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
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
Authentication tokens
Threat Model: Eavesdropper

- Adversary can 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: \( T \rightarrow M: \text{id}_T \)

- **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->M: 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

- **Countermeasure:** one-time passwords
“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: \text{T} \rightarrow \text{B}: \quad \text{T, MAC}(\text{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
There are numerous algorithms available to use for generating the MAC, but for various reasons we have chosen the Advanced Encryption Standard (AES) algorithm, which is a symmetric block cipher. The AES algorithm supports key sizes of 128, 192, and 256 bits. Its use as a MAC generator is discussed further under the heading "Section 2.1.1 Rolling Windows" on page 5.

2.1.1 Rolling Windows

The concept of simply ignoring messages having old sequential numbers leaves one problem: What if the counter value overflows and wraps back to 0? This section describes a solution.

Handling the sequential counter is best described by two examples, given in Figure 2-4. The first example shows a situation where the last received valid message had a counter value A. As there is always the possibility that the transmitter has been activated a number of times outside the receiver's range, the receiver must accept values up to some limit, labeled C in the figure. The simple approach of accepting all values larger than the last received value won't work, as is apparent in the second example where point A is close to the upper end of the counter value range. The dark segment from point A to C shows the window of acceptance for counter values. Point B is an example of a value that would be accepted while point D is a value that would be rejected. When a value is accepted, the window starting point moves to that point.

This scheme ensures that old messages are never accepted unless the head of the rolling window has reached the old counter values. By choosing a large enough counter span and limiting the window size itself, this scheme effectively prevents replay attacks with old messages.

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
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...
Unique challenge: MACs

Assume: M 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->M: I want to authenticate with T
2. M: invent unique nonce N
3. M->T: N
4. T: t=MAC(N; k_T)
5. T->M: id_T, t
6. M: 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 L: what if key database is stolen?
EPC Gen2v2 RFID Cards
Unique challenge: Dig Sig

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

1. U->M: I want to authenticate with T
2. M: invent unique nonce N
3. M->T: N
4. T: s=Sign(N; k_T)
5. T->M: id_T, s
6. M: lookup (uid, K_T) for id_T;
   U is authenticated as uid if Ver(N; s; K_T)

U2F
Two-factor with PIN

Assume: M also stores a PIN for each token, i.e., a set of tuples (id_T, uid, k_T, pin), and T stores k_T

1. U→M: I want to authenticate with T
2. M: invent unique nonce N
3. M→T: N
4. T→U: Enter PIN on my keyboard
5. U→T: pin
6. T: compute \( t = \text{MAC}(N, \text{pin}; k_T) \)
7. T→M: id_T, t
8. M: lookup (uid, \text{pin}, k_T) for id_T;
   U is authenticated as uid if \( t = \text{MAC}(N, \text{pin}; k_T) \)
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
Estimating clock value

- Each device D has a clock C_D
  - model C_D as an non-decreasing, positive function of real time
- Server needs to estimate C_T(t_code): the time the token's clock displayed when the code was computed
- Clocks run at different rates and thus drift apart
  - we assume drift rate is bounded by a constant ρ
  - If C_T(t) = C_S(t) then |C_T(t') - C_S(t')| <= 2ρ(t'-t)
- Messages take time d_min – d_max to deliver
- Clock estimation:
  - C_T(t_prev) <= C_T(t_code)
  - C_T(t_code) ∈ [C_S(t_curr) + Δ_prev + d_min - 2ρ(t_curr - t_prev), C_S(t_curr) + Δ_prev + d_max + 2ρ(t_curr - t_prev)]
  - To authenticate: check all possible times in range
  - On successful authentication, update t_prev
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 nonce with current time
  - use L to input PIN
  - server checks ±10 minutes to allow for clock drift
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

...
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: $H^1(x)$, $H^2(x)$, $H^3(x)$, ..., $H^n(x)$
OTPs from hash chains

- Given a randomly chosen, large, secret seed $s$...

**Bad idea:** generate a sequence of OTPs as a hash chain: $H^1(s), H^2(s), \ldots, H^n(s)$
  - 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: $H^n(s), \ldots, H^1(s)$
  - 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 (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 to S's hash burden

- S stores \texttt{last}: last successful OTP for id\_Hu, where \texttt{last} = H^{n+1}(s)
- S receives \texttt{next}: next attempted OTP, where if all is well \texttt{next} = H^n(s)
- S checks its correctness with a single hash:
  \[ H(\texttt{next}) = H(H^n(s)) = H^{n+1}(s) = \texttt{last} \]
- And if correct S updates last successful OTP: \texttt{last} := \texttt{next}

Next problem: what if Hu and S don't agree on what password should be used next? i.e., become \textit{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: running out of passwords: have to bother sysadmin to get new printed passwords periodically; might run out while traveling
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 (uid, n_S, salt, last), Hu stores (pass, n_u)

1. U→L→S: uid
2. S: lookup n_S for uid
3. S→L→U: n_S
4. U: n = min(n_u, n_S) - 1;
   if n<=0 then abort
   else let p = H^n(pass, salt); // lookup on paper
   n_u := n // cross off on paper
5. U→L→S: n, p
6. S: if n<n_S and H^{n-n}_S(p)=last
   then n_S := n;
   last := p;
   uid 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
Solution to human computation

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

... 
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
...