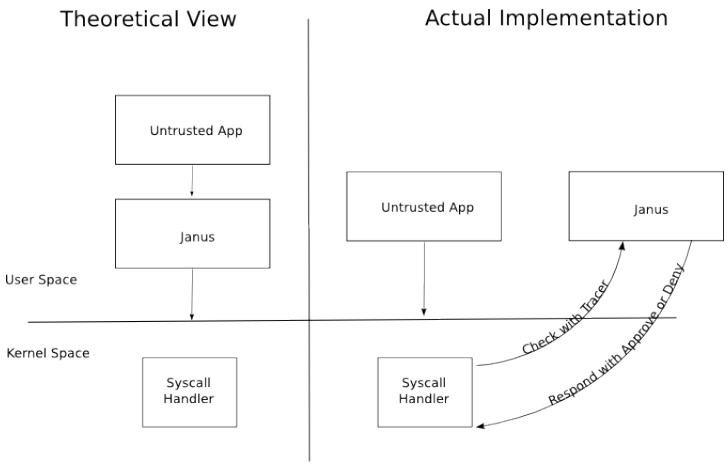
CS261 Notes - Sandboxing (Janus)

Lecturer: David Wagner Scribe: Barret Rhoden

2007-09-20

1 Description of Janus

The original Janus was a user-mode tool that would intercept system calls from an untrusted application. The goal was to have Janus act as a reference monitor directly between the application and the operating system; the actual implementation had Janus run as a separate process that would use the tracing mechanisms of the OS to intercept the syscalls. Janus would either approve or deny the syscalls based on a set of policy modules.



This method introduces two new techniques for sandboxing: virtualization and interposition. Janus attempts to virtualize the OS syscall interface, and interposes itself at that layer.

Compared to Software-based Fault Isolation, Janus provides the ability for the sandboxed application to make system calls. SFI implied that it had no syscall interface, though the SFI technique could be used to rewrite every syscall that the untrusted application makes to point to its policy routines.

Janus is especially suitable for applications such as mobile code, least privilege for network daemons, privilege separation, and programming contests (you can't trust CS students).

2 Problems with Janus

2.1 Portability

Janus is not easily ported to other operating systems, since it must be customized for each OS to handle the syscall interposition.

2.2 Creating Policies

Every application needs to have a policy. As a user, you may want to use Janus for numerous applications. Someone needs to write those module policies, and this is often beyond the scope of a regular user. This issue is exacerbated by applications which interact with the OS in undocumented or unexpected ways. The application developers are the best equipped to write these policies, but getting them to write the policies involves numerous issues such as administration, motiviation, and resources.

Another alternative is to use a "Learning Mode" in which the application runs with no restrictions and Janus logs the behavior to build a baseline policy. After a suitable learning period, Janus can enforce those rules. The problem is that it may be difficult to generate a thorough coverage of the syscalls the code may make over its lifetime. Additionally, the system must be secure during this learning period so that it does not learn attacks. For instance, it would be a bad idea to have your system face the Internet and "learn" for a month.

2.3 Static Grants

Consider a text editor. What can it open, based on policies? A user would like to open any file with it, but only files that the user explicitly wants. This is difficult to express for a policy that is set in advance. A solution would be to use Dynamic Grants, such that the editor is allowed to access files from its policy as well as any files passed in on the command line. Still, this is difficult for a user who wants to use a command from within the editor to open files. Janus ignored this issue and simply allowed access to all of /tmp, but this is a serious hurdle for using Janus with every-day applications.

2.4 Composition Problem

Multiple untrusted processes with different capabilities can collude to circumvent security policies, if they are allowed to communicate. An example

involves a process that is not allowed to access the network when it is accessing a Top Secret file. If there is another cooperating, evil process that has network access, then the first process can pass the contents to the networked-process, which can then communicate. There are many variations on this theme. A moral of this is that security systems need to take a close look at concurrent processes and threads.

2.5 OS is part of the TCB

The operating system is part of the Trusted Computing Base, meaning that any errors in the OS can result in a failure of Janus. Consider the event of a bug, such as a null pointer being dereferenced in the kernel. A malicious process can use this bug to execute arbitrary code and break out of Janus's sandbox. For example, say there is a part of the kernel code that has a function pointer such as:

(p → ops → read) (... arguments); where p → ops == NULL

The kernel will try to offset from $p \rightarrow ops$ by whatever *read* is, for example 0x20. Since $p \rightarrow ops$ is 0, it will try to execute the function pointed to by the address 0x20. The hack is to mmap the 0th page block and put a function pointer to code of your choosing in 0x20.

3 Holes in Janus

3.1 TOCTTOU Syscall Arguments

Janus is susceptible to time-of-check-to-time-of-use attacks. Consider an open() syscall. The kernel copys-in the filename from the untrusted application's memory. The kernel then sends to the Janus process a pointer pointing to that spot in the untrusted memory. Janus checks the contents of that memory and makes a determination of whether or not to allow the access. The problem is that a concurrent thread that has access to that memory can change it before Janus checks it. This requires the other process to be context-switched in before Janus checks. To extend the time window of the attack, the attacker could put the filename in memory that crosses a page break. Janus will block on the page fault, and then a malicious, concurrent thread can change the part that is in memory.

3.2 Symlinks

Symlinks present a host of problems. One must evaluate what the symlink points to. As a simple example, consider your policy allows access to /tmp/* via the open() syscall. An attacker can make a symlink such as /tmp/foo

→ /etc/passwd. The link passes the check, but you are still hosed. This leads to the next problem.

3.3 TOCTTOU Filesystem

Janus is further vulnerable to TOCTTOU attacks within the file system. One attack involves making a maze of symlinks and directories, such as n → x/x/x/x/.../x/x/harmless-file. While Janus hops down the directory tree, another thread can change n to point to somewhere that violates the policy. To increase the window of the attack, the initial process can also make sure the IO buffer cache is empty of any inodes for that directory tree so that Janus must block for IO.

However, some file systems block inodes together, and this may not give enough delay. An alternative is to make a longer maze with more symlinks, such as n → x/x/x/x/.../x/x/link → /y/y/.../y/y/link → etc → harmless-file. On Linux, you can have up to forty links, each being 4096 bytes long.

Another issue with the filesystem and symlinks is in relative links. Consider /tmp/etc/passwd and /tmp/d1/d2/link → ../../etc/passwd, which are normally allowed. Rename /tmp/d1/d2 to /tmp/d3. Your result is /tmp/d3/link is still ../../etc/passwd, but now it points to the real /etc/passwd. It is hard to develop a policy that maintains what is invariant, especially in a file system.

3.4 Shadow State

Janus maintains state for each process it traces. Some of its policies' responses depend on the current state Janus is tracking. This state is supposed to mirror the real state that the OS maintains. For instance, Janus may have a policy that allows a TCP socket to bind to port 25, but denies a UDP socket to bind to port 25. In UNIX, it takes two calls to do this operation.

```
filedesc = socket( ... tcp ...);
connect(filedesc, address, port);
```

Janus must track that *filedesc* is a TCP socket. The problem is that there is a syscall called dup2 that copies file descriptors. If Janus does not know this, then its state will fall out of sync with the real state maintained by the OS. This becomes very difficult, since Janus must know everything about every syscall. Here's how the attack works:

```
fd1 = socket( ... tcp ...);
fd2 = socket( ... udp ...);
dup2(fd2, fd1);  // closes fd1 and duplicates fd2 on to fd1
```

```
connect(fd1, address, 25); // Janus thinks fd1 is TCP and allows this
```

The moral of this story is that whenever a security system deals with shadow state, it must be very careful and ensure that it maintains the shadow exactly the same as the real state. To make matters worse, Janus cannot ask the OS about its state. Another point to consider is that the presence of a shadow state implies that your security policy may be at the wrong level. The real level your system wants to monitor and interpose on is actually lower, which is what your shadow state is trying to track.