Passwords
Overview

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

• Binding of identity to subject
  – Identity: external entity (e.g., Matt)
  – Subject: computer entity (process, etc.)
How to Assert “Identity”: Ideas

• 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 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
Password

- 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:
  – $A = \{ \text{strings of 8 chars or less} \}$
  – $C = \{ 2 \text{ char hash id } || 11 \text{ char hash} \}$
  – $F = \{ 4096 \text{ versions of modified DES} \}$
  – $L = \{ \text{login, su, ...} \}$
  – $S = \{ \text{passwd, nispasswd, passwd+, ...} \}$
Anatomy of Attack

• 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 $l(a)$
Defense

— 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 $l$)
Dictionary Attack

• 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
  – $N \geq \frac{TG}{P} = \frac{(365 \times 24 \times 60 \times 60) \times 10^4}{0.5} = 6.31 \times 10^{11}$
  – Choose $s$ such that $\sum_{j=0}^{s} 96^j \geq N$
  – $s \geq 6$: passwords must be at least 6 chars long
Password Selection Ideas

• 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. $cv$, $vc$, $cvc$, $vcv$
  - Examples: helgoret, juttelon are; przbqxdfl, zxrptglfn are not

- Problem: too few

- Solution: key crunching
  - Run long key through hash function
  - Convert to printable sequence
  - Use this sequence as password
Users select weak passwords

- Problem: people pick easy to guess passwords
  - Based on account names, usernames, 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.)
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: *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 original 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 using login function 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 Protocols

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

user → \text{request to authenticate} → \text{system} \\
user \leftarrow \text{random message } r \hspace{1cm} \text{(the challenge)} \rightarrow \text{system} \\
user \rightarrow \text{ } f(r) \hspace{1cm} \text{(the response)} \rightarrow \text{system}
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:
  \[ h(k) = k_1, \ h(k_1) = k_2, \ldots, \ h(k_{n-1}) = k_n \]
- Passwords are chosen in 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
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)
More “biometric” ideas

• 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
Caution

• **Usually a terrible idea to use biometrics** as primary authentication
  – Why? 😊
• Also they can often be fooled!
  – Assumes biometric device accurate *in the environment it is being used in!*
  – Transmission of data to validator is tamperproof, correct
Use Location in Authentication?

• 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 or similar device gives location signature of entity
    • Host uses LSS (location signature sensor) to get signature for entity
Combine multiple factors

• 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 Auth. 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
Sample PAM File

authsufficient /usr/lib/pam_ftp.so
authrequired /usr/lib/pam_unix_auth.so use_first_pass
authrequired /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
Q: What about usernames?

- How to chose usernames online?
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