Mac OS X
Security Framework

Leon Towns-von Stauber, Occam's Razor

Seattle SAGE Group, February 2004

http://www.occam.com/osx/

X

Contents

Opening Remarks

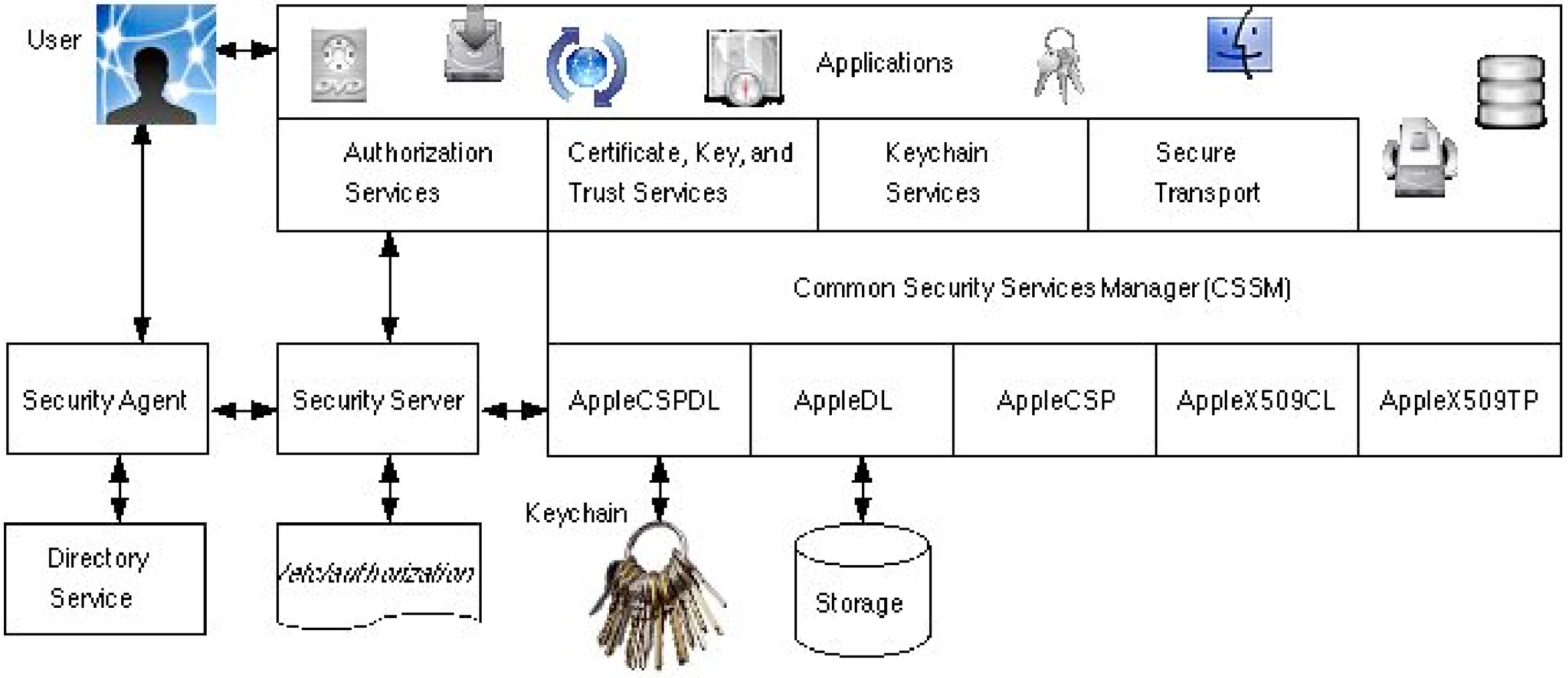
- I'm assuming basic familiarity with UNIX operating system design
- Where I'm coming from:
  - UNIX user and some-time admin since 1990
  - Full-time UNIX admin since 1995
  - NeXTstep/OS X user and admin since 1991
- This presentation primarily covers Mac OS X 10.3.2

Legal Notices

-
- Trademark notices
  - Apple®, Mac OS®, Finder™, Panther™, and other terms are trademarks of Apple Computer. See `http://www.apple.com/legal/appletmlist.html`.
  - Other trademarks are the property of their respective owners.

Overview

- Provides infrastructure for security-related functionality to Mac OS X apps
  - Privileged access, encryption, certificate handling, password storage, etc.
- Examples
  - Login Window uses the framework to authenticate graphical logins
  - Installer and Software Update verify permission to install software
  - Finder permits superuser-level access
  - Disk Copy creates encrypted disk images
  - Apple Mail Server supports SSL connections
  - Keychain Access is entirely dependent on the Security framework



Security framework and associated components

Overview

- What is a Mac OS X framework?
  - Like a versioned shared library, encapsulated with resources in a structured directory with a `.framework` extension (a "bundle")
    - Resources include headers, images, NIBs, property lists, etc.
    - [Show framework]
  - Panther (Mac OS X 10.3) splits Security framework into more pieces
    - `/System/Library/Frameworks/Security.framework` provides the bulk of the API (in C)
    - `/System/Library/Frameworks/SecurityFoundation.framework` provides an abstracted Objective-C API to Authorization Services
    - `/System/Library/Frameworks/SecurityInterface.framework` and `/System/Library/PrivateFrameworks/SecurityHICocoa.framework` provide Obj-C APIs for GUI elements

- Introduction
- Security Server
- Security Agent

- Every app linked to the Security framework maintains its own instance of the framework in its address space
  - Think of the large box in the Security framework diagram as the address space of a single process
- To put distance between apps and sensitive data, external daemons handle most passwords and private keys
- Daemon executables located in `/System/Library/CoreServices/`

- The Security Server processes authorization requests, stores keys in memory, performs cryptographic computations on keys, evaluates ACLs in keychains, and manages keychain master keys
  - Executable is named `SecurityServer`
- Started by `SecurityServer` startup item (in `/System/Library/StartupItems/`)
- Contacted via a privileged Mach IPC port, providing some assurance that processes are talking to the right daemon when making sensitive requests

- The Security Agent handles user interaction, acquiring user secrets (passwords, biometric data, smart card keys, etc.), thus further separating such data from the app requesting access
  - Executable is named `SecurityAgent`
- When the Security Server requires user interaction, it launches a Security Agent running under the user's UID
  - The Agent remains to handle further requests until the user logs out
- Interacts with user via onscreen dialogs
  - Any Security framework requests are sent to the user at the graphical console, even if the request came from another user's process
    - If no one is logged into the console, requests automatically fail
    - Darwin, lacking Aqua GUI, doesn't include the Security Agent, and many Security framework capabilities don't work with Darwin

- Introduction
- Policy Database
- Sequence of Events
- `AuthorizationExecuteWithPrivileges`
- Command-Line Tools

- Authorization Services enables programs to determine whether a user should be permitted to take certain actions
  - This is what give members of group `admin` much of their privileged access
  - Kind of like `sudo`, but primarily intended for GUI functions
- [Examples: System Preferences, Finder]
- An action is associated with an authorization **right**, configured in a **policy database**

- The policy database is an XML file, `/etc/authorization`
  - Fills a role analogous to `sudoers`, by defining authorization requirements
- Example entry in `/etc/authorization`, used by `authopen`:

```
<key>sys.openfile.</key>
<dict>
	<key>class</key>
	<string>user</string>
	<key>group</key>
	<string>admin</string>
	<key>mechanisms</key>
	<array>
		<string>builtin:authenticate</string>
	</array>
	<key>shared</key>
	<false/>
	<key>timeout</key>
	<integer>300</integer>
</dict>
```

- After transformation into NeXT property list format with `/usr/bin/pl < /etc/authorization`:

```
"sys.openfile." = {
        class = user;
        group = admin;
        mechanisms = ("builtin:authenticate");
        shared = 0;
        timeout = 300;
};
```

- `sys.openfile.`: The name of the right
  - This is a wild card entry, indicated by the trailing dot
- `class`: Requirement for successful authorization
  - The `user` class requires membership in a specified `group`
- `group`: Users who authenticate as members of group `admin` satisfy the requirements for this right

- `sys.openfile.` (cont'd.)
  - `mechanisms`: Actions taken to authorize
    - In this case, user authentication is performed
  - `shared`: If true, other applications need not reauthorize (within the `timeout` period)
    - In this case, authorization credentials are not shared with other apps
  - `timeout`: After the specified number of seconds since last acquisition of the right, the credentials cached by the Security Server are dropped, requiring reauthorization
    - A `timeout` of 0 means that reauthorization is never required

- Another example, used by the Login Window application:

```
"system.login.console" = {
     class = "evaluate-mechanisms";
     mechanisms = (
          "loginwindow_builtin:login",
          authinternal,
          "loginwindow_builtin:success",
          "builtin:getuserinfo",
          "builtin:sso"
     );
};
```

- Requested by Login Window after bootup, causing the Security Server to start a Security Agent (running as `root`) to display the login dialog
  - Upon login, this Agent is killed
- Class of `evaluate-mechanisms` causes Security Server to execute each of the routines listed for the `mechanisms` key
- The `authinternal` mechanism creates a shared credential that negates the need for reauthentication by applications after logging in

- Another example, used by PAM:

```
"system.login.tty" = {
    class = "evaluate-mechanisms";
    mechanisms = ("push_hints_to_context",authinternal);
};
```

  - Requested by the `pam_securityserver.so` module, referenced in several service config files in `/etc/pam.d/`
    - There is a `system.login.pam` right, which is unused (Huh?)

- Thanks to this, successful password-based authentication through a PAM-enabled service results in the creation of a shared credential within the context of that service (SSH session, etc.)
  - Possible for remote logins due to `push_hints_to_context`, which forwards auth info to the Security Server, bypassing the Security Agent (which normally needs to put up a graphical dialog)
  - [Demonstrate using `authorize` and `sudo`]

- Another example:

```
"system.login.screensaver" = {
        class = rule;
        rule = "authenticate-session-owner-or-admin";
};
```

  - The `rule` class uses an entry defined in the `rules` section of `/etc/authorization` to determine how authorization is performed

    - Rules offer reusable collections of values

```
"authenticate-session-owner-or-admin" = {
        "allow-root" = 0;
        class = user;
        group = admin;
        mechanisms = ("builtin: authenticate");
        "session-owner" = 1;
        shared = 0;
};
```

  - `allow-root`: If true, user logged in as `root` requires to authentication
  - `session-owner`: Right granted by successful authentication as user whose GUI session is up on the system's console

- Default entry applies when requested right doesn't match any other entry:

```
"" = {
        class = rule;
        rule = default;
};
```

  - The `default` rule produces a shared credential if a person can authenticate as an administrative user, which times out in 5 minutes:

```
default = {
        class = user;
        group = admin;
        mechanisms = ("builtin: authenticate");
        shared = 1;
        timeout = 300;
};
```

- Changes to `/etc/authorization` take effect on the fly, no signaling of `SecurityServer` required (unlike `syslogd`, `xinetd`, etc.)
- [Show `/etc/authorization`, processed through `/usr/bin/pl`]
- Naming conventions
  - Most rights prefixed by `system.`, indicating they're provided with the OS
  - Entries by third parties are encouraged to follow a convention like that used for Java classes: reversed DNS domain name associated with the organization, followed by segments describing the right (e.g., `com.occam.syslog.reload`)

- When an application requests authorization for an action (such as starting a network service from System Preferences), here's what happens:
  - 1) The application uses the Authorization Services API to contact the Security Server, requesting a named right (like `system.preferences`)
  - 2) The Security Server looks for the right in the policy database (`/etc/authorization`), and determines the requirements
  - 3) If the right requires authentication, and the Security Server doesn't already possess cached credentials that can be shared with the application, it triggers the Security Agent
  - 4) The Security Agent prompts the user logged into the console, then attempts to auth the user through the Directory Services framework
  - 5) Directory Services reports success or failure, which is passed back through the Security Agent to the Security Server
  - 6) The Security Server returns result to the application

- The `AuthorizationExecuteWithPrivileges` (`AEWP`) routine lets a program specify an external command to be run with superuser privileges
- Command invoked by `/System/Library/CoreServices/AuthorizationTrampoline`, which is `setuid root`
- Access to `AEWP` controlled by `system.privilege.admin` right in `/etc/authorization`, and is thus limited to `root` and to those in the `admin` group
- Since granting an app full superuser privileges to run an arbitrary command opens the possibility of a security hole, Apple warns against use in its developer documentation
- Used by some software, such as installers that set the `setuid` bit on binaries that copy files to restricted areas
  - Be sure you trust software that asks you to authorize a privileged action

- `/usr/libexec/authopen`
  - Lets authorized users read and write files to which they wouldn't normally have access
  - Example:
    - `nidump group . | authopen -w /etc/group`
    - Request made for `sys.openfile./private/etc/group` right
      - Note symlink resolution
  - If authorization successful, text from standard input replaces contents of `/etc/group`
  - Man page contains more details

- `authorize`
  - Available from `http://www.occam.com/tools/`
  - Takes name of right as argument, returns result of authorization attempt
  - Not really intended for high-security uses
    - Opportunity for malicious interference in exchange of information between `authorize` and tool invoking it
      - Better to include calls to Authorization Services in same code performing privileged actions, rather than calling an external authorization tool
- `sudo` is more flexible for CLI uses than `authorize`
- Mainly intended to demonstrate how `/etc/authorization` works
  - Could also be useful for casual authorization scenarios

- authorize.c

```
#include <stdlib.h>
#include <Security/Authorization.h>

int main(int argc, char *argv[]) {
     char *commandName = argv[0];
     char *rightName;

     if (argc == 2)
          rightName = argv[1];
     else {
          fprintf(stderr, "usage: %s
authorization_right_name\n", commandName);
          exit(EXIT_FAILURE);
     }

     OSStatus status;
     AuthorizationItem right = { rightName, 0, NULL, 0 };
     AuthorizationRights rightSet = { 1, &right };
     AuthorizationFlags flags = kAuthorizationFlagDefaults |
          kAuthorizationFlagExtendRights |
          kAuthorizationFlagInteractionAllowed;
```

- `authorize.c` (cont'd.)

```
        status = AuthorizationCreate(&rightSet,
            kAuthorizationEmptyEnvironment, flags, NULL);

        if (status == errAuthorizationSuccess) {
            fprintf(stdout, "success\n");
            exit(EXIT_SUCCESS);
        } else {
            fprintf(stdout, "failure\n");
            exit(EXIT_FAILURE);
        }
    }
```

- Compile with `cc -framework Security -o authorize authorize.c`

- Introduction
- Modules
- System Services
- STOS Projects

- The Common Data Security Architecture (CDSA) is a standard originally developed by Intel, and now promoted by The Open Group
- Apple has implemented it as part of the Mac OS X Security framework
  - Nearly all of it part of open-source Darwin project
- At the base are plug-in modules of various types
- Access to module functionality is through the Common Security Services Manager (CSSM) API, the centerpiece of the CDSA

- CSP (Cryptographic Service Provider): Random number generation, encryption/decryption, key generation, hashes, digital signatures
  - Symmetric encryption algorithms: ASC (Apple Secure Compression), RC2, RC4, RC5, DES, 3DES, AES/Rijndael
  - Asymmetric encryption algorithms: FEE (NeXT's Fast Elliptic Encryption), RSA, DSA, Diffie-Hellman
  - Message digesting algorithms: MD2, MD5, SHA-1
  - Pseudo-randum number generation algorithm: Yarrow
- DL (Data Library): File-based storage of certificates, keys, etc.
- CSP/DL: Manages keychains
- X509CL (Certificate Library): Manages X.509 certificates in memory
- X509TP (Trust Policy): Determines validity of X.509 certs

- Atop the CSSM, Apple provides higher-level APIs as part of the Security framework
  - Secure Transport: Implements SSL/TLS
    - Offers cleaner integration and abstraction for developers,, and possibly greater performance, but OpenSSL is more familiar and cross-platform
  - Certificate, Key, and Trust Services: Manages certs and public keys
  - Keychain Services: Programmatic interface to keychains

- The Secure Trusted Operating System (STOS) Consortium (`http://www.stosdarwin.org/`) brings together representatives from the U.S. federal government, academia, and private industry to work on advanced security capabilities using the Darwin kernel as a starting point
- Several projects involving Apple's CDSA implementation:
  - Apache SSL module using CDSA instead of OpenSSL
  - PGP implementation using CDSA
  - OpenSSH on CDSA

- Introduction
- Keychain Contents
- Sequence of Events
- Keychain Files
- Tools

- A keychain is a file containing keys, certificates, passwords, and other secured data
- Contents are encrypted, protected by an access password
- Convenience: a single password unlocks access to a multitude of passwords used for web sites, mail servers, file shares, etc.
  - Makes it practical to use unique, well-chosen passwords for each

- Keychains contain three kinds of objects: **keys**, **access control lists** (**ACLs**), and other **items**
  - Storage and access of objects in keychains is done by the AppleCSPDL module, via Keychain Services and the CSSM
- Each key in the keychain has an ACL, processed by the Security Server, which determines which applications can access it, and how
  - An app is identified by a hash of the invariant parts of the app binary
    - Thus, application access to keys must be re-established after a software update
    - Provides reasonable assurance that app is the same as when the user gave it permission to access a key
      - Only way to modify ACLs is by user direction through a Security Agent, and the Security Server will only accept ACL changes from an Agent that it has started

- Keychain items you see in Keychain Access aren't actually keys
  - Each item (password, secure note, etc.) is stored in the keychain, and encrypted with its own key, which is also stored in the keychain
  - ACLs are applied to per-item encryption keys, but they're made to look as if they're properties of the items the keys protect
- A master signing key is used to sign the per-item keys and their ACLs
- Per-item keys and the master signing key are themselves encrypted with a master key
  - ACLs can't be encrypted, since they're needed to determine whether access to encrypted keys should be permitted
- The master key is encrypted with the keychain password
- The master key and master signing key are also stored in the keychain, making the keychain file completely portable, requiring only the password to unlock its contents

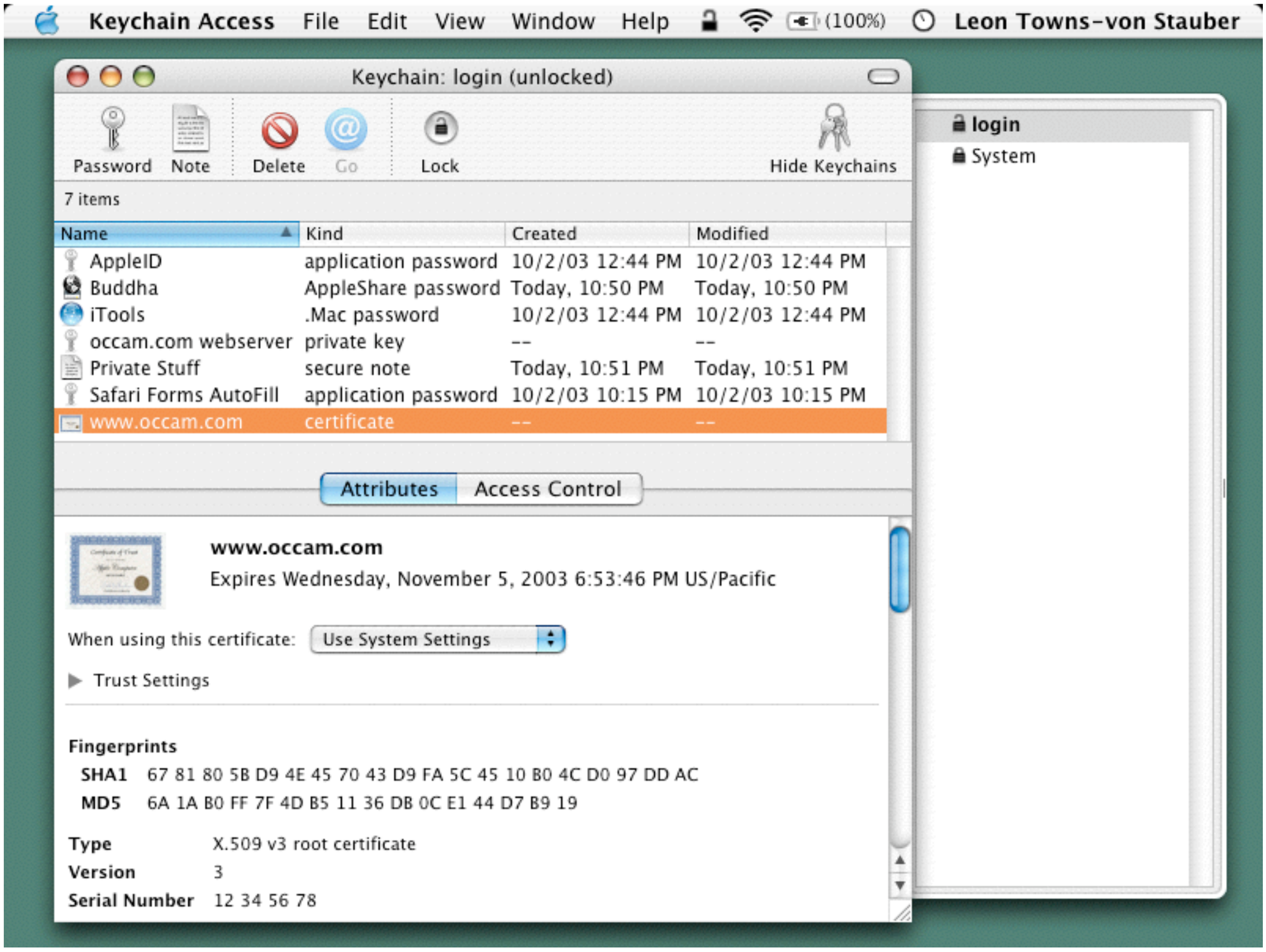
- Keychain items you see in Keychain Access aren't actually keys
  - Each item (password, secure note, etc.) is stored in the keychain, and encrypted with its own key, which is also stored in the keychain
  - ACLs are applied to per-item encryption keys, but they're made to look as if they're properties of the items the keys protect
- A master signing key is used to sign the per-item keys and their ACLs
- Per-item keys and the master signing key are themselves encrypted with a master key
- The master key is encrypted with the keychain password
- The master key and master signing key are also stored in the keychain, making the keychain file completely portable, requiring only the password to unlock its contents

- Here's what happens when an app desires access to a keychain item:
  - 1) App makes a request with Keychain Services, through the CSSM, which calls on the AppleCSPDL
  - 2) If the default keychain is locked, AppleCSPDL retrieves the encrypted master key and hands it to the Security Server
  - 3) The Security Server has a Security Agent prompt the user for the keychain password, which is passed back to the Security Server
  - 4) The Security Server uses the password to decrypt the master key, then caches it in memory (A keychain is unlocked when the Security Server has its decrypted master key cached in memory.)
  - 5) Once the keychain is unlocked, the AppleCSPDL retrieves the desired item, the item's encryption key, and the key's ACL, handing them to the Security Server

- Accessing a keychain item (cont'd):
  - 6) The Security Server verifies the signature on the key and ACL with the master signing key
  - 7) If the signature checks out, the Security Server processes the ACL, resulting in denial, permission, or another prompt through the Security Agent
  - 8) If access is permitted, the Security Server decrypts the keychain item and hands it to the AppleCSPDL, which passes it back up the software stack to the application
- Note that the application process never sees the user's keychain password, nor any of the decrypted keys in the keychain

- A keychain is created as `~/Library/Keychains/login.keychain` upon first login
  - Keychain password same as login password, synced when changing login password through GUI
  - If keychain and login passwords are the same, the keychain is automatically unlocked upon login
- The **login keychain** is also the initial **default keychain**, meaning that new items are added to it, and that it's the default argument for many of the `security` utility's commands
- On first boot, the `SecurityServer` startup item uses `systemkeychain` to create `/Library/Keychains/System.keychain`
  - Also creates `/var/db/SystemKey`, which presumably contains a randomly generated keychain password
  - Used by system processes running as `root` (daemons, boot processes)

- Keychain Access is the primary UI for keychain management
  - In the Settings dialog (under the Edit menu), you can set the keychain to lock after some idle timeout, and/or when the machine sleeps
    - "Locking" means to have the Security Server throw away its cached copy of the keychain's decrypted master key
    - You should consider lowering the idle timeout
  - Keychain First Aid (under Window menu) meant to fix certain problems that can corrupt a keychain, but with certain setups in the past it has caused more damage than it repairs
    - Run it in Verify mode, and try to fix things manually
  - Password Assistant (under Edit->Change Password...->Details->i) can help choose strong password
- Keychain Status menu bar item provides convenient access to functions



Keychain Access

- The `security` command can help manage keychains from the CLI
  - `security list-keychains`
  - `security lock-keychain` *`keychain`*
  - `security show-keychain-info` *`keychain`*
  - `security dump-keychain` *`keychain`*
  - See man page for more

Resources

- Web Sites
- Mailing Lists
- Books

- Security Framework developer documentation
  - `http://developer.apple.com/techpubs/macosx/CoreTechnologies/coretechnologies.html`
- Apple CDSA site
  - `http://developer.apple.com/darwin/projects/cdsa/`
- Intel CDSA site
  - `http://www.intel.com/ial/security/`
- The Open Group CDSA site
  - `http://www.opengroup.org/security/l2-cdsa.htm`
- CDSA specification
  - `http://www.opengroup.org/onlinepubs/009608599/`

- apple-cdsa (Apple)
  - `http://lists.apple.com/mailman/listinfo/apple-cdsa/`
  - Very low traffic, mostly developers
- MacOSX-admin (Omni Group)
  - `http://www.omnigroup.com/developer/mailinglists/macosx-admin/`
  - Moderate to heavy traffic

- Mac OS X Panther in a Nutshell
  - Chuck Toporek, Chris Stone
- Mac OS X Panther for Unix Geeks
  - Brian Jepson, Ernest E. Rothman
- Both contain references for Security framework-related CLI commands (written by yours truly)