L7: Authentication

Hui Chen, Ph.D.
Computer Science
Dept. of Math & Computer Science
Virginia State University
Petersburg, VA 23806

Acknowledgement

- Revised based on slides are from the author of the Textbook (Matt Bishop, Introduction to Computer Security, Addison-Wesley Professional, October, 2004, ISBN-13: 978-0-321-24774-5.)

Overview

- Basics
- Passwords
  - Storage
  - Selection
  - Breaking them
- Other methods
- Multiple methods

Authentication

- Binding of identity to subject
  - Identity is that of external entity (e.g, your username)
  - Subject is computer entity (e.g., process)

Establishing Identity

- One or more of the following
  - What entity knows (*e.g.,* password)
  - What entity has (*e.g.,* badge, smart card)
  - What entity is (*e.g.,* fingerprints, retinal characteristics)
  - Where entity is (*e.g.*, in front of a particular terminal)

Authentication System

- Consisting of 5 components (*A*, *C*, *F*, *L*, *S*)
  - *A:* authentication information that proves identity
  - *C:* complementary information stored on computer and used to validate authentication information
  - *F:* complementation function
    - $f : A \to C$
  - *L:* authentication functions that prove identity
    - $l: A \times C \to \{\text{true}, \text{false}\}$
  - *S:* selection functions enabling entity to create or alter information in *A* or *C*

Example: Cleartext password

- Password system, with passwords stored on line in cleartext
  - *A:* the set of strings making up passwords
  - $C = A$
  - *F:* the singleton set of identity function { *I* }
  - *L:* the single equality test function { *eq* }
  - *S:* the function to set/change password

Example: Encrypted password

- Password system, with passwords stored on line in an encrypted form
  - *A:* the set of strings making up passwords
  - *C:* the set of strings making up encrypted passwords
  - *F:* the set of encryption or hash functions that computes the encrypted form of a password
  - *L:* the set of test functions that takes a password, finds its encrypted form, and check if it is equal to a stored one
  - *S:* the function to set/change password

Passwords

- Sequence of characters
  - Examples: 10 digits, a string of letters, *etc*.
  - Generating passwords
    1. randomly
    2. by user
    3. by computer with user input
- Sequence of words
  - Examples: pass-phrases

Storage

- Store as cleartext
  - If password file compromised, *all* passwords revealed
- Store in encipher file
  - Need to have decipherment, encipherment keys in memory
  - Reduces to previous problem
- Store one-way hash of password
  - If file read, attacker must still guess passwords or invert the hash

Example: Linux/UNIX

- Linux/UNIX system *standard* hash function
  - Hashes password into a character string using a hash function
- 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+*, … }
- Latest Linux/Unit have enhancements

Example: Linux/UNIX

- Read manual pages

  man 1 passwd

  man 5 passwd

  man 3 crypt

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:
  - Dictionary attack type 1: direct approach, as above, compute $f(a)$
  - Dictionary attack type 2: 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*)

Exercise L7-1: Linux Shadow Passwords

- In Linux, read manual page

  man 5 shadow

- Examine two files
  - /etc/passwd
  - /etc/shadow
  - Answer the questions in the context of the description in slide 13.
    - Who can read from and write to /etc/passwd?
    - Who can read from and write to /etc/shadow?

Exercise L7-2: Linux Login Failure

- In Linux, log out
- When log back in, enter a wrong password intentionally
- Describe what you observe in the context of the description in slide 13.

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

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: Determine Password Length

- 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
  - $N \geq TG/P = (365\times24\times60\times60)\times10^4/0.5 = 6.31\times10^{11}$
  - Choose s such that $\sum_{j=0}^{s} 96^j \geq N \geq 6.31 \times 10^{11}$
  - So $s \geq 6$, meaning passwords must be at least 6 chars long

Assumptions in Anderson’s Formula

- Time required to test a password is a constant
  - This is reasonable
- All passwords are equally likely to be selected
  - However, this can be remotely different from reality

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
  - 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)

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*

Example: Salted Passwords

- 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

Exercise L7-3: Examine Linux Password Salt

- Read manual page

  man 5 crypt

- Examine /etc/passwd

  What are the salt used in the passwords?

Guessing Through Authentication Function *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
  - 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

Example One-Time Passwords: 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}.

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

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

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

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.

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

Challenge-Response 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

Encrypted Key Exchange

- Defeats off-line dictionary attacks
- Idea: random challenges enciphered, so attacker cannot verify correct decipherment of challenge
- Assume Alice, Bob share secret password *s*
- In what follows, Alice needs to generate a random public key *p* and a corresponding private key *q*
- Also, *k* is a randomly generated session key, and R_A and R_B are random challenges

EKE Protocol

$$Alice \xrightarrow{\text{Alice} \mid\mid E_s(p)} Bob$$

$$Alice \xleftarrow{E_s(E_p(k))} Bob$$

Now Alice, Bob share a randomly generated secret session key *k*

$$Alice \xrightarrow{E_k(R_A)} Bob$$

$$Alice \xleftarrow{E_k(R_A R_B)} Bob$$

$$Alice \xrightarrow{E_k(R_B)} Bob$$

Biometrics

- Automated measurement of biological, behavioral features that identify a person
  - Fingerprints: optical or electrical techniques
    - Maps fingerprint into a graph, then compares with database
    - Measurements imprecise, so approximate matching algorithms used
  - Voices: speaker verification or recognition
    - Verification: uses statistical techniques to test hypothesis that speaker is who is claimed (speaker dependent)
    - Recognition: checks content of answers (speaker independent)

Other Characteristics

- Can use several other characteristics
  - Eyes: patterns in irises unique
    - Measure patterns, determine if differences are random; or correlate images using statistical tests
  - Faces: image, or specific characteristics like distance from nose to chin
    - Lighting, view of face, other noise can hinder this
  - Keystroke dynamics: believed to be unique
    - Keystroke intervals, pressure, duration of stroke, where key is struck
    - Statistical tests used

Cautions

- These can be fooled!
  - Assumes biometric device accurate *in the environment it is being used in!*
  - Transmission of data to validator is tamperproof, correct

Location

- If you know where user is, validate identity by seeing if person is where the user is
  - Requires special-purpose hardware to locate user
    - GPS (global positioning system) device gives location signature of entity
    - Host uses LSS (location signature sensor) to get signature for entity

Multiple Methods

- Example: “where you are” also requires entity to have LSS and GPS, so also “what you have”
- Can assign different methods to different tasks
  - As users perform more and more sensitive tasks, must authenticate in more and more ways (presumably, more stringently) File describes authentication required
    - Also includes controls on access (time of day, *etc*.), resources, and requests to change passwords
  - Pluggable Authentication Modules in Linux

PAM in Linux

- Idea: when program needs to authenticate, it checks central repository for methods to use
- Library call: *pam_authenticate*
  - Accesses file with name of program in */etc/pam_d*
- Modules do authentication checking
  - *sufficient*: succeed if module succeeds
  - *required*: fail if module fails, but all required modules executed before reporting failure
  - *requisite*: like *required*, but don't check all modules
  - *optional*: invoke only if all previous modules fail

Example PAM File

```
auth sufficient /usr/lib/pam_ftp.so
auth required   /usr/lib/pam_unix_auth.so use_first_pass
auth required   /usr/lib/pam_listfile.so onerr=succeed \
                      item=user sense=deny file=/etc/ftpusers
```

For ftp:

1. If user “anonymous”, return okay; if not, set PAM_AUTHTOK to password, PAM_RUSER to name, and fail
2. Now check that password in PAM_AUTHTOK belongs to that of user in PAM_RUSER; if not, fail
3. Now see if user in PAM_RUSER named in /etc/ftpusers; if so, fail; if error or not found, succeed

Exercise L7-4: Examine PAM in a Linux system

- Read manual page

  man 7 pam

  man 8 pam_unix

  man 8 pam_tally2
- Examine /etc/pam.d/login
- Configure the system so that it locks a user account after 4 failed logins
  - **Create a new user to test this (otherwise, you may be locked out)**

Summary

- Authentication is not cryptography
  - You have to consider system components
- Passwords are here to stay
  - They provide a basis for most forms of authentication
- Protocols are important
  - They can make masquerading harder
- Authentication methods can be combined
  - Example: PAM