University of Rochester CSC290B Introduction to Computer Security

Authentication

January 27, 2009

Previously, on CSC290B: Access control

- Given subjects *S*, objects *O*
  - What subject is allowed
  - what type of access
  - to what object
- Enough for a realistic system?
  - How do we know a request really is from *s*?
  - And the object accessed really is *o*?

Today’s Overview

- Authentication and Identity
  - Basics
  - Passwords
    - Storage
    - Selection
    - Breaking them
  - Other methods
  - Multiple methods

Basics

- Authentication: binding of identity to subject
  - Identity is that of external entity (my identity, Bukys, *etc*.)
  - Subject is computer entity (process, *etc*.)

Establishing Identity

- One or more of the following factors:
  - What entity knows (*eg*. password)
  - What entity has (*eg*. badge, smart card)
  - What entity is (*eg*. fingerprints, retinal characteristics)
  - Where entity is (*eg*. In front of a particular terminal)
- Requiring more than one is called *N*-factor authentication (usually 2-factor authentication)

Authentication System: Formal Definition

- A: set of *authentication information*
  - used by entities
- C: set of *complementary information*
  - used by system to validate authentication information
- F: Set of *complementation functions*
  - $f : A \to C$
  - Generate appropriate $c \in C$ given $a \in A$
- L: set of *authentication functions*
  - $l: A \times C \to \{$ **true, false** $\}$
  - verify identity
- S: set of *selection functions*
  - $s : ? \to A$
  - Generate/alter A and C

Authentication System

- $(A, C, F, L, S, (f, l))$ such that
  - $\forall f \in F, \forall\ l \in L, \exists\ (f, l)$ in the system such that
    $\forall\ a \in A, \forall\ c \neq f(a) \in C$:
    - $l(a, f(a)) \rightarrow$ **true**
    - $l(a, c) \rightarrow$ **false**
    - with high probability
      - Does not have to be perfect (hash-based systems generally are not), but finding a that violate these conditions should be infeasible.

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

Example

- Password system, with passwords stored on line 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

Authentication Systems: Hashed Passwords

- Information known to subject
- Complementation Function
  - Identity – requires that c be protected
  - One-way hash – function such that
    - $f(a)$ easy to compute
    - $f^{-1}(c)$ difficult to compute
  - Better ideas?

Example

- 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+*, … }

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: *L0phtCrack*, *john-the-ripper*
  - *Off-line and pre-computed (high storage but rapid attack on a particular encrypted file)*
    - *Example: RainbowCrack*
  - *On-line*: have access to functions in L and try guesses g until some $l(g)$ succeeds
    - Example: trying to log in by guessing a password

Dictionary Attacks in this notation

- Dictionary attack: Trial and error guessing
  - Type 1: attacker knows A, f, c
  - Type 2: attacker knows A, l
  - Difficulty based on $|A|$, time g to compute f or l
    - Probability P of breaking in time T: $P \geq Tg/|A|$
    - Problem: often smaller in practice than in theory

Anatomy of Attacking

- 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:
  - Direct approach: as above
  - Indirect approach: as $l(a)$ succeeds iff $f(a) = c \in C$ for some c associated with an entity, compute $l(a)$

Preventing Attacks

- How to prevent this:
  - Hide one of a, f, or c
    - Prevents obvious attack from above
    - Example: UNIX/Linux shadow password files
      - Hides c's
  - 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)

Using Time

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

Salting

- Goals:
  - Slow pre-computed dictionary attacks
  - Increase storage required for pre-computed hashed dictionaries
  - Cause stored hashes to differ even if cleartext passwords are identical
- 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

Approaches: 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

- "LlMm*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

Example: OPUS

- Goal: check passwords against large dictionaries quickly (with a Bloom Filter):
  - Run each word of dictionary through *k* different hash functions $h_1, \ldots, h_k$ producing values less than *n*
  - Set bits $h_1, \ldots, h_k$ in OPUS dictionary
  - To check new proposed word, generate bit vector and see if *all* corresponding bits set
    - If so, word is in one of the dictionaries to some degree of probability
    - If not, it is not in the dictionaries

Example: *passwd+*

- Provides little language to describe proactive checking
  - test length(“$p”) < 6
    - If password under 6 characters, reject it
  - test infile(“/usr/dict/words”, “$p”)
    - If password in file /usr/dict/words, reject it
  - test !inprog(“spell”, “$p”, “$p”)
    - If password not in the output from program spell, given the password as input, reject it (because it’s a properly spelled word)

Guessing Through *L* (online attacks)

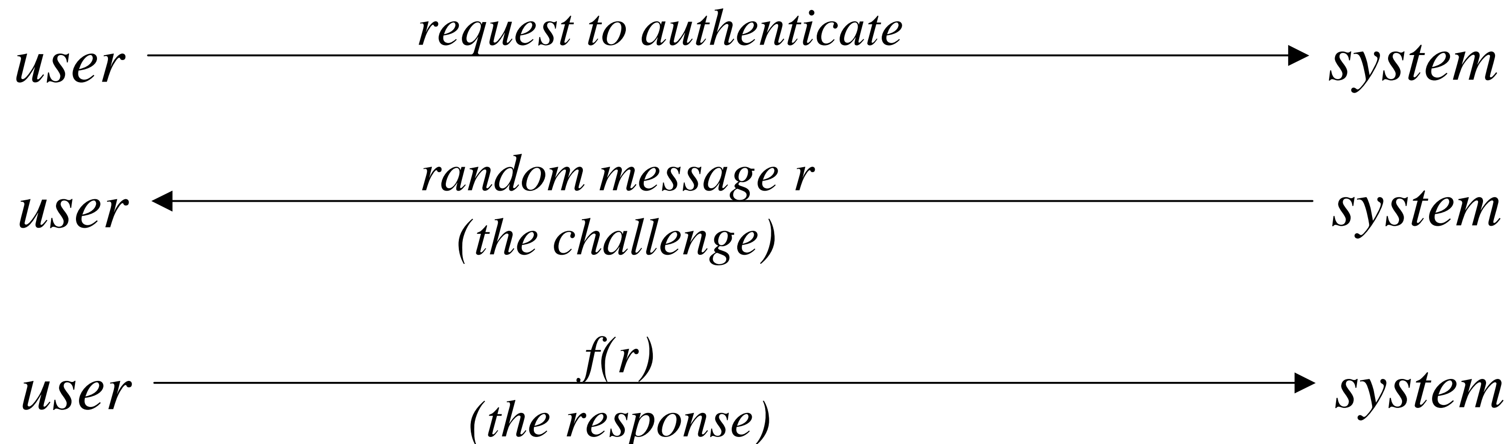
- Cannot prevent these
  - Otherwise, legitimate users cannot log in
- Make them slow
  - Backoff
  - Disconnection
  - Disabling
    - Be very careful with administrative accounts!
  - Jailing
    - Allow in, but restrict activities

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)



Pass Algorithms

- Challenge-response with the function *f* itself a secret
  - Example:
    - Challenge is a random string of characters such as “abcdefg”, “ageksido”
    - Response is some function of that string such as “bdf”, “gkip”
  - Can alter algorithm based on ancillary information
    - Network connection is as above, dial-up might require “aceg”, “aesd”
  - 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:

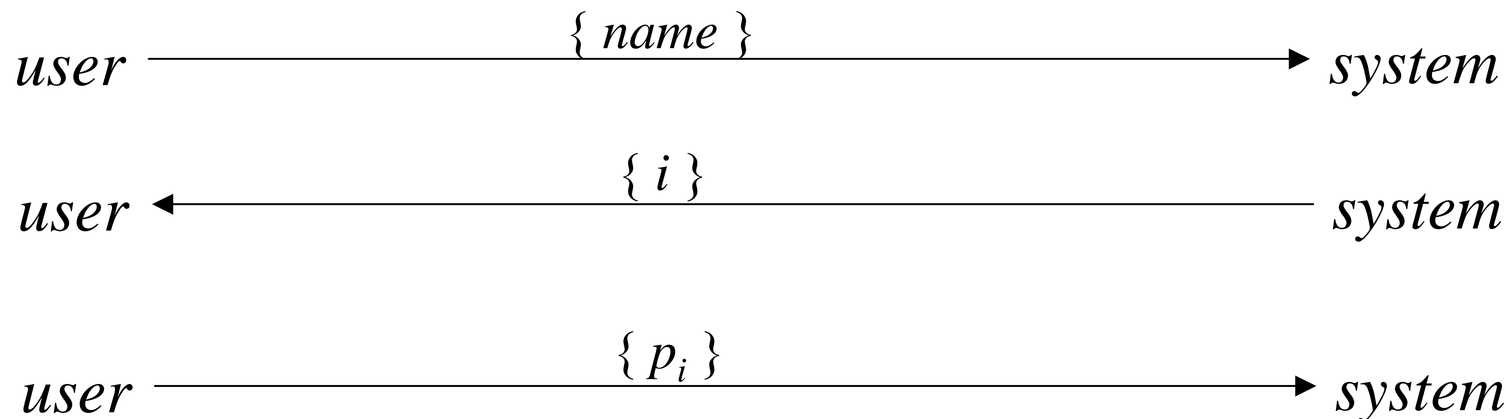
$$h(k) = k_1,\ h(k_1) = k_2,\ \ldots,\ h(k_{n-1}) = k_n$$

- 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}.



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.

Hardware Support

- Token-based
  - Used to compute response to challenge
    - May encipher or hash challenge
    - May require PIN from user
- Temporally-based
  - Every minute (or so) different number shown
    - Computer knows what number to expect when
  - User enters number and fixed password
- Sequence-based

C-R and Dictionary Attacks

- Same as for fixed passwords
  - Attacker knows challenge *r* and response *f*(*r*); if *f* encryption function, can try different keys
    - May only need to know *form* of response; attacker can tell if guess correct by looking to see if deciphered object is of right form
    - Example: Kerberos Version 4 used DES, but keys had 20 bits of randomness; Purdue attackers guessed keys quickly because deciphered tickets had a fixed set of bits in some locations

Acknowledgements

- Substantial portions of these slides are courtesy of Matt Bishop and Chris Clifton and used with permission