CSE 509 Lecture 24

Prof. Rob Johnson, Scribe:Arun Rathakrishnan

December 2, 2013

1 Capability Based Systems

In addition to Hardware Capabilities (fat pointers) and OS Capabilities (file descriptors), we have Programming Language Capabilities. Objects should serve as a capability mechanism. The model is called Object Capability model.

The major motivation for capability based systems is to enforce the Principle of least privileges. A consumer must be granted the least privileges on an object it needs, in order to perform a particular task. The least privilege depends on the level of enforcement an underlying machine can provide. For example, when adding a vector of numbers, a routine needs to only read the values at the pointer location. But if hardware does not support read only memory locations, we choose the ability to access only the passed addresses within the bounds, as the least privilege.

Capability based systems must support,

- Unforgeability.
- Possession of a handle must mean having access to the resource.
- Ability to pass capability.
- Attenaubility - ability to restrict capabilities, as they are passed from one component to another.

Joe-e is a Java like language that makes use of the idea of objects as capabilities.

1.1 Pitfalls of Ambient Authority

A system enforcing ambient authority does not identify a source of authority with a particular permission. It might grant permission for a subject to access an object, even if the requesting source does not give a permission to do that, but if another source grants the permission. For example, the billing administrator gives permission to compiler binary to read billing.txt file, while the user does not. It is possible for the ambient authority to confuse the administrator's permission with the user's and thus wrongly provide access to the user. We consider how a language supports capability based enforcements in place of ambient authority. We particularly look at an untrusted module that has run in the same address space as trusted code.

2 Enforcement of Restricted Access in Programming Languages

Programming languages (Java in particular) enforce restricted access by the below mechanisms.

Address Space Memory safety and Type safety enforced in the JVM prevent memory errors and isolate separate modules from writing into each other.

Access Specifiers Public, private, protected access specifiers of classes, methods and fields, restrict the visibility and hence the usability of components to a limited scope.

Static Public fields Can be accessed from any method. They are the sources of ambient authority. Constructors representing system resources is also one source of authority, whose exposure/usage must be limited.

2.1 File class

For example consider the File class. Let us assume it has been modified to support least access. We provide only one method that returns a File object, called openAt. Similar to open_at system call, it allows only files in the sub-tree of directory represented by the File object to be accessed.

```
class File{
    public File openAt(String path){
        //check for '..' or absolute path in path
    }
}
class TrustedClass{
   public void init(File f){
        File obj = f.openAt("tmp");
        untrustedObject.doWord(obj);
   }
}
```

A method like getParent(), which gives the parent directory will leave room to overcome the enforcements of this model. Hence such methods should not be available to untrusted modules.

2.2 Getting an Object reference in Java

A method can get Object references only in the below ways.

Call a Constructor. We restrict the constructors made available to an untrusted module, like the File class above.

Returned from a method other than constructor can return object references.

Reference is passed as argument. The trusted code can control this.

Direct access to global static Object. We can restrict access to be granted only to immutable final public members. Immutable objects are deeply immutable.

Thrown as an Exception.

Reflection API can allow one to access non-public members that could not be accessed using the standard API. We could modify the Reflection API to prevent access to certain private/protected members.

By limiting the exposed constructors, and public methods and fields we can ensure that the untrusted capabilities are limited by the parameters passed to a untrusted module. As a side effect capability based programming makes code review of plugin/components easier, to determine which interfaces are exposed by the host system.

3 Attenuating Capabilities

We can wrap objects to create attenuated capabilities. These are especially useful, if an component needs to pass the capability to another untrusted module. Consider a ROFile class which allows only the read operation.

```
class ROFile extends File{
    public ROFile(String fname){
        super(fname);
    }
    public read(InputStream i){
        super.read(i);
    }
    public write(OutputStream i) throws InvalidOperationException{
        throw new InvalidOperationException();
    }
}
```

This wrapper class will not allow write method on the object. We must ensure that the method is not final in the parent class. If a method is final, we can contain a File object in a wrapper class and support all allowed valid method by calling the same on the contained object. The disallowed operations on File are simply not performed by a suitable implementation in the wrapper class. APIs can exclusively use wrapper classes, ensuring that the passed parameters represent capability.