CSE 509 Lecture 22

Prof. Rob Johnson, Scribe:Arun Rathakrishnan

November 26, 2013

1 Capability Based Systems

Capability based systems provide a mechanism for granting acccess to resources through,

- providing unforgeable references/handles.
- possessing handle equates to having access to resource.
- handles may also specify operations that a holder can perform on the resource.

Usually the operating system can be counted upon to provide references that are representative of some actions on a resource and ensure that the reference can not be forged by a user process.

2 Hardware Capability Based Systems

In native client systems, the untrusted code can be given more fine grained access to resources in the trusted region if the hardware supports fat pointers and typed/ tagged memory (which differentiates a pointer data from non-pointer data in the memory). The OS has the capability to provide boundaries of stack, heap and data section in pre determined register locations and provides a restrict instruction for selecting a smaller pointer address range within the valid range.

```
store 0, A
store 1<<64, A+1
store p, A+2
load A, %p0
```

This code causes assigning the valid address range to entire pointer range. But the addresses A to $A+2$ are tagged as non-pointers which will cause the load instruction into a pointer type register to fail.

With such a system in place a trusted entity can pass pointers to addresses in trusted region, to an untrusted code. The untrusted code can access addresses which are provided as arguments, respecting the bounds placed by the fat pointers but not any other region in the trusted entity. The caller can specifiy if the pointers when deferenced can be read from or written to by extra permissions, passed alongside the arguments in the call.

3 Capability Based Systems used by Operating System

3.1 Passing file descriptors across processes

A process can pass a file descriptor in its file descriptor table to another process, via the kernel through pipes, which causes the second process to incorporate the file descriptor entry in its own file descriptor table. Expect for the file descriptor index, both processes have the copied file descriptor representing identical file state. It should be noted that the permissions obtained in the copied file descriptor, may not be achieved through a regular open by the second process.

3.2 File Descriptors in Unix

With respect to capability, we can evaluate the capability enforced by file descriptors based on,

Forgeability

- Can only be created by the operating system.
- Can not be accessed across address spaces or modified by a process bypassing the kernel.

Holding Reference

- Specify access rights of a file completely. No other information is required further for accessing a file, once a file descriptor is held by a process.

Delegating Capability

- Can be delegated through pipes as noted above.

3.3 Confused Deputy Problem

Consider an operating system which enforces unix like file access permissions for users and groups. In addition, an administrator can set a list of files that an executable file can open. In this setting, consider a compiler program which takes a source file as input, and writes the compiled binary to an output file. The user running the compiler program must have permissions to access, the input and output files. The compiler also writes usage information to a billing file, for which the permission is granted by an administrator. No user of the compiler is granted access to write the billing file.

```
compile(){
    fd1 = open(input_arg, "r");
    fd2 = open(output_arg, "w");
    read(fd1, ...);
    process(fd1);
    truncate(fd2);
```

```
        write(fd2, ...);
        fd = open("billing_file", "w");
        write(fd, ...);
    }
```

For successful compilation the user must have correct permissions to input_arg and output_arg. Even though the user does not have permission to write to "billing_file", during open OS accepts the extra permissions granted to compiler binary, to open and write to that file.

Assume that a user passes, "billing_file" as output_arg. Now the OS confuses the extra permissions granted to compiler binary, with the permissions of the user and lets the file be truncated, leading to the actual billing_file being overwritten.

The compiler program serves as a deputy to two masters, users - who use the compiler, and administrators - who perform the billing. Here the deputy used the permissions granted by one master (access to billing file), to perform an unauthorised operation (user does not have access to billing file) for another master.

3.4 Establishing capabality boundaries through exec

exec system call flushes the address space of a process and overwrites it with code in the argument provided. However, it does not change the file table descriptor. Consider the code snippet below.

```
fd1 = open(...);
fd2 = open(...);
exec(p2, "--input1 = fd1", "--input2 = fd2");
```

The process p2 may not have access to fd1 and fd2, but on exec it can use the descriptors with the same permissions the original process possessed. In terms of capability, having access to file descriptors means that they can be used by p2. We can use this approach to solve the confused deputy problem.

Here's the code for "compiler.c" which opens the input and output files, only if the user possesses the permission to do so. This binary does not have special permission to open "billing_file".

```
fd1 = open(input_arg, "r");
fd2 = open(output_arg, "w");
if(fd1 < 0 || fd2 < 0) exit(1);
exec("_compiler", "--input = fd1", "--output = fd2");
```

Here's the code for "_compiler.c" whose binary is granted write access to file descriptor of "billing_file" through a special mechanism, different from open system call.

The code is execed and can access file descriptors fd1 and fd2. It performs the actual compilation and writes to the billing file.

```
fd1 = parse_args(...);
fd2 = parse_args(...);
billing_fd = get_fd(...);
...
read(fd1);
...
write(fd1);
write(billing_fd, ...);
```

By separating the permission granted by different masters across different permission domains, we can solve this instance of confused deputy problem. The compiler may need to read several include files, and library code independent of user input. A capability based system can provide a similar mechanism, different from open for the compiler to read these files, thus making the compiler code, independent of the ambient security model enforced by the file system.

3.5 Secure file dialog

Based on the ability to pass file descriptors, we can allow untrusted applications that can not call open system call directly, to access files through a Secure File Dialog server. The secure file dialog server receives requests from the untrusted application and then shows a prompt to user asking if access to the file can be provided. If the user agrees, the server then opens the file (it has the necessary permissions) and passes the file descriptor through pipe to the untrusted application. Here too, by just holding file descriptors, the untrusted applications gains capability to access the desired file.

4 Eliminating Ambient Authority

Consider a scenario in which a process is provided capability to access only parts of the file system indicated by file descriptors. We can assume that only files in the subtree of the file descriptors (obtained through a open like mechanism) at the start of the process can be accessed. We can borrow ideas from open_at system call.

```
open_at (dir_fd, path, ...);
```

path is a relative path, present in the path represented by dir_fd, which provides the file descriptor of a parent directory. If open_at can enforce that file descriptors returned by it are strictly in the subtree of dir_fd path, then we can use it for supporting the intended capability based system. There are two general principles that must be followed in capability based systems.

- Global access to resources must be prevented from being held.

- Capability leaks must be avoided.

Thus we must find a way to check paths which are absolute (global access) and use ".." to navigate to parent directory (capability leak).