CS 261 Scribe Notes

Matt Finifter

October 2, 2008

Administrative

- Homework 2 due Monday 10/6

Capabilities

- Assigned readings weren't that great
- Small group of people working on capabilities, and they haven't written much down
- We'll talk mainly about language support for capabilities
  - Allows for privilege separation within a program at a fine level of granularity
  - Easier to reason about what code has what privileges
- Problem: monolithic applications running under your uid
  - Any app you run can do anything you can do
  - If any part of the code goes wrong, there is potentially a big problem
- Problem: too much privilege. Applications get capabilities by default.
  - Solution: by default, code gets no privileges. Opt-in instead of opt-out. Code only gets privileges explicitly granted to it.

- Problem: needs to be some way to get privileges. Invoked code needs privileges.
  - Solution: Explicitly delegate privileges to callee. For example, consider logging messages to a file:

    ```
    void append(String filename, Privilege p, String message)
    ```

  - But this is irritating from a programmer's standpoint because you have to pass around a lot of privileges.
  - This approach contrasts with stack inspection, where privileges are enabled/disabled and stored globally somewhere.
- Problem: specifying intent twice. In the example above, we specify both the filename and the privilege to access that file separately. The filename designates, and the privilege authorizes.
  - Solution: bundle designation with authorization. Continuing with the example, change the API to:

    ```
    void append(File f, String message)
    ```

  - Using a file object designates a file on the hard disk. Also, we declare the semantics of the File object to mean that having a file object allows you to read and write to the file.
  - This reference to the File object is called a capability. It both designates and conveys authorization.
- Important principle: can carve powerful capabilities up into smaller pieces. For example, can take a capability to the entire hard drive, and generate a capability to a particular file on the hard disk. Then, can use this to generate a capability to the file that allows only read access.
  - Hopefully libraries provide some common abstractions that do this for you in a lot of cases. For example, File, ReadOnlyFile, AppendOnlyFile, etc.
  - Extensibility is also a goal, so that the user can do this himself when appropriate.
  - Example where you write a wrapper for a file that allows read-only access:

```
class ROFile {
    private File f;
    read { f.read(); }
    write() { throw ... }
}
```

  - Discussion that maybe this lends itself to dangerous laziness on the part of the programmer, since he has to explicitly pare down a capability before handing it off. It is easier for a programmer to pass around a capability to the entire hard disk than it is to generate and pass capabilities to specific files. Capabilities provide a framework to write more secure code, but don't force you into it. You can write bad code in any language, but in a capability language, it is easy to recognize this type of misuse by looking at the APIs.

- In an object-capability language, objects have entry points (in Java, public methods) and state (in Java, private fields). Having a capability to an object does not allow you to access its private state (encapsulation).

- Graph model of capabilities. Objects are the nodes, capabilities are the edges.

- Basic idea behind capabilities is simple. Can build access control on top of capability systems if you want it.

- The idea is that the only way to affect the outside world is through a capability. For example, there is no publicly available global method `eraseHardDrive` that erases the whole hard drive. If you wanted this functionality, then in capability discipline, you would have to have a capability to the hard drive.

  - "No ambient authority"
  - Authority: power to modify the state of the system. Authority can be direct or indirect (e.g. direct access to a file vs. access to someone who you can ask to write a file for you).
  - Ambient authority is authority available to all the code (like a global variable, with unlimited scope). Capability folks say this

is bad because it violates the principle of least privilege (POLP). Shouldn't have global variables that hold capabilities. Use locals, parameters, etc. instead.

- It is possible to know just by looking at code exactly which capabilities are in scope. This allows you to reason about your code more easily.
- Question: what prevents a piece of code from constructing something it shouldn't have access to?
  - For example, `new File("/etc/passwd")`
  - In Java, this is okay. But in a capability language, this constructor would be turned off because it effectively turns nothing into something.
- Question: where do the capabilities come from in the first place?
  - Answer 1: at some point, you "enter" the capability world. The "creator" parcels out authority, giving each module exactly what it needs to do its job, and nothing more. Then, everything else runs within the capability world.
  - Answer 2: the whole world is capabilities, including the OS. Even processes communicate using capabilities.
- Compare shell command `$ cp foo bar` to `cat < foo > bar`. Second one is more capability-like because file descriptors are passed instead of filenames.
- How much of current systems are we willing to throw away? If willing to start over, could use capabilities all the way from the ground up. If not, need some adapter between non-capability world and capability world.

Confused Deputy

Imagine a game called nethack with a high scores file that it writes to. Want to be able to specify a log file to the game when we run it. Without any protection, could potentially trash the high scores file by specifying its name as the desired log file. This is an example of a confused deputy.

- Attempt 2: enable privilege for high score file, then use it, then disable privilege for high score file before doing anything to the log. This defends against the confused deputy by being explicit about the source of authorization. Disadvantage of this approach is that you have to explicitly insert enable/disable privilege calls. Also, what if you wanted to pass 2 filenames – one for the log file and one for something else? There would be no way to differentiate the source of authorization for these two files.

- Attempt 3: use file descriptors instead of file names. Pass the privilege along with the filename. Every time a file is opened, explicitly specify the source of authorization.

- Attempt 4: bundle authority and name together in an object. This makes it more convenient to pass it around. Still have to explicitly enable and disable privilege before use.

- Attempt 5: let the caller open the file and pass you a capability.

- With capabilities, can't trick nethack into writing to a file that you couldn't have written to yourself.

- Ambient authority makes code susceptible to confused deputy bugs.

Capability revocation

Have to build revocation at the application level. See Figure . It is also possible to create a design that allows you to both revoke and un-revoke a capability. See Figure 2.

There is a potential for problems with equality comparison when using revocable capabilities. == is no longer useful, but `equals` can be made to work (in Java). Highly language dependent, since languages handle equality differently.

There is also the potential for problems with e.g. a revocable directory. Need to get it right so that it wraps any files gotten from it in revocable files. Have to be very careful about leaking non-revocable capabilities.

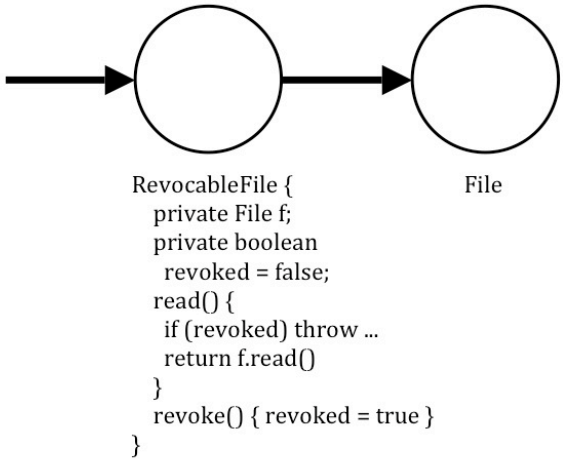

Figure 1: Graphical representation of a revocable file. Normally, the `RevocableFile` calls methods on the `File` it holds, but after being revoked via the `revoke` method, it throws an exception when its methods are called.

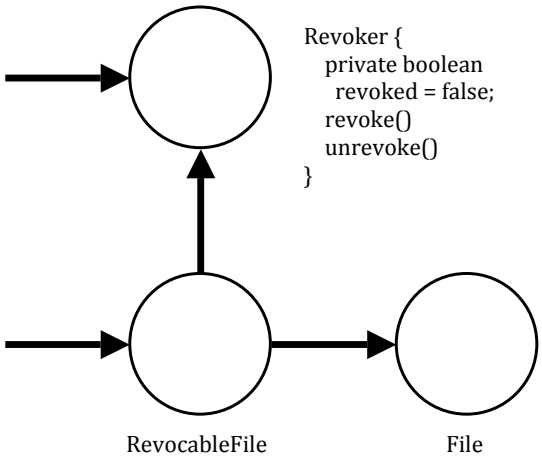

Figure 2: Revocable file capability that can also be un-revoked. Alice holds a capability to the `Revoker`, which has the `revoke` and `unrevoke` methods. The `RevocableFile` privately holds a capability to the `Revoker`, which it consults before performing any operation. If the `Revoker` is in the revoked state, any attempt at performing an operation on the `RevocableFile` throws an exception; otherwise it calls the corresponding method on the `File` that it holds. Alice can pass a capability to the `RevocableFile` to Bob, and she can revoke and unrevoke Bob's access to the underlying `File` using her `Revoker`.