CIS 610: Advanced Topics in Systems Security
Capabilities: Past and Present

Prof. Kevin Butler

Fall 2011
Evolution of Capabilities

• Original: HYDRA
• Extended to other systems (SCAP, ICAP, EROS)
• Android Permissions as capabilities
HYDRA

• Kernel base for a collection of OSes (e.g. Linux kernel vs. distro)

• Ran on C.mmp multiprocessor
  ‣ 16 processors, each with up to 32 MB memory, crossbar architecture

• Provide an environment for effective utilization of the hardware resources
Design Philosophy

• Separation of mechanism and policy
  ‣ Policy: high level e.g., scheduling and protection
  ‣ Mechanism: low level e.g., message dispatching

• Multiple instances of the systems coexist together

• Rejection of strict hierarchical layering

• Protection

• Reliability
Design Philosophy

• Need to decide *what belongs to the kernel and what does not.*
  ‣ A kernel provides facilities for building an OS
  ‣ OS defines an *abstract machine* by providing resources
  ‣ OS efficiently allocates (hardware) resources
  ‣ Objects belong to several distinct “types”
  ‣ Reference count and garbage collection
Hydra Environment

• Key terms: Procedures, Local Namespaces (LNS), Processes and Capabilities (dependent & independent)

• Hydra procedures support protection facilities via templates (formal parameters list)

• LNS: record of execution environment at the time of invocation
  ‣ generated when procedure is invoked based on capabilities

• Capabilities are manipulated by the kernel so they can’t be forged.
  ‣ Capabilities are stored in C-lists
Protection Mechanism

• Protection vs. Security: protection is a mechanism, security is a policy
  ‣ procedure-based as opposed to process-based

• Procedure an object with C-list (caller independent capabilities), inherited from called process

• Procedures have templates (formal parameters list), access rights of the actual parameters are checked at the time of invocation.
Procedure-Level Protection Domains

• **HYDRA**
  ‣ Each procedure defines a new protection domain

• **Procedure**
  ‣ Code
  ‣ Data
  ‣ Capabilities to other objects
    ‣ Caller-independent
    ‣ Caller-dependent templates

• **Local Name Space**
  ‣ Capabilities are bound here
  ‣ Record of a procedure invocation (procedure instance)

• **Process**
  ‣ Stack of LNSs
Walk Right

• *Walk primitive*: produces the capability in specified position of the object named by the parameter capability
  ‣ The walk primitive, like all kernel primitives, is an access right protected by the "kernel rights" bits in a capability.

• Because of the walk primitive, procedure environment extends beyond capabilities in its LNS
  ‣ closure of set of objects reachable along a path (originating in the LNS) such that every capability along the path (except possibly the last) grant the walk right

• Where have we seen path-based enforcement?
Q: Which object defines the protection domain?
Implications of Fine-Grained Protection

• Programmer
  ‣ Must define *templates* for procedure
  ‣ Connect the procedure rights together

• Performance Impact

• **Q:** Do we need to manage rights at this level?
Security Implications

• What if a node is compromised? Particularly in networked system?

• HYDRA unconfined right disables capability propagation.... what happens?

• Protection vs Security
• Add the identity of the holder into the capability
  ‣ (client ID, obj ID, rights, integrity)

• Authorize transfer of capabilities
  ‣ C1 constructs a signed message to grant a capability to C2 –
    C1 must be the identity in the capability
  ‣ C2 presents capability and signed grant on first use
  ‣ Server authorizes based on security policy and creates a
    capability for C2 to use
Re-presenting Capability

- server creates object for client C1 with capability 
  \((Object, Random0)\)

- C1 gets external capability \((Object, Rights, Random1)\)

- \(Random1 = f(C1, Object, Rights, Random0)\)

- \(f\) is a one-way function (e.g., hash)

- Random0 kept secret, so capability should be unforgeable

- **What conditions need to be in place for this to be secure?**
Capability Propagation

- C1 wants to delegate capability to C2
- Must present request to access server, policy check
- Server creates external capability \((Object, Rights, Random2)\)
- \(Random2 = f(C2, Object, Rights, Random0)\)
Revocation

- Based on event list and propagation tree
- Server updates list based on revocation request
- Ensure only certain subjects can revoke by embedding multiple identities into capability
  - e.g., \((Object, Rights, C1, C2, C3, Random3)\)
- Can also store the tree in the server
- **What assumptions about the server are made?**
Android

• Android uses permissions as capabilities
  ‣ Delegation not always supported though

• Permissions granted either at time of use or install time depending on OS
  ‣ iOS: asks for access to e.g., location
  ‣ Android: defined in manifest file

• Refresher on Android components/permissions...
Android Architecture

• The Android smartphone operating system is built upon Linux and includes many libraries and a core set of applications.
• Uses the Binder component framework
• Applications consist of many *components* of different types
• Applications interact via components
Component Model

• While each application runs as its own UNIX uid, sharing can occur through *application-level* interactions
  ‣ Interactions based on components
  ‣ Different component types
    • Activity
    • Service
    • Content Provider
    • Broadcast Receiver
  ‣ Target component in the same or different application
Components

• Activity: components make up the UI, actions by user tell activities to start other activities

• Service: components perform background processing and provide interfaces between applications

• Content provider: interface for data sharing, modeled as a database

• Broadcast receiver: handle intent messages, allow subscribing to specific actions
Intents

• Intents are objects used as inter-component signaling
  ‣ Starting the user interface for an application
  ‣ Sending a message between components
  ‣ Starting a background service
Android Manifest

• Manifest files are the technique for describing the contents of an application package (i.e., resource file)

• Each Android application has a special AndroidManifest.xml file (included in the .apk package)
  ‣ describes the contained components
    • components cannot execute unless they are listed
  ‣ specifies rules for “auto-resolution”
  ‣ specifies access rules
  ‣ describes runtime dependencies
  ‣ optional runtime libraries
  ‣ required system permissions
• Android focuses on *Inter Component Communication* (ICC)
• The Android manifest file allows developers to define an access control policy for access to components
  ‣ Each component can be assigned an access *permission label*
  ‣ Each application requests a list of permission labels (fixed at install)
• Android’s security model boils down to the following:

![Diagram showing access control permissions between applications.]

• However, there are a number of exceptions
Permission Re-delegation

• Occurs when an application without a permission gains additional privileges through another application

• A special case of the confused deputy problem
  ‣ Privilege obtained through user permissions
  ‣ e.g., a carelessly written program that exposes public services
  ‣ exploited by untrusted program
Potential Deputies

• Preconditions: possesses *dangerous permission* and has public interface

• Study by Felt et al.: 872 manifests examined, 16 system apps, 756 most popular, 100 recently uploaded

• 320 apps were candidates based on manifest

• Call graph analysis used to find paths from public entry points to protected API calls

• Attacks: enable wifi/GPS from unprivileged app, disable alarm clock from stopping vibrating/sleeping
Why not Capabilities?

• Capability could be required by deputy
  ‣ Argument: badly-written deputy could use own capabilities when making calls rather than those of system

• Access control on privileges: require approval before granting communication ability
  ‣ similar to ICAP model
  ‣ Argument: requires analysis of deputies to determine what privileges would be involved (examined in Felt et al.’s CCS’11 paper)
IPC Inspection

• Reduce deputy permissions to intersection of recipient and requester permissions

• Similar in concept to Java stack inspection
  ‣ inspect permissions set, captures communication history
  ‣ permission reduction reduces set of permissions
  ‣ similar to LOMAC

• Create different instance of application per request, each with own current permission set
Impact on Applications

• Permissions become transitive
  ‣ e.g., barcode scanner needs CAMERA permission, but app calling it does not
  ‣ with this model, if app doesn’t have CAMERA permission then the intersection won’t

• Assumption: deputies bad at checking caller privilege
  ‣ Main challenge: How to get context of caller privilege?
    • provenance of IPC invocation (Quire paper)

• Requires rewriting of certain apps
Takeaway

• Capability systems have evolved based on requirements
• An essential model used in many systems
• Lessons keep on recurring in even modern systems
  ‣ How to get better context for capabilities/permissions?
  ‣ Understanding information flows
  ‣ Model checking and analysis
Next Day

• VM Security
• Take-home midterm

• (slide thanks: Trent Jaeger, Will Enck, Patrick McDaniel, Machigar Ongtang, Adrienne Felt, Alex Kachurin, Mohamed Saad Laaseel)