Authentication

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

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

- One or more of the following
  - 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)
Authentication process

- Obtain authorization information from an entity
- Analyze the data
- Determine if it is associated with that entity

- The computer must save some information about the entity
- A mechanism to manage this information is required
  - Authentication system
Authentication System \((A, C, F, L, S)\)

- **Authentication information** \((A)\)
  - information that proves identity
- **Complementary information** \((C)\)
  - information stored on computer and used to validate authentication information
- **Set of complementary functions** \((F)\)
  - Generates the complementation information from the authentication information i.e. for \(f \in F, f: A \rightarrow C\)
- **Set of authentication functions** that verify identify functions \(\text{(L)}\)
  - For \(l \in L, l: A \times C \rightarrow \{\text{true, false}\}\)
- **Set of selection functions** \(\text{(S)}\)
  - Enable entity to create, alter information in \(A\) or \(C\)

*Computer Security: Art and Science*
©2002-2004 Matt Bishop
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

- Information associated with an identity that confirms the entity's identity.
- Authentication mechanism *based on what people know*
  - User supplies known password, system validates it
Passwords

- **Sequence of characters**
  - Examples: 12 digits \((\text{password space} = 10^{12} \ \text{elements})\), a string of letters, \(\text{etc.}\).
  - 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
- **Store one-way hash of password**
  - If file read, attacker must still guess passwords or invert the hash
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:
  - Method 1: Hide enough information such that one of $a$, $f$, or $c$ cannot be found
    - Prevents obvious attack

- Method 2: Prevent access to the authentication functions $L$.
  - Block access to all $l \in L$ or result of $l(a)$
    - Prevents attacker from knowing if guess succeeded
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

- Dictionary
  - List of random strings
  - Set of strings in decreasing order of probability of selection.
Using *Time* to counter password guessing

- Maximize the time the defender needs to determine the password

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 during which guessing occurs
- $N$ number of possible passwords ($|A|$)
- Then $P \geq TG/N$
Approaches: Password Selection

- Random selection
  - Any password from A equally likely to be selected
    - The expected time to find the password is highest when the selection of any set of possible passwords is equi-probable.
  - Removal of short passwords
  - Quality of pseudo-random number generator
Example of randomly generated passwords gone wrong

- $A = \{\text{8-char composed of lower case alphabets and digits}\}$
- Password space = $(26 + 10)^8$
- If it takes 0.00156 seconds per encryption (at that time), time to try all possible passwords is 140 years.
- A pseudo-random number generator is used to pick a random password.
- The total output of the PRN generator = $2^{15}$
  - The total number of unique passwords is $2^{15}$
  - It takes only 102 seconds to try all $2^{15}$ passwords
Human factors

• Can remember and repeat up to eight meaningful items
• One 8-char password
  • Can remember
• Two 8-char passwords
  • Write it down!
  • On the monitor, on the keyboard, in the wallet
• Solution:
  • Apply a transformation \( t \) to the written down password \( X \) to generate \( A \) i.e \( X \rightarrow A \)
  • e.g. “Capitalize the third letter in the word and append a 2 at the end”
  • \( t \) is easy to remember
Pronounceable Passwords

- Compromise between random, un-memorizable passwords and writing 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
  - Use long keys(passwords) – sentence or phrase that has some meaning
  - Run long key through hash function and convert to acceptable password sequence
DES key crunching for safer cypher keys by Lynn Grant

Key crunching

FIGURE 2. DES Key Crunching

FIGURE 1. DES Message Authentication
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, 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.)
1. Passwords based on account names
   a. Account name followed by a number
   b. Account name surrounded by delimiters
2. Passwords based on user names
   a. Initials repeated 0 or more times
   b. All letters lower- or uppercase
   c. Name reversed
   d. First initial followed by last name reversed
3. Passwords based on computer names
4. Dictionary words
5. Reversed dictionary words
6. Dictionary words with some or all letters capitalized
7. Reversed dictionary words with some or all letters capitalized
8. Dictionary words with arbitrary letters turned into control characters
9. Dictionary words with any of the following changes: a → 2 or 4, e → 3, h → 4, i → 1, l → 1, o → 0, s → 5 or $, z → 5.
10. Conjugations or declensions of dictionary words
11. Patterns from the keyboard
12. Passwords shorter than six characters
13. Passwords containing only digits
14. Passwords containing only uppercase or lowercase letters, or letters and numbers, or letters and punctuation
15. Passwords that look like license plate numbers
16. Acronyms (such as “DPMA,” “IFIP/PTC11,” “ACM,” “IEEE,” “USA,” and so on)
17. Passwords used in the past
18. Concatenations of dictionary words
19. Dictionary words preceded or followed by digits, punctuation marks, or spaces
20. Dictionary words with all vowels deleted
21. Dictionary words with white spaces deleted
22. Passwords with too many characters in common with the previous (current) password
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”
  1. Always invoked
  2. Can detect, reject bad passwords for an appropriate definition of “bad”
  3. Discriminate on per-user, per-site basis
  4. Needs to do pattern matching on words
  5. Needs to execute subprograms and use results
     1. Spell checker, for example
  6. Easy to set up and integrate into password selection system
Example: 4.3 BSD UNIX “passwd”

- Advise users that passwords must be at least 5 characters long, or at least 6 if they consist of monocase letters.
- Error message asks user to use a longer password
- If the user enters the password 3 times, requirements are waived
- Fails almost all requirements
- Newer UNIX versions do require password aging, but other requirements are still not met.
Example: *passwd+

- Provides a “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)
<table>
<thead>
<tr>
<th>Function</th>
<th>Action</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>length($p$)</td>
<td>Length of value</td>
<td>length(&quot;gueSS/This1!&quot;) = 12</td>
</tr>
<tr>
<td>alpha($p$)</td>
<td>Number of letters</td>
<td>alpha(&quot;gueSS/This1!&quot;) = 9</td>
</tr>
<tr>
<td>substr($p$, 2, 3)</td>
<td>Return substring</td>
<td>substr(&quot;gueSS/This1!&quot;, 2, 3) = “ue”</td>
</tr>
<tr>
<td>lcase($p$)</td>
<td>Make all letters lowercase</td>
<td>lcase(&quot;gueSS/This1!&quot;) = “guess/this1!”</td>
</tr>
<tr>
<td>rev($p$)</td>
<td>Reverse the string</td>
<td>rev(&quot;gueSS/This1!&quot;) = “!1sihT/SSeuG”</td>
</tr>
<tr>
<td>reflect($p$)</td>
<td>Reflect the string</td>
<td>reflect(&quot;hello&quot;) = “hellolehl”</td>
</tr>
<tr>
<td>trans($p$, a, b)</td>
<td>Change all a’s to b’s</td>
<td>trans(&quot;ax-ya&quot;) = “bx-yb”</td>
</tr>
</tbody>
</table>

**Figure 12–2  Examples of functions.**
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 guess hash $n$
Guessing Through \( L \)

- Cannot prevent these
  - Otherwise, legitimate users cannot log in
- Make them slow
  - Backoff
    - eg. Exponential backoff \((x^n)\), arithmetic series \((x, 2x, 3x ..., nx)\)
  - Disconnection
    - After \(x\) number of attempts, the system will get disconnected and must reconnect
  - Disabling
    - After \(n\) unsuccessful attempts (security manager)
    - Alerts security manager of potential threat
- Jailing
  - Allow in, but restrict activities
  - Use of honeypots
Password Aging

- Hide F, A, and C.
  - Or *change before attacker discovers the correct value*
  - If a password can be determined in 180 days, then change password more frequently than every 180 days.
- Force users to change passwords after some time has expired
  - How do you force users not to re-use passwords?
    - Record (and disallow) 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
Authentication with password (over a network)

Alice says “I am Alice” and sends her secret password to “prove” it.

<table>
<thead>
<tr>
<th>Alice’s IP addr</th>
<th>Alice’s password</th>
<th>“I’m Alice”</th>
</tr>
</thead>
</table>

Failure scenario??
Authentication with password

Alice says “I am Alice” and sends her secret password to “prove” it.

playback attack: Trudy records Alice’s packet and later plays it back to Bob.
Challenge-Response

- Reusable passwords can be “replay” attacked.
- User, system share a secret function $f$ (in practice, $f$ is a known function with unknown parameters, such as a cryptographic key)

user \[\text{request to authenticate}\] \rightarrow \text{system}

user \[\text{random message } r\] \rightarrow \text{system}
(user) \text{(the challenge)}

user \[f(r)\] \rightarrow \text{system}
(user) \text{(the response)}

Compute $f(r)$ for comparison with received response