

---

# CS 5430

---

## Tokens

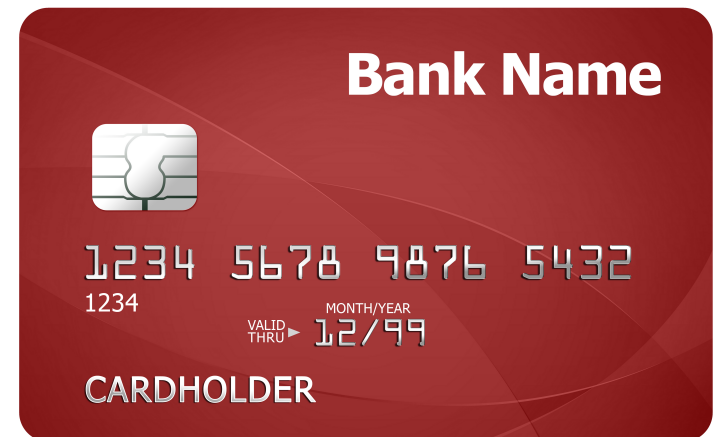
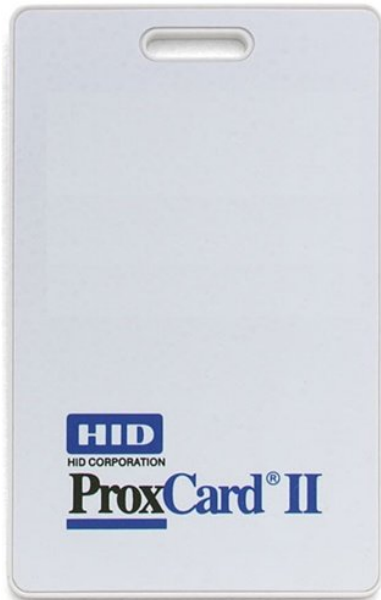
Prof. Clarkson  
Spring 2017

# 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  $H_u$  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: short-range radios (e.g., RFID)
- **Enrollment:** associate identifier  $id_T$  with identifier  $id_{Hu}$

# **CASE STUDY 1: KEYLESS ENTRY**

# Keyless entry



- Into car or garage
- **Activated** typically with some physical action (button press, handle pull)
- Provide entry past some **barrier** (gate, door)
- Brutally simple *one-pass* protocols: only one message sent



# Keyless entry: fixed codes

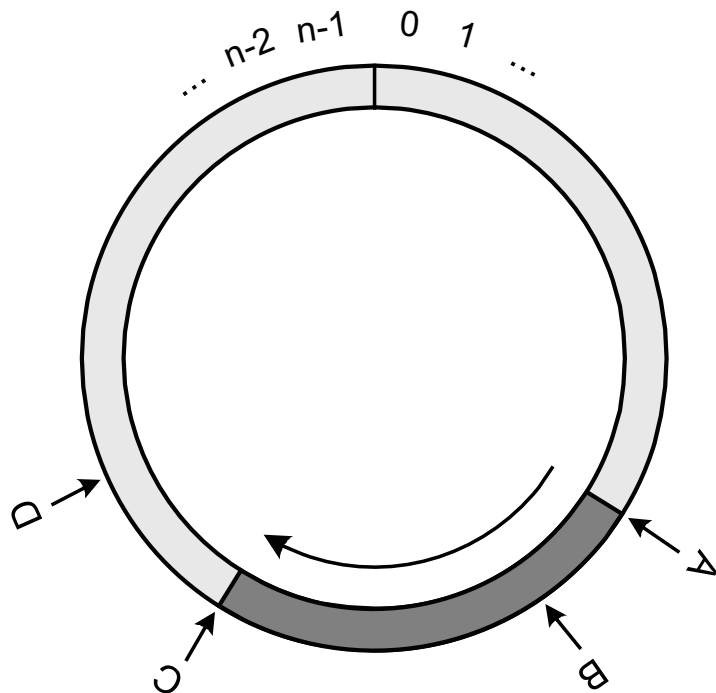
- Token stores its serial number, call it T
- Barrier stores all serial numbers for authorized tokens
- To enter: **T**→**B**: **T**
- **Attack:** replay: thief sits in car nearby, records serial number, programs another token with same number, steals car
- **Attack:** brute force: serial numbers were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

# “Rolling” codes

- There is a master key,  $mk$ , for the barrier
- 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:  **$T \rightarrow B: T, MAC(T, N; k)$** 
  - And  $T$  increments  $N$
  - So does  $B$  if MAC tag verifies
- **Problem:** desynchronization of nonce
- **Partial solution:** accept “rolling window” of nonces

# Rolling window

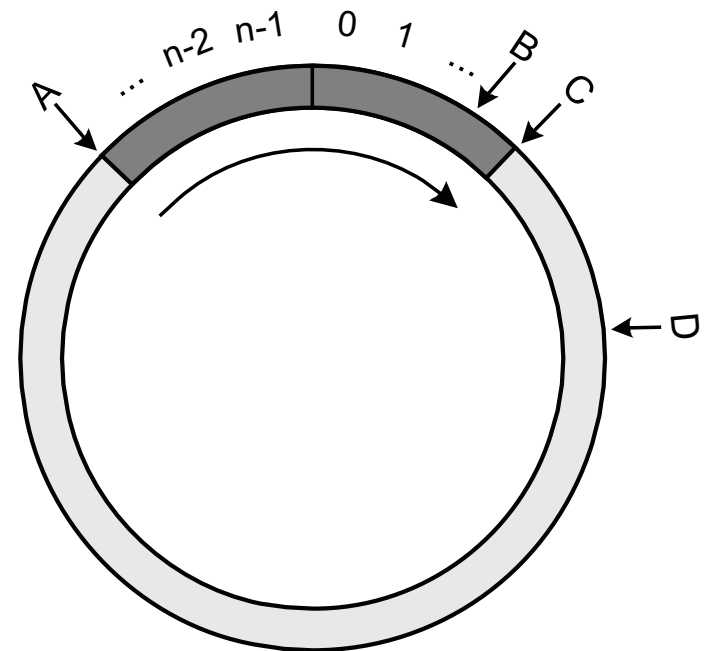
Example 1



A - Value from last valid message

B - Accepted counter values

Example 2



C - End of window

D - Rejected counter values

**CHALLENGE-RESPONSE**

# Fixed challenge

**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

# Unique challenge: Dig Sig

**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)

Quasi-problems: cost? performance? power? patents?

# Unique challenge: 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 we won't solve:** key storage at L: what if key database is stolen?

**THEFT**



# Theft

- **Expanded threat model:** eavesdropper, or 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  $t = \text{MAC}(N_L, \text{pin}; kT)$
7. T->L: id\_T, t
8. L: lookup (id\_Hu, pin, kT) for id\_T;  
id\_Hu is authenticated  
if  $t = \text{MAC}(N_L, \text{pin}; kT)$

# 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  $H_u$  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**

# Remote authentication

- **Expanded goal:** authenticate human  $H_u$  to remote system  $S$  using local system  $L$  and token  $T$
- **Expanded threat model:**
  - on  $T \leftrightarrow L$  channel: eavesdropper, theft
  - on  $L \leftrightarrow S$  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  $t = \text{MAC}(N_S, \text{pin}; k_T)$
8. T->L->S: id\_T, t
9. S: lookup (id\_Hu, pin, kT) for id\_T;  
id\_Hu authenticated if  $t = \text{MAC}(N_S, \text{pin}; k_T)$

Note: L is just an intermediary but could hijack session

## **CASE STUDY 2: SECUREID**

# 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**

# S/KEY: Paper “token”

...

~~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

41: 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: Pre-registered OTPs
- **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:  $H^1(x), H^2(x), H^3(x), \dots, H^n(x)$

# OTPs from hash chains

- Given a randomly chosen, large, secret seed  $s$ ...
- **Bad idea:** generate a sequence of OTPs as a hash chain:  
 $H^1(s), H^2(s), \dots, H^n(s)$ 
  - 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:  $H^n(s), \dots, H^1(s)$ 
  - Suppose untrusted public machine learns  $H^i(s)$
  - Next password is  $H^{i-1}(s)$
  - Computing that is hard!

# Leslie Lamport



b. 1941

Turing Award Winner 2013

*For fundamental contributions to the theory and practice of distributed and concurrent systems, notably the invention of concepts such as causality and logical clocks, safety and liveness, replicated state machines, and sequential consistency.*



# 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** =  $H^{n+1}(s)$
- S receives **next**: next attempted OTP, where if all is well **next** =  $H^n(s)$
- S checks its correctness with a single hash:  
 $H(\mathbf{next}) = H(H^n(s)) = H^{n+1}(s) = \mathbf{last}$
- 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 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 computes  $\min(m_S, m_{Hu})$ , sends that along with OTP for it
  - S and Hu 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  $H^n(\text{pass}, \text{salt})$
- 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  $n \leq 0$  then abort  
    else let  $p = H^n(\text{pass}, \text{salt})$ ; // lookup on paper  
         $n_{Hu} := n$  // cross off on paper
5. Hu->L->S: n, p
6. S: if  $n < n_S$  and  $H^{n_S - n}(p) = \text{last}$   
    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

- [next Friday] A4 due

*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*