Scribe Notes: Janus

Scrivener: Peter Alvaro

September 21, 2009

1 Sandboxing helper applications using Janus

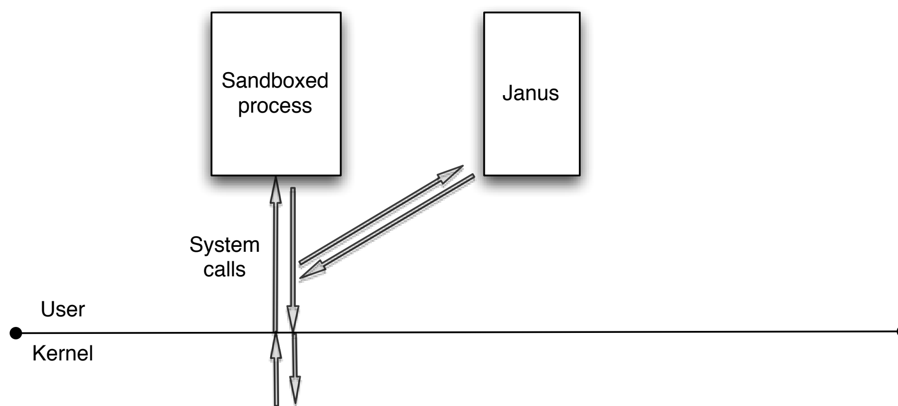

Figure 1: Janus architecture.

Bottom line: the containment approach limits the harm that an attacker can do. This means that it does not:

1. protect against the vulnerabilities themselves
2. detect / repair compromised systems

2 Problems with the basic Janus approach

2.1 Portability

Using the tracing facilities provided by an OS means poor portability. Originally implemented on solaris: the linux port was difficult.

Worse, the system call interposition approach won't work on Windows at all. UNIX has O(100) syscalls, while Windows has thousands.

2.2 Policies

Authoring policies is hard, and easy to get wrong. Applications ask to do things they don't need to do: sometimes when the call is forbidden the application can proceed anyway. An example is a routine that traverses up from the PWD to root and back to determine the fully qualified path.

2.2.1 An idea: what if we design the policy in two phases?

In the first, we run the application and log the system calls,"learning" the minimal set of necessary calls. Then we design our policy around this. The problem is that coverage is hard to get right. "Training mode" needs to exercise every possible feature of the app. And we critically assume that we are not attacked during training!

2.2.2 Idea: ask the user

Windows personal firewalls take this approach: "should I allow this system call?" Anecdotally, users get trained to "swat away" such dialog boxes, limiting the usefulness of the approach.

2.2.3 Idea: Static Analysis

What does the code access? In general, allow accesses that are consistent with the source code. This may lead to permissive policies.

2.3 TOCTOU vulnerabilities with regard to syscalls

Example: take a syscall with a string argument (ie open). The process passes an address to the kernel, so that it can copy the string to kernel buffers when it wants to access it. Janus itself makes a copy of the string. Then later, if the call is allowed, the kernel makes another copy!

How could this be exploited? Another thread in the helper process gets scheduled and changes the string between the copy to Janus and the kernel copy. Or on a multicore system, the process itself does so concurrently. Or a signal is sent to the process, and a handler gets executed that modifies the string. In general, we must try to eliminate all sources of concurrency to address this :(

Sidenote: it was obviously necessary to prevent sandboxed processes from using the tracing infrastructure; otherwise an app could register its own handlers and subvert everything.

2.4 Symlinks are tricky

We'd like to allow RW access to /tmp/*. But what about

```
/tmp/foo      -->     /etc/passwd
```

This is hard because we're operating at the system call level (ie looking at names and not inodes). We can stat files to check where they really point, or canonicalize names, but these checks will in general be vulnerable to TOCTOU tricks.

2.4.1 Idea: track safe symlink creation

When a helper process creates a symlink, check RW access to the pointed-to file. If allowed, allow the creation (because this link adds no new privileges).

2.4.2 Theoretical problem with this idea

A non-sandboxed process creates a file:

```
/tmp/bar     -->      /etc/passwd
```

Then our sandboxed process creates:

```
/tmp/foo     -->      /tmp/bar
```

Allowed! Our thread model assumes no colluding non-sandboxed (ie, trusted) processes.

2.4.3 Real problem with the idea

```
mkdir /tmp/etc
touch /tmp/etc/passwd
mkdir /tmp/d1/d2
cd /tmp/d1/d2
ln -s ../../etc/passwd foo
cd /
mv /tmp/d1/d2 /tmp/d3
```

Now we have:

```
/tmp/d3/oo     -->     /etc/passwd
```

Basically, this approach to symlinks is broken. We *should have* identified the safety invariants and ensured that they are preserved across *all* operations. In this case, the invariant we wanted is that every symlink resolves only to files to which the sandboxed process has access. This invariant was preserved in the given design for symlink creation, but not for rename.

Other simpler but more restrictive approaches could have involved disallowing relative symlinks, the rename operation, all symlinks, etc.

3 TOCTOU in the FS

Just how easy can it be to exploit time of check to time of use vulnerabilities? In the right environment, almost arbitrarily easy. Take this "maze":

```
# MAZE1
n                 -->      x/x/x/x/x/x/x/ [...] x/x/l
l                  -->      some_harmless_file
```

An application checking the safety of the link begins traversing the x's using stack calls. Meanwhile, a colluding malicious process changes where n points (it cleared caches before calling the checker app :)). Because paths can be 4096 bytes, this traversal could follow 2048 directories (so possibly this many seeks). This buys 10 - 100 ms to mount an attack.

What about this one?

```
[MAZE1]    -->     [MAZE2]    -->     [MAZE3]
```

Linux allows up to 40 symlinks in a path!

Observation: in database-ese, the TOCTOU class of vulnerabilities are examples of RW-conflict schedules. A transactional filesystem would address these issues (at the cost of overhead).

4 Ostia

Why not have Janus execute the system call itself, and define an API to communicate with sandboxed processes? Tricky only in the case of calls that affect the user process state (malloc, etc).

This is the approach Ostia takes: replace the stdlib with functions that make IPC calls to the framework instead of syscalls. This amounts to virtualizing the OS at the system call level, and as such, entails duplication of OS functionality (like data structures for FDs, etc). Still need to deal with calls that affect the process, and reflect them to the sandboxed process. For getpid() this is trivial. fork() is harder... What about malloc?

5 Shadow State

A particularly difficult issue is shadow state. Take the following (pseudo) socket API calls:

```
int sd = socket(TCP);
int ret = connect(sd, host, 25);
```

What if our policy wanted to prevent UDP connections to port 25? Since we're sniffing at the syscall level, we don't know what state is associated with 'sd' in kernel structures. To capture the state and distinguish between protocols to implement our policy, we'd need to duplicate that state in Janus and provide efficient lookups. But doing so is non-trivial; if we merely recorded the arguments to socket() and checked them on calls to connect(), something like this could happen:

```
int fd2 = socket(UDP);
dup2(fd, fd2);
```

Oops, forgot to track dup2(), so our policy is violated again (internal hash table is now stale). There are in fact several ways to dup socket handles (ie fcntl() w/ F_DUPFD) In general, the challenge is keeping the framework's "shadow state" in sync with the OS state, which necessarily means some duplication of functionality in userland.

6 Plash

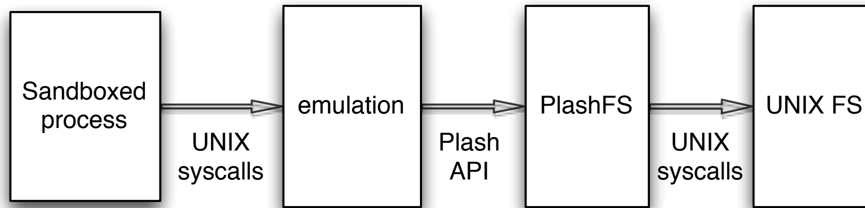

Figure 2: Plash architecture.

Idea: an API designed for interposition, providing an Object-Oriented view of file and directory objects. Use references instead of descriptors, and object methods for read, write, ls, etc. Now we can, for example, write method wrappers to special-case get_entries(), etc.

This is an example of "paravirtualization:" translating a hard-to-interpose API into a more manageable one, performing the interposition, then translating back.