Capabilities

Yunlong (George) Li

September 25, 2012

1 Background

This is the last class on software security. In the security world, the idea of capabilities go back to the 70's. There used to be a lot of work on capability-based OS and hardware in the 70's but they died out, become unpopular. Nevertheless, it is interesting to look at the capability community's critiques of the conventional approaches to OS and designing secure systems.

2 Goals

Goals for capabilities	
1.	Minimal substrate for access control
2.	Least privilege/authority
2'.	Radical privilege separation
3.	Reviewable security
3'.	Way to reason about security
4.	~~Legacy~~

2.1 Minimal substrate

In the first paper, Miller and Shapiro argued for the notion of a minimal substrate. *The platform should provide a minimal set of primitives that are flexible enough so that applications can enforce their own access control policy based on them.* This is reminiscent of the end-to-end principle (but in a security context).

A standard argument against capability is the lack of revocation. But as the paper showed through the caretaker pattern, it is possible to build revocation on top of the platform using its primitives.

2.2 Least Privilege/Authority

There is a subtle but important difference between permission and authority. While a permission is directly defined by a system's access control policy, an authority is derived from the arrangement of permissions.

An example would be the printer example from the last lecture. If Bob has access to the printer but Alice does not, we say that Bob has the *permission* to the printer. If Alice can persuade Bob to print something on her behalf, then she has the *authority* to the printer.

The authors point out that it is not enough to have the Principle of Least Privilege (because the term privilege is a bit fuzzy when it comes to permission vs. authority) but really we want the Principle of Least Authority (or POLA).

2.3 Reviewable security

Capability folks want reviewable security: not only do they want their systems to be secure but also they want to be able to review and confirm the security properties of their systems. If they choose the set of primitives that the platform supports wisely, it would make the review process simpler and more rigorous.

2.4 ~~Legacy~~

One of the main criticisms of the capability way of building systems is that the capability folks seem to only care about building new systems using their paradigm and not so much about interfacing with existing legacy code. This forces software engineers to re-write their entire software suite should they decide to use capability to enforce their security policies.

3 Critiques

Critiques of conventional approach
1. Goals not met
2. Coarse granularity of permissions
3. Ambient authority
4. Authority separated from designation

3.1 Goals Not Met

Obviously the goals mentioned in the previous section are not met.

3.2 Coarse granularity of permissions

The reason why we can't implement the Principle of Lease Privilege is the lack of a minimal substrate. The granularity at which we grant permissions to applications is too coarse. In traditional operating systems, if one wants to grant an authority for an application to run, then he would have to grant it the authority to run as him, giving the application more permissions than necessary.

3.3 Ambient Authority

An example of of ambient authority is given by the `File` class in Java. If a block of code gets access to an `File` object, then it can get access to the rest of the filesystem, a very undesirable ambient authority. Every line of code in that block has ambient authority and it doesn't need to justify to the OS if it decides to exercise that authority. Why is ambient authority bad? Ambient authority violates the Principle of Least Privilege and makes privilege separation difficult. Further, it doesn't help reviewable security because if ambient authority is present in a program, then security analysis must be done everywhere in the code.

3.4 Authority Separated from Designation

Capability folks would point out that it is bad to separate authority from designation.

```
domytaxes(String filename, Permission p) {
  ...
}

domytaxes2(File f) {
  ...
}
```

In the above example, the file descriptor `f` both designates what file we are operating on as well as contains within it the authority to operate on that file, where as the filename `filename` provides a designation but does not provide authorization to that file.

Why is the separation bad? Because programmers don't want to pass in two arguments every time they need to invoke a method that needs access to a file. With the separation, now they need to do twice the work, because both `filename` and its corresponding permission object `p` must be coupled for a single purpose.

It might even become tempting for programmers to just pass in the global permission object every time so that they don't have to separately construct a file-specific permission object prior to the method invocation.

If there is no separation between authority and designation, then least privilege and privilege separation come for free because we can run a method by default with no permissions other than the ones passed to it and free of ambient authority.

4 Confused Deputy Problem

The confused deputy problem involves a scenario where a computer program (the deputy) is tricked by another party (a client) into misusing its authority. For example, consider the game NetHack and consider that the user `client` has access to her own resource `R2` but not to the high score file `hiscore.db` stored in the filesystem.

```
|--------|     |--------|     |-------------|
| client | --> | Deputy | --> | Resource/OS |
|--------|     |--------|     |-------------|
    |           nethack        /var/nethack/hiscore.db
    v
  |----|
  | R2 |
  |----|
```

Imagine that NetHack has an additional feature that lets the deputy save a log of the entire game.

```
$ nethack -l log.txt
```

Since `nethack` runs with its owner's permissions via `setuid`, a plausible attack would be to specify a file that the client doesn't have access to but `netback` does:

```
$ nethack -l /var/nethack/hiscore.db
```

Even though the programmer might have thought that he's doing the "logging" on behalf of the client, the OS has lost the notion of for whom its doing the work. Because the UNIX operating system system call separates designation from authorization, we run into this confused deputy problem.

Another example of the confused deputy problem is Cross-Site Request Forgery (CSRF) in the web browser.

```
|---------|     |----------|
| browser | --> | bank.com |
|---------|     |----------|
     ^
     |
|----------|
| evil.com |
|----------|
```

When loading `evil.com` sends a network request (include an in-line image) to `bank.com`, the browser will send any session cookie associated with `bank.com` to it because the browser has an ambient authority policy. We discussed how to defend against CSRF in a previous lecture.

A capability system is a natural solution to the confused deputy problem. In the CSRF example, a random CSRF token would be given to the client by `bank.com` so that `evil.com` cannot guess the correct url. Thus the url with the CSRF token (128-bit random string) would serve as a *capability* to the client to make a network request to `bank.com` to do some operation on client's behalf.

An interesting web service that has both the access control security model and the capability model is Google Docs. In Google Docs, one can specify explicitly who can have access to a file or generate a random url and use it as a capability to send to certain recipients.

5 Object-Capability Rules

The capability model says, there are two kind of things in the world, only two. There are objects and there are capabilities (object references). Everything in the world should be represented by an object. The only way for an object to interact with something in the rest of the world is to have a capability to it. A capability can be obtained by one of the following rules:

- An object `A` has a capability to itself.
- If `A` has a reference to `B` and `C`, then `A` can introduce one to another.
- An object `A` can allocate new objects. If `A` allocates `B`, then `A` gets a reference to it.

In Java, the first rule is achieved through the `this` reference and the second rule is achieved via a method call. `A` can call a method of `B`, passing it a reference to `C` as an argument. Using these rules, given a state of an object-capability graph, we can prove some bounds on what operations are allowed and what this graph can do. This bound is very conservative. If we can look at the code and the behavior of each object, we can get a more precise bound on what can happen in the system.