Passwords
Authentication

• Basics
• Passwords
  – Storage
  – Selection
  – Breaking them
• Other methods
• Multiple methods
Authentication Basics

• Binding of identity to subject
  – Identity: external entity (e.g., Matt)
  – Subject: computer entity (process, etc.)
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)
  – Where entity is (e.g. at particular terminal)
Authentication System

• \( (A, C, F, L, S) \)
  – \( A \) information that proves identity
  – \( C \) information stored on computer and used to validate authentication information
  – \( F \) complementation function; \( f: A \rightarrow C \)
  – \( L \) functions that prove identity
  – \( S \) functions enabling entity to create, alter information in \( A \) or \( C \)
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
Passwords

• Sequence of characters
  – Examples: 10 digits, a string of letters, *etc.*
  – Generated randomly: by user, computer with user input

• Sequence of words
  – Examples: pass-phrases

• Algorithms
  – Examples: challenge-response, one-time passwords
Storage

- **Store as clear-text**
  - If password file compromised, *all* passwords revealed

- **Encipher file**
  - Need to have decipherment, *en-cipherment* keys in memory
  - Reduces to previous problem

- **Store one-way hash of password**
  - If read, attacker must guess password or invert the hash
Example

• UNIX system standard hash function
  – Hashes password into 11 char string using one of 4096
    (*we find out why this number later*) hash functions

• As authentication system:
  – \( A = \{ \text{strings of 8 chars or less} \} \)
  – \( C = \{ \text{2 char hash id || 11 char hash} \} \)
  – \( F = \{ \text{4096 versions of modified DES} \} \)
  – \( L = \{ \text{login, su, …} \} \)
  – \( S = \{ \text{passwd, nispasswd, passwd+, …} \} \)
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 $a, f, \text{ or } c$
    • 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 $l$)
Dictionary Attacks

• Trial-and-error: 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

• 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 $\sum_{j=0}^{s} 96^j \geq N$
  – $s \geq 6$: passwords must be at least 6 chars long
Approaches: Password Selection

• Random selection (not realistic)
  – any password from $A$ selected equally likely
• Pronounceable passwords
• User selection of passwords
Pronounceable Passwords

• Generate phonemes randomly
  – Phoneme is unit of sound, eg. \( cv, vc, cve, vcv \)
  – Examples: helgoret, juttelon are; przbqxdlfn, zxrptglnf are not

• Problem: too few

• Solution: key crunching
  – Run long key through hash function
  – 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, places
  - 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 setup/integrate into password selection system
Example: OPUS

- **Goal**: check passwords against large dictionaries quickly
  - “Bloom Filters”: allow to test set membership in less space (!)
    - 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” (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

• Main goal: slow down 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 $n$ times
Examples

• Vanilla UNIX method
  – Use DES to encipher message with password as key; iterate 25 times
  – Perturb DES in one of 4096 ways according to 12 bit salt

• Alternate methods
  – Use salt as first part of input to hash function
Guessing Through $L$

- Cannot prevent these
  - Otherwise, legitimate users cannot log in
- Make them slow
  - Back-off
  - Disconnection
  - Disabling
    - Be very careful with administrative accounts!
  - Jailing
    - Allow in, but restrict activities
    - “Honey-pots”
Password Aging

• Force users to change passwords after some time has expired
  – How do you force users not to re-use passwords?
    • Record previous (n) 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 (CR) Protocols

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

user \rightarrow_{request to authenticate} \rightarrow system

user \leftarrow_{random message r \ (the \ challenge)} \rightarrow system

user \rightarrow_{f(r) \ (the \ response)} \rightarrow system
Pass Algorithms

• Challenge-response where $f$ itself is a secret
  – Example:
    • Challenge: random string 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”
  – 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 (Lamport)
- \(h\) one-way hash function (e.g., SHA256)
- User chooses initial seed \(k\)
- System calculates:
  \[
  h(k) = k_1, \quad h(k_1) = k_2, \quad \ldots, \quad h(k_{n-1}) = k_n
  \]
- Passwords are chosen in reverse order:
  \[
  p_1 = k_n, \quad p_2 = k_{n-1}, \quad \ldots, \quad p_{n-1} = k_2, \quad p_n = k_1
  \]
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
Dictionary Attacks in CR

• 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 only 20 bits of randomness; Purdue attackers guessed keys quickly because deciphered tickets had a fixed set of bits in some locations 😊
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
    - Imprecise: approximate matching algorithms used
  - Voice: 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
    • E.g. distance from nose to chin
    • Lighting, view of face, other noise can hinder this
  – Keystroke dynamics: believed to be unique
    • 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 alike (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 (PAMs)
PAM: Pluggable Authentication Module

- 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

```bash
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
Key Points

• **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