October 3: Capabilities

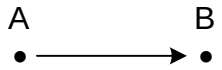
Jonathan "Jon" Kotker

1 Capabilities: Goals

1.1 Simple substrate, programmatic policies

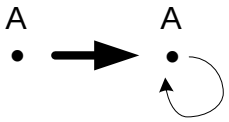
With access control lists (ACLs), we need to add new users every time a new policy is made. With capabilities, however, we have a simple set of substrates to work with.

In the *dots-and-arrows model*, the dots represent the entities and the arrows represent which entity has capability for which other entity. For example, in the diagram below, A has a capability for B.

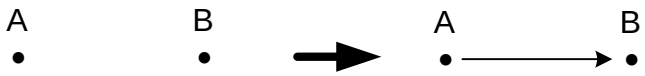


We have a few rules that we can use to propagate capabilities and evolve the capability graph:

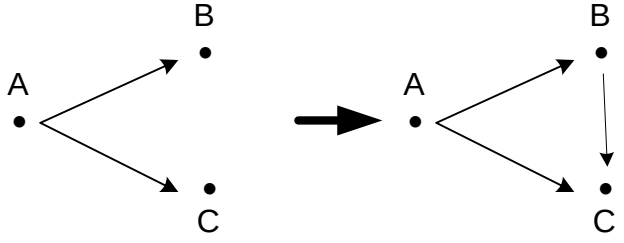
1. *Self-loop.* An entity has a capability for itself.



2. *Allocation/Instantiation.* An entity A gets a capability for another entity B. This is equivalent to calling a constructor on an object, but the instance has no capabilities yet: what can A do with its capability for B?



3. *Passing/Introduction.* If an entity A has a capability for entities B and C, it can provide a capability to B for C.



4. *Removal.* An entity A can remove its capability for B.

There are no other ways for capabilities to propagate. We can thus use these rules to formally derive how a capability graph can evolve, and to prove certain properties about the capability graph.

Student question. Is an arrow a reference or a capability?
Answer. In an object-capability model, the reference *is* the capability.

Student question. Where do we get a capability initially?
Answer. We set initial conditions that apply during bootstrapping and employ rule 2 above.

1.2 Build a system with least privilege.

Capabilities allow us to build a system with the least privilege. It can be recursive: the program could itself be privilege-separated. This makes it easier for programmers to reason about privilege.

1.2.1 No ambient authority.

In UNIX, if a program is run, it is run with the user's ID, so it has access to all the files that the user has. Any line of code can write to any file. This is a case of *ambient authority*, where authority is floating around waiting to be taken. With capabilities, however, the default is to have no ambient authority: we have to obtain something special to access files.

Student question. How about Java?
Answer. People have suggested using Java with the security manager and stack inspection as the security substrate (which is, in fact, the standard), instead of an object-capability language. But, what about ambient authority? System-level code has all permissions: any line of code from the file system can access the rest of the filesystem. For example, in Java, the line

```
new FileWriter(new File("...")).write(...)
```

allows access to the file, but only if the program is considered trusted by the class loader. It is also restricted while running: this is referred to as *restricted ambient authority.*

1.2.2 Bundle designation with authority.

The system can be clunky: every single method may be given a lot of permission objects as parameters simply to do something:

```
f(String pathname, ..., FileSystemPermission p)
{
   ...
   File f = new File(pathname, p);
   f.write();
}
```

The argument `pathname` is the *designation* and the argument `p` is the *authorization*. This is not how actual systems work, because of the extra arguments and because this might give the function too much permission: permission is given to `f` for the whole filesystem.

So, we should combine designation and authorization. `f` can act as a capability, so we can pass the file object in:

```
f(File f, ...) { ... f.write(...); ... }
```

Now, the permission is restricted to just that file, since the function only gets the capabilities that it needs.

1.2.3 Security distinctions represented by object hierarchy.

Suppose the system defines an API for a `FileSystem` object, but you find that this is too coarse-grained and you would like a `File` object. Internally, the `File` object can contain a `FileSystem` object and the name of the file. Users of the object can therefore use the `File` object without direct access to a `FileSystem` object. A `ReadOnlyFile` object can similarly be built upon `File` object. The security distinctions are thus represented by object hierarchy. The caveat here is that the classes have to be written carefully, because if one class is compromised, a malicious user could have access to the internal object.

Student question. It looks like this system requires full language support.
Short answer. Yes, you would have to code from scratch.
Long answer. People have researched implementing capabilities at the OS level: there are attempts to build a UNIX emulator on a capability-based OS.

Student question. How can we modify Java to be a capability-based language?
Answer.

- *Get rid of global variables*, since they violate the principle of "no ambient authority"; less strictly, we could prevent mutable goals.
- Public instance variables may still be allowed, since we can treat them as private with public getter or setter methods that are part of the API.
- *Tame reflection. Reflection* is the idea that if a user has a reference to an object and the `String` of a method name, then the user can call the method of the object with the same name. This is proper if the methods accessed were anyway public. However, in Java's reflection mechanism, private methods can also be accessed! (This is allowed for trusted code, but disallowed for applets.)
- *Native methods.* In the line `A = new B();` we note that a user can call public constructors of classes. This implies that with the line `A = new File(...);` A is making a whole new capability for itself! A can take in a string and generate a file handle: this is more than what regular methods are allowed to do. This violates the capability model.
- *Replace Java library methods.* We make new ones that don't violate the model. There is a shortcut: we tame the methods instead by providing access to a subset of the API.

Question. Are capabilities used?

Answer. Unfortunately not. There are a few notable exceptions: Kaha, for example, is a capability-version of JavaScript developed by Google. It is used by Yahoo! for mashups.

Question. Why are capabilities not used?
Answer.

- Hard to pick up: it affects the way programmers think and architect systems.
- Hard to abstract the changes away and to incrementally change existing code.
- Hard for the capability world to talk to the non-capability world: for example, the UNIX emulator may not understand the capability-based OS.
- Languages have to be redesigned entirely.
- Capability OSs have inter-process communication overheads.

  Student question. How do we bootstrap a capability-based system? *Answer.* The loader is given all the capabilities at the outset: it is responsible for dispensing the capabilities. In CapDesk, a prototype of a capability-based OS, there are two kinds of grants: a *static grant* is approved or denied by the user when the program was installed, and a *dynamic grant* is given using "powerboxes": capabilities are given based on UI actions, so if a user selects a file from a dialog box, then the program receives a capability for that file.