CS 261 Scribe Notes

Lecture 10/04/07 - Capabilities

1 Administrative Notes

- Reading: The Oz-E Project: Design Guidelines for a Secure Multiparadigm Programming Language
- This lecture represents the end of discussions on software security. Next lecture will start discussing network security.

2 Capabilities

Object capabilities are a language-based approach to security. A capability is an unforgeable reference to an object. In the world of capabilities everything is either an object or a capability.

- Unforgeable: The object can't be created out of thin air.
- Unspoofable: The paper mentions that capabilities are unspoofable as well as unforgeable. It is unclear what is meant by this, but a likely analogy is the Java typesystem. For example, in Java, a String type can't be spoofed – no one can create an object that gets treated like a String, but isn't actually a String object.

2.1 The Goals of a Capability System

1. Follow the Principle of Least Privilege.
2. Provide reasoning about and restriction of privilege.

Some examples of violating the Principle of Least Privilege:

- Statically assigned access control rights. E.g. A word processor can read and write to the entire file system.
- Granting a process an entire user's privilege set.

2.2 Achieving the Goals of a Capability System

2.2.1 Avoid Ambient Authority

Authority: Describes what a process or a module can do, how it can affect the outside world, what permissions it has (which files it can read or write), and whether it can cause something to be done indirectly.

Ambient Authority: Authority granted through some global namespace.

The OS maintains a list of every user's permissions and a process executes with a bunch of rights. When the process executes a system call (e.g. write) the OS determines which right to yield (e.g. use the user's permissions to allow the open file call). The process's authority to open the file was granted through the global list of the user's permissions. In this system there is a Confused Deputy: there is no differentiation between authority granted through the process's permissions and authority granted through the user's permissions. In addition, the default level is the maximum permission which violates the Principle of Least Privilege.

Ambient Authority

open ("/tmp/foo"); //OS decides if this is allowed by checking all variables in scope to see if any allow access.

Authority fixed with Capability

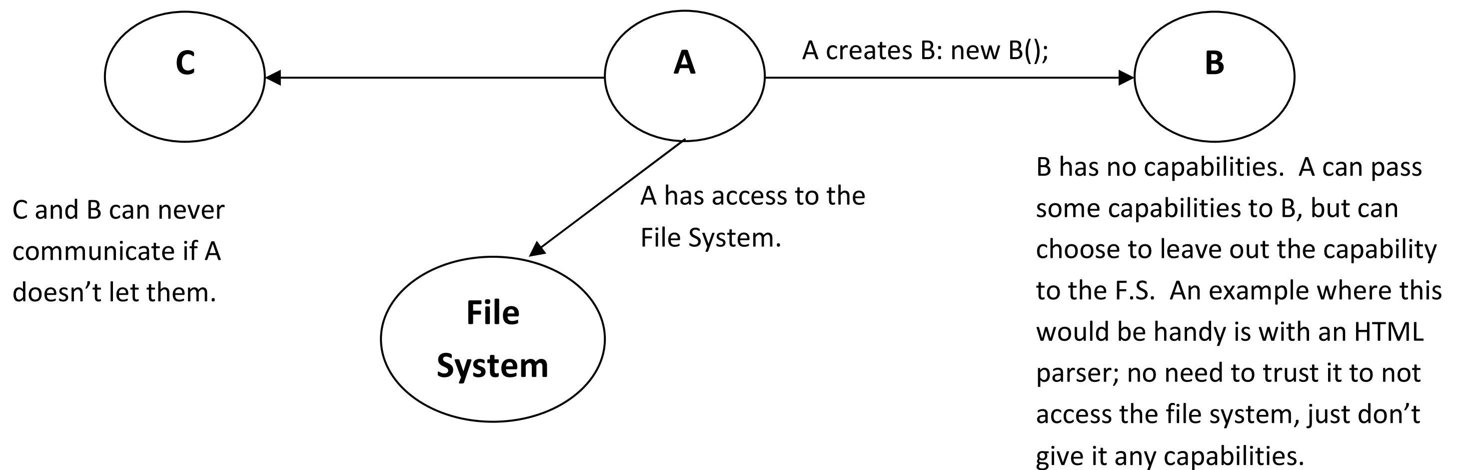
open (file); //Specifies exactly which authority to use.

2.2.1.1 How Avoiding Ambient Authority Follows the Principle of Least Privilege

In a capability system all code has no privilege by default. All privileges must be explicitly granted (this is the opposite of using ambient authority). Writing down a list of all the privileges each module should have is difficult and doesn't scale well so a capability system tries to solve this by following two basic rules:

1. Universal scope conveys no authority. The default is no privilege and there are no global variables (variables accessible by all code) that contain capabilities.
2. Can only obtain a capability in one of three ways:
    a. Method Arguments: someone invokes a method and passes it a capability.
    b. Return Value.
    c. Parenthood: The process creating a new object gets a reference to that object. Since the new object is born with no authority, having authority to it is no extra authority. (Side note: someone brought up the fact that this is in fact creating a new capability out of thin air – a violation of the unforgeable property. While this is true, there is no added authority being created here.)

A diagram to illustrate the use of capabilities



2.2.1.2 How Avoiding Ambient Authority Provides Reasoning About and Restriction of Privilege

```
//Global variables have to have no authority-granting capabilities, e.g. constant values

Class C
{
	//these variables could be in scope
	foo (File x)
	{
		//these variables could be in scope of capabilities passed in argument
	}
}

//Only need to look at what's in range to have an idea of what capabilities could be available.
```

2.2.2 Capability Discipline

Capability Discipline: Association between objects and the resources they represent should be kept intact.

One of the costs in capability-based programming is designing classes and interfaces to follow the idea of Capability Discipline.

Two examples of no discipline:

```
Object                                    Resource
Class File  ----------------------------> file on disk
//An object in address space.
//Operations on File are the way to interact with the bits on disk.

//Should only supply methods to operate securely on a file.
read();
write();

//This is bad.  File should not have this method.
eraseHardDrive();
```

```
Object                  Resource
Class Dir  ------------> subtree of filesystem hierarchy
list ();
get (String name);

//Dir shouldn't have this method.
getParentDir ();
```

One of the advantages of a capability systems with discipline enforced is that it can allow for finer-grained control than original capability designers may have thought of.

Two examples of capability discipline leading to fine grained control:

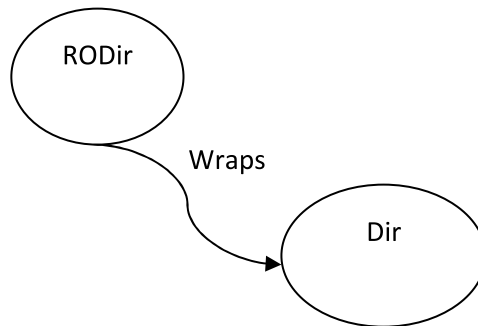
```
/*Example 1*/
//Create a new guard object.  Override methods of Dir to get whatever you want.

//Read only directory
Class RODir{

//Return a read-only directory
override get(..);

}
```

```
/*Example 2*/
//Create a revocable object.  Use a wrapper class with extra flag and reference to real class.
//Check the flag before performing operations of wrapped class.
```



2.2.3 Bundle Designation with Authority

Example: Defending against Confused Deputy

A process is running with two sources of authority, when the OS needs to check authority for a particular action, it just checks if any of the granted rights will suffice, it doesn't check to see the source of the rights.

The reason the confused deputy can occur is because the string filename is separated from the justification for having the right to access it. Code that separates these two things might be susceptible to confused deputy attacks. A capabilities system combines these two things.

Confused Deputy:

```
nethack (String logfile)
{
        s = open ("/var/nethack/highscore");    //by game's group authority.
        write_scores(s);
        p = open (logfile);                     //by user's id authority.
        writelog (p);
}
```

The malicious user runs (-s flags tells program to write log. File given should be in user's path, e.g. /usr/bin/nethack):
>nethack –s /var/nethack/highscore

Solution 1 - Use Java's enablePrivilege:

```
nethack (String logfile)
{
        enablePrivilege (HSFile);           //sets some state associated with the thread to avoid
                                            //passing in more arguments.
        s = open("/var/nethack/highscore");
        write_scores(s);                    //have file descriptor (a capability),
                                            // don't need privileges anymore.
        disablePrivilege();
        …
}
```

Solution 1 gets clumsy when dealing with multiple filename arguments because enablePrivilege can only have one source of authority. For example the call, append ("/var/nethack/highscore", logfile), would require two extra privileges, one for each argument of the method.

Solution 2 – Authority wrapper class:

```
class AuthFile
{
        String filename;
        Privilege p;
}

nethack (AuthFile af){…}
append (AuthFile src, AuthFile dest){…}
open (AuthFile af){…}
```

Two advantages of Solution 2 are: there are no strings as file names, only AuthFiles are used; and it reduces the chance of a confused deputy occurring as long as conversion from String to AuthFile is done correctly. However, Solution 2 may not be a natural programming style.

Solution 3 – A more natural AuthFile:

```
class AuthFile
{
        private String filename;
        private Privilege p;
        open();
        read();
        write();
}
```

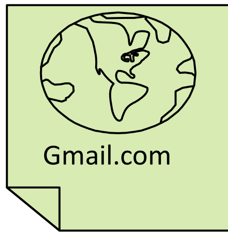
Solution 3 implements a capabilities system: a reference to one of these AuthFile objects is a capability.

2.3 Implementing Capabilities

- Not much implementation experience.
- Capabilities have not yet been validated to understand the cost of trying to design the systems.
- "Capabilities are the way of the future. They always have been and always will be."

2.4 A Practical Example of Confused Deputy attack

Cross Site Request Forgery (CSRF)



Firefox

A user types in her password initially.
Every subsequent “click” on the Gmail interface results in her cookie authenticating her for every request.

Firefox

Html:
<A href = “gmail.com/delete_all_your_email”>

If the user clicks on the link, the request goes to Gmail and will be authenticated as her. This is an example of Ambient Authority. The browser looks through all cookies and sends the appropriate one with each HTTP request. There is no notion of tying cookie to a specific link.

Problem: Evil.com can name a URL on Gmail without having the right to it.

Solution: Tie URL with the right to access it with a random token appended to the URL for the session.