
- To enter: `T->B: T`
- **Attack:** replay: thief sits in car nearby, records serial number, programs another token with same number, steals car
- **Attack:** brute force: serial numbers were 16 bits, 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

Keyless entry: "rolling" codes

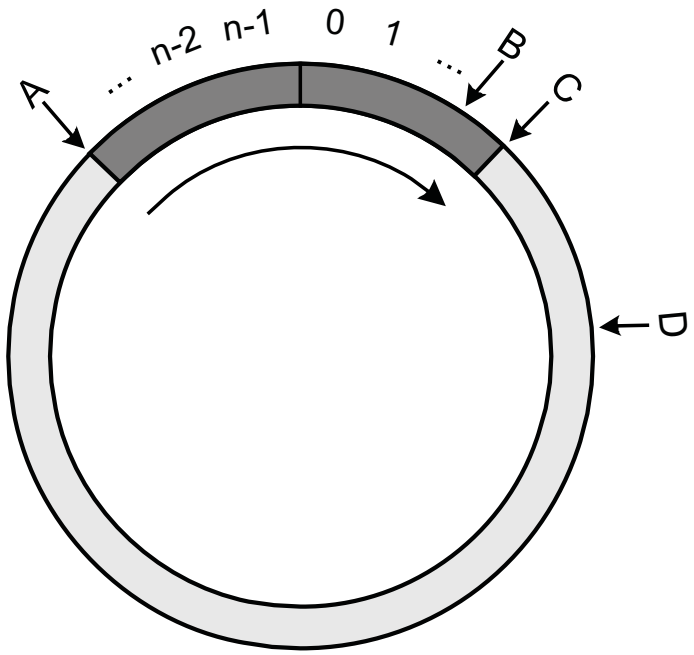
- Token stores:
  - serial number T
  - nonce N, which is a sequence counter
  - shared key k, which is H(mk, T)
- Barrier stores:
  - all those values for all authorized tokens
  - as well as master key mk
- To enter: **`T->B:  T, MAC(T, N; k)`**
  - And T increments N
  - So does B if MAC tag verifies
- **Problem:** desynchronization of nonce
- **Solution:** accept rolling window of nonces

Rolling window

Example 1



Example 2



A - Value from last valid message

B - Accepted counter values

C - End of window

D - Rejected counter values

Image source: Atmel

Digital signatures

Assume: L stores a verification key for each token, i.e., a set of tuples (id_T, id_Hu, K_T), and T stores signing key k_T

```
1. Hu->L: I want to authenticate with T
2. L: invent unique nonce N_L
3. L->T: N_L
4. T: s=Sign(N_L; k_T)
5. T->L: id_T, s
6. L: lookup (id_Hu, K_T) for id_T;
      id_Hu is authenticated if Ver(N_L; s; K_T)
```

Note: human might implement channel between T and L

Problem: cost? performance? power? patents? asymmetric crypto sometimes considered too expensive to use on tokens

MACs

Assume: L stores a MAC key for each token, i.e., a set of tuples (id_T, id_Hu, kT), and T stores kT

```
1. Hu->L: I want to authenticate with T
2. L: invent unique nonce N_L
3. L->T: N_L
4. T: t=MAC(N_L; kT)
5. T->L: id_T, t
6. L: lookup (id_Hu, kT) for id_T;
      id_Hu is authenticated if t=MAC(N_L; kT)
```

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

Problem: key storage at L: needs to store many keys, can't hash and salt them because plaintext is needed; could use secure co-processors

Theft

- With protocols so far, a thief can impersonate the human
- **Expanded threat model:** eavesdropper and thief
- **Countermeasure:** two-factor authentication

Two-factor with PIN

Assume: L also stores a PIN for each token, i.e., a set of tuples (id_T, id_Hu, kT, pin), and T stores kT

```
1. Hu->L: I want to authenticate with T
2. L: invent unique nonce N_L
3. L->T: N_L
4. T->Hu: Enter PIN on my keyboard
5. Hu->T: pin
6. T: compute t=MAC(N_L, pin; kT)
7. T->L: id_T, t
8. L: lookup (id_Hu, pin, kT) for id_T;
      id_Hu is authenticated
        if t=MAC(N_L, pin); kT)
```

Protect the PIN

From offline guessing:

- Can salt and iterate hash, as with password; T and L need to store the salt
- Can instead store PIN on T, which authenticates Hu with PIN then authenticates to L with key

From online guessing:

- Wherever PIN is stored needs to rate limit guessing
- Practical problem if PIN stored on T: children who get ahold of token

Remote authentication

- With protocols so far, authenticate only locally
- **Expanded goal:** authenticate human Hu to remote system S using local system L and token T
- **Expanded threat model:**
  - on **`T<->L`** channel: eavesdropper, theft
  - on **`L<->S`** channel: Dolev-Yao
  - **`L`** is trusted by **`T`** and **`S`**
- **Countermeasure:** secure channel

Remote authentication

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

```
1. Hu->L: I want to authenticate with T to S
2. L and S: establish secure channel
3. S: invent unique nonce N_S
4. S->L->T: N_S
5. T->Hu: Enter PIN on my keyboard
6. Hu->T: pin
7. T: compute t=MAC(N_S, pin; kT)
8. T->L->S: id_T, t
9. S: lookup (id_Hu, pin, kT) for id_T;
      id_Hu authenticated if t=MAC(N_S, pin; kT)
```

Note: L is just an intermediary but could hijack session

Case study 2: SecureID

- Token: displays code that changes every minute
  - LCD display
  - Internal clock
  - No input channel
  - Can compute hashes, MACs
  - Stores a secret
- Ideas used:
  - replace nonce with current time
  - use L to input PIN

Hypothetical protocol

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

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

Engineering challenge: clock synchronization [Schneider 5.2]

Case study 3: S/KEY

```
...
~~50: MEND VOTE MALE HIRE BEAU LAY~~
~~49: PUG LYRA CANT JUDY BOAR AVON~~
48: LOAM OILY FISH CHAD BRIG NOV
47: RUE CLOG LEAK FRAU CURD SAM
46: COY LUG DORA NECK OILY HEAL
45: SUN GENE LOU HARD ELY HOG
44: GET CANE SOY NOR MATE DUEL
43: LUST TOUT NOV HAN BACH FADE
42: HOLM GIN MOLL JAY EARN BUFF
40: KEEN ABUT GALA ASIA DAM SINK
...
```

One-time passwords

- A one-time password (OTP) is valid only once, the first time used
  - Similar to changing your password with every use
  - Rules out replays entirely
  - But man-in-the-middle could still succeed
- **Use case:** login at untrusted public machine where you fear keylogger
- **Use case:** recovery
  - "main password" is lost
  - phone is lost during two-factor authentication (e.g., Google backup codes)
- **Older use case:** send cleartext password over network

One-time passwords

- Strawman implementation:
  - Each user chooses many OTPs and stores them at server (in hashed salted form)
  - At use, server checks whether provided password hashes to any of stored passwords, then deletes stored password if so to prevent re-use
- Expensive!
  - User has to pick and remember lots of strong passwords
  - Server has to store passwords and search for right password
- **Solution:** algorithmic generation of OTPs
  - SecureID can be seen as an instantiation: each code is a OTP valid for only 60 sec.
  - Iterated hashing is another possibility...

Hash chains

- Let $H^i(x)$ be i iterations of H applied to x
  - $H^0(x) = x$
  - $H^{i+1}(x) = H(H^i(x))$
- 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!

Protocol (almost)

Assume: S stores a set of tuples (id_Hu, n_Hu, s_Hu)

```
1. Hu->L->S: id_Hu
2. S: lookup (n_Hu, s_Hu) for id_Hu;
      let n = n_Hu;
      let otp = H^n(s_Hu);
      decrement stored n_Hu
3. S->L->Hu: n
4. Hu: p = H^n(s_Hu)
5. Hu->L->S: p
6. S: id_Hu is authenticated if p = otp
```

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

Solution to S's hash burden

- S stores **`last`**: last successful OTP for id_Hu, where **`last`** = $H^{n+1}(s)$
- When S receives **`next`**: next attempted OTP, where **`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)
  - note: so far, Hu is happy to re-use old passwords

Solution to desynchronization

- 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 responds with otp for whichever is smaller, m_S or m_Hu, as well sending S that index value (so S knows what is being received)
  - Both adjust their stored index

Next problem: humans can't compute an iterated hash

Solution to human computation

Pre-printed passwords:

$50: H^{50}(s)$

$49: H^{49}(s)$

...

$1: H^{1}(s)$

Next problem: humans aren't good at typing long bit strings

Solution: represent bit strings as short words

i.e., divide hash output into chunks, use each chunk as index into dictionary, where each word in dictionary is fairly short

Pre-printed passwords

```
...
~~50: MEND VOTE MALE HIRE BEAU LAY~~
~~49: PUG LYRA CANT JUDY BOAR AVON~~
48: LOAM OILY FISH CHAD BRIG NOV
47: RUE CLOG LEAK FRAU CURD SAM
46: COY LUG DORA NECK OILY HEAL
...
```

Next problem: running out of passwords: have to bother sysadmin to get new printed passwords periodically; might run out while traveling

Solution: replace system-chosen seed with user-chosen password plus salt

Salted passwords as seed

- 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

Final protocol

Assume: S stores a set of tuples (id_Hu, n_S, salt, last), Hu stores (pass, n_Hu)

```
1. Hu->L->S: id_Hu
2. S: lookup n_S for id_Hu
3. S->L->Hu: n_S
4. Hu: n = min(n_Hu, n_S) - 1;
       if n<=0 then abort
       else let p = H^n(pass, salt);
            n_Hu := n
5. Hu->L->S: n, p
6. S: if n<n_S and H^(n_S-n)(p)=last
      then n_S := n;
           last := p;
           id_Hu is authenticated
```

S/KEY

[RFC 1760]:

- Instantiation of that protocol for particular hash algorithms and sizes
- But same idea works for newer hashes and larger sizes
- Many software calculators for passwords available

Upcoming events

- [today] A4 due
- [next week] Spring Break

It is the part of men to fear and tremble, when the most mighty gods by tokens send such dreadful heralds to astonish us. – William Shakespeare