Applied Access Control

CSC 348·648

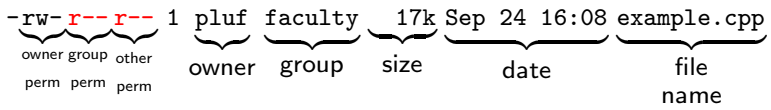
WAKE FOREST
UNIVERSITY
Department of Computer Science

Spring 2013

Unix Access Control

- Unix resources include the following
  - Files, which includes as devices and IPC
  - Network, TCP/UDP port address space
- Each process has a real UID (ruid), effective UID (euid), saved UID (suid); similar for GIDs
  - Real is ID of the user who started the process
  - Effective is ID that determines effective access rights
  - Saved, used to swap IDs, gaining or losing privileges
- If an executable's setuid bit is set, it will run with effective privileges of its owner, not the user who started it
  - If I run `lpr`, real UID is pluf (13630), effective UID is root (0), saved UID is pluf (13630)

Unix File Access Control



- New files created according to *umask*
- setuid bit for executables
  - Changes uid to uid of file owner on exec() of file
- Execute on directory implies access to directory

File System ACL

I want to do a project with a new group, how do I share files with only them without involving my sysadmin?

- Lameo
  - Create a directory accessible but not readable by everyone
  - Make directory (or files) in that directory with secret name
  - Hand out name as a capability
  - Smells bad and has many limitations
- ACL is a better solution
  - User created groups, user setable list of groups/users on files/directories
  - AFS, NTFS, NFSv4, FreeBSD, Solaris, ZFS...

Unix Identities and Abilities

- Highest privilege uid is root (uid = 0), has most privilege
  - Access privileged ports, override file permissions, etc...
  - Have root, then you own the system
- Process with a certain euid can send signals to other processes with that uid, ptrace() those processes, etc...

  So what?

- Good news
  - Works for multi-user system where programs are not malicious
  - Shortcomings can be fixed with file system ACL's

- Bad News
  - Difficult to contain damage of malicious root process
  - Lots of stuff needs root
  - Users can't protect themselves from bad applications since applications are completely **trusted**

Dropping Privilege

- To prevent programmer errors
  - Assume role of less privileged user for most operations (seteuid), let OS do its job
  - Only keep privilege long enough to do what you need
  - Temporary, program can resume root privilege at will
  - Essential for some things in Unix (e.g. race free file open)
- To prevent malicious code from inflicting damage
  - Drop privilege permanently (often accompanied by `fork()`)
  - Works well if there is a before-time (with privilege)/ after-time(without privilege) model for example daemons that need root to listen to privileged port

TOCTOU

- A user should only access a file if they have the permission to do so...
  - *What if user is running as setuid-root?*
- A printing program is usually setuid-root to access the printer
  - Runs *as if* the user had root privileges
  - But a root user can access any file...

How does the printing program know that the user has the right to read (and print) any given file?

System Call to the Rescue

- Consider an `access()` system call, result of `man access`

```
access -- check the access permissions of a file or pathname
⋮
Access() is a potential security hole and should never be used.
```

  - Separating time of check and time of use is not good
- As soon as you get a result from `access()`, its invalid, TOCTOU
  - Permissions could have changed, checking Permission and obtaining resource must be atomic!

Access, Open, and Exploit

- Goal is to trick setuid-root program into opening an inaccessible file
- Create a symbolic link to a harmless user file
  - `access()` will say that file can be read
- After `access()`, before `open()` switch symbolic link to inaccessible file
  - For example, `/etc/shadow` is a root readable password file
- Attack program must run concurrently with the victim and switch the link at exactly the right time
  - Interrupt victim between `access()` and `open()`

    How hard is this?

Confused Deputy Problem

- Pay to play compiler service (an old problem)
  1. Compiler service takes file and compiles it for user
  2. User hands compiler path for compiler's own billing file
  3. Compiler overwrites billing file

  What?

- *How can this be addressed?*
  - Drop privilege to that of user to open file (more error prone, common source of bugs)
  - Have user pass capability to compiler (less error prone, a reason to use capabilities)

Dangers of BAD ACM

- Too much privilege
  - Original Unix architecture suffers severely
  - Windoze world even worse
  - Once developers expect this, very hard to go back!
- Wrong granularity
  - To coarse grain $\rightarrow$ too much privilege
  - To fine grain $\rightarrow$ too many controls to set, difficult to use (SELinux complex and not very intuitive)
  - Compromise, protection profiles, roles, etc... use fine grain permissions to build up common case profiles.

Change Root

- Basic idea, allow process to change file system root

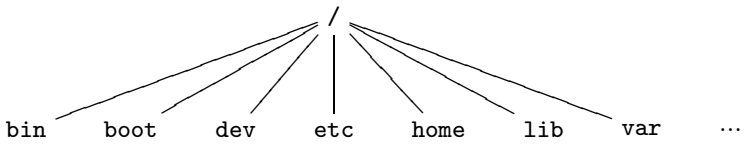
```
chroot("/home/apache/base");
```

> *"... any future system calls issued by the process will see that directory as the filesystem root. It becomes impossible to access files and binaries outside the tree rooted on the new root directory. This environment is known as a chroot jail."*

  - Can be used to create a separate virtualized copy of the system
- Variety of sharp edges due to power of root user, `ptrace()`, etc.
  - BSD Jail tries to fix this
- Fundamental problem, limited controlled sharing

Root Directory?

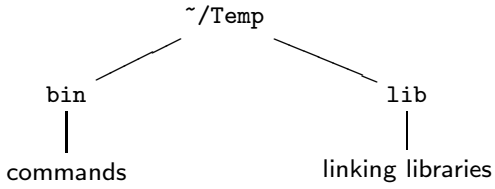
- The first or top-most directory in a hierarchy
  - In Unix and Unix-like systems, the root directory is denoted "/"
- In Unix all filesystem entries are *branches* of root



  - Each subdirectory under root contains certain files
  - `bin` contains command binaries, `lib` contains libraries, `etc` contains configuration files, `home` contains user directories, ...
- `chroot` will recreate part of this structure (essential components)

`chroot` Steps

- Assume you want an environment that only allows certain commands
  1. Create the new root directory, let's call it `Temp`
  2. Create `bin` directory within `Temp`
  3. Copy necessary Unix commands to `Temp/bin`
  4. Create `lib` directory within `Temp`
  5. Copy linking libraries to `Temp/lib`
  6. Issue the command `chroot  /Temp`
- The resulting tree the user would see is minimal... but functional



- For example

Terminal

```
> mkdir Temp
> mkdir Temp/bin
> cp /bin/bash Temp/bin
> cp /bin/ls Temp/bin
> mkdir Temp/lib
> cp /lib/* Temp/lib
> chroot  /Temp
bash-3.2#
```

- Copying all libraries was lazy... be more selective (use `ldd command`)

In this example what can the user do once root path has changed?

POSIX Capabilities

- Partitioning power of root, dividing into smaller capabilities
  - For example a process just needs ability to open a raw socket
  - Traditional Unix would require UID of zero, which is too much...
    Whoa, whoa, whoa, pump da brakes...?

- Some example capabilities that can be applied to a process
  - `CAP_DAC_OVERRIDE` ignore normal file permissions
  - `CAP_SYS_NET_BIND_SERVICE` bind to network sockets below 1024
    Tahw? 1024?

  - `CAP_NET_RAW` allow raw sockets (for example `ping` needs this)

```
chmod u-s /bin/ping
setfcaps -c cap_net_admin=p -e /bin/ping
```

- Improvement on setuid but fixed granularity

Sandboxing

- A security mechanism for safely running programs
  - An environment with controlled set of resources such as disk, system memory, network access and input devices
  - Can be considered an example of virtualization
- Some examples include
  - Applets (Java, Flash, and Silverlight) are self-contained programs that run in a virtual machine or scripting language interpreter that does the sandboxing
  - Jail is a set of resource limits imposed on programs by the operating system kernel, used to provide virtual hosting
    An example?

  - Virtual machines emulate a complete host computer, on which a conventional operating system run as on actual hardware

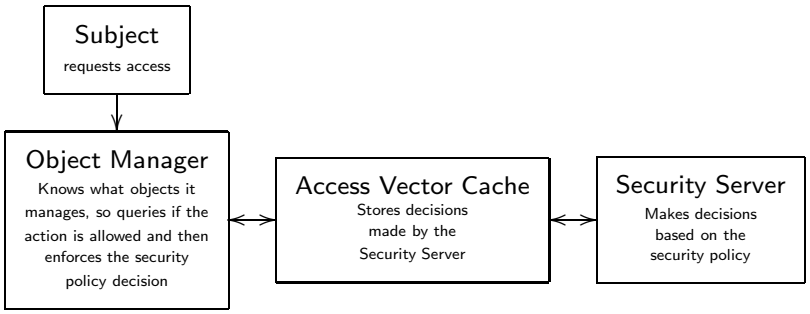
SELinux

- Added access control to Linux

  > *" ...a Linux feature that provides a mechanism for supporting access control security policies, including United States Department of Defense style mandatory access controls, through the use of Linux Security Modules (LSM) in the Linux kernel."*

- Flexible policy architecture
  - Has a general reference model in place, you select additional access control models using LSM
- Default uses combination of RBAC and Multi-Level Security (MLS)
  - Possible to implement BLP and Biba types of policies
  - Subjects and objects are given labels (sensitivity and category)
- Checks applied if existing Unix model succeeds e.g. if access fails, additional checks not invoked

- At a very high level, the following are the SELinux core components



1. Subject causes an action to be taken by an object (such as read a file)
2. Object Manager that knows the actions required of the particular resource (such as a file) and can enforce those actions
3. Security Server that makes decisions regarding the subjects rights to perform the requested action on the object, based on security policy
4. Security Policy describes rules (SELinux policy language)
5. Access Vector Cache (AVC) improves performance by caching decisions

Android

- Open source platform
  - Part of the *Open Handset Alliance*
  - Native development in Java (although C is possible)
- Platform overview
  - Linux kernel
  - Webkit-based browser
  - SQL-lite database support
  - Open SSL, Bouncy Castle crypto API, and Java library
  - Bionic LibC (small code, but no GPL)
  - Other support include media codecs, network APIs, etc...

Android Challenges

- Battery life
  - Developers should be aware of power consumption
  - Applications store state, can be stopped and restarted
  - Most foreground activity is never killed
- Android market
  - Not reviewed by Google, unlike what happens at iTunes
  - Cannot stop bad applications from appearing on the market
  - Hackers focus is on privilege escalation instead of remote exploits
  - However the application has to inform the user of activities

> *"A central design point of the Android security architecture is that no application, by default, has permission to perform any operations that would adversely impact other applications, the operating system, or the user. This includes reading or writing the user's private data (such as contacts or e-mails), reading or writing another application's files, performing network access, keeping the device awake, etc. "*

Android Application Design

- Framework requires programmers to use a certain structure
  - For example programs do not have `main()` or single entry point
  - This allows for sharing of code between applications
- Applications can use other application **components**
  - Assume your application needs to display a list of images and another application has developed a suitable scroller
  - You can call upon that scroller component to do the work
- Developers must design applications in terms of *components*
  - Activity, Service, Intents, Content Provider, and Broadcast Receiver

Android Component Types

- **Activity** - define an application's user interface
  - Typically developer defines one activity per *screen*
  - Only one activity can have focus and keyboard at a time
- **Service** - components that perform background processing
  - If an activity needs to perform an operation that must continue after the user interface disappears
  - Can be used as application-specific daemons, starting at boot
- **Content Providers** - store and retrieve data from database
  - Each has an authority describing the content it contains
  - Other components use the authority as a handle to perform queries
- **Broadcast Receiver** - act as mailboxes for messages
  - Broadcast message to specific destinations

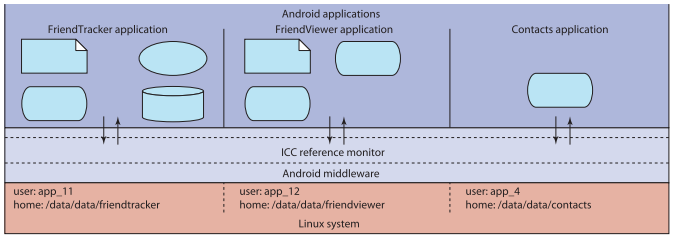
Manifest File

- Before starting a component, must learn the component exists
  - Applications declare their components in a manifest file
- File holds the application code, files, and resources
  - Specify what external Intents (messages) should be delivered
  - Also defines specifies rules for auto-resolution, specifies access rules, runtime dependencies, optional runtime libraries, and required system permissions

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest . . . >
3     <application . . . >
4         <activity android:name="com.example.project.FreneticActivity"
5                   android:icon="@drawable/small_pic.png"
6                   android:label="@string/freneticLabel"
7                   . . .  >
8         </activity>
9         . . .
10    </application>
11 </manifest>
```


Component Interaction

- Primary mechanism for component interaction is an **intent**
  - A facility for late run-time binding between components in the same or different applications
  - A message object containing destination address and data
- Android API defines methods that accept intents to do the following
  - Start activities, start services, or send broadcasts



- Inter-Component Communication (ICC) is an **action**, similar to IPC

Application Signing

- All Android applications (`.apk` files) must be signed
  - A certificate whose private key is held by their developer
  - This certificate identifies the author of the application
- The certificate does not need to be signed by a certificate authority

  "... it is perfectly allowable, and typical, for Android applications to use self-signed certificates."

  Tahw? What is the certificate providing?

- Can grant or deny applications access to *signature-level permissions*

Android Security Architecture

- A privilege-separated operating system
  - Each application runs with a distinct system identity (Linux IDs)
  - Parts of the system are also separated into distinct identities
  - Isolates applications from each other and from the system
- Kernel sandboxes applications from each other
  - applications must explicitly share resources and data
  - Must declare **permissions** for additional capabilities they require
- Android prompts the user for permission consent at installation

> *"Every app has to define which resources it wants access to. When you install an app (via the Market), you are shown a bunch of very clear warnings that detail what data and services the app will have access to. The great thing about this system is that apps can't lie. An app can't, after two weeks, gain access to your email."*

User ID and Security

- At install time, Android gives each package a distinct Linux user ID
  - Remains constant for the package's life on that device
- Middleware uses a *reference monitor* to mediate ICC access
  - Access is based on the protection level specified in the manifest

Using Permissions

- Basic application has no permissions
  - Difficult for it to do any damage to the system
- Application must **request** to use device *features*
  - Include `<uses-permission>` tag in `AndroidManifest.xml`

```
1 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
2     package="com.android.app.myapp" >
3     <uses-permission android:name="android.permission.RECEIVE_SMS" />
4     ...
5 </manifest>
```

> *"...permissions requested by the application are granted to it by the package installer, based on checks against the signatures of the applications declaring those permissions and/or interaction with the user. No checks with the user are done while an application is running: it either was granted a particular permission when installed, and can use that feature as desired, or the permission was not granted and any attempt to use the feature will fail without prompting the user."*

- Permission failure will result in `SecurityException` being thrown

Declaring and Enforcing Permissions

- Enforce your own permissions using the `AndroidManifest.xml`
  - Availability of your application's components to others

```
<manifest xmlns:android=
    "http://schemas.android.com/apk/res/android"
    package="com.me.app.myapp" >
    <permission android:name=
          "com.me.app.myapp.permission.DEADLY_ACTIVITY"
        android:label="@string/permlab_deadlyActivity"
        android:description="@string/permdesc_deadlyActivity"
        android:permissionGroup=
          "android.permission-group.COST_MONEY"
        android:protectionLevel="dangerous" />
    ...
</manifest>
```

- `<protectionLevel>` tells the system who is allowed to use
  - `"normal"` - always granted
  - `"dangerous"` - requires user approval

  - `"signature"` - matching signature key
  - `"signatureOrSystem"` - same signature and system application

Is this a lattice? Enforcing Biba and/or BLP?

- `<permissionGroup>` is group name associated with this permission
  - Only used to help the system display permissions to the user

iPhone Application Protection

- All applications run in a sandbox
  - Not like `chroot` or jails from Linux/FreeBSD
  - Unlike Android UID-based segregation, apps run as one user
- Enforced via a MAC module called **SeatBelt**
  - Based on TrustedBSD MAC framework
  - Currently undocumented, but developers do have access to some limited information
  - Public API in the near future for both Mac OS and iPhone OS?
- SeatBelt policy enforces the sandbox

SeatBelt

- Policy obtained from a Jailbroken iPhone
  - Fairly easy to read policy language
- Example rules

```
; System is read only
(allow file-read*)
(deny file-write*)
```

- More complex rules allow for things like regular expressions
  - Not documented at all, mostly speculation

iPhone Code Signing

- Applications signed by Apple
  - Developer certificate signed with Apple root certificate used for development and testing
- Code reviewed by Apple before publication in the App Store
  - Review process is unknown, but seems thorough
- Jailbreaking circumvents this protection

Steamy Andriod Malware

- 3/2011 Google removed 50 trojan apps from the market
  - *Infected* apps had the same malicious code, `Android.Pjapps`
  - Many of the apps were *knock-offs/clones* of other apps
  - Apps had access to data, network, etc... *security failure?*
- Steamy Window was one of the cloned applications
  - Once installed, malicious version asked for extra privileges...

Is this a failure of the Android security model?

- `Android.Pjapps` builds a botnet controlled by different C&C servers
  - It is able to install applications, navigate to websites, add bookmarks to your browser, send text messages, and optionally block text message responses (*messin' with yo GF/BF?*)
  - Registers its own service to operate in background without user noticing
  - Service will be started whenever signal strength changes, it tries to connect to a C&C server to register the infection

    Does Google have an uninstall switch?