Identity Authentication

Authentication

- Basics
- Passwords
- Challenge-Response
- Biometrics
- Location
- Multiple Methods

Basics

- Authentication: binding of identity to subject
  - Identity is that of external entity (my identity, Van, *etc*.)
  - Subject is computer entity (process, *etc*.)
- Note:
  - message authentication is a different topic and already mentioned in the applications of hash functions

Establishing Identity

- One or more of the following
  - What entity knows (*eg.* password)
  - What entity has (*eg.* Identity card, smart card)
  - What entity is (*eg.* fingerprints, retinal characteristics)

Authentication System

- Authentication process
  - Obtaining the authentication information from an entity
  - Analyzing the data
  - Determining if it is associated with that entity
- We need a formal definition, rather abstract view, of an AS
- A 5-tuple (*A*, *C*, *F*, *L*, *S*)
  - *A – a set of Authentication Information:* information that proves identity
  - *C – a set of Complementary Information:* information stored in system and used to validate authentication information
  - *F: a set of* complementation functions; $f : A \rightarrow C$
    - *To compute complementary information from authentication information*
  - *L:* authentication functions that prove identity; $l: A \times C \rightarrow \{true, false\}$
  - *S:* functions enabling entity to create, alter information in *A* or *C*

Example

- Password system, with passwords stored online in clear text
  - *A* set of strings making up passwords
  - *C* = *A*
  - *F* singleton set of identity function { *I* }
  - *L* single equality test function { *eq* }
  - *S* function to set/change password

Passwords

- Sequence of characters
  - Examples: 10 digits, a string of letters, *etc*.
  - Generated randomly, by user, by computer with user input
- Sequence of words
  - Examples: pass-phrases
- Algorithms
  - Examples: challenge-response, one-time passwords

Storage

- Store as cleartext
  - If password file compromised, *all* passwords revealed
- Encipher file
  - Need to have decipherment, encipherment keys in memory
  - Reduces to previous problem → need something else
- Solution: Instead store one-way hash of password
  - Got the file, attacker must still guess passwords or invert the hash values

Example: Unix

- By definition, a 5-tuple (*A*, *C*, *F*, *L*, *S*)
  - *A – a set:* information that proves identity
    - *A* = { strings of 8 chars or less }
  - *C – a set:* information stored on computer and used to validate authentication information
    - *C* = {hash values of password}
  - *F: a set of* complementation functions; $f : A \to C$
    - *F* = { versions of modified DES }
  - *L:* authentication functions that prove identity
    - *L* = { *login*, *su*, … }
  - *S:* functions enabling entity to create, alter information in *A* or *C*
    - *S* = { *passwd*, *nispasswd*, *passwd+*, … }

Example: Unix

- By definition, a 5-tuple (*A*, *C*, *F*, *L*, *S*)
  - *A – a set:* information that proves identity
    - *A* = { strings of 8 chars or less }
  - *C – a set:* information stored on computer and used to validate authentication information
    - *C* = {hash values of password}
  - *F: a set of* complementation functions; $f : A \to C$
    - *F* = { versions of modified DES }
  - *L:* authentication functions that prove identity
    - *L* = { *login*, *su*, … }
  - *S:* functions enabling entity to create, alter information in *A* or *C*
    - *S* = { *passwd*, *nispasswd*, *passwd*+, … }

Attacking passwords

- Goal: find $a \in A$ such that:
  - For some $f \in F$, $f(a) = c \in C$
  - c is associated with entity
- Two ways to determine whether *a* meets these requirements:
  - By trying computing f(a) for a set of a values until succeed
  - By trying calling l(a) until succeed (l(a) returns true)

Preventing Attacks

- How to prevent this:
  - Hide one of *a*, *f*, or *c*
    - Prevents obvious attack from above
    - Example: UNIX/Linux shadow password files
      - Makes the files containing complementary information readable only by root
  - Block access to all $l \in L$ or result of $l(a)$
    - Prevents attacker from knowing if guess succeeded
    - Example: preventing *any* logins to an account from a network
      - Prevents knowing results of l (or accessing l)

Dictionary Attacks

- Trial-and-error from a list of potential passwords
  - *Off-line*: know f and c's, and repeatedly try different guesses $g \in A$ until the list is done or passwords guessed
    - Examples: *crack*, *john-the-ripper*
  - *On-line*: have access to functions in L and try guesses g until some $l(g)$ succeeds
    - Examples: trying to log in by guessing a password

Success probability over a time period

Anderson's formula:

- P probability of guessing a password in specified period of time
- G number of guesses tested in 1 time unit
- T number of time units
- N number of possible passwords ($|A|$)
- Then $P \geq TG/N$

Example

- Goal
  - Passwords drawn from a 96-char alphabet
  - Can test 10^4 guesses per second
  - Probability of a success to be 0.5 over a 365 day period
  - What is minimum password length?
- Solution
  - $N \geq TG/P = (365 \times 24 \times 60 \times 60) \times 10^4 / 0.5 = 6.31 \times 10^{11}$
  - Choose s such that $\Sigma^{s}_{j=0}\, 96^j \geq N$
  - So $s \geq 6$, meaning passwords must be at least 6 chars long

Example

- Goal
  - Passwords drawn from a 96-char alphabet
  - Can test 10^4 guesses per second
  - Probability of a success to be 0.5 over a 365 day period
  - The password length is at most s
  - What is minimum value of s?
- Solution
  - $N \geq TG/P = (365 \times 24 \times 60 \times 60) \times 10^4 / 0.5 = 6.31 \times 10^{11}$
  - Choose s such that $\Sigma^{s}_{j=0} 96^{j} \geq N$
  - So $s \geq 6$long

Example 2

- Goal
  - Passwords drawn from a 100-char alphabet
  - Can test 10^5 guesses per second
  - The password length is at least 5 and at most *s*
  - Probability of a success to be at most 0.1 over a 365 day period
  - What is minimum value of *s*?

On password selection

- Random selection
  - Any password from *A* equally likely to be selected
- Pronounceable passwords
- User selection of passwords

Pronounceable Passwords

- Generate phonemes randomly
  - Phoneme is unit of sound, eg. *cv*, *vc*, *cvc*, *vcv*
  - Examples: helgoret, juttelon are; przbqxdfl, zxrptglfn are not
- Problem: too few
- Solution: key crunching
  - Run long key through hash function and convert to printable sequence
  - Use this sequence as password

User Selection

- Problem: people pick easy to guess passwords
  - Based on account names, user names, computer names, place names
  - Dictionary words (also reversed, odd capitalizations, control characters, “elite-speak”, conjugations or declensions, swear words, Torah/Bible/Koran/… words)
  - Too short, digits only, letters only
  - License plates, acronyms, social security numbers
  - Personal characteristics or foibles (pet names, nicknames, job characteristics, *etc*.

Picking Good Passwords

- "LIMm*2^Ap"
  - Names of members of 2 families
- "OoHeO/FSK"
  - Second letter of each word of length 4 or more in third line of third verse of Star-Spangled Banner, followed by "/", followed by author's initials
- What's good here may be bad there
  - "DMC/MHmh" bad at Dartmouth ("Dartmouth Medical Center/Mary Hitchcock memorial hospital"), ok here
- Why are these now bad passwords? ☹

Proactive Password Checking

- Analyze proposed password for “goodness”
  - Always invoked
  - Can detect, reject bad passwords for an appropriate definition of “bad”
  - Discriminate on per-user, per-site basis
  - Needs to do pattern matching on words
  - Needs to execute subprograms and use results
    - Spell checker, for example
  - Easy to set up and integrate into password selection system

Salting

- Goal: slow dictionary attacks
- Method: perturb hash function so that:
  - Parameter controls *which* hash function is used
  - Parameter differs for each password
  - So given *n* password hashes, and therefore *n* salts, need to hash guess *n*

Examples

- Vanilla UNIX method
  - Use DES to encipher 0 message with password as key; iterate 25 times
  - Perturb E table in DES in one of 4096 ways
    - 12 bit salt flips entries 1–11 with entries 25–36
- Alternate methods
  - Use salt as first part of input to hash function

Unix actually is …

- UNIX system standard hash function
  - Hashes password into 11 char string using one of 4096 hash functions
- As authentication system:
  - A = { strings of 8 chars or less }
  - C = { 2 char hash id || 11 char hash }
  - F = { 4096 versions of modified DES }
  - L = { *login*, *su*, … }
  - S = { *passwd*, *nispasswd*, *passwd+*, … }

Guessing Through *L*

- Cannot prevent these
  - Otherwise, legitimate users cannot log in
- Make them slow
  - Backoff
  - Disconnection
  - Disabling
    - Be very careful with administrative accounts!
  - Jailing
    - Allow in, but restrict activities

Password Aging

- Force users to change passwords after some time has expired
  - How do you force users not to re-use passwords?
    - Record previous passwords
    - Block changes for a period of time
  - Give users time to think of good passwords
    - Don’t force them to change before they can log in
    - Warn them of expiration days in advance

Challenge-Response

- User, system share a secret function *f* (in practice, *f* is a known function with unknown parameters, such as a cryptographic key)

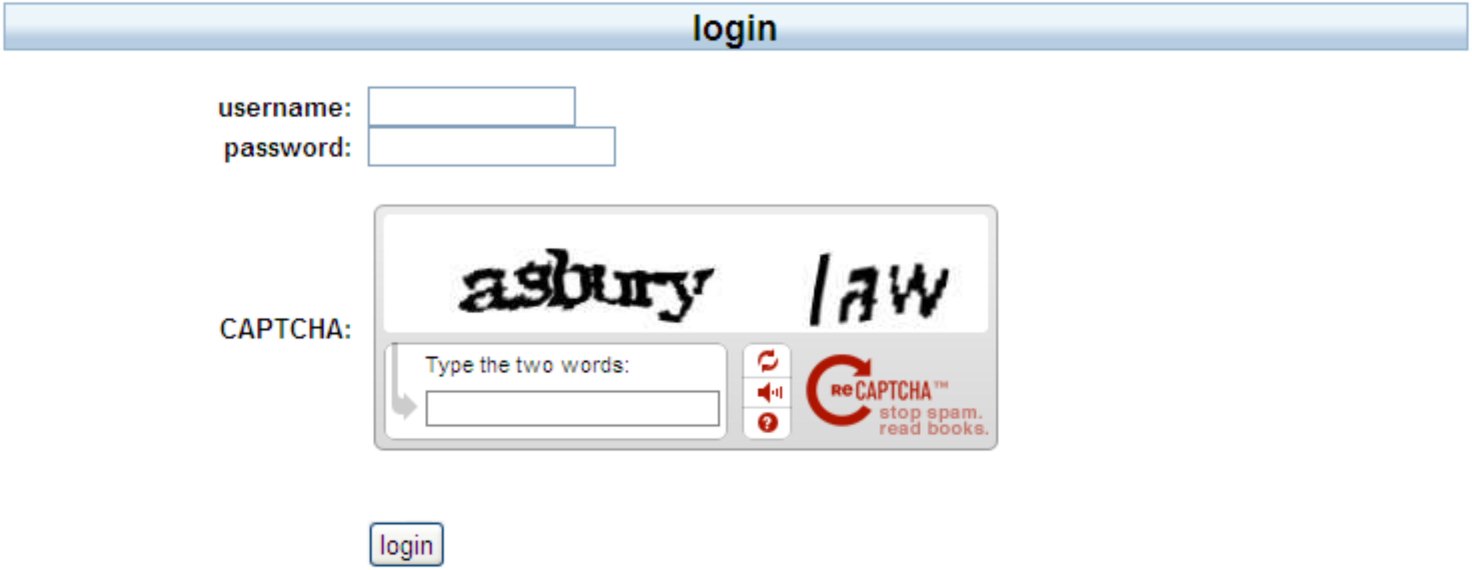
user $\xrightarrow{\textit{request to authenticate}}$ *system*

user $\xleftarrow[\textit{(the challenge)}]{\textit{random message } r}$ *system*

user $\xrightarrow[\textit{(the response)}]{f(r)}$ *system*

Pass Algorithms

- Challenge-response with the function *f* itself a secret
  - Challenge is a random string of characters
  - Response is some function of that string
  - Usually used in conjunction with fixed, reusable password



One-Time Passwords

- Password that can be used exactly *once*
  - After use, it is immediately invalidated
- Challenge-response mechanism
  - Challenge is number of authentications; response is password for that particular number
- Problems
  - Synchronization of user, system
  - Generation of good random passwords
  - Password distribution problem

S/Key

- One-time password scheme based on idea of Lamport
- h one-way hash function (MD5 or SHA-1, for example)
- User chooses initial seed k
- System calculates:

$$h(k) = k_1,\ h(k_1) = k_2,\ \ldots,\ h(k_{n-1}) = k_n$$

- Passwords are reverse order:

$$p_1 = k_n,\ p_2 = k_{n-1},\ \ldots,\ p_{n-1} = k_2,\ p_n = k_1$$

S/Key Protocol

System stores maximum number of authentications n, number of next authentication i, last correctly supplied password p_{i-1}.

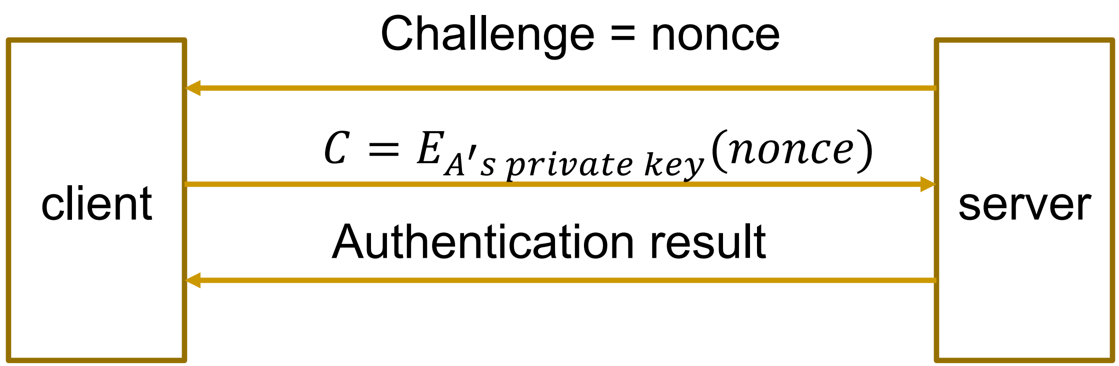
$$\textit{user} \xrightarrow{\{\ \textit{name}\ \}} \textit{system}$$

$$\textit{user} \xleftarrow{\{\ i\ \}} \textit{system}$$

$$\textit{user} \xrightarrow{\{\ p_i\ \}} \textit{system}$$

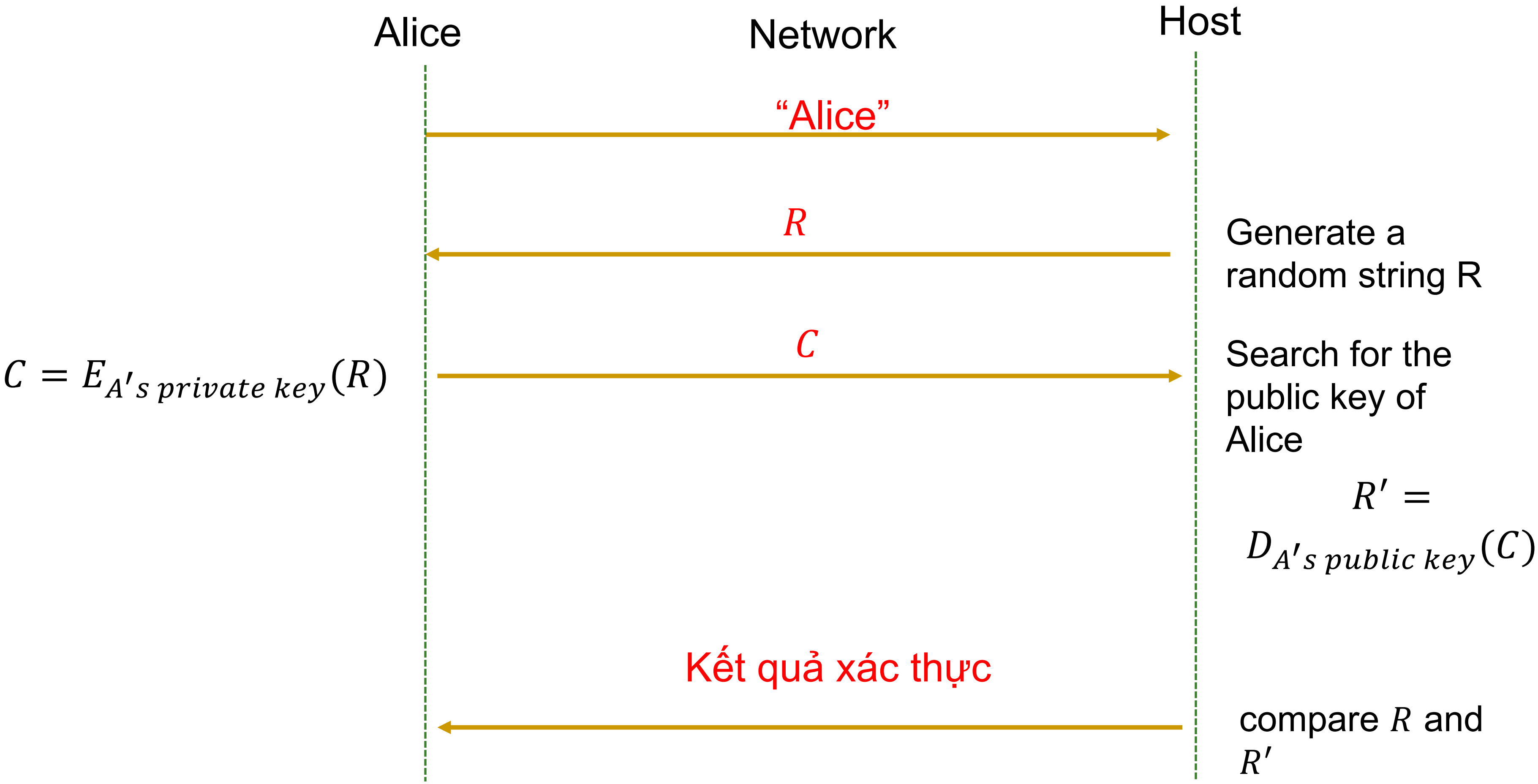
System computes $h(p_i) = h(k_{n-i+1}) = k_{n-i} = p_{i-1}$. If match with what is stored, system replaces p_{i-1} with p_i and increments i.

Authentication by using asymmetric key cryptography



- Server needs to know the public key of client

Authentication by using asymmetric key cryptography



Projects

- Survey and implement Oauth
- Survey and implement OpenID Connect