September 23, 2008 Scriber: Himanshu Sharma

Shadow State

The lecture started with the discussion of last flaw of Janus – shadow state.

- described in the context of Janus, but a more general potential issue.
- Suppose, Janus wants to restrict network connectivity and for that purpose it has a policy that outbound connection can only be on Port 25. In BSD Unix, to bind a TCP socket to port 25, two calls are required

```
fd=socket(TCP);
connect(fd,1.2.3.4,25);
```

- To enforce the policy, we have to see the combination of the system calls. Remember the protocol associated with the socket, see the port number, and then decide if its allowed/not allowed.
- Now, suppose the code is

```
fd1=socket(TCP);
fd2=socket(udp);
dup2(fd2,fd1);
connect(fd1,address,25);
```

  If Janus does not know about dup2 system call that copies file descriptors then its state will fall out of sync with the state in OS. This becomes very difficult, since Janus must know everything about every system call.
- Operating systems often perform non-trivial processing to interpret system call arguments. Duplicate implementation of kernel algorithms, can become inconsistent if there is a kernel upgrade. Systems dealing with shadow state must ensure that it maintains the shadow exactly the same as the real state.

Some suggestions to deal with shadow state –

- Query methods
- Quality testing
- Context free meaning
- Virtualization

Then we looked at other representative sandboxing projects – Systrace, Secomp, Secomp++, Ostia and Plash.

1. Systrace

- Its architecture is similar to Janus.
- Systrace prevents argument, file race conditions (an adversary may redirect the file access by changing a component in the filename to a symbolic link after the policy check) by replacing the system call arguments with the arguments that were resolved and evaluated by Systrace. The replaced arguments reside in kernel address space and are available to the monitored process via a read-only look-aside buffer. This ensures that the kernel executes only system calls that passed the policy check.
- A dedicated kernel module is used for tasks such as system call interposition, canonicalizing pathnames, fetching system call arguments etc.

- Downside: Requires kernel module.

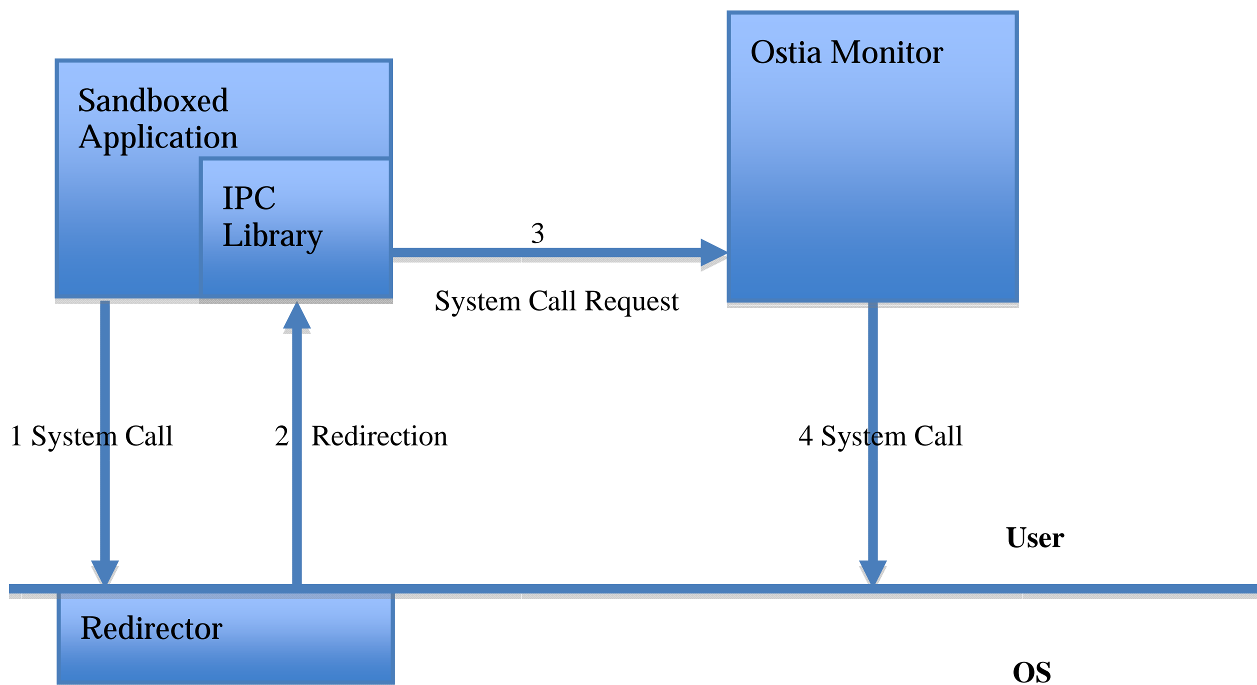
2. Secomp

- Secomp is a simple, Linux kernel extension meant initially for grid computing.
- Users should be able to run some compute-oriented task on their local machines and report back the results, in a secure manner.
- Secomp provides the sandboxed process three system calls only (read(), write(), exit()). The parent Secomp process forks a child process that runs in Secomp mode.
- Downside: Can't fork, do I/O.

3. Secomp++

- It can be thought of as an extension to Secomp allowing readv(), writev(), fork system calls, sending file descriptor apart from read(),write(), exit().

4. Ostia



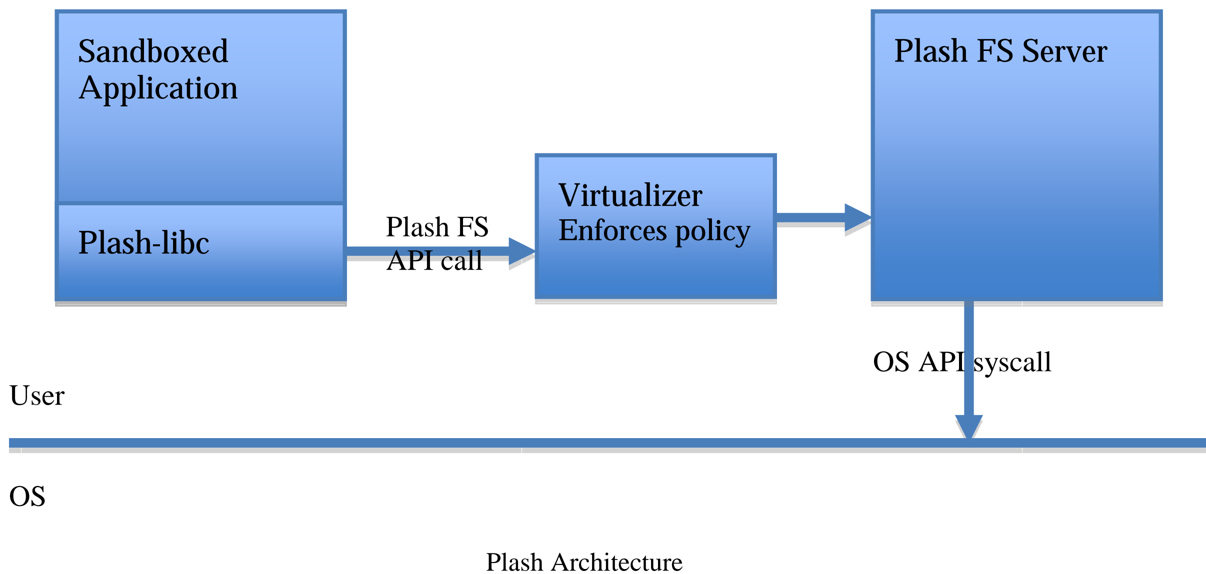
Ostia Architecture

1. Sandboxed application makes a system call
2. Redirector, a small kernel module redirects calls to an IPC Emulation library.
3. IPC Library transforms the system call into a request to Ostia Monitor

4. If the system call request is permitted by policy, Ostia Monitor accesses the requested resource (possibly executing one or more system calls) and returns the result (e.g. return code, descriptor) to the client.

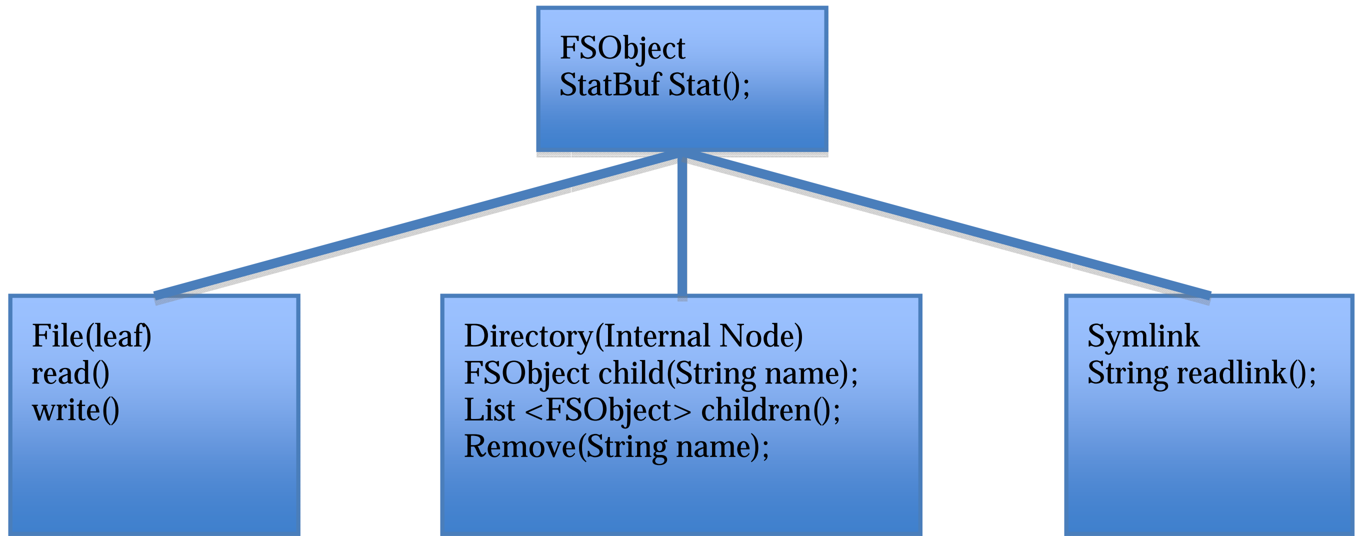
- For performance reasons, calls that do not provide access to sensitive resources but merely use resources the client has already obtained (e.g. read, write) are executed directly by the client.
- To ensure that file system requests are interpreted correctly, monitor emulates the relevant file system state of sandboxed process before interpreting a request.
- Solves most race conditions, such as argument races because an external process cannot modify the state (file descriptor space, current working directory, etc.,) held exclusively by the Ostia monitor. Ostia sandbox makes all accesses to resources itself; so all accesses can be performed in a manner respecting OS conventions for providing race-free operations on the file system.
- Downside: makes more system calls, poor performance

5. Plash



Plash Architecture

- Plash virtualizes filesystem accesses
- Plash File server mediates the operations by sandboxed process on the file namesace
- Plash-libc modifies file systems calls (such as open()) so that they make remote procedure calls (RPCs) to Plash FS. It disables these system calls by putting the sandboxed process in a minimal chroot() jail, and by running the process under a dynamically-allocated user ID and group ID.
- Plash will translate them back to Unix system calls.
- When a sandboxed process sends a request to open a file to the FS Server, it can grant the request by sending a file descriptor in reply
- Most frequently called system calls, such as read() and write(), are not affected.



Object Oriented API

- file server, uses its own filesystem object abstraction internally, has its own functions for resolving pathnames and following symbolic links which do not use the kernel's facility for following symbolic links
- interprets `..' itself rather than using the `..' parent directory facility provided by the underlying filesystem; parent of a directory is the directory that it was reached through, after symlinks have been expanded.
- provides functionality similar to chroot(). Facilities for creating new namespaces for use with chroot are limited. Plash moves the interpretation of filenames into user space allowing the creation of file namespaces to be more flexible.
- read-only can be implemented by subclassing existing classes, and adding or removing methods as needed.

Downside: Plash does not prevent a process from connecting to or listening on network sockets.
Solution: prevent a process from doing connect() and bind() system calls.

Paravirtualization