September 8: Capabilities

Scribe: Riyaz Faizullabhoy

1 Motivation for Capabilities

In *The Confused Deputy*, the compiler had two sets of rights: one from the user to access user files such as source code to compile, and another set granted to the compiler intended to access a special statistics file that would collect information about language feature usage.

The compiler was installed in the `SYSX` directory, where it was granted access to write to any file in this directory. Naturally, the statistics file `SYSX/STAT` was also located in this directory such that the compiler (`SYSX/FORT`) could add language feature information. To achieve this, the compiler's privileges were set with a *home files license* that was allowed by the operating system to write to any file in the `SYSX` directory.

A billing information file `SYSX/BILL` was also stored in `SYSX` – due to its sensitive nature, this billing file was not directly accessible to users. However, since the compiler was granted access to the entire `SYSX` directory, users could subvert their access restrictions on the billing information file by using the compiler like so:

```
SYSX/FORT foo.f -o SYSX/BILL
```

The above user-invoked command to the compiler would allow the compiler to first read the user's source code in `foo.f`, but then the compiler's privilege to `SYSX` would cause the `SYSX/BILL` file to be overwritten with the user's binary. In effect, the compiler is the *confused deputy* having to reconcile both its own and the user's set of privileges, but unfortunately allows for unintended behavior.

2 UNIX Solutions for The Confused Deputy

The UNIX operating system offers some primitives that allow for better methods for understanding the confusing privileges presented in *The Confused Deputy*.

2.1 setuid

`setuid` ("set user ID upon execution") - a UNIX construct to allow running a process with the privilege of the owner. The owner is not necessarily the same as the caller, and so `setuid` could be used to escalate user privilege. A similar `setgid` primitive exists to change group privilege.

2.2 Strawman Solutions

While `setuid` provides a promising function, it is limited in this setting. Consider the following scenarios:

2.2.1 User privilege setuid

If we use `setuid` on the compiler to user privilege with only `SYSX/STAT` access, then the compiler itself cannot open user files (and therefore cannot compile the desired user source code).

2.2.2 Root privilege setuid

If we use `setuid` on the compiler to root privilege, then we remain with the original problem of being able to overwrite `SYSX/BILL`.

2.2.3 File open checks

A different approach could be to check every file open in the following manner, in order to ensure that users are allowed access:

1. Check that the user has access to the file
2. If so, `open()...write`

While this approach could notice that the user should not have access to `SYSX/BILL`, it suffers from a TOCTTOU (time of check to time of use) bug! Between the access check and actual access steps, an attacker could change the file and point the open call to a different (and even disallowed) file.

2.3 General Problems

In general, while UNIX provides primitives for and notions of privilege, it suffers from overarching classes of problems that make privilege separation more difficult to understand:

1. **Ambient authority**: Authority and privileges are automatically used by the process due to the context (ex: firewalls, UNIX groupids).
2. **Complex permission checks**: It is difficult to understand and decide when and how to check permissions.

3 Capabilities and Capsicum

3.1 Capability

For a finer-grained and more transparent notion of privileges, we introduce capabilities.

A **capability** is simply a file descriptor. To enable a process to open a certain subset of files, we can pass capabilities (file descriptors) of these files to the process to describe its privileges. Note that for a process to generate a file descriptor, the process must be able to open the corresponding file and convert to a descriptor – this prevents processes from escalating privilege by delegating capabilities to child processes for files it cannot open.

3.1.1 Confused Deputy Solution

One can use capabilities to solve the problem presented in *The Confused Deputy* by wrapping the compiler with a `FORT_FRONTEND` that runs with user privilege. `FORT_FRONTEND` would then spawn

`FORT` with `setuid` user 0, and pass the file descriptors for the input and output files from the original user invocation as capabilities.

This approach would prevent the compiler from overwriting `SYSX/BILL` – consider the user invoking the compiler as follows, attempting to confuse it as before:

```
SYSX/FORT_FRONTEND foo.f -o SYSX/BILL
```

Because `FORT_FRONTEND` runs with user privilege, it cannot actually open the `SYSX/BILL` and generate a file descriptor for it, so `FORT_FRONTEND` cannot grant a capability to `FORT` for the billing file as an output.

3.2 Capsicum

Capsicum is a system presented by Watson, Anderson, Laurie, and Kennaway that builds on top of the mechanism of capabilities to reduce the privilege of untrustworthy applications, effectively sandboxing them. Capscium's architecture borrows elements from previous sandbox approaches while adding the notion of capabilities.

3.2.1 Classical Sandbox Designs

1. **Virtual machines**: emulation of a particular computer system.
   Advantages:

   - Great isolation
   - Can sandbox unmodified code

   Disadvantages:

   - Large CPU + memory overhead
   - Difficult to share data between VMs, and to the user's original OS

2. **Discretionary Access Control (DAC)**: each object has a permissions list, access is determined by the privilege of the process in combination with the permissions of an object. DAC was designed to protect users from each other.
   Advantages:

   - Currently used in UNIX
   - Can customize permissions per object

   Disadvantages:

   - Laborious and difficult to fine-tune permissions of *every* file and ensure granted files are as intended

   Mandatory Access Control (MAC): fixes a policy detailing which application can do which operations on certain files, access is determined by how the policy applies to the given process and object in question. MAC was designed to enforce system policies.
   Advantages:

- The policy is fixed and cannot change at runtime
- Paradigm is used commonly, often in firewalls

Disadvantages:

- Difficult to predict which operations should be allowed ahead of time for a certain application
- May need to specify different operations for different users on the same application
- Cannot support a program to run separate tasks with different privileges on behalf of the same user

3.2.2 Capsicum Design

Capsicum extends UNIX file descriptors in order to use capabilities to grant permission to certain file objects for the sandboxed process. Capsicum also assigns file descriptors to each process, for compatibility with Capsicum capabilities. For Capsicum to access objects with its file descriptors, it passes the file descriptor to the kernel which stores a table for the process ID and looks up the file descriptor to find the underlying file.

Capsicum adds new primitives to the kernel in order to restrict system calls from accessing global name spaces. As a result, Capsicum does not allow use of the `open()` syscall, and instead users must use `open_at(directory_fd, name)` where `directory_fd` refers to the file descriptor capability for a directory. Capsicum imposes further restrictions on the scope of capabilities for the `name` argument – for example, the "`..`" capability is not allowed, such that directories cannot attempt to traverse past capabilities for a given directory, and absolute paths are also disallowed.

Capsicum introduces two different constructs for sandboxing processes: one for sandboxing the current process, and another for starting a different process in a sandbox:

1. `cap_enter()`: Sets a flag for capability mode for the current process, only allowing this process to open files for which it has already received a capability
2. `lch_start()`: Starts another process in capability mode, and passes it `fd_list`: a list of file descriptors that the other process is allowed to access. In this case, the caller can have a superset of capabilities from the new process. Moreover, spawning a new process prevents the new sandbox from accessing the memory of the caller.

3.2.3 Sandboxing a component with Capsicum

In order to use Capsicum, components must be modified to be compatible with capabilities and abide by Capsicum's restrictions. The necessary transformations can be summarized in three steps:

1. Component cannot use pathnames or other global name spaces
2. For every necessary directory and file to the component, the file descriptor must be opened ahead of time (to get the capability)
3. Component must use `open_at()` to interact with these files

In the paper, the authors implement Capsicum for a variety of programs including tcpdump and Chromium. In the instance of Chromium, `lch_start()` can be used to spawn each rendering engine into a separate Capsicum sandbox.

3.2.4 Evaluation of Capsicum

Advantages:

- Low performance overhead
- Fine tuned privileges, can run different tasks with different capabilities

Disadvantages:

- Need to change application code to be compatible (ex: using `open_at()`)
- Limited to FreeBSD