James Ide
2009-09-30

CS 261 - Capabilities

Principle of Least Privilege

- Minimize the impact that an attacker could have – a goal of capabilities
- Current OSes allow all programs a wide set of a privileges
  - Distinction at the user level (e.g. processes may run with the current user's privileges)
  - OS doesn't know what the program may want to do – e.g., it doesn't know if the program is Solitaire (needs few privileges) or Word (may need many privileges)
- Capabilities provide *finer-grained privileges*

Ambient Authority

- A set of globally available privileges, usually derived from the privileges of the user
- E.g., Unix makes all of a user's privileges available to a process
  - OS maintains a conceptual "bag" of privileges and looks in the bag to authorize actions that a process wishes to perform
  - The set of privileges is the union of those available to the current euid, gid, and suids
    - Need a simple wrapper script to drop privileges if we e.g. wish to run a setuid program with the owner's privileges but not those of the current user
  - Example of ambient authority vs. capabilities:
    - `cp foo bar` The `cp` program works because it has access to `foo` and `bar` due to ambient authority
    - `cat <foo >bar` The shell has access to `foo` and `bar`, and delegates the capability to respectively read and write to these files by passing their file descriptors to `cat`
- Capabilities proponents say that processes don't need all of the rights of the user – ambient authority does not follow the principle of least privilege
- All code in a process, including untrusted modules, have the same rights under ambient authority

An Opt-in Model

- Code begins with no privileges and must be explicitly delegated privileges to have any rights
- **Goal:** Want an easy way to delegate privileges so that it is not a burden on the programmer
- **Possibility 1:** Caller must explicitly pass privileges as function arguments

    - E.g., `public void append(String filename, Privilege p, String message);`
    - Annoying for programmers to have to keep designators (file names) with authority (privileges)
- **Possibility 2:** Simplify by bundling file name and privilege
    - E.g., `class BundledPrivilege { private Privilege p; private String filename; }`
    - Doesn't adhere to OOP very well since actions are separated from objects
- **Possibility 3:** Build privileges directly into objects
    - A reference to a `File` object is the capability to access the file
    - Requires modifications to core APIs

Bundling Designation with Authorization

- Facilitate privilege delegation by building privileges into resource identifiers
- Design a capabilities language
    - Need *encapsulated objects with private members* and *unforgeable object references*
    - Must ensure that the only way to affect the system is with a capability

Case Study – CapDesk

- CapDesk was a capabilities-based windowing system
- User actions (e.g., double-clicking a file) gave the associated program the capability to access the file
- When the target resource is unknown, a trusted system component could be used to let the user select it
    - E.g., the "Save As" option in a word processor needs the user to select a file, so a trusted file chooser dialog would receive input from the user and pass the capability to access that file back to the word processor
- Infers policy from UI events – user-friendly so that humans do not need to explicitly grant capabilities to processes

Other Benefits of Capabilities

- Can effectively sandbox untrusted code by passing it only a subset of safe capabilities
- Software developers can delegate privileges to smaller components within an application to enforce *privilege separation*

Problems with Capabilities

- Low compatibility with legacy code - may need to port application to the new language or API with capabilities support, especially to interact with external libraries or programs that use capabilities

- Fine-grained privilege management is tedious and difficult because a user's system may have many configuration possibilities, and does not scale
- Poor past performance when capabilities were built into the processor
- Revocation – without it, capabilities last indefinitely
    - Newer systems use *revocable forwarders* as proxies
    - Forwarder proxies method calls through to the underlying capability
    - An "enabled" flag in the forwarder can be turned off by the capability revocation system at which point the forwarder will no longer proxy requests
    - Pitfalls:
        - An underlying object may have a method that returns itself or another capability. The forwarder must be sure to wrap the returned object in another forwarder before returning it to the caller, or else the caller will gain access to a non-wrapped capability.
        - Also need to unwrap objects when passing them into methods that expect the underlying objects as arguments
        - Many forwarders (i.e., levels of indirection) can hurt performance
- Difficult to prevent propagation of capabilities
    - Code that has the capability to communicate with other code (usually can, due to covert channels) can act as a proxy, serving requests to use its capabilities on the behalf of others
- Hard to incrementally move to a capabilities-based system – needs to be built from the ground up

OS Support

- OS could maintain capabilities per process
- Can pass capabilities along processes
- Similar to microkernel architecture in that each component can perform a specific set of tasks
- Granularity is at the process level, as opposed to the user or group level

Access Control Lists (ACLs) vs. OS-based Capabilities

ACLs	Capabilities
OS maps resources (e.g., files) to users who may access those resources. Since processes run with the user's privilege, the OS effectively maps resources to processes.	OS maps processes to capabilities. Each process has a set of privileges.
Need to set up just once	Hard to maintain a mapping because processes are relatively transient, so the mapping is inferred
Only the owner of a resource may edit the ACL	Can delegate capabilities from process to process