CS261 Scribe: Recent Approaches to Sandboxing

Lecturer: David Wagner Scriber: David Zhu

October 2, 2007

Introduction

Last lecture, we discussed Janus, which uses system call interposition to achieve software sandboxing isolation. We talked about a few problems that Janus had, particularly race conditions that can potentially allow malicious code to break software isolation property. Since then, a few more sandboxing projects have been implemented to address such issues. Today, we are going to look at a few representative ones and compare among them and with the Janus approach.

Systrace

Systrace has a similar architecture to Janus. However, it addresses the argument race conditions by making a copy of the parameters and pass the copy to the reference monitor to check against its policies. It also performs atomic file canonicalization in the kernel and pass the result of that to the policy check. Systrace is successful but not widely adopted, partly due to the large number of changes required in OS kernel. Currently, it is integrated into NetBSD and available on several flavors of Unix systems.

Seccomp

Seccomp is a sandboxing mechanism that was originally designed to enable grid computing. Ideally, average users should be able to run some compute-oriented task on their local machines and report back the results. However, this has to be done in a secure manner. Seccomp provides the sandboxed process three system calls only. (read(), write(), exit()) The parent Seccomp process opens all the file descriptor the sandboxed child needs and pass them to the child. Because Seccomp is only suitable for a particular kind of work load, it is not flexible as other sandboxing techniques.

Ostia

Ostia is a sandboxing tool designed by Tal Garfinkel after his detailed analysis of the Janus implementation. He concludes that the a binary decision based

on file name is difficult. Its sandboxing is based on a proxy-like approach. As shown in Figure 1, the proxy is responsible for forwarding all syscalls for the sandboxed application. For performance reasons, the sandboxed application is allowed to make several system calls such as read/write/dupe. However, since all open requests are done by the proxy and passed to sandboxed application, this is safe. There are potential issues with ownership of the file. When a

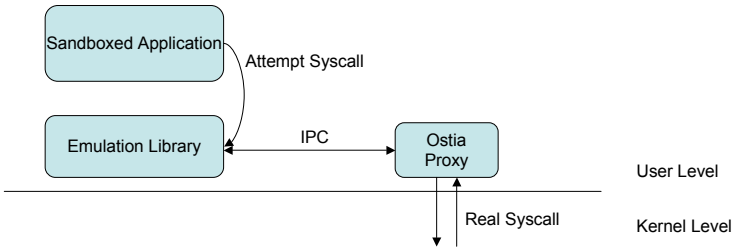

Figure 1: Ostia Architecture

file is created by Ostia for the sandboxed application, the system will record the incorrect user as the owner/creator for the file. The operating system can potentially allow Ostia to open files that sandboxed application should not have access to based on the user permission. This can be solved in the following two ways.

1. Always run Ostia and the sandboxed application in the same user name and password.

2. Ostia keeps a list of users on the system. Ostia can do some permission checking for that particular user and disallow file access.

Ostia pays a price for the ability to proxy all system calls. Many possible race conditions result from symbolic links in Unix file system. To effectively determine whether a file access should be granted, the Ostia reference monitor

needs to traverse the symbolic link at user level. This is not a trivial task and a simplified version is explain below in psedo ML format.

```
(* Takes a directory fd and a list of path components;
  traverses path, relative to that directory. *)
lookup : (fd, string list) -> fd
lookup(fd, ".."::_) = throw "Not implemented"
lookup(fd, x::nil) = openat(fd, x, O_RDWR | O_NOFOLLOW)
lookup(fd, x::xs) =
 try {
   // openat(,O_DIRECTORY) throws if it's not a directory
   lookup(openat(fd, x, O_DIRECTORY | O_NOFOLLOW), xs)
 } catch {
   // readlink() throws if it's not a symlink
   lookup(resolve(fd, readlinkat(fd, x)), xs)
 }

(* Takes a directory fd and a relative pathname; traverses path. *)
lookup2 : (fd, string) -> fd
lookup2(fd, path) = lookup(fd, split(path, "/"))

(* Takes the current working directory and an absolute or relative
  pathname; resolves it to a fd. *)
resolve : (fd, string) -> fd
resolve(_, "/"+path) =
 resolve(open("/", O_DIRECTORY), path)
resolve(cwd, path) =
 lookup2(cwd, path)
```

Unix file open operation has a flag `O_NOFOLLOW`. However, this flag only applies to the last element in the path. For example, in the path "`a/b/c/d`", `a`, `b`, `c` will be followed but `d` will not be. Ostia solves this problem by getting file descriptor one component at a time. Because Unix file descriptors corresponds to an inode in kernel representation, once a file descriptor is obtained by Ostia, it will not be fooled by external modification to the file system. If symbolic links exist in the path, they will be recursively resolved before the rest of the path is traversed. As shown above, "`..`" is not supported. To support this, the path traversal code needs to remember a stack of path traversed so far. Instead of asking the operating system for the parent directory, it will simply go to the last node traversed for its parent. User-level path traversal has other applications in addition to sandboxing. Consider an application that periodically deletes the temporary file directory. The naive implementation will follow all symbolic links and potentially delete unintended files. The only effective way to implement such utility is to use user level path traversal.

Plash

Plash is also a proxy-based solution to sandboxing. Its approach is very similar to paravirtualization. It implements its own file system interface, which is a subset of Unix file system interface. The emulation library translates file systems calls into PlashFS API calls and Plash will translate them back to Unix system calls. (Figure 2)

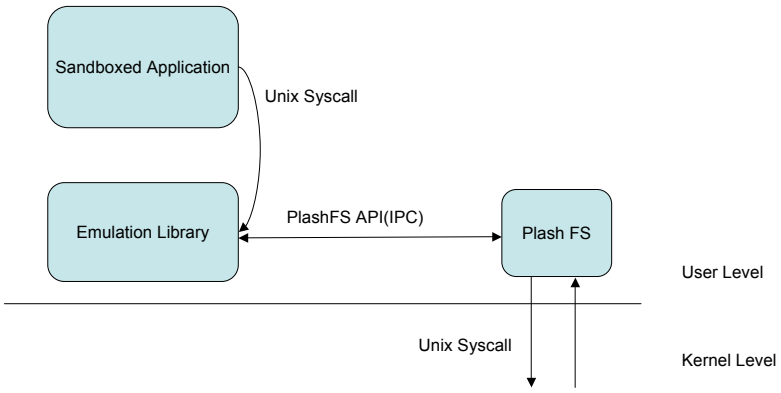

Figure 2: Plash Architecture

global namespace	sandbox application namespace
`/jail/mail/usr`	`usr`
`/jail/mail/var`	`var`
`/tmp/mail`	`tmp`

Table 1: Namespace remapping

PlashFS remaps the global namespace to give the sandboxed application a restricted view on the file system. Table 1 is an example of such mapping. In PlashFS, file system is modeled with an object oriented tree structure as shown in Figure 3. This file access model has several advantages.

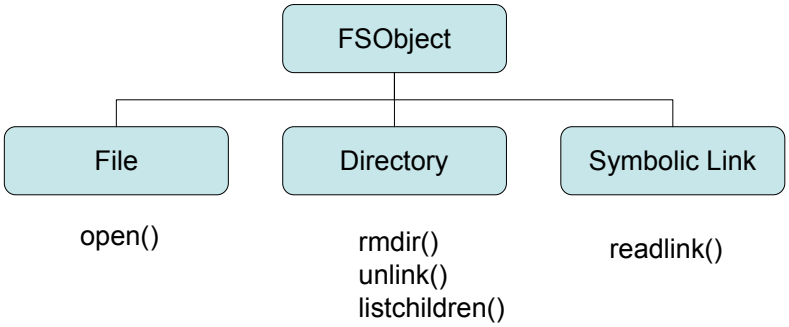

Figure 3: File Access Model

1. No support for `../` in the file system, so the sandboxed application cannot escape the jail
2. PlashFS API is the trusted computing base. The security of the system does not depend on the emulation library.
3. Can easily implement read-only by subclassing existing classes, and add or remove methods as needed.

However, Plash has bad isolation properties. It makes no attempt to restrict network access. File system protection is only one part of a successful sandbox.

Takeaway Points

There are a few takeaway points in this lecture. They are summarized below.

- Paravirtualization
- Interfaces for Interposition
  - Avoid implicit state

– Use message passing to avoid TOCTOU vulnerability
– Design interfaces with limited expressibility