Capabilities in JOS

6.828
December 5th, 2016

Jeremy Rubin

1 Introduction

When developing code, it can be difficult to reason about what a process is able to do, should be able to do, and does (these are three separate things). Often when designing secure software, it is desirable to follow the principle of least privilege, that is, an element can only perform the tasks which is must perform. Capabilities exist as a way of assisting the programmer properly permission and sandbox code. The core idea is that a master environment creates all the resources (such as file descriptors) that will be needed for the rest of the programs life, then removes its (and its children's) ability to create further resources, then passes them to its children as appropriate. These file descriptors can be bundled with a set of permissions such as read only, append only, etc such that the master process can get fine grained control of what the child process may affect. In addition, certain syscalls can be "revoked" from an environment. These practices makes reasoning about application security much easier. For my final project, I augmented JOS with an inter environment capabilities system.

2 Implementation Details

2.1 JOS Limitations

2.1.1 File Descriptors

Because of the microkernel architecture of JOS, it was not straightforward to implement capabilities because capabilities require an *unforgeable token* for permissions storing. In a typical Monolithic Kernel, file descriptors are managed by the kernel and are therefore unforgeable. A user has a number (fd) which can be passed to the kernel for mapping to a struct fd. This is not the case in JOS, here the user themself has access to the FD. This has its drawbacks. This means that the file descriptors cannot be trusted to have accurate information, nor can they be safely shared between envs.
To get around this restriction the design decision I made was to make each server responsible for storing its own permission metadata. The downside of this is that the FD securing code is not centralized, any service wishing to make use of it must track all env's capabilities to its resources. This also implies that a non-service based FD will not be able to be passed in the same way (such as the pipe and console) because they give users direct access to

protected resource (through page sharing or syscalls). To support pipes and console would require creating a device server which can track permissions.

2.1.2 Syscalls

Syscall capabilities in JOS did not present as signifigant of a barrier; it was fairly straightforward.

2.2 Implementation

2.2.1 General

In order to engage capabilities, a user thread calls the syscall *cap_enter*. This enters an environment into capabilities mode and it cannot be exited by it or by it's child processes. In this mode, there are a number of system calls which have effect on the environment. These are implemented as a single syscall and are multiplexed with the first argument. By using these calls an environment can limit it (and it's decendent's) access to various syscalls. Sycall capabilities are copied on fork.

Call	Summary
cap_sys_mask	applies a mask to the effective syscalls set
cap_sys_mask_no	disable a syscall in the effective set
cap_sys_set	enable a set of syscalls in the effective set from the allowed set
cap_sys_restore	reset the effective syscalls to the allowed set
cap_sys_get	retrieve either the effective or allowed set of syscall capabilities
cap_sys_grant	if an environment can call a certain syscall, it can grant it to another

The data on syscalls is stored in two bitvectors kept in the *struct Env*.

In additon, there are a number of file system specific IPC calls. These are handled by the file system server, which independently tracks its permissions.

Call	Summary
cap_inherit	takes an fd and inherits the parents permissions to the FD if the parent has their inheritable bits set.
cap_grant	grants permissions to an FD to another Env.
cap_view	read own env's capabilities on an FD
cap_fd_restore	reset priviledge to an FD to allowed
cap_fd_mask	mask away some FD priviledge

2.2.2 File System File Descriptors

In order to track the information neccessary, each OpenFile has a permissions bitmap which it tracks to make sure any operation can be performed when in cap mode. This is allocated on use, and is only a 4MB overhead.
Additionally, a table of Env id's is kept to make sure that the accessor is from the correct generation (preventing someone from reusing a stale env to gain access to other's fd's).

3 Using Capabilities in JOS

3.1 Typical Use

A typical interaction with JOS Capabilities is given in "user/testcap". The environment should first create it's resources, then should call *cap_enter*, toggle permissions if desired, and then fork. The child should then either inherit the capability, or wait for the parent to grant (as seen in "user/cap_fdblocked").

3.2 Inherit V.S. Grant

There are two models of passing capabilities to another process, granting and inheriting.
In a granting model, a child should check a fd's permission before using, and spin (or do another task) until the parent grants it the right capability (see "user/cap_fdblocked").
In an inheriting model, a parent configures its resource and then the child calls inherit on it, and gets the parents effective capability. Syscalls are always inherited in my design.
The tradeoff is subtle; inheritence is a much more convenient way to give children a permission, however, if the parent desires to change their effective set after a child has inherited they must be careful becasue a child could attempt to re-inherit their effective priviledge. One solution to this would be to add $cap\_(set|check)\_heir$ call and only allow inheriting when explicitly named heir, but this is less convenient.

4 Future Work

4.1 More Server

One of the key issues I ran against is that only one type of devfile is running as a server. Because of this, it is impossible to "truly" protect the resources without having some access control kernel or server. It would be a neat job to go through and add more servers. Indeed, I began doing this for pipes in a branch (having a pipe server which facilitates IPC's) but what I found is that it would be a lot of extra work because each server must implement its own capability scheme, and the shcemes must all be congruent.

4.1.1 Unified Library

The above suggests a better way, which could be to develop a shared server-server which lets you permission objects to envs. This allows for some of the odd tasks to be centrally done (ie, revoking all when the generation changes) as well as lessening the attack surface/need to reimplement.

4.2 Better Inheritence

As suggested above, a nicer inheritence model might be possible and could make capabilities more seemless. Some thoughts are heir setting, ability to revoke a childs ability to request inheritence after the child sends the "done inheriting" signal.