Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools

Tal Garfinkel
talg@cs.stanford.edu

Computer Science Department, Stanford University

Abstract

System call interposition is a powerful method for regulating and monitoring application behavior. In recent years, a wide variety of security tools have been developed that use this technique. This approach brings with it a host of pitfalls for the unwary implementer that if overlooked can allow his tool to be easily circumvented. To shed light on these problems, we present the lessons we learned in the course of several design and implementation cycles with our own system call interposition-based sandboxing tool. We first present some of the problems and pitfalls we encountered, including incorrectly replicating OS semantics, overlooking indirect paths to resources, race conditions, incorrectly subsetting a complex interface, and side effects of denying system calls. We then present some practical solutions to these problems, and provide general principles for avoiding the difficulties we encountered.

1 Introduction

The explosion of interest in OS-based intrusion detection and confinement in recent years has brought with it a host of approaches to security that rely on system call interposition[1] for confinement [24, 2, 5, 3, 9, 11, 19, 14] and intrusion detection [15, 30, 21, 27]. The system call interface is a compelling boundary on which to interpose , as it allows virtually all of an application's interactions with the network, file system, and other sensitive system resources to be monitored and regulated. Unfortunately, building tools that securely interpose on this interface can be quite subtle. If the implementer is not careful, his tool can easily be circumvented.

Over the course of several design and implementation iterations to build the current incarnation of Janus, our own system call interposition-based application sandboxing tool, we have encountered a host of mistakes and challenges related to building this class of systems. Given the perspective of this experience, we believe that past work in this area has often painted an incomplete or inaccurate picture of the steps required to ensure the correctness of an interposition-based tool. Often the details of the interposition mechanism, tool architecture, and underlying OS semantics are only given cursory consideration. Most work has instead focused on higher-level issues such as policy specification [3, 8] and analysis of system call patterns [21, 15] for intrusion detection. Work that has focused on mechanism [13, 11, 16] has been more concerned with general design issues and performance than with the pitfalls of the mechanism's use in practice, where interactions between interposition mechanism, tool architecture, and the system call API must be taken into account.

We have found that the problems that arise from these interactions can be quite subtle. The Unix API is tremendously complex, as are the specifics of process-tracing interfaces. Because work on system call interposition often fails to take these details into account, its results may be based on assumptions that are incomplete, unjustified or simply wrong. Overlooking a "minor detail" can often undermine a tool's security. The changes that must be made to a tool's architecture to deal with a "minor detail" can be significant, and can have important implications for the system's performance and the range of security policies it can enforce.

Presenting a detailed analysis of Unix for the purposes of building interposition tools is far beyond the scope of a single paper. In this work we will instead present the pitfalls and problems we encountered in the course of building Janus, the solutions that we explored, and the principles we distilled out of our experience for designing, building, and auditing this class of systems. Although these lessons are presented in the context of our work on a sandboxing tool, most of the problems and solutions have equal relevance to intrusion detection tools that rely on

[1] System call interposition is sometimes referred to as system call interception or system call tracing. The latter term typically refers to an architecture where the system call interface is passively monitored.

system call interposition.

Our paper will proceed as follows. In Section 2 we provide background on the current version of Janus, discuss related work on system call interposition-based security tools and examine the relationship of interposition-based sandboxing and intrusion detection tools. In Section 3 we provide a basic model of how Janus works to provide context for our discussion. Section 4 discusses the pitfalls and problems that we encountered while building Janus; these are divided into the following categories: incorrectly replicating OS semantics, overlooking indirect paths to resources, race conditions, incorrectly subsetting a complex interface, and side effects of denying system calls. We explain each category of error, then present concrete examples in the context of Janus. In Section 5 we present solutions to some of the problems discussed in Section 4, consider the design choices and trade-offs we made in Janus, and present some alternative designs. We also offer design principles to help avoid the types of pitfalls that we encountered. Section 6 presents a discussion of some remaining open questions and topics for future work in the design of system call interposition-based security tools. We conclude our discussion in Section 7.

2 Background and Related Work

The current Janus implementation was built through a process of successive improvements and re-writings, starting from the original prototype described in Goldberg et. al. [14]. The original Janus work presented an architecture for restricting an application's interactions with the underlying operating system by interposing on the system calls made by the application via standard process tracing mechanisms. Several other systems for restricted execution were subsequently developed based upon this architecture including MapBox [3] and Consh [5].

Unfortunately, as detailed by Wagner [29], the `ptrace` [22] interface, initially used by Janus for interposition in Linux, has a variety of limitations that make it a poor choice of mechanism for application sandboxing, and other security-sensitive applications. Additional discussion of the limitations of `ptrace` in the presence of hostile applications is offered in Cesare [7].

There have been attempts to overcome the limitations associated with `ptrace` through the creative use of other standard Unix mechanisms; their results have been lackluster. Subterfugue [1] provides a means of overcoming the problem of argument races (see Section 4.3.3) with standard Unix mechanisms; unfortunately, their approach exacts a severe performance penalty. Jain and Sekar [16] demonstrate a more efficient user-level approach to interposition using `ptrace`, but as the authors note, it relies on a technique for avoiding system call argument races that is not completely secure. The Solaris `/proc` interface, another prominent process tracing mechanism, also does not provide an ideal mechanism for security applications [29].

To address the problem of supporting secure system call interposition, the present version of Janus relies on its own dedicated system call interposition mechanism, implemented in Linux as a loadable kernel module. A similar approach to addressing this problem taken by Systrace [24], another system call interposition-based security tool whose design closely resembles the present version of Janus.

Several groups have looked at developing novel system call interposition mechanisms for a broader set of applications than simply sandboxing. Work on SLIC [13] and Interposition Agents [17] looked at providing an infrastructure for using interposition as a general mechanism for OS extensibility. Ufo [4] examined the potential for applying this technique as a means of implementing a user-level filesystem.

Completely in-kernel system call interposition based tools for application sandboxing have also been studied by several groups [11, 2]. Several commercial products using this approach are currently available [8, 9].

Intrusion detection via analysis of system call traces has received a great deal of attention [10, 15, 21, 27]. Typically this work abstracts away many of the real details of the system, choosing (explicitly or not) to work with an idealized model of system call tracing. The problem of secure system call tracing is similar to that of secure system call interposition. In the system call tracing case, the "viewer" is only interested in what calls an application makes, and not in modifying them or denying them. However, the same problems that can allow a system call interposition-based sandbox to be circumvented can also be used to evade a system call tracing-based IDS. The IDS case is in some ways more difficult. A sandbox can deny system calls that would make interpreting the effects of other calls difficult, and can ignore issues like being overloaded by calls to analyze. An IDS is generally restricted to passive observation, and thus must deal with these problems. Additionally, users are often willing to pay the overhead associated with secure interposition for sandboxing for some additional "guaranteed" degree of security; the potential to simply detect an intruder may be less enticing.

Our discussion of system call interposition requires some basic knowledge of the Unix API. We recommend readers looking for additional background on this topic refer to Stevens [26] or McKusick et. al. [22].

3 The Janus Architecture

Intuitively, Janus can be thought of as a firewall that sits between an application and the operating system, regulat-

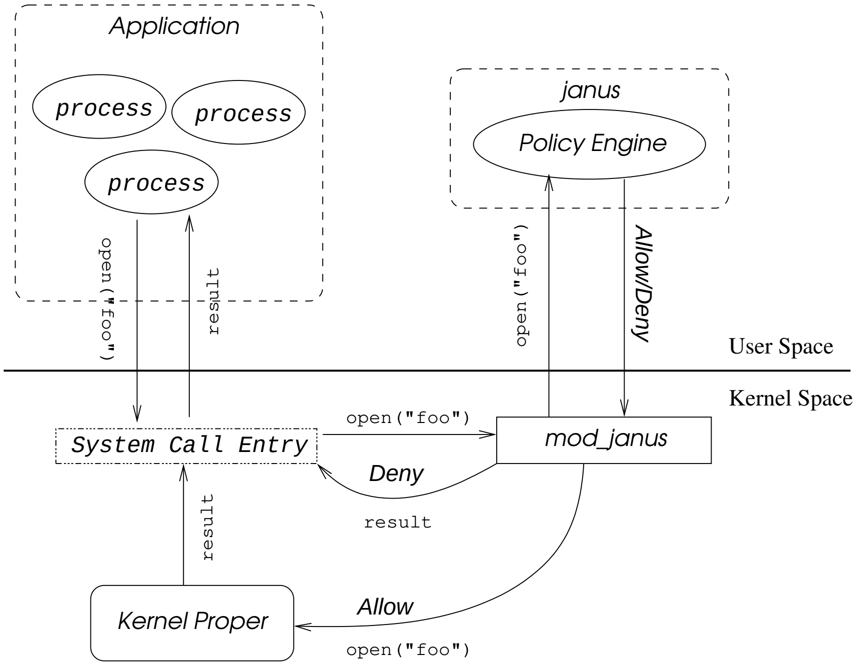

Figure 1. System Call Interposition in Janus

ing which system calls are allowed to pass. This is analogous to the way that a firewall regulates what packets are allowed to pass. Another way to think about Janus is as an extension of the OS reference monitor that runs at user level.

Concretely, Janus consists of `mod_janus`, a kernel module that provides a mechanism for secure system call interposition, and `janus`, a user-level program that interprets a user-specified policy in order to decide which system calls to allow or deny.

To gain a better understanding of Janus's basic operating model we will look at the lifetime of a program being run under Janus:

1. At startup, `janus` reads in a policy file that specifies which files and network resources it will allow access to.

2. `janus` then `forks`, the child process relinquishes all of its resources (closes all of its descriptors, etc.), and the parent attaches to the child with the tracing interface provided via `mod_janus`. At the user level, this consists of attaching a file descriptor to the child process. `janus` then `selects` on this descriptor and waits to be notified of any interesting events.

3. The child `execs` the sandboxed application.

4. All accesses to new resources via `open`, `bind` etc. is first screened by Janus to decide whether to allow the application access to the descriptor for the resource.

5. The program continues to run under Janus's supervision until it voluntarily ends its execution or is explicitly killed by Janus for a policy violation. If a sandboxed process `forks`, its new children will have new descriptors attached to them, and will be subjected to the security policy of their parents by the `janus` process.

To further examine how Janus screens system calls, let us consider the sequence of events that occurs when a sandboxed process attempts the call `open("foo")` as depicted in Figure 1.

1. A sandboxed process makes a system call `open("foo");` this traps into the kernel at the system call entry point.

2. A hook at the system call entry point redirects control to `mod_janus`, since `open` is a sensitive system call.

3. `mod_janus` notifies `janus` that a system call has been requested and puts the calling process to sleep.

4. `janus` wakes up and requests all relevant details about the call from `mod_janus`, which it uses to decide whether to allow or deny the call. It then notifies `mod_janus` of its decision.

5. If the call is allowed, control is returned to the kernel proper and system call execution is resumed as normal. If `janus` decides to deny the call, an error is returned to the calling process and the system call is aborted.

This is the basic model of a program running in a Janus sandbox. Readers may also wish to refer to other sources [12, 23] on how `janus` and `mod_janus` work , as well as referring back to the description of the initial Janus prototype [29].

4 Problems and Pitfalls

This section presents a selection of problems and pitfalls encountered in the course of building Janus. These can be broken into the following general categories: incorrectly replicating OS semantics, overlooking indirect paths to resources, race conditions, incorrectly subsetting a complex interface, and side effects of denying system calls. We introduce each problem with a general description, then present concrete examples from our own work.

4.1 Incorrectly Replicating the OS

In order to make policy decisions, Janus must obtain and interpret OS state associated with the application it is monitoring. Achieving this can lead to replicating the OS in two ways. First, we may try to replicate OS state. Necessarily, we must keep around some state in order to track what processes we are monitoring. This state overlaps with state managed by the OS. In order to interpret application behavior (e.g. the meaning of a system call) we must replicate OS functionality. In both cases, replication introduces the possibility of inconsistency that can lead to incorrect policy decisions.

4.1.1 Incorrectly Mirroring OS State

Janus often needs OS state in order to make a policy decision. For example, if we observe that a process wants to call `ioctl` on a descriptor, we might want to know more about that descriptor. Is it open read-only, or read-write? Is it associated with a file or a socket? Does it have the `O_SYNC` flag set? One solution to this problem is to infer current OS state, by observing past application behavior. This option is certainly attractive in some ways. Inferring state means we don't need to modify the OS if this information is not readily available. It also eliminates the system call overhead of querying the OS. Unfortunately, trying to infer even the most trivial information can be error-prone, as we discovered in the course of building Janus.

Janus needs to know the protocol type of IP sockets in order to decided whether or not to let a monitored process `bind` them. Our initial solution to this problem was to record a socket's protocol type when it was created, then later look up this information when the need arose. Unfortunately, we had not taken into account all of the system calls that could change our descriptor space. As a result, a malicious process could fool Janus about the protocol type of a given socket, and violate certain security policies. Consider the following example:

Suppose we are enforcing the following security policy: our application, for example a web server, is only allowed to `bind` TCP sockets on port 80, and is not allowed to `bind` UDP sockets, but is still allowed to create and `connect` them, perhaps to talk to a DNS server.

Suppose we see the following sequence of calls:

1. `6 = socket(UDP,...)`. Janus logs 6 as a UDP socket.

2. `7 = socket(TCP,...)`. Janus logs 7 as a TCP socket.

3. `close(7)`.

4. `dup2(6,7)`. Janus's state about the type of 7 is now inconsistent as it still believes that 7 is a TCP socket.

5. `bind(7,...some address...port 80)`. Janus allows this call to proceed as it believes that 7 has type TCP, our security policy has thus been violated.

Because we failed to take `dup2` into account, the sequence of calls shown could leave Janus in a state where its internal representation of the process's descriptor space indicates that 7 has protocol type TCP, a state inconsistent with its actual protocol type of UDP. We could have modified Janus to account for all possible means of duplicating descriptors; e.g. `dup, dup2, fcntl,` etc. but a more robust solution was simply to query the kernel directly. [2] In this case the state was readily available from the kernel; however the interface to access it was initially overlooked. We took several lessons from this.

First, avoid replicating state in the monitor. Any time one attempts to shadow OS state at user level, one runs the risk that his copy will become inconsistent with the OS. Second, policy decisions should minimize the amount of state they require, as this reduces the chance of relying on inconsistent state. Another important lesson this mistake illustrates is not to underestimate the complexity of Unix interface semantics. The initially incorrect approach taken to solving this problem was written by programmers with

[2] Directly accessing this information was achieved through the Janus kernel module, although using the `/proc` interface would also have been a viable solution.

significant experience in the Unix programming environment. Their mistake was not detected until a rewrite, several years later, after a variety of other parties had looked over the code and failed to detect the problem.

4.1.2 Incorrectly Mirroring OS Code

Operating systems often perform non-trivial processing to interpret system call arguments. Duplicating this processing runs the risk of incorrectly implementing the kernel's functionality or having the duplicate implementation become inconsistent over time (real-world kernels do change, after all). An ideal example of where this type of problem can arise is the canonicalization of path names.

Janus needs canonical file names to allow files to be uniquely specified in the context of its access control policies, for example, suppose Janus sees the call `open("../foo",O_RDWR)`. It needs to be able to tell that this is the same file as `/tmp/foo` in its security policy. Superficially, this appears to be a simple problem. If we are given a relative path then we simply need to resolve its constituent parts. For example, if the path contains `".."` or `"."` we need to resolve these to the correct directories, however, there are some subtle issues here. If the last component of our path is a symbolic link, should we expand it? What should we consider as our file system root when we resolve a path name? If the application being monitored is allowed to use `chroot`, it may have a different file system root than the Janus process watching it. Even individual processes of the same application may have different roots. Basic path resolution regularly confounds implementers[3] of applications that require this functionality.

Other subtleties can arise due to the fact that file system behavior may differ on a per-process basis. Consider Linux, where `/proc/self` contains information reflecting the state of the process accessing the directory. Thus, if we resolve `/proc/self/cwd` (the current process's current working directory) in the context of the monitor we may find a very different result than if we resolve this path in the context of the untrusted process that the monitor is watching.

Janus initially tried to canonicalize at user level, and got it wrong by overlooking a few of these issues. The solution we adopted was to canonicalize in the kernel, in the context of the monitored process, by calling the appropriate kernel function after file names are copied into the kernel. The complete description of our approach to dealing with this problem is given in Section 5.4. Duplicating OS functionality/code should be avoided at all costs. Such duplication adds complexity to the tool in question, and often the duplicated functionality overlooks corner cases.

4.2 Overlooking Indirect Paths to Resources

Resources are often accessible through what we call *direct paths* and *indirect paths*. Direct paths are those that are obvious and clearly specified, and typically those that are interposed upon. To access the Linux file system the direct paths are "normal" file system API calls that are used to access descriptors such as `open` or to modify the file system, such as `rename` or `unlink`. Most monitoring and interposition tools will carefully guard these interfaces. However, these same tools will often ignore or overlook indirect routes to access the file system. Consider the following routes:

- Unix domain sockets: Unix domain sockets can be bound to locations in the file system. While these cannot be opened like normal files, they can still be used to create things like lock files.

- Core dumps: Even if an application cannot open any files directly, it can still create files by dumping core.

- Descriptor passing: Unix domain sockets allow descriptors to be passed back and forth between processes via `sendmsg` and `recvmsg`. If this path is overlooked, applications can gain access to unauthorized files or network sockets without any checking from a monitor. Consider two sandboxed processes that have different network access policies, but access to a common portion of the file system where they can open and access Unix domain sockets (but not files). If one process has access to sensitive data but no access to an external network, and the other process has external network access, but no access sensitive data, the two processes can collaborate to leak sensitive data, and thus violate the system security policy. Admittedly, this example could also utilize the presence of shared memory or other interfaces that processes can use to share data. In the limit, this runs up against the issue of the confinement problem [20].

- Unhelpful processes: Another common way that an application can gain access to parts of the file system not specified in its security policy is through other processes that unwittingly help it. An ideal example of this is the Name Service Cache Daemon (or `nscd`). `nscd` can be used by glibc to retrieve entries from `/etc/passwd` and `/etc/groups` via interprocess communication over a Unix domain socket. If the unwary user grants read access to the standard `nscd` socket in `/var` they may inadvertently grant

[3] A cursory examination of recent reports on BUGTRAQ reveals several serious bugs in network services that got canonicalization wrong the first time.

access to potentially sensitive parts of their system. If a malicious process is able to create sockets in this part of the file system, it can impersonate nscd, and thus trick other processes into using its own version of the `passwd` and `groups` files.

One of the key difficulties of interposing on an interface as complex as the Unix API is simply knowing all of the side effects and non-obvious ways that one can affect system resources. It is important to identify every possible way for a process to access or modify resources, both alone and working in concert with other processes.

4.3 Race Conditions

Race conditions in system call interposition-based security tools most commonly occur due to the non-atomicity of permission checking at user level in Janus and access granting in the kernel [25]. These are sometimes referred to as time-of-check/time-of-use bugs [6]. The following is a basic outline of what such a race condition looks like:

1. Janus grants permission to perform an operation A, that relies on some mutable shared state.
2. That state changes, making the result of performing operation A illegal (i.e. the meaning of A in context has changed).
3. The operation A is performed by the operating system (i.e. access is granted, but not the access that Janus approved).

This type of race condition is a significant problem for sandboxing tools and can be used to mislead intrusion detection systems. Often, operations that system calls perform rely on mutable shared state that if altered will fundamentally change the impact of the system call. We will see many examples of this type of race in the following sections.

4.3.1 Symbolic Link Races

Symbolic link races [6] are a familiar problem to the security conscious Unix programmer. A symbolic link race occurs when a symbolic link is modified to point to a different file between the time that the file it refers to is checked for a given property, and when an access that relies on that check occurs. Consider the following example:

Suppose Janus is enforcing a security policy that allows write access to `/tmp/foo` and `/tmp/bar` and read access to `/tmp/baz`. Also suppose that `/tmp/foo` is initially a symbolic link pointing to `/tmp/bar`.

Now consider the following sequence of events:

1. Process A calls `open("/tmp/foo", O_RDWR)`.
2. Janus checks that access to both `/tmp/foo` and `/tmp/bar` is allowed and, noting that it is, it will allow the call to proceed.
3. Process B removes `/tmp/foo`, and creates a new symbolic link `/tmp/foo`, that points to `/tmp/baz`.
4. The OS executes `open("/tmp/foo",O_RDWR)` for process A, thereby granting write access to `/tmp/baz`.

Process A has just violated our security policy since it now holds a descriptor to `/tmp/baz` with both read and write access. In this example and in later examples, Process B does not necessarily need to make the call to modify `/tmp/foo` after Janus makes its check, the modifications just need to happen after Janus's check i.e. the order that calls are made does not strictly dictate the order that they complete in.

Symbolic link races are one of the most commonly known and frequently confounding problems in system call interposition based sandboxing tools. Frequently tools attempt to solve the problem of symbolic link races by first canonicalizing pathnames and then checking the canonical name against their policy, or in some cases, forcing the application to use a canonicalized name for file system access by altering the arguments to the call in question. This also does not solve the problem of symbolic link races because it fails to address the root of the problem, which is concurrency. Even after resolving `/tmp/foo` to `/tmp/bar`, another process can still change `/tmp/bar` to be a symlink to `/tmp/baz`, before the `open` call executes, they can even change `/tmp` to be a symlink to `/etc` if they have the appropriate permissions. We discuss the difficulty of formulating a correct solution to this problem and some of the missteps that we made in Section 4.4.

4.3.2 Relative Path Races

Relative path races exploit the fact that the location of a given inode can change between the time that Janus resolves a path relative to that inode to check an access, and the time that the kernel resolves a path relative to that inode to perform the access. To illustrate a potential exploit of this race, consider the following sequence of events:

Assume the current working directory of Process A is `/tmp/foo/bar` and that Janus allows the monitored application read and write access only to the contents of the `/tmp` directory.

1. Process A calls `open("../../etc/shadow", O_RDWR)`.

2. Janus resolves this path to /tmp/etc/shadow, notes that this is in /tmp, and allows the call.

3. Process B, also in the sandbox, renames /tmp/foo/bar to /tmp/bar.

4. The OS executes open("../../etc/shadow",O_RDWR) in the context of Process A. Since the current working directory of A is now /tmp/bar, Process A gains read and write access to /etc/shadow.

Relative path races have a similar flavor to symbolic link races, as both rely on malicious processes collaborating to modify shared file system meta-data, and fool the monitor. Interestingly, relative path races are not a well studied problem like symbolic link races, in spite of the fact that they do not appear to be unique to the problem of system call interposition-based sandboxing.

4.3.3 Argument Races

An argument race occurs when arguments to a system call are modified between the time that they are checked by Janus, and when they are used by the operating system to perform the system call. Argument races can occur if system call arguments reside in a portion of memory that is accessible by more than one process. Consider the following scenario:

Suppose Janus's policy is to allow read access to /tmp/foo and deny all other file system access.

1. Process A calls open(path,O_RDONLY), where path is "/tmp/foo".

2. Janus traps process A.

3. Janus reads path from the memory of A, notes that /tmp/foo can be opened according to its policy, and allows the call to proceed.

4. Process B modifies path to point /etc/shadow.

5. The OS executes the open call, returning a descriptor for /etc/shadow.

Generally, argument races are only a concern for non-scalar system call arguments. Scalar system call arguments are passed to the OS in registers that are immediately copied into the kernel by the system call trap handler. Scalar arguments in the kernel can only be tampered with by user space processes via ptrace and other tracing interfaces.

Non-scalar arguments, such as path names and socket addresses (e.g those used by connect, bind, etc.) are not immediately copied from user memory into the kernel. Instead, they are copied by individual system calls immediately before use. This potentially leaves a window of opportunity for an attacker to strike, after the time that a system call has been initiated and arguments have been examined by Janus, but before the arguments have been copied into the kernel.

This type of race can only occur in the presence of shared memory. Examples of mechanisms supporting this in Linux include: the SYSV shared memory facilities, mmap, and memory shared among multi-threads created via the clone call. Debugging interfaces such as ptrace that allow processes to modify the memory of other processes must also be taken into account.

4.3.4 File System Information Races

In Linux, when two threads share file system information it means that those threads share their root directories and current working directories. Sharing current working directories is highly problematic as it easily facilitates a race for any file system operation that uses a relative path. For example, consider the following sequence of calls, assuming access to /etc/shadow is forbidden by Janus:

1. Thread A calls open("shadow",...) with the current working directory /tmp.

2. Janus traps and approves the call.

3. Thread B calls chdir("/etc") and Janus approves the call.

4. Thread A's open("shadow",...) executes with the current working directory /etc, returning a descriptor to /etc/shadow. Thus, our security policy has been violated.

While this class of race conditions seems similar to the previous two, it is really in a class of its own. The problem is not that file system meta-data is shared, but rather that process meta-data is shared between two or more threads. The sharing of root directories is also problematic as a similar race condition can be induced using chroot.

4.3.5 Shared Descriptor Space Races

In the case where two threads share descriptor spaces we encounter the possibility for a race in checks that rely on the type of descriptors for their correctness.

For example, suppose that Janus has the policy that it only allows TCP sockets to be bound to port 80. By executing the following sequence of calls a multi-threaded application with threads A and B can violate this policy.

1. Thread A calls socket to create a TCP socket as fd 7.

2. Thread A calls `socket` to create a UDP socket as fd 6.

3. Janus traps a call to `bind` from thread A to attach fd 7 to port 80, and allows the call to proceed as fd 7 is a TCP socket.

4. Thread B closes fd 7.

5. Thread B calls `dup2(6,7)`.

6. Thread A's `bind` call proceeds, binding a UDP socket to port 80, and violating the security policy.

Races of this type, along with argument races, make dealing with multi-threaded application highly problematic. Due to these difficulties Janus currently does not support multi-threaded applications. We discuss this problem further in Section 6.

4.4 Incorrectly Subsetting a Complex Interface

Application sandboxes generally enforce policy by only allow the use of a subset of the system call interface, we refer to this practice as subsetting. The system call interface can be subsetted at the granularity of whole system calls, (e.g. disallow `chroot` calls) or on a more granular by disallowing certain arguments to calls (e.g. only allow `open(path)` if `path` = `foo`).

In spite of its seeming simplicity, subsetting can often be tricky. Unanticipated interactions between different system calls can make it difficult to check that a subsetting solution is correct. In order to better illustrate the limitation of this approach, and the type of difficulties that can arise, we will consider a series of incorrect subsetting solutions to the problem of symlink races motivated by our own experience.

Attempt 1: Deny the creation of symlinks to files to that we do not allow unrestricted access.

The argument for the correctness of this solution goes: if an untrusted application is only allowed to create symlinks to files it could access without restriction then even if a symlink race occurs there is no harm done.

Problems: Pre-existing symlinks, other sources of symlinks.

This solution is a good start, but it is incomplete. Some notable oversights include the implicit assumption that we are only worried about symlinks that the untrusted application creates. This ignores the possibility of pre-existing unsafe symlinks and unsafe symlinks created by other processes. If either of these conditions exists, then the above policy is not sufficient to prevent symlink races. For example, the untrusted application can rename a pre-existing symlink to facilitate a race in the former case is true.

Attempt 2: Deny the creation and renaming of symlinks.

The argument for the correctness of this solution is that if we remove the untrusted applications ability to create symlinks with a given name, we have removed its ability to create a race.

Problems: Directory renaming and relative symlinks.

Preventing symlinks from being renamed prevents the attack that broke our previous solution, but there are still problems. Suppose for example that our untrusted process can rename `/tmp` to `/quux` and replace it with another directory, that also contains a file named `foo`, that is a symlink to `../quux/baz`. Again, we can violate the security policy. So completely disallowing direct manipulation of symlinks does not prevent this attack. This attack is even possible if we check that all reachable symlinks are safe when we start Janus.

Attempt 3: Deny access through symlinks.

Suppose that we give up on allowing access through symlinks in general. To implement this, we have Janus `stat` files before it allows access to them, and if they are symlinks, deny the call. Clearly our renaming attack above still works, and we are aware that a race might still occur, so we try so to ensure that our open will fail if someone does try to slip us a symlink. We modify the open call slightly by adding the `O_NOFOLLOW` flag before we let it execute. This flag will cause the open to fail if it finds that `/tmp/foo` is a symlink. It seems like now we are safe, but as it turns out, we can still get burned, for any component of the path `/tmp/foo` may be a symlink, including `tmp`. Thus, we are only protected by this approach if our path consists only of `foo`.

Problem: Symlinks in intermediate components in our path.

Providing a general solution to this problem that yields a tool that is not fragile in the presence of symlinks does not seem possible using just subsetting, however, there are other mechanisms that can be called upon. We will present some potential alternative approaches to addressing this problem in section 5.2.

We believe that this example clearly illustrates the potential for mistakes when subsetting a complex and stateful interface such as the Unix file system. It also serves as a clear illustration of the limitations of a purely subsetting approach to user level application sandboxing.

4.5 Side Effects of Denying System Calls

System call interposition-based sandboxes restrict an application's behavior by preventing the execution of any system call that would violate a predetermined security policy. Preventing the execution of a system call, or causing a system call to return in a manner inconsistent with its normal semantics, can have a detrimental impact on the operation of the application, potentially undermining its reliability and even its security.

Denying calls that an application uses to drop privilege frequently introduces serious security flaws. This type of problem is most often found in applications that run as root and drop privilege using `setuid`. Many applications that rely on `setuid` fail to check its return value, and if `setuid` fails, will continue to function in a compromised state. Upon casual examination we were able to discover this condition in several common FreeBSD daemons, and it appears that this problem is quite widespread. We also found that applications fail to check the return values from other privilege-reducing calls, or simply fail-open. We frequently saw failures being ignored for other privilege reducing calls including: `setrlimit` to reduce resource limits, and `fcntl` to drop privilege from descriptors.

Given that aborting privilege-dropping calls will often undermine the security model of a sandboxed application, it seems generally advisable to allow all such calls. For `setuid` and related calls, it seems most prudent to abort the application entirely if we wish to deny a call.

Forcing system calls to return with a value that is not part of its specified interface, or that the application designer simply did not anticipate, is another potential source of problems. For example, the Solaris `/proc` process tracing interface will only allow aborted system calls to return `EINTR`. This sometimes leads applications to hang, repeatedly retrying the aborted system call with the expectation that it will eventually complete [14, 13].

5 Solutions and Helpful Hacks

In this section we present techniques for solving or avoiding some of the problems that we presented in the Problems and Pitfalls Section. We consider several different approaches for: avoiding argument races, avoiding file system races, and denying system calls without adversely effecting applications. For each set of solutions, we first provide a brief review of the problem being addressed, we then give a high level summary of the solutions. We then present the details of implementing these solutions and that trade-offs associated with each approach. We give special attention to the solutions we chose for Janus, and the rationale behind our choices. We conclude with some more general principles for recognizing and avoiding problems in this class of system.

5.1 Avoiding Argument Races

To recap our discussion in section 4.3.3, argument races occur when system call arguments are modified between the time that a monitor reads the arguments for a permission check, and when the operating system uses the arguments. The canonical example of this problem is a process changing the argument string `"/tmp/foo"` to `"/etc/shadow"` in the memory of another process that has just used the string as an argument to an `open` call. Here we consider two approaches to preventing this class of race condition:

- Copy arguments into a "safe" place in memory, e.g. private memory,[4] kernel address space, or the address space of a trusted process. This guarantees that sensitive arguments cannot be modified between when they are read by the monitor and when they are used by the kernel.
- Leave arguments in place and ensure that the memory they reside in is private (i.e. "safe").

Using these strategies we have sought to make arguments accessible only to the kernel and trusted process (e.g. Janus) and when safe, the thread of control requesting the system call.

5.1.1 Copying Sensitive Arguments into the Kernel

OS kernel's prevent argument races by copying arguments into kernel memory before use. This approach makes sense; the cost of this copy is usually minimal, and it is easy to verify the safety of this approach. `mod_janus` protects system call arguments using a variation on this theme by copying volatile system call arguments into kernel memory before they are used.

When `mod_janus` traps a system call it immediately looks in a table to see if this call has any arguments that reside in user memory. If arguments are present, `mod_janus` will copy these arguments into a thread-specific buffer in the kernel and twiddle the argument pointers in the trapped thread's registers to point to this new location in memory. It will then set a flag indicating to the kernel that it is OK for the current system call to obtain its arguments from kernel memory. When Janus subsequently examines the arguments of the untrusted application, it will fetch a copy of them directly from the thread-specific buffer, not from the memory of the untrusted process.

The primary advantage of this approach is the simplicity of verifying its correctness. We can say with a high degree of certainty that arguments in a kernel buffer cannot be modified by untrusted processes, even in the presence of multi-threading.

The disadvantage of this approach is that it adds some complexity to the kernel resident portion of Janus; about

[4] For memory to be private in the sense we mean here, the memory must be modifiable only by the monitor processes and the process that owns the memory. This means not only that the memory is not explicitly shared between processes (e.g. created via. `mmap` with the `MAP_PRIVATE` flag), but also that it is not shared between multiple threads, modifiable via. a process tracing interface, etc.

25% of mod_janus is dedicated to performing this task (mod_janus is well under 2K lines of C total, so this is not a major penalty). Individual system calls require special treatment in order to copy their arguments into the kernel. Fortunately, the arguments of interest are typically of two standard types, socket addresses and path names. Most of the work of specifying per-call copying behavior is reduced to filling in a per-call entry in a table which specifies its argument types.

Another worrisome property of this approach is that system calls are permitted to fetch their arguments from kernel memory instead of user memory. If mod_janus failed to twiddle a threads argument pointers (or pointers in the arguments themselves, when there are arguments with nested pointers), the untrusted application might be able to gain unauthorized access to kernel memory. The difficulty of verifying this is somewhat ameliorated by our table-driven approach. This minimizes code duplication and simplifies auditing.

We considered the possibility of moving some of this complexity into Janus by giving it finer grained control over moving arguments to and from this per-thread scratch space at user level, but we ultimately decided against this approach as we believe it provided too much power to the tracing process, and greatly increased the possibility of creating an exploitable hole in mod_janus, which is intended to be accessible by unprivileged processes.

5.1.2 Protecting Arguments in User Memory

Another solution to the problem of preventing argument races is copying arguments into a read-only section of memory in the address space of the untrusted process. Setting up this section can be accomplished either by dedicated kernel code or by forcing the process to mmap and mprotect a region of memory. Before system call arguments are checked, they can be copied into this region by Janus. This approach was taken by earlier versions of Janus that would create a read-only memory region by calling mmap in the context of the untrusted process when that process was "attached", then keep track of the location of this scratch space in the untrusted application's address space. The correctness of this approach relies on Janus judiciously guarding the mprotect, mmap and mremap interfaces to ensure that this read-only section of memory is not tampered with.

This approach is attractive, as much less work must be done in the kernel in order to implement it. The Subterfugue [1] system actually does this using only ptrace, although the inefficiency of using ptrace to copy arguments makes this prohibitively expensive, given a more efficient mechanism this seems like an attractive approach. Concerns about assuring that access to the user scratch space was sufficiently restricted and efficiency lead us to abandon this approach in Janus.

5.1.3 Checking that Arguments Do Not Reside in Shared Memory

Argument races can only occur if arguments reside in unprotected shared memory. On approach to preventing this is restricting the interfaces that allow the creation of shared memory, such clone for creating multiple threads, certain uses of mmap and the SYSV facilities for creating shared memory areas.

For many applications this is a viable solution. Relatively few Linux applications use multi-threading. The same is true of the BSD-based operating systems, which until recently [31] did not provide kernel support for multi-threading. The use of other shared memory facilities is also not terribly widespread.

We can check that arguments do not reside in shared memory at user level by examining the permissions on the virtual memory area that arguments currently reside. This can be accomplished through the /proc filesystem under Linux. For this approach to be correct we must be ensure not only that the virtual memory area that an argument resides in is not shared, but also that the untrusted process has its own copy. For private mmaped memory regions this can be enforced by reading our of the process's memory and writing them back to the same location to ensure that the process has a private copy. This is necessary as a privately mmaped files may reflect changes in the underlying file, a process is not guaranteed to get its own copy of an area of an mmaped file until it writes to that area.

While this approach enforces some limitation on the generality of a tool, it requires no kernel modifications and minimal effort on the part of the implementer.

5.2 Avoiding File System Race Conditions

In order to verify that Janus sees exactly what filesystem accesses a process makes, file system access must take place in a manner that ensures that no race condition can take place. In section 4.4 we demonstrated that trying to achieve this by simply subsetting away problematic behaviors left us with a policy for file system access that was quite cumbersome and difficult to use safely. In this subsection, we present some potential approaches to solving this problem. Our approaches work by coercing applications into always accessing the file system using an access pattern that we can easily verify is safe (e.g. via shared library replacement), and then simply disallowing all access that does not conform to this access pattern i.e. we even disallow potentially safe operations which would simply be hard to verify.

5.2.1 What is Good Behavior?

Unix applications can obtain access to files without encountering symlink races. This is important for normal application programmers who, for example, might want to write an ftp server that securely checks file system accesses against a security policy. The programmer can accomplish this by leveraging the fact that the current working directory of a process is private state[5] and will not change between the time that it performs a check on a file relative to this directory, and the time that the call completes. The programmer can leverage this to perform a race free open by recursively expanding (via `readlink`) and following a path one component at a time until they have reached a file, or until they have found that the path violates policy. A similar sequence of calls can be used to perform other file system operations without races. If we are monitoring an application performing such a sequence of calls, we can also check each call in the sequence without the risk of a race condition. This follows from the fact that we perform our check after the application makes the call, if there cannot be a race that can fool the application, there cannot be a race that will fool the monitor. Clearly this is a very specific behavior pattern that we would not normally expect applications to conform to. However, we can play some tricks to coerce applications into always conforming to good behavior patterns.

5.2.2 Enforcing Good Behavior

There are several mechanisms we can use to "force" an application to conform to our definition of good behavior, i.e. to access the filesystem in a manner that we can easily verify is safe.

1. Induce safe call sequences: The monitored process can be forced to directly execute a safe sequence of calls using a process tracing mechanism. For example, if a monitored process makes the call `open("/tmp/foo",...)`, we could force the operating system to make the appropriate safe sequence of system calls specified above in the context of the traced process.

2. Static or dynamic library replacement: [6] Using this approach we replace problematic library calls with code that converts these calls to their easy-to-check counterparts. Again, we could have a shared library replacement for `open`, which if called with `/tmp/foo` would make the safe sequence of calls given above. When using this approach we still rely only on Janus for the security of our system, the libraries are not trusted, but merely serve to facilitate easy checking.

3. Force access through a proxy: Instead of letting an application access the file system directly we can require it to go through a proxy process which accesses the file system on the application's behalf. This proxy can in turn use safe operations when it accesses files, preventing it from falling victim to file system races. We have had promising results with preliminary experiments in facilitating proxy-based file system access with shared library replacement. It is interesting to note that this approach also solves the problem of argument races.

[5] As mentioned above under Linux this invariant can be violated if we allow threads to clone themselves with shared file system state.

[6] This approach also requires that we also use a modified loader. This "trick" does not work with the loader, because the loader must first access the file system directly in order to load the shared libraries before it can use them.

5.3 Denying System Calls without Breaking Applications

Often the reason that we wish to deny a system call is that it allows a process to modify some sensitive global state, such as a sensitive file, its own resource limits, or its uid. However, it is often the case that the process can be given its own local copy of this state without affecting its functionality. This approach of giving an application its own copy of some global state is what we call virtualization. Virtualization is a powerful technique because it allows us to isolate an untrusted application from sensitive resources, while preserving normal system semantics, thus obviating the risk of breaking an application. There are several ways that we can virtualize the sensitive resources.

We can **emulate** the normal semantics of an unauthorized portion of the operating system interface using shared library replacement e.g. this technique can be used to simulate the semantics of running as root for processes running without privilege. This technique is used by tools like fakeroot [18], to simplify packaging software.

We can **redirect** calls to sensitive resources to a copy of those resources. For example, through modifying the arguments to system calls, either directly through the tracing mechanism as done in MapBox [3] or indirectly through shared library replacement.

We can **replicate** resources using the normal operating system facilities for this task; for example, using `chroot` we can give an untrusted application its own copy of the file system.

If possible it is always preferable to virtualize the resources than deny access as this gives us the highest level of certainty that we have not broken our sandboxed application.

5.4 Let the Kernel Do the Work

If the kernel does some complex operation, don't try to replicate that code yourself, just call the code in the kernel. In section 4.1 we discussed the problem of canonicalizing file names. Sometimes the OS will provide a system call for just this purpose[7]. Janus addresses this problem by having `mod_janus` canonicalize path names at the same time that they are fetched from the untrusted process upon system call entry. This is advantageous for two reasons. First, the file system namespace varies on a per process basis, canonicalizing path names in the execution context of the monitored processes ensures that these differences are taken into account. Second, because the Janus kernel module simply calls the kernel's canonicalization code, we can be sure that we are getting the correct canonicalization. A final advantage of letting the kernel do the work of canonicalization for us is that it simplifies our policy engine by several hundred lines.

5.5 Lessons for the Implementer

To summarize the lessons from our experience:

- Avoid replicating OS state and functionality. Reuse OS functionality and query the OS directly for state whenever possible. Beware of inconsistency.
- Be conservative in your design. Don't underestimate the complexity of the system call API. Don't overestimate your understanding of its nuances.
- Be aware of race conditions that can occur between the OS and monitor. Consider all the state that a system call relies upon to perform its function. Think about what parts of the system can modify that state. Think about what can happen between the time you make a policy decision about a system call and when the system call finishes.
- Be aware of the multi-threading semantics of your particular operating system.
- Be explicit. Document and justify the decisions you have made in your design and the assumptions that must hold in order for your implementation to be correct. These assumptions may be violated as the OS evolves, when your tool is ported to another platform, etc. The security of system call interposition-based tools often rely on a complex set of assumptions that rarely make it beyond the mind of the implementer.
- Be aware of all direct and indirect paths to resources. Know all the ways that a process can modify the file system, network, and other sensitive system resources.
- The file system is a huge chunk of mutable shared state. It is fraught with race conditions of the obvious and non-obvious variety. Dealing with the file system interface is the most difficult part of confining/monitoring an application.
- Any time you change the behavior of your operating system, for example by aborting system calls, you risk breaking your applications and potentially introducing new security holes. Avoid making changes that conflict with normally specified OS semantics, or diverge from application designer's expectations.

6 Future Work

There are still a variety of problems to be solved in order to demonstrate a system call interposition-based sandbox that can support the full range of potential applications in a secure fashion. The most notable omission in our list of solutions was an answer to the question of how to support multi-threaded applications. We are not aware of any user-level system call interposition-based sandboxing tool that has addressed this problem. One potential solution is to offer functionality in a kernel module to allow locking of per thread meta-data. Since this is per-process state, and not globally shared state like file system meta-data, it seems quite possible that a user level process could safely be allowed to lock it. The performance implications of such a solution are unclear and require further study.

An important trend from first generation sandboxes such as the Janus prototype, MapBox, Consh, etc., to second generation sandboxes as exemplified by Janus and Systrace, has been to abandon a purely user-level approach to application sandboxing and instead embrace a hybrid [24] solution where a dedicated kernel module/patch is used for tasks such as system call interposition, canonicalizing pathnames, fetching system call arguments, etc. Significant performance and security benefits have already been realized through the reliance on a small amount of additional kernel code (well under 2K lines of C for `mod_janus`, and a comparable number in the Systrace kernel patch). It is not clear that the correct balance between user and kernel space functionality has yet been found. Pushing a small amount of additional functionality into the kernel to deal with file system access policy could potentially eliminate the race checking file system access control that we examined in Sections 4.3.1 and 4.3.2. Subdomain [8] has demonstrated that such a solution is feasible with the addition of a modest amount of additional kernel code.

[7] Solaris provides the `resolvepath` system call to canonicalize file names. However, the `resolvepath` interface is not fool proof. The issue of differences in per process views of the file system still remains a problem. Also, having to call `resolvepath` could add several additional system calls to your policy engines critical path.

Another important question to be addressed is whether the system call boundary remains best place to interpose on applications access to sensitive resources at all. Under Linux, an alternative approach will likely soon be available in the form of the Linux Security Module(LSM) [32] that provides low-level hooks for adding new access control mechanisms to the kernel. LSM does not provide a complete solution, however, it does provides a common foundation on which to build other mechanisms that could potentially yield a cleaner abstraction for controlling access to sensitive resources.

We have briefly touched upon the relationship between this work and host-based intrusion detection. We believe there are likely to be challenges unique to that application of interposition and would hope to see comparable study examining the interactions between policy, mechanism and implementation in the context of a real interposition-based HIDS system. Currently the closest work to this has been work by Wagner et. al. [28] examining weaknesses in the policy models of system call interposition-based anomaly detection tools such as those proposed by Hofmeyr et. al. [15].

7 Conclusion

We have presented a variety of problems and pitfalls that can occur in the design and implementation of system call interposition based security tools. We have broadly categorized these problems under the headings of: incorrectly replicating OS semantics, overlooking indirect paths to resources, race conditions, incorrectly subsetting a complex interface, and side effects of denying system calls. We have shown how these problems can allow sandboxes and related tools to be circumvented. We have considered a variety of solutions to the problems we have identified in this domain, as well as noting principles that can aid the implementer in avoiding common pitfalls. Finally, we have touched on a number of problems in this area that we believe merit further study.

8 Acknowledgments

This work, and the present implementation of Janus would not exist without the significant effort and guidance provided by David Wagner. This work also could not have been done without the development of the first version of Janus of by Ian Goldberg, David Wagner, and Marti Hearst. Steve Gribble, and Nikita Borisov provided invaluable feedback and encouragement in the course of this work. This work also greatly benefited from feedback and discussions with Niels Provos, Constantine Sapunzakis, Michael Constant and Ben Pfaff. Ben Pfaff and Steven Bergsieker offered significant editorial help. Eric Brewer, David Culler, Mendel Rosenblum, and Dan Boneh provided the excellent research environment in which this work was conducted. This material is based upon work supported in part by the National Science Foundation under Grant No. 0121481 and in part by NSF CAREER CCR-0093337.

References

[1] Subterfugue: strace meets expect. `http://subterfugue.org/`.

[2] D. A., M. A., R. R., and S. D. Chakravyuha(cv): A sandbox operating system environment for controlled execution of alien code. Technical Report 20742, IBM T.J. Watson Research Center, Sept. 1997.

[3] A. Acharya and M. Raje. MAPbox: Using parameterized behavior classes to confine untrusted applications. In *Proceedings of the Ninth USENIX Security Symposium*, Aug. 2000.

[4] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. Extending the operating system at the user level: the ufo global file system. In *Proc. of the USENIX Annual Technical Conference*, January 1997.

[5] A. Alexandrov, P. Kmiec, and K. Schauser. Consh: A confined execution environment for internet computations, 1998.

[6] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Spring 1996.

[7] S. Cesare. Linux anti-debugging techniques. `http://www.big.net.au/~silvio/linux-anti-debugging.txt`, January 1999.

[8] C. Cowan, S. Beattie, G. Kroach-Hartman, C. Pu, P. Wagle, and V. Gligor. Subdomain: Parsimonious server security. In *Proceedings of the Systems Administration Conference*, Dec. 2000.

[9] Entercept Security Technologies. System call interception whitepaper. `http://www.entercept.com/whitepaper/systemcalls/`.

[10] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996.

[11] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1999.

[12] T. Garfinkel and D. Wagner. Janus: A practical tool for application sandboxing. `http://www.cs.berkeley.edu/~daw/janus`.

[13] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. Slic: An extensibility system for commodity operating systems. pages 39–52, June 1998.

[14] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications, 1996.

[15] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[16] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Proc. Network and Distributed Systems Security Symposium*, 2000.

[17] M. B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Symposium on Operating Systems Principles*, pages 80–93, 1993.

[18] Joost Witteveen. fakeroot: a fake root enviroment. `http://packages.debian.org/stable/utils/fakeroot.html`.

[19] C. Ko, T. Fraser, L. Badger, and D. Kilpatrick. Detecting and countering system intrusions using software wrappers. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[20] B. W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, oct 1973.

[21] Y. Liao and V. R. Vemuri. Using text categorization techniques for intrusion detection. In *Proc. 11th USENIX Security Symposium*, August 2002.

[22] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implimentation of the 4.4 BSD Operating System*. Addison-Wesley, 1996.

[23] V. Nakra. Architecture study: Janus - a practical tool for application sandboxing.

[24] N. Provos. Improving host security with system call policies. Technical Report 02-3, CITI, November 2002.

[25] Steve Bellovin. Shifting the Odds, Writing More Secure Software. `http://www.research.att.com/~smb/talks/odds.ps`.

[26] R. W. Stevens. *Advanced Programming in the Unix Environment*. Addison-Wesley, 1992.

[27] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proc. IEEE Symposium on Security and Privacy*, 2001.

[28] D. Wagner and P. Soto. Mimicry attacks on host based intrusion detection systems. In *Proc. Ninth ACM Conference on Computer and Communications Security*, 2002.

[29] D. A. Wagner. Janus: an approach for confinement of untrusted applications. Technical Report CSD-99-1056, 12, 1999.

[30] A. Wespi, M. Dacier, and H. Debar. Intrusion detection using variable length audit trail patterns. In *RAID 2000*, pages 110–129, 2000.

[31] N. J. Williams. An implementation of scheduler activations on the netbsd operating system. In *USENIX Annual Technical Conference*, 2002.

[32] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Linux Security Modules: General Security Support for the Linux Kernel*, 2002.