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 $n$
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

Guessing Through $L$

• 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)
  
  $\text{user} \quad \text{request to authenticate} \quad \text{system}$
  
  $\text{user} \quad \leftarrow \quad \text{random message } r \quad \text{(the challenge)} \quad \text{system}$
  
  $\text{user} \quad \leftarrow \quad f(r) \quad \text{(the response)} \quad \text{system}$
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
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:
  \[
  h(k) = k_1, \quad h(k_1) = k_2, \quad \ldots, \quad h(k_{n-1}) = k_n
  \]
- Passwords are 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 \( p_{i-1} \).

\[
\begin{align*}
  \text{user} & \rightarrow \{ \text{name} \} \rightarrow \text{system} \\
  \text{user} & \leftarrow \{ i \} \rightarrow \text{system} \\
  \text{user} & \rightarrow \{ p_i \} \rightarrow \text{system}
\end{align*}
\]

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

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
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

\[ \begin{align*}
\text{Alice} & \quad \{ \text{Alice} \| E_s(p) \} \quad \rightarrow \quad \text{Bob} \\
\text{Alice} & \quad \left\{ E_s(E_p(k)) \right\} \quad \leftarrow \quad \text{Bob} \\
\text{Alice} \quad \left\{ E_s(R_A) \right\} \quad \rightarrow \quad \text{Bob} \\
\text{Alice} \quad \left\{ E_s(R_A,R_B) \right\} \quad \leftarrow \quad \text{Bob} \\
\text{Alice} \quad \left\{ E_s(R_B) \right\} \quad \rightarrow \quad \text{Bob}
\end{align*} \]
Biometrics

- Automated measurement of biological, behavioral features that identify a person
  - Fingerprints, voices, eyes, faces
  - Keystrokes, timing intervals between commands
  - Combinations
- Cautions: 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

PAM

- 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

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
Overview

- Access control lists
- Capability lists
- Locks and keys
- Rings-based access control
- Propagates access control lists

Access Control Lists

- Columns of access control matrix

<table>
<thead>
<tr>
<th></th>
<th>file1</th>
<th>file2</th>
<th>file3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andy</td>
<td>rx</td>
<td></td>
<td>rwo</td>
</tr>
<tr>
<td>Betty</td>
<td>rwxo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Charlie</td>
<td>rx</td>
<td>rwo</td>
<td>w</td>
</tr>
</tbody>
</table>

ACLs:
- file1: \{(Andy, rx) (Betty, rwxo) (Charlie, rx)\}
- file2: \{(Andy, r) (Betty, r) (Charlie, rwo)\}
- file3: \{(Andy, rwo) (Betty, r) (Charlie, rwo)\}
Default Permissions

- Normal: if not named, no rights over file
  - Principle of Fail-Safe Defaults
- If many subjects, may use groups or wildcards in ACL
  - UNICOS: entries are \((user, group, rights)\)
    - If \(user\) is in \(group\), has rights over file
    - \('*' is wildcard for \(user, group\)
      - (holly, *, r): holly can read file regardless of her group
      - (*, gleep, w): anyone in group gleep can write file

Abbreviations

- ACLs can be long … so combine users
  - UNIX: 3 classes of users: owner, group, rest
    - \(rwxrwxrwx\)
      - rest
      - group
      - owner
  - Ownership assigned based on creating process
    - Some systems: if directory has setgid permission, file group owned by group of directory (SunOS, Solaris)
ACLs + Abbreviations

- Augment abbreviated lists with ACLs
  - Intent is to shorten ACL
- ACLs override abbreviations
  - Exact method varies
- Example: IBM AIX
  - Base permissions are abbreviations, extended permissions are ACLs with user, group
  - ACL entries can add rights, but on deny, access is denied

Permissions in IBM AIX

attributes:
base permissions
  owner(bishop): rw-
  group(sys):   r--
  others:      ---
extended permissions enabled
  specify      rw-  u:holly
  permit       -w-  u:heidi, g=sys
  permit       rw-  u:matt
  deny         -w-  u:holly, g=faculty
ACL Modification

- Who can do this?
  - Creator is given \textit{own} right that allows this
  - System R provides a \textit{grant} modifier (like a copy flag) allowing a right to be transferred, so ownership not needed
    - Transferring right to another modifies ACL

Privileged Users

- Do ACLs apply to privileged users (\textit{root})?
  - Solaris: abbreviated lists do not, but full-blown ACL entries do
  - Other vendors: varies
Groups and Wildcards

• Classic form: no; in practice, usually
  – AIX: base perms gave group sys read only
    
    \texttt{permit -w- u:heidi, g=sys}

    line adds write permission for heidi when in that group
  – UNICOS:
    • holly : gleep : r
      – user holly in group gleep can read file
    • holly : * : r
      – user holly in any group can read file
    • * : gleep : r
      – any user in group gleep can read file

Conflicts

• Deny access if any entry would deny access
  – AIX: if any entry denies access, \textit{regardless or rights given so far}, access is denied

• Apply first entry matching subject
  – Cisco routers: run packet through access control rules (ACL entries) in order; on a match, stop, and forward the packet; if no matches, deny
    • Note default is deny so honors principle of fail-safe defaults
Handling Default Permissions

• Apply ACL entry, and if none use defaults
  – Cisco router: apply matching access control rule, if any; otherwise, use default rule (deny)

• Augment defaults with those in the appropriate ACL entry
  – AIX: extended permissions augment base permissions

Revocation Question

• How do you remove subject’s rights to a file?
  – Owner deletes subject’s entries from ACL, or rights from subject’s entry in ACL

• What if ownership not involved?
  – Depends on system
  – System R: restore protection state to what it was before right was given
    • May mean deleting descendent rights too …
Windows NT ACLs

- Different sets of rights
  - Basic: read, write, execute, delete, change permission, take ownership
  - Generic: no access, read (read/execute), change (read/write/execute/delete), full control (all), special access (assign any of the basics)
  - Directory: no access, read (read/execute files in directory), list, add, add and read, change (create, add, read, execute, write files; delete subdirectories), full control, special access

Accessing Files

- User not in file’s ACL nor in any group named in file’s ACL: deny access
- ACL entry denies user access: deny access
- Take union of rights of all ACL entries giving user access: user has this set of rights over file
Capability Lists

- Rows of access control matrix

<table>
<thead>
<tr>
<th></th>
<th>file1</th>
<th>file2</th>
<th>file3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andy</td>
<td>rx</td>
<td>r</td>
<td>rwo</td>
</tr>
<tr>
<td>Betty</td>
<td>rwxo</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>Charlie</td>
<td>rx</td>
<td>rwo</td>
<td>w</td>
</tr>
</tbody>
</table>

C-Lists:
- Andy: { (file1, rx) (file2, r) (file3, rwo) }
- Betty: { (file1, rwxo) (file2, r) }
- Charlie: { (file1, rx) (file2, rwo) (file3, w) }

Semantics

- Like a bus ticket
  - Mere possession indicates rights that subject has over object
  - Object identified by capability (as part of the token)
    - Name may be a reference, location, or something else
    - Architectural construct in capability-based addressing; this just focuses on protection aspects
- Must prevent process from altering capabilities
  - Otherwise subject could change rights encoded in capability or object to which they refer
Implementation

• Tagged architecture
  – Bits protect individual words
    • B5700: tag was 3 bits and indicated how word was to be treated (pointer, type, descriptor, etc.)

• Paging/segmentation protections
  – Like tags, but put capabilities in a read-only segment or page
    • CAP system did this
  – Programs must refer to them by pointers
    • Otherwise, program could use a copy of the capability—which it could modify

Implementation (con’t)

• Cryptography
  – Associate with each capability a cryptographic checksum enciphered using a key known to OS
  – When process presents capability, OS validates checksum
  – Example: Amoeba, a distributed capability-based system
    • Capability is (name, creating_server, rights, check_field) and is given to owner of object
    • check_field is 48-bit random number; also stored in table corresponding to creating_server
    • To validate, system compares check_field of capability with that stored in creating_server table
    • Vulnerable if capability disclosed to another process
Copying

- Copying: systems usually use copy flag
- Other approaches possible
  - Example: Amoeba again; suppose Karl wants to let Matt read file Karl owns, but not propagate this right
    - Karl gives capability to server, requests restricted capability
    - Server creates new capability (read only here), and sets check_field of new capability to \( h(\text{rights} \oplus \text{check_field}) \)
    - Server gives this to Karl, who gives it to Matt
    - Matt presents it to server to read file
    - Server looks in table to get original check_field, recomputes new check_field from original one and rights in capability
      - If this matches the one in the capability, honor it
      - If not, don’t