Janus: an approach for confinement of untrusted applications

by David A. Wagner

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Eric Brewer
Research Advisor

(Date)

* * * * * * *

Professor Alex Aiken
Second Reader

(Date)

Abstract

Security is a serious concern on today's computer networks. Many applications are not very good at resisting attack, and our operating systems are not very good at preventing the spread of any intrusions that may result. In this thesis, we propose to manage the risk of a security breach by confining these untrusted (and untrustworthy) applications in a carefully sanitized space. We design a secure environment for confinement of untrusted applications by restricting the program's access to the operating system. In our prototype implementation, we intercept and filter dangerous system calls via the Solaris process tracing facility. This enables us to build a simple, clean, user-mode mechanism for confining untrusted applications. Our implementation has negligible performance impact, and can protect pre-existing legacy code.

Contents

List of Figures

Acknowledgements

Some of these results have been published previously as joint work with Ian Goldberg, Randi Thomas, and Eric Brewer in [40]. Some of the text from that paper (as well as some text from the unpublished manuscript [70], jointly written with Ian Goldberg and Eric Brewer) has been adapted for use herein.

We gratefully acknowledge the efforts of Michael Kaminski, who helped code up the necessary extensions to apply Janus to `sendmail`.

Thanks also to Steven Bellovin, David Oppenheimer, Armando Fox, Steve Gribble, Alex Aiken, and various anonymous reviewers for their helpful comments. Thanks especially to Alex Aiken for a number of insightful observations about the design of Janus.

Finally and most importantly, I am deeply indebted to Eric Brewer for his direction and advice. His comments, both high-level and low-level, have greatly improved this thesis. Thank you, Eric.

Chapter 1

Introduction

Over the past several years the Internet environment has changed drastically. This network, which was once populated almost exclusively by cooperating researchers who shared trusted software and data, is now inhabited by a much larger and more diverse group that includes pranksters, crackers, and business competitors. This increases the risk of serious attacks on critical systems. At the same time, the stakes are higher than ever before: corporate entities want to engage in electronic commerce over the net, and the value of the data carried over these networks is increasing constantly.

We can identify one common theme underlying many of the vulnerabilities found in today's network. The software and data exchanged on the Internet is very often unauthenticated, so it could easily have been constructed by an adversary. At the same time, much of the software we run is not very good at handling adversarial inputs, and our operating systems are not very good at confining any intrusions that may result.

We propose to address this problem by providing a first cut at a tool that can reduce the harmful effects of security compromises. Our approach is to confine the untrusted software in a limited sandbox by monitoring and restricting the system calls it performs. As an example of this approach, we built Janus[1], a secure environment for untrusted applications; a crucial implementation technique is to take advantage of the Solaris process tracing facility to enforce the desired restrictions on system calls.

1 Motivation

We see two important reasons one may classify an application as untrusted. First, the application may have come from an untrusted source—either downloaded over an insecure link, or written by an uncertified programmer. This first scenario represents the case of an application with potentially malicious intent. Alternatively, the application may be untrusted if it is exposed to outside attack yet not trusted to protect itself against adversarial inputs. This second scenario represents the case of a programmer with good intentions but insufficient coding skills to prevent catastrophic failure. In either case, we will want to protect ourselves against the potential agent of disaster.

[1]Janus is the Roman god of entrances and exits, who had two heads and eternally kept watch over doorways and gateways to keep out intruders.

One simple solution is to avoid running such untrusted programs. But in practice this is insufficient; many of these programs are too useful to abandon, and secure alternatives with comparable functionality often do not exist. Therefore, for some software, we must find a way to manage the risk.

This thesis develops a more realistic approach: confinement. We propose to confine untrustworthy and dangerous programs inside an environment that restricts the actions they can perform, thus managing the risk by limiting the harm that a potential security compromise could cause. We argue that a powerful technique for confining untrusted applications is to interpose a user-level reference monitor on the interface the OS presents to untrusted applications. We implement this abstract model by intercepting system calls invoked by the untrusted application and filtering potentially harmful requests before they are executed. This has a number of powerful advantages:

- We retain significant flexibility to transparently enforce per-application security policies from user-level code.
- We can secure legacy code, such as `sendmail` or Netscape.
- We can safely execute mobile code inside our restricted sandbox.
- We can attain much higher assurance than the existing OS mechanisms can provide.

2 The challenges

What security requirements are demanded from a successful protection mechanism? Simply put, an outsider who has control over the untrusted application must not be able to compromise the confidentiality, integrity, or availability of the rest of the system, including the user's files or account. Any damage must be limited to the application's display window, temporary files and storage, and associated short-lived objects. In other words, we insist on the Principle of Least Privilege: the application should be granted the most restrictive collection of capabilities required to perform its legitimate duties, and no more. This ensures that the damage a compromised application can cause is limited by the restricted environment in which it executes. In contrast, an unprotected Unix application that is compromised will have all the privileges of the account from which it is running, which is unacceptable.

Imposing a restricted execution environment on untrusted applications is more difficult than it might seem. Many traditional paradigms such as the reference monitor and network firewall are insufficient on their own, as discussed below. In order to demonstrate the difficulty of this problem and appreciate the need for a novel solution, we explore five possible approaches.

1. Building security directly into each untrusted application: Taking things to the extreme, we could insist all untrusted applications be rewritten in a simple, secure form. We reject this as completely impractical; it is simply too much work to re-implement them, at least in the short term. More realistically, we could adopt a reactive philosophy, recognizing individual weaknesses as each appears and engineering security patches one at a time. Historically, this has been a losing battle, at least for

large applications: for instance, explore the sad tale of the `sendmail` "bug of the month" [1, 2, 3, 4, 13, 14, 15, 16, 17, 18, 19, 21]. In any event, attempts to build security directly into untrusted applications would require each program to be considered separately—not an easy approach to get right on short notice. For now, we are stuck with many useful programs that offer only minimal assurances of security; therefore what we require is a general, external protection mechanism.

2. Adding new protection features into the OS: We reject this design approach for several reasons. First, it is inconvenient. Development and installation both require modifications to the kernel. This approach, therefore, has little chance of becoming widely used in practice. Second, wary users may wish to protect themselves without needing the assistance of a system administrator to patch and recompile the operating system. Third, security-critical kernel modifications are very risky: a bug could end up allowing new remote attacks or allow a compromised application to subvert the entire system. The chances of exacerbating the current situation are high. It is better to find a user-level mechanism that lets users protect themselves and lets pre-existing access controls serve as a backup; then even in the worst case, introducing the mechanism cannot reduce system security.

3. The pre-existing reference monitor: The traditional operating system's monolithic reference monitor cannot protect against attacks on untrusted applications directly. At most, it could prevent a penetration from spreading to new accounts once a user's account has been compromised, but by then serious damage has often already been done. In practice, even that goal is unattainable. Against a motivated attacker, most operating systems fail to prevent the spread of penetration; once one account has been subverted, the whole system typically falls soon thereafter.

4. The conventional network firewall: Packet filters cannot distinguish between different types of HTTP traffic, let alone analyze the data or mobile code contained therein for security threats. In theory, a proxy could, but it would be extremely hard-pressed to understand all possible file formats, interpret the often-complex application languages, and squelch all dangerous data. This would make for a very complex and thus untrustworthy proxy.

5. Java: Java has two significant limitations. First, legacy systems have little to gain from Java: legacy code cannot run inside a protected Java environment, and rewriting an entire application in Java is often too much work. In contrast, because it is orthogonal to application code, Janus can help protect legacy code as easily as new code. Second, many security researchers have concerns over Java's ability to contain mobile code reliably with high assurance. In this regard, we can help. One can use Janus as a wrapper around Java as a form of "belt-and-suspenders" security—an adversary would have to penetrate both Java *and* Janus to compromise security—so this approach offers strictly better security for mobile code than Java can provide on its own.

6. Proof-carrying code: Proof-carrying code [54, 59] and its ancestor, software fault isolation [71], provide a way for a compiler to prove to a runtime verifier that the code it generates satisfies some security policy; this is typically done by embedding extra checks at any operation that cannot statically be proven safe. This provides a powerful and flexible framework for executing mobile code, and it allows us to specify very fine-grained

security policies (so long as we can formalize them in the appropriate formal logic) such as memory safety. However, proof-carrying code systems force us to specify the security policy at compile time, instead of at run time, and existing proof-carrying code systems cannot handle pre-compiled legacy code. Therefore, Janus seems preferable when the policy might change or when we need to handle legacy code.

This overview shows that the traditional paradigms are either impractical or insufficient. Application re-writes are too much work; kernel modifications are too difficult and too risky to deploy. At the same time, standard host security and firewall mechanisms are inadequate for the task at hand. Finally, the need to protect legacy applications rules out Java. A new approach is needed.

3 An overview of this thesis

We have organized this thesis around four high-level topics: implementation approaches, applications, lessons, and other work.

In Chapter 2, we describe the design and implementation of the Janus prototype. Section 4 addresses design issues, Section 5 touches on a few interesting features of our implementation, Section 6 briefly examines assurance issues in our tool, and Section 7 outlines the core security policy that we have converged on.

Next, Chapter 3 discuss a number of interesting applications for Janus. It covers protection for mobile code, web browser helper applications, web browsers, `sendmail`, other security-critical system daemons, and a few other possibilities. This shows that Janus is a powerful tool with broad applicability.

Third, we analyze the implications for OS designers in Chapter 4. We examine various interfaces for system-call interposition in depth, conclude that Janus requires only minimal support, and argue that OS designers ought to include this support.

Fourth, Chapter 5 addresses other work relevant to Janus. Section 15 surveys some related work, Section 16 discusses some limitations of our approach, Section 17 proposes some areas for future work,

Finally, Chapter 6 concludes the thesis.

Chapter 2

The Design and Implementation of Janus

Janus is a powerful tool for confining untrusted applications. In this chapter, we describe the design and implementation of our prototype. To support the design and implementation choices we made, we also briefly discuss why we believe our tool is secure and outline a basic security policy for it.

4 Design

Our design, in the style of a reference monitor, centers around the following basic assumption:

> AN APPLICATION CAN DO LITTLE HARM IF ITS ACCESS TO THE UNDERLYING OPERATING SYSTEM IS APPROPRIATELY RESTRICTED.

Our goal, then, was to design a user-level mechanism that monitors an untrusted application and disallows harmful system calls[1].

A corollary of the assumption is that an application may be allowed to do anything it likes that does not involve a system call. This means it may have complete access to its address space, both code and data. Therefore, any user-level mechanism we provide must reside in a different address space. Under Unix, this means having a separate process.

One of our basic design goals was SECURITY. The untrusted application should not be able to access any part of the system or network for which our program has not granted it permission. Like others before us, we use the term *sandboxing* to describe the concept of confining an untrusted application to a restricted environment, within which it has free reign. This term was introduced before in, e.g., [71] (although it was used there in a slightly different setting).

[1] As an aside, it is worth mentioning that the boxed principle above is not sufficient on its own to prevent denial-of-service attacks. Fortunately, in practice not much extra work is required: we simply set limits on the application's resource usage using the Unix `setrlimit(2)` primitive before transferring control to the application. See also Section 5.3.

To achieve security, a slogan we kept in mind was "keep it simple" [47]. Simple programs are more likely to be secure [25, Theorem 1]; simplicity helps to avoid bugs, and makes it easier to find that which creep in. We would like to keep our mechanism simpler than the applications that would run under it.

Another of our goals was VERSATILITY. We would like to be able to allow or deny individual system calls flexibly, perhaps programatically depending on the arguments to the call. For example, the `open` system call could be allowed or denied depending on which file the application was trying to open, and whether it was for reading or for writing. Maintaining flexibility allows us to use the tool to protect many different types of applications.

Our third goal was CONFIGURABILITY. Different sites have different requirements as to which files the application should have access, or to which hosts it should be allowed to open a TCP connection. In fact, our program ought to be configurable in this way even on a per-user or per-application basis.

On the other hand, we did *not* strive for the criteria of safety or portability of applications. By *safety*, we mean protecting the application from its own bugs. We allow the user to run any program he wishes, and we allow the executable to play within its own address space as much as it would like. In other words, our tool focuses on security for the untrusted application, not correctness.

We adopted for our program, then, a simple, modular design: a *framework*, which is the essential body of the program, and dynamic *modules*, used to implement various aspects of a configurable security policy by filtering relevant system calls. This decomposition allows us to separate mechanism from policy [47].

The framework reads a configuration file, which can be site-, user-, or application-dependent. This file lists which of the modules should be loaded, and may supply parameters to them. For example, the configuration line

```
path allow read,write /tmp/*
```

would load the `path` module, passing it the parameters "`allow read,write /tmp/*`" at initialization time. This syntax is intended to allow files under `/tmp` to be opened for reading or writing.

Each module filters out certain dangerous system-call invocations, according to its area of specialization. When the application attempts a system call, the framework dispatches that information to relevant policy modules. Each module reports its opinion on whether the system call should be permitted or quashed, and any necessary action is taken by the framework. We note that, following the Principle of Least Privilege, we let the operating system execute a system call only if some module explicitly allows it; the default is for system calls to be denied. This behavior is important because it causes the system to err on the side of security in case of an under-specified security policy.

Each module contains a list of system calls that it will examine and filter. Note that some system calls may appear in several modules' lists. A module may assign to each system call an arbitrary function that validates the arguments of the call *before* the call is executed by the operating system.[2] The function can then use this information to optionally

[2]In addition, a module can assign to a system call a similar function which gets called *after* the system call has executed, just before control is returned to the untrusted application. This function can examine

update local state, and then suggest allowing the system call, suggest denying it, or make no comment on the attempted system call.

The suggestion to *allow* is used to indicate a module's explicit approval of the execution of this system call. The suggestion to *deny* indicates a system call that is to be denied execution. Finally, a *no comment* response means that the module has no input as to the dispatch of this system call.

Modules are listed in the configuration file from most general to most specific, so that the last relevant module for any system call dictates whether the call is to be allowed or denied (unless it has "no comment"). For example, a suggestion to allow countermands an earlier denial. Note that a *no comment* response has no effect: in particular, it does not override an earlier *deny* or *allow* response.

Normally, when conflicts arise, earlier modules are overridden by later ones. To escape this behavior, for very special circumstances modules may unequivocally allow or deny a system call and explicitly insist that their judgement be considered final. In this case, no further modules are consulted; a *super-allow* or *super-deny* cannot be overridden. The intent is that this feature should be used quite rarely, for only the most critical of uses. Write access to `.rhosts` could be super-denied near the top of the configuration file, for example, to provide a safety net in case we accidentally miswrite a subsequent file access rule. We have never used the *super-allow* keyword, and in retrospect it probably should have been omitted from the design.

Relying on the ordering of the policy rules is potentially error-prone, but it gives the policy specification language great power and flexibility by allowing successive refinement of the policy. For instance, it allows us to implement the principle of least privilege easily: we can deny a broad class of system calls at the top of the configuration file, and later craft out limited exceptions to this broad rule with modules listed further down in the configuration file. See Figure 2.2 for an example where this technique is used to initially deny file access to most absolute pathnames and later allow access to relative paths and selected absolute paths. However, in hindsight we did not use the ordering rules as much as we initially expected, and this feature could probably be eliminated with better module design and with a slightly more powerful syntax for pattern matching.

Note also that the existence of *super* keywords may cause the tool to behave in unexpected ways if modules have visible side effects, since a *super* judgement will prevent later modules from executing. In practice, none of our modules cause *super* judgements where side effects could occur, but in retrospect, this was blind luck rather than careful design. This provides additional evidence that relying on the ordering of policy rules may have been a mistake.

To sum up, in designing the framework we aimed at providing simplicity and versatility as much as possible, though these goals often conflict. In retrospect the policy specification language probably could have been simplified with little loss in power, but on the whole we are quite happy with the rest of our framework for dispatching system calls.

These goals of versatility and configurability provided sound reasons to reject an approach based on an in-kernel implementation. It is extremely difficult for kernel code to read configuration files. Also, ease-of-development and similar benefits must be discarded

---

the arguments to the system call, as well as the return value, and update the module's local state.

with any kernel implementation. Finally, as we mentioned before, a kernel implementation runs the risk of introducing new security holes to the system, and system administrators are much less likely to install any tool that requires modifications to the kernel. Therefore, we focused on building a user-level solution.

One of the core benefits of our design is its simplicity. Simpler programs are typically more likely to be correct and easier to audit, and our final implementation contains only a few thousand lines of code. Furthermore, much of the complexity can be found in just one or two modules; those modules need not be loaded for applications that don't need them, so for such applications the tool should have even better assurance properties. By allowing the implementation to be as simple as possible[3], we have improved our chances of getting the code reasonably correct.

5 Implementation

5.1 Choosing an operating system

In order to implement our design, we needed to find an operating system that allowed a user-level process to watch the system calls executed by another process and to control the second process in various ways (such as causing selected system calls to fail).

Luckily, most operating systems have a process-tracing facility, intended for debugging purposes. In such cases, typically one can find a program, such as `trace`, `strace`, or `truss`, that can observe the system calls performed by another process as well as their return values. Under the hood, these programs often use the `ptrace` system call, which allows the tracer to register a callback to be executed whenever the traced process issues a system call. Unfortunately, `ptrace` offers only very coarse-grained all-or-nothing tracing: we cannot trace a few system calls without tracing all the rest as well. Another serious limitation of the `ptrace` interface is that many OS implementations provide no way for a tracing process to abort a dangerous system call without killing the traced process entirely. Both of these shortcomings make `ptrace` unsuitable for our purposes.

Some more modern operating systems, such as Solaris 2.4 and OSF/1, however, offer access to a better process-tracing facility through the `/proc` virtual filesystem. This interface allows direct control of the traced process's address space. Furthermore, it supports fine-grained control: we can request callbacks on a per-syscall basis. Finally, the `proc` interface provides a way for the tracer to abort a system call requested by the tracee before the syscall executes.

There are only slight differences between the Solaris and the OSF/1 interfaces to the `/proc` facility. One of them is that Solaris provides an easy way for the tracing process to determine the arguments and return values of a system call performed by the traced process. Also, Solaris operating system is somewhat more widely deployed. For these reasons, we chose Solaris 2.4 for our implementation.

[3] ...but no simpler—with thanks to Albert Einstein.

5.2 The policy modules

The policy modules are used to select and implement security policy decisions. They are dynamically loaded at runtime, so that different security policies can be configured for different sites, users, or applications. We implemented a sample set of modules that can be used to set up the traced application's environment, and to restrict its ability to read or write files, execute programs, and access the network. In addition, the traced application is prevented from performing certain system calls, as described below. The provided modules offer considerable flexibility themselves, so that one may configure them simply by editing their parameters in the configuration file. However, if different modules are desired or required, it is very simple to compile new ones.

Policy modules need to make a decision as to which system calls to allow, which to deny, and for which a function must be called to determine what to do. The first two types of system calls are the easiest to handle.

Some examples of system calls that are always allowed (in our sample modules) are `close`, `exit`, `fork`, and `read`. The operating system's protection on these system calls is sufficient for our needs.

Some examples of system calls that are always denied (in our sample modules) are ones that would not succeed for an unprivileged process anyway, like `setuid` and `mount`, along with some others, like `chdir`, that we disallow as part of our security policy.

The hardest system calls to handle are those for which a function (a "guard") must, in general, be called to determine whether the system call should be allowed or denied. The majority of these are system calls such as `open`, `rename`, `stat`, and `kill` whose arguments must be checked against the configurable security policy specified in the parameters given to the module at load time.

5.3 The framework

Reading the configuration file: The framework starts by reading the configuration file, which can be specified on the command line or as a system-wide default. This configuration file consists of lines like those shown in Figure 2.2: the first word is the name of the module to load, and the rest of the line acts as a parameter to the module.

For each module specified in the configuration file, `dlopen(3x)` is used to dynamically load the module into the framework's address space. The module's `init()` function is called, if present, with the parameters for the module as its argument.

The list of system calls and associated values and functions in the module is then merged into the framework's *dispatch table*. The dispatch table is an array, indexed by system-call number, of linked lists. Each value and function in the module is appended to the list in the dispatch table that is indexed by the system call to which it is associated.

The result, after the entire configuration file has been read, is that for each system call, the dispatch table provides a linked list that can be traversed to decide whether to allow or deny a system call.

Setting up the traced process: After the dispatch table is set up, the framework gets ready to run the application that is to be traced: a child process is `fork()`ed, and the child's state is cleaned up. This includes setting a `umask` of 077, setting limits on virtual memory use, disabling core dumps, switching to a sandbox directory, and closing unnecessary file descriptors. Modules get a chance to further initialize the child's state; for instance, the `putenv` module sanitizes the environment variables. The parent process waits for the child to complete this cleanup, and begins to debug the child via the `/proc` interface. It sets the child process to stop whenever it begins or finishes a system call. (Actually, only a subset of the system calls are marked in this manner; see Section 5.4, below.) The child waits until it is being traced, and executes the desired application.

In our sample security policy, the application is confined to a sandbox directory. By default, this is a new subdirectory created in `/tmp` with a random name, although the `SANDBOX_DIR` environment variable can be used to override this choice. We create a new temporary-use subdirectory in `/tmp` and confine the program to that subdirectory, rather than simply allowing the application full access to `/tmp`, because we want to isolate the untrusted application from other processes running on the same system. If the untrusted application needs to receive data from other processes, we instruct the other processes (via some out-of-band hint) to place their output in an appropriate temporary subdirectory where the untrusted application can find it.

RUNNING THE TRACED PROCESS: The application runs until it performs a system call. At this point, it is put to sleep by the operating system, and the tracing process wakes up. The tracing process (Janus) determines which system call was attempted, along with the arguments to the call. It then traverses the appropriate linked list in the dispatch table, in order to determine whether to allow or to deny this system call.

If the system call is to be allowed, Janus simply wakes up the application, and the system call is executed. If, however, the system call is to be denied, the tracing process wakes up the application with the `PRSABORT` flag set. This causes the system call to abort immediately, returning a value indicating that the system call failed and setting `errno` to `EINTR`. In either case, the tracing process goes back to sleep immediately.

The fact that an aborted system call returns `EINTR` to the application presents a potential problem. Some applications are coded in such a way that, if they receive an `EINTR` error from a system call, they will retry the system call. Thus, if such a application tries to execute a system call that is denied by the security policy, it will get stuck in a retry loop. We detect this problem by noticing when a large number (currently 100) of the same system call with the same arguments are consecutively denied. If this occurs, we assume the traced application is not going to make any further progress, and just kill the application entirely, giving an explanatory message to the user. We would prefer to be able to return other error codes (such as `EPERM`) to the application, but Solaris does not support that behavior.

When a system call completes, the tracing process has the ability to examine the return value if it so wishes. If any module had assigned a function to be executed when this system call completes, as described above, it is executed at this time. This facility is not widely used; it was developed primarily to support the `fork()` system call.

Since all children of untrusted processes are presumed to be untrusted as well, and since our security policy requires that all untrusted processes be traced by Janus, we see that we need to handle the creation of child processes to ensure that this policy is obeyed. When a `fork()` or `vfork()` system call completes, the tracing process checks the return value and then `fork()`s itself. The child of the tracing process then detaches from the

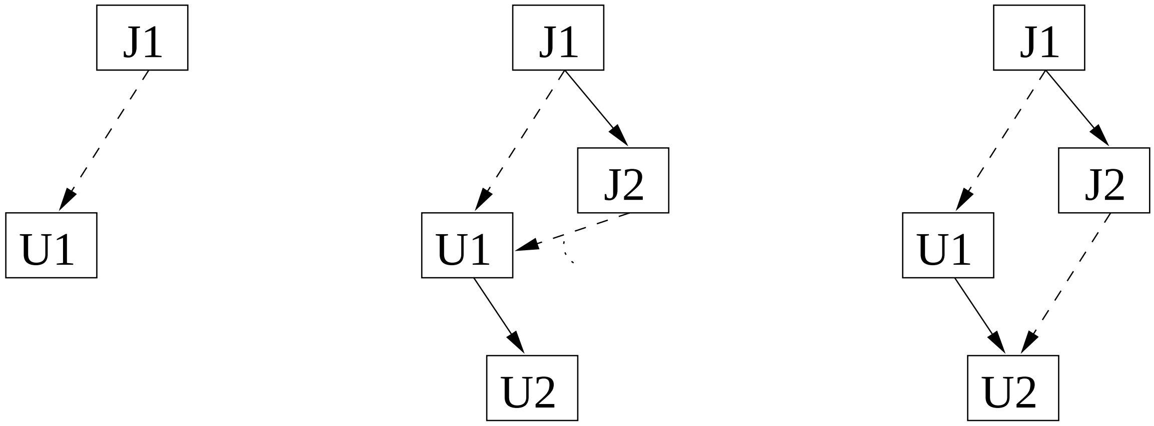

Figure 2.1: Handling the `fork` system call. Solid lines depict the parent-child relationships, and dashed lines are drawn from tracer to tracee. In the leftmost picture, untrusted application U1 is being traced by Janus process J1. In the middle picture, U1 has forked an untrusted child process U2, and Janus responds by pausing U2 and forking a copy of itself J2 to handle U2. (With the `/proc` filesystem, by default J2 inherits the tracing relationship with U1.) In the rightmost picture, J2 has detached from U1 and attached to U2; now U2 may be restarted.

application and begins tracing the application's child. Once the child Janus process has enabled process tracing on the application's child, the application's child is allowed to begin executing. See Figure 2.1 for a graphic depiction of this process. This method safely allows the traced application to spawn a child (as `ghostview` spawns `gs`, for example) by ensuring that all children of untrusted applications are traced as well.

We have not aimed for extensive auditing, but logging of the actions taken by the framework would be easy to add to our implementation if desired.

We should point out that the Solaris tracing facilities will not allow a traced application to `exec()` a setuid program. Therefore, untrusted applications cannot increase their privileges. Furthermore, traced programs cannot turn off their own tracing.

5.4 The optimizer

Our program has the potential to add a nontrivial amount of overhead to the traced application whenever we intercept a system call. In order to keep this overhead down, we obviously want to intercept as few system calls as possible. However, we do not wish to give up security to gain performance. Therefore, we apply several optimizations to the system-call dispatch table when it is created.

One optimization is a simple form of constant propagation. We note that one common case arises when a module's system-call handler always returns the same allow/deny value (and leaves no side effects); this special case allows us to remove redundant values in the dispatch table.

The most important optimization observes that certain system calls, such as `write`, are always safe. This strategy is appropriate because IO in Unix is built around a capability model: one requests capability via the `open` system call, and if the `open` call is allowed by the kernel, a reference is returned that may be safely used with e.g. the `write` system call. Since all access checks are done when the capability is bound by the `open` call to a file object, we only need to check the `open` call; and since we trust the kernel to execute `write` requests only when an appropriate capability is presented, we need not check the `write` call.

When we know in advance that a system call will always be allowed, we need not register a callback with the OS for them. This avoids the extra context switches to and from the tracing process each time the traced application makes such a system call, and thus those system calls can execute at full speed as though there were no tracing or filtering in place. By eliminating the need to trace common system calls such as `read` and `write`, we can greatly speed up overall performance.

Fortunately for us, the structure of the Unix system-call interface allows us to apply this type of optimization to many of the system calls most critical to performance. Thus we benefited from an interface that separates binding-time access checks for an object from fast direct access to that object. See also Section 13 for more discussion on this point.

6 Security and assurance

There is no universally accepted way to assess whether our implementation is secure; however, there are definite indications we can use to make this decision.

We believe in security through simplicity, and this was a guiding principle throughout the design and implementation process. Our entire implementation consists of approximately 2100 lines of code: the framework has 800, and the modules have the remaining 1300. Furthermore, we have attempted to minimize the amount of security-critical state where possible. Since the design concept is a simple one, and because the entire program is small, the implementation is easier to understand and to evaluate. Thus, there is a smaller chance of having a security hole go undetected.

We performed some simple sanity checks to verify that our implementation appropriately restricts applications. More work on assurance is needed.

Most importantly, the best test is outside scrutiny by independent experienced security researchers; a detailed code review would help improve the assurance and security offered by our secure environment. All are encouraged to examine our implementation for flaws.

Note that we rely on the OS kernel to enforce some minimal protection properties. For instance, we assume that the operating system will protect Janus from the untrusted application it is tracing.[4] As a slightly more subtle example, we rely on the invariant that the kernel will not allow the traced application to perform a `write` on a file descriptor not obtained via `open`: this allows us to perform the optimization of monitoring only `open` calls

[4] Note that these types of invariants have occasionally been violated when bugs are found in the kernel. A recent example is a bug that let one kill any other process, bypassing permission checks by using idiosyncrasies of the signals interface. Nonetheless, it is our experience that serious security holes in the kernel are far less common than security exposures in application code.

and ignoring all `write` and `read` calls. In general, we have tried to rely on only the most stable, well-understood protection properties guaranteed by the kernel, but if the kernel operates with reckless disregard for security (or worse, with hostile intent), there is clearly nothing Janus can do to remedy the situation.

7 A security policy

We implemented a sample security policy to test our ideas, as a proof of concept. This one policy turned out to be very useful at confining a variety of interesting applications; but certainly if it were not sufficient for some needs, other policies could be easily implemented.

Sandboxed processes may `fork` children, which we then recursively trace. Traced applications can only send signals to themselves or to their children, and never to an untraced application. Our policy carefully limits resource usage and sanitizes environment variables at initialization time.

We impose severe limits on access to the filesystem. We place the untrusted application in a particular directory and allow it full access to files in or below this directory; To prevent escape from this sandbox directory, we ensure that it cannot `chdir` out of this directory and always deny access to paths containing "`..`". We provide the untrusted application with read access to certain carefully controlled files referenced by absolute pathnames, such as shared libraries and global configuration files.

We concentrate all access control in the `open` system call, and always allow `read` and `write` calls. This is safe, because `write` is only useful when used on a file descriptor obtained from a system call like `open`. This approach simplifies matters and also permits an important performance optimization (see Section 5.4).

Of course, protecting the filesystem alone is not enough. Network access must be carefully controlled, lest the untrusted application escape its sandbox and cause harm elsewhere.[5] Yet nearly any practical application will legitimately require access to network resources.

For example, some programs may need to open a window on the X11 display to interact with the user. In our security policy, we allow network connections only to the X display, and this access is allowed only through a safe X proxy. In theory, we could instead buy a second monitor and set the X display of the sandboxed application to point to the second monitor; this would provide robust security, but at great expense. Therefore, we find that we must securely multiplex access to the X display for both trusted and untrusted applications. We found it most natural to enforce this by carefully controlling network access so the untrusted application can only access the display through a trusted application-level proxy.

As X access control is all-or-nothing, X11 does not itself provide the security services we require. A rogue X client has full access to all other clients on the same server, so an otherwise confined application could compromise other applications if it were allowed

[5]This is a crucial limitation of the `chroot` primitive. `chroot` can help control filesystem access reliably, but its Achilles heel is that it does not control network access; for these reasons, `chroot` environments can often be escaped if extra precautions are not taken.

uncontrolled access to X. Fortunately the firewall community has already built several safe X proxies that understand the X protocol and filter out dangerous requests [44, 51]. We integrated our Janus prototype with `Xnest` [51], which lets us run another complete instance of the X protocol under `Xnest`. `Xnest` acts as a server to its clients (e.g. the sandboxed applications), and its entire display is painted within one window managed by the root X server. In this way, untrusted applications are securely encapsulated within the child `Xnest` server and cannot escape from this sandbox display area or affect other normal trusted applications. (For instance, attempting to cut-and-paste from a untrusted window inside `Xnest` to a trusted window will fail, as will the reverse direction.) `Xnest` is not ideal—it is not as small or simple as we would like—but any further advances in X protocol filtering from the firewall community would likely provide immediate benefits for Janus as well.

Additional controls over network access are typically also necessary. For instance, some applications need to be able to resolve hostnames via DNS. As another example, web browsers need to be able to access web servers (typically on TCP port 80).

At the moment, we handle DNS by opening up a small hole in the sandbox to allow UDP connections to port 53 on a single, specially designated local DNS server. We take care to ensure that the designated DNS server runs the most secure DNS code we can find (with no known holes), so a malicious application can't easily escape this way. Also, to help manage the risk, we only grant DNS access to the small subset of programs that actually need it. If Berkeley had a firewall, we would point our untrusted applications to a DNS server outside the firewall that knows nothing about internal hosts. Note that this does introduce a covert channel [48, 31], but from our perspective, we are not worried about covert channels, because we have attempted to prevent untrusted applications from ever getting access to any confidential information in the first place.

It is interesting to observe that the problem of handling DNS securely for sandboxed applications is very similar to the problem of handling DNS securely in a firewall. In the firewalls community this is well-known to be a tricky problem. Our current policy is akin to what most simple packet filters do today; this policy is not perfect, but it can be easily improved. Better would be to interpose a filtering DNS proxy between the sandboxed application and the external DNS server. Such secure DNS proxies have been described in the firewalls literature [23], and we are in a good position to leverage that work. There is no fundamental reason this is not in place at the moment; we simply haven't added it to our prototype yet.

Our approach for handling web browsers is also very similar to policies found in today's packet filters. As before, this could be improved by borrowing techniques from application-level firewalls and interposing a filtering web proxy between the sandboxed application and the outside world.

This seems to be a general principle: techniques for handling network access by sandboxed applications can almost always be stolen directly from the firewalls community. This has very nice benefits, as we can benefit from the experiences of firewalls researchers, as well as borrow pre-existing code. In general, we may be able to do even better than firewalls can, since we can set a per-application policy on network access.

Sample modules: Our modules implementing this sample policy are as follows. The `basic` module supplies defaults for the system calls which are easiest to analyze, and takes no configuration parameters. The `putenv` module allows one to specify environment variable settings for the traced application via its parameters; those environment variables that are not explicitly mentioned are unset. The special parameter `display` causes the confined application to inherit the parent's `DISPLAY`. The `net` module allows us to restrict TCP and UDP connections (both outgoing and incoming) by host and/or port; the default is to disallow all connections. The `path` module, the most complicated one, lets one allow or deny file access according to one or more patterns.

Because this policy is just an example, we have not gone into excruciating detail regarding the specific policy decisions implemented in our modules.

A sample configuration file for this policy can be seen in Figure 2.2.

Figure 2.2: A sample configuration file.

```
# 'basic' sets a simple policy for core interfaces (signals, fork, ioctl, ...)
# This simple policy will be refined by later modules.
basic

putenv display
putenv HOME=.  TMP=.  PATH=/bin:/usr/bin:/usr/ucb:/usr/local/bin:/usr/local/X11/bin
:/usr/bin/X11:/usr/contrib/bin:/usr/local/bin XAUTHORITY=./.Xauthority
LD_LIBRARY_PATH=/usr/local/X11/lib

net allow connect display

path super-deny read,write,exec */.forward */.rhosts */.klogin */.ktrust

# This is the paradigm to deny absolute paths and allow relative paths.
# (Of course, we will later allow selected absolute paths.)
# Assumes someone will put us in a safe sandboxed dir.
path allow read,write *
path deny read,write /*

# Allow certain explicit paths.
path allow read /dev/zero /dev/null /etc/netconfig /etc/nsswitch.conf /etc/hosts
/etc/resolv.conf /etc/default/init /etc/TIMEZONE /etc/magic /etc/motd
/etc/services /etc/inet/services /etc/hosts /etc/inet/hosts

# Note:  subtle issues here.
# Make sure tcpconnect is loaded, to restrict connects!
# /dev/ticotsord is the loopback equivalent of /dev/tcp, ticlts same for udp.
path allow read,write /dev/tcp /dev/udp /dev/ticotsord /dev/ticlts

# Where libraries live; includes app-defaults stuff too.
path allow read /lib/* /usr/lib/* /usr/local/X11/lib/* /usr/local/X11R6/lib/*
/usr/share/lib/zoneinfo/* /usr/local/lib/* /usr/openwin/lib/*

# This is where binaries live; it should look a lot like your PATH.
path allow read,exec /bin/* /usr/bin/* /usr/ucb/* /usr/local/bin/*
/usr/local/X11/bin/* /usr/bin/X11/* /usr/contrib/bin/* /usr/local/bin/*
```

Chapter 3

Applications

There are many applications for the Janus tool. All are centered around the confinement problem, but each has somewhat different operational and policy-level characteristics. In this chapter, we explain how Janus can help to solve these problems, including how we have used it to help improve the security of troublesome legacy system components such as

- `sendmail` and other security-critical daemons,
- execution of Java applets, and
- Netscape and web browsing.

We will also describe how we have used Janus to enable new functionality, such as allowing users to upload executable customization settings to an Internet service.

The primary uses of Janus can be divided into two broad categories. First, it can be used to confine wholly untrusted applications (such as mobile code) to a subset of entirely harmless privileges. In this case, we attempt to devise a tight enough sandbox to prevent the applications from causing any harm whatsoever; this is the classical formulation of confinement. Alternatively, Janus can be used to reduce the impact of future security holes in dangerous but important system services (such as `sendmail`) that are not themselves malicious but may fail to protect themselves adequately against adversarial inputs. These services often contain subtle security holes that might let an adversary take control and execute malicious code under the service's good name. For these applications, we try to limit the harm they can cause if compromised, and the techniques we use tend to look more like compartmentalization then confinement.

In this chapter we will discuss each of these categories in turn, indicating our experience with how well Janus has performed in each of the problem spaces.

8 Classical confinement and wholly untrusted applications

Quite often in computer security we encounter applications that are thoroughly untrusted (or untrustworthy), and we desire some way to restrict them from causing any harm. Perhaps the program's author is unknown, uncertified, untrusted, or known to be malicious; or perhaps the code was downloaded over an insecure link or via an untrusted intermediary. In all these cases, we want to protect the system from the untrusted software, and in no case should we allow the untrusted code to perform any harmful operation.

Fortunately, in many cases we can devise a very limited set of privileges that is both sufficient for the application's needs and limited enough to prevent deleterious consequences. The confinement problem then is reduced to defining a cautious security policy and expressing it in Janus's policy language. This can require significant domain-specific knowledge, so we describe several important examples.

8.1 Mobile code

The case of sandboxing unsigned mobile code, such as Java applet execution, is an excellent example of this sort of usage of the Janus tool. We can set a highly restrictive security policy (such as "no filesystem access, no network access except back to the host where the applet originally came from, access to the X display only when this display content is well-labeled as untrusted," to take a well-known example) and build a relatively straightforward Janus policy configuration file (disable all `open` system calls, carefully filter `connect` calls, and so on).

In practice for best security we would want to use Janus as a wrapper around the pre-existing Java VM subsystem, in a "belt-and-suspenders" configuration. This does have one small disadvantage: it may require adding extra permissions to the Janus policy file to accomodate the Java VM's needs (such as access to system shared libraries and so on). Fortunately, so long as the security policy is simple and severe, the "policy creep" will typically be minimal.

The advantage of this approach is that we gain defense in depth. Java [41] is seeing widespread deployment, but a number of implementation bugs [31] have started to shake confidence in its security model. Each of these loopholes resulted in total system access for malicious applets, which is an extremely severe failure mode. The vulnerabilities found so far have been patched, and Java's security features seem to be stabilizing as it matures, but it leaves a number of fundamental concerns about the Java's security architecture[31]. For example, the Java implementation includes many thousands of lines of security-critical code, and yet security-critical programs of this scale are notorious for being riddled with bugs and vulnerabilities. Janus helps us address these issues by providing a way to build a robust second line of defense: even if the Java VM is subverted, the attacker will still have to get past Janus's (largely independent) protection mechanisms. Although the user may not trust either Java or Janus alone to protect their critical systems, we argue that the "belt-and-suspenders" combination ought to be good enough for many practical uses.

In our experience this approach to confining mobile code works reasonably well for security policies that are simple and highly restrictive, but complexity introduces significant implementation difficulties. Of course, complex policies are hard to get right. Also, they are often hard to capture in a simple policy language, and Janus's low-level system-call-oriented policy language quickly becomes cumbersome, at best, or unworkable, at worst, for expressing higher-level security policies. Consider the following example: "applets may send arbitrary email messages, except when they contain obscenities or commercial solicitations."

The problem with complex policies goes deeper than this. When using both Java and Janus for security, it is difficult to assure that the policies they implement remain coherent, especially when the underlying policies are complicated. Contributing to this effect is a fundamental mismatch in the granularity of their mechanisms: Janus can only implement policies that are expressible in terms of system-call filtering, while Java supports a much richer set of application-level security semantics. Thus, many natural Java access rules are simply not expressible (or implementable) in Janus's model; as the security policies we wish to enforce become more complex and more expressive, the mismatch between the Java model and the Janus model will only become more and more glaring. In the face of such a conflict, we have only two choices: make Janus's policy either more permissive or more restrictive than Java's. Neither tactic is particularly attractive: the former reduces the benefit of Janus as a second line of defense, whereas the latter may prevent many legitimate applications from running. This is especially problematic, since the primary motivation behind developing more complex security policies is usually to support more sophisticated applications. We don't see any good solution to this. The primary appeal behind Janus stems from its simplicity and its focus on controlling a relatively simple, low-level interface; any change in this philosophy would seriously impact its assurance level. This is an important lesson: Janus is weak at implementing complex security policies.

But we are happy with the Janus model. Because of the difficulty of getting complex policies right[1], we are fundamentally suspicious of any use of Janus that relies on complex policy specifications. We prefer an approach that is simple (and secure) to one that is complex and expressive (and insecure).

8.2 Uploading personalized agents

Sandboxing is also an important technique for securing mobile agents and other uploaded code. To demonstrate the applicability of Janus in this arena, we extended TranSend [36, 35] (a proxy-based Internet service for speeding up web browsing performance, built on top of a general service-construction toolkit called TACC [36]) to support secure agent uploading. TACC is designed to support scalable, highly available, massively customized Internet services running on clusters of commodity workstations. In the TACC model, the service developer builds "workers" to transform and aggregate data on behalf of the end user; TranSend's workers perform data transcoding and compression. We designed and implemented a secure mechanism for dynamically uploading workers to a TACC-based service such as TranSend. This enables construction of extensible services, reduces barriers to third-party innovation, and allows us to take service customization and personalization farther than previously considered. As a natural by-product, our mechanism also immediately enables secure use of mobile agents.

Security is especially critical in such an environment. The security problem arises because the users uploading new workers are invariably unknown and untrusted. If we simply ran their uploaded code without further care, they would be free to execute arbitrary code on the system, compromise its integrity, crash it at will, snoop on other users, rewire its internal logic, and generally take total control of the service. Because Internet services

[1]Even simple policies often exhibit unexpected behavior, to say nothing of surprises in more complex specifications.

may serve millions of users, both the exposure to risk and the potential impact of security compromise are extremely high.

A pivotal technique in our implementation is the use of a transparent wrapper for security. We take the untrusted worker, and apply a wrapper around it to get a nested worker; the wrapper securely confines the underlying untrusted code in an appropriately restricted sandbox, so the nested worker is now a secure, trustworthy, safely usable object. The crucial aspect of this technique is transparency: the nested worker looks just like a normal worker to the PTM, frontend, etc., so the wrapper is transparent to the rest of the system; furthermore, the programmer need not do anything special when developing an untrusted worker, so the security wrapper is transparent to the underlying worker.

This sort of transparency is an effective way to minimize the burden of security and avoid excess complexity. For example, the PTM programmer doesn't want to have to keep track of which workers are trusted and which are not; similarly, we don't want the PTM to be security-critical, so we wouldn't trust the PTM's lists even if it were to maintain them. Also, because our protection mechanism is transparent, to implement Perl workers (for example) we can run the entire Perl interpreter inside the sandbox; this way, we need not trust the correctness of the Perl interpreter, which is far too large to trust directly.

A sandboxing approach is especially attractive in this context because TACC workers tend to satisfy several key properties:

- Distillers are typically stateless. (More specifically, they typically keep no state across requests. If they do need to manage state, they will typically store it in the cache or in the user-profile database.)
- Distillers typically require little or no interaction with the rest of the system.
- Distillers typically need little access to the underlying filesystem, network, and other machine resources. Almost all data flow goes via `stdin` and `stdout` (or rather, their TACC analog). Workers are largely a vehicle for expressing well-encapsulated data transformation tasks.

Since the TACC model is roughly analogous to the Unix pipeline model—it encourages programmers to organize their computation into a series of small, modular, stateless filters that implement some sort of transformation on the data—we see that these desirable properties are largely a consequence of the TACC model. Nonetheless, we expect that they would also apply to many other programming models for mobile agents.

In short, the approach of wrapping an untrusted object with a secure sandbox tool has many compelling advantages. We found Janus to be very well-suited for this task: it allowed us to protect arbitrary workers, no matter what language they were written in (including C, Perl, Java) at no extra cost; and it allowed us to keep the size and complexity of the trusted computing base down, thereby increasing the assurance level of our implementation.

There are some downsides to sandboxing. Sandboxing typically is poorly suited to controlling communications-oriented interaction, and this problem is especially severe with Janus. In the face of heavy use of network-based fine-grained interaction, we can only control the initial access to the underlying network medium—for instance, we can specify which (host,port) pairs the untrusted worker is allowed to connect to—but we have essentially no control over the content of the messages. In other words, Janus gives us only coarse-grained control over worker-system interactions. Since we didn't need fine-grained control over network communications, these disadvantages didn't affect us much, but it is conceivable that they might be more problematic in other settings.

Re-using existing Janus code made implementation of sandboxing relatively easy. Not many changes to Janus were needed: we added extra support to Janus for mediating the `sendto` system call, for controlling some `setsockopt` and `getsockopt` features that TACC used, and for allowing IP multicast. Apart from these minor, well-confined modifications, though, we used Janus as is. The hardest part was picking an appropriate security policy.

We derived our security policy by initially prohibiting access to all interesting machine resources and watching what broke when workers run under this incomplete environment. This gave us a fairly complete list of the resources to which workers (and worker stubs) need access. We went through this list and evaluated each one to see whether allowing it would open any new security vulnerabilities. Finally, with a list of minimal necessary permissions (verified safe and refined to respect the principle of least privilege) in hand, we enabled them in the Janus policy configuration file.

To summarize the policy, we allow:

- **Network access** to a few specific locations: untrusted workers may contact two local trusted high-security DNS servers, the Harvest cache, the PTM, and multicast groups corresponding to the monitor and the PTM. These permissions are specified at a (host,port) level of granularity.

- **Filesystem access** to a number of files: local configuration files, libraries, TranSend executables, TranSend's configuration file, and the area where the executables for uploaded workers live. These permissions are exclusively limited to read access (with a few specific exceptions for write access, e.g., `/dev/null`).

- **Hand-screened OS system calls** that perform basic, required, roughly risk-free functionality. These primarily consist of system calls that can be used to modify one's own sandbox but cannot affect anything outside the sandbox. Most of them are necessary for program execution, to support dynamically linking in shared libraries, managing signal handlers, and similar tasks. This is built from core Janus code that has remained stable for over a year or more.

This policy also ensures per-user isolation, which helps protect mutually distrusting users from each other so one user can't assail the integrity of another user's web browsing experience. Selection and verification of this policy was made easier because it is primarily a refinement of a policy developed earlier (see Section 7) for confining generic mobile code for web browsing. We gave a detailed argument that this policy is safe in [68]; see that document for more details.

We also found it important to develop several other mechanisms to help protect the security of the system, including extensive auditing for problem tracing, simple "trusted path" techniques [32] to deter Trojan horses and viruses, and human-factors engineering to help prevent accidental security lapses caused by configuration and other administration errors. However, Janus remained the bedrock of our security architecture; the other components were relatively minor.

In short, Janus performed admirably well at this task. It adapted well: although it was not initially designed for this application, only minor extensions to the code were needed. Our experience with implementing secure agent uploading helped convince us that Janus is useful as a general-purpose tool for implementing confinement and sandboxing mechanisms.

8.3 Programming contests

We imagine that Janus might also be appropriate for a programming contest environment in which contestants must have very restricted access to the underlying system. However, we have no experience with this configuration, so we can only suggest it as a potential area for further work.

9 Compartmented systems and partially untrusted applications

Next we discuss the case of partially untrusted applications. Classically, confinement is intended for wholly untrusted code; here we show how to apply confinement techniques to protect a system that includes partially untrusted code. The idea is to partition the system into a number of non-interacting subsystems, one for each potential point of compromise, and then wall them off from each other. This way if one subsystem is penetrated, at least the rest of the system stands a chance of remaining secure. This approach is analogous to the concept of "watertight compartments" in maritime construction, where each compartment is separated from the others by a very strong bulkhead. Such a containment strategy enhances robustness in ship design, and it does likewise for system security.

9.1 Helper applications for web browsing

One example of securing partially untrusted applications with Janus can be found in helper applications for web browsers. Web browsers and `.mailcap` files make it convenient for users to view information in a wide variety of formats by demultiplexing documents to helper applications based on the document format. For example, when a user downloads a Postscript document from a remote network site, it may be automatically handled by `ghostview`.

The helper applications these browsers rely upon are security-critical, as they handle unauthenticated data from the network, but the implementations are not particularly trustworthy themselves. Since that downloaded data could be under adversarial control, it is completely untrustworthy. We are concerned that an adversary could send malicious data that subverts the document viewer (through some unspecified security bug or misfeature), compromising the user's security. Older versions of `ghostscript`, for example, allowed maliciously generated documents to spawn processes and to read or write an unsuspecting user's files [20, 26, 27, 66, 72]. In general, the helper applications are generally too big and complex to be bug-free, and large bloated programs are notoriously insecure. (For instance, `ghostscript` is more than 60,000 lines of C; and `mpeg_play` is more than 20,000 lines long.) Also, many helper programs were initially envisioned as a viewer for a friendly user and were not designed with adversarial inputs in mind. Furthermore, `ghostscript` implements a full programming language, with complete access to the filesystem; many other helper applications are also very general. Finally, careless coding (such as using `gets`, `strcpy`, or `sprintf` without protecting against buffer overflow) may allow remote adversaries to totally subvert the helper application and replace it with an arbitrary executable that has full access to the machine. In short, there are significant security concerns.

As a result, we consider helper applications largely untrusted, and we would like to place them outside the host's trust perimeter. We propose to reduce the risk of a security breach by creating a secure restricted environment to contain the untrusted programs, so that even if a remote adversary gains total control over the helper application, the adversary cannot harm the rest of the system. This ensures that the damage a compromised application can cause is drastically curtailed by the restricted environment in which it executes. In contrast, an unprotected Unix application that is compromised will have all the privileges of the account from which it is running, which is unacceptable, because in practice that will typically lead to compromise of the whole system.

In short, we take the view that vulnerabilities in these complex applications are a *fait accompli* which we must accept and live with as best as possible. Rather than trying to prevent such compromises, we pessimistically assume that these programs will exhibit adversarial behavior at the most inconvenient times possible, so we focus on preventing harmful effects. We institute severe restrictions on what privilege the applications may obtain, and scrutinize their actions closely.

With this philosophy firmly in place, it should be clear how confinement techniques can be used to good effect. We advocate the use of Janus to help secure unwieldy helper applications, based on the security policies applied to mobile code above.

Our measurements indicate that the use of Janus imposes virtually no performance penalty on these applications [40]. The negligible performance impact can be attributed to the unintrusive nature of our implementation. Of course, all computations and memory references that do not involve the OS will execute at full speed, so system calls can be the only source of performance overhead. System calls are already so time-consuming that the additional overhead of the Janus filtering is often relatively insignificant compared to the total process execution time. Furthermore, most of the heavily used system calls (such as `read` and `write`) require no access checks and therefore run at full speed. By staying out of the application's way and optimizing for the common case, we have allowed typical applications to run with negligible performance overhead.

9.2 MIME-aware mail agents

Nowadays many of the most popular mail agents are MIME-aware, which means that they can interpret many formats of data. Such support is almost always implemented by use of specialized helper applications, one for each data format that is too complex to implement in the mailer itself. These are usually the same helper applications found in web browsers; for instance, Postscript attachments might be displayed by invoking `ghostview` on the data, and GIF images by executing `xv`. Of course, email can come from arbitrary sources, and the data should be considered untrusted. This presents a serious problem, since many of the helper applications are too complex to handle adversarial data securely.

So it should be clear that the solution to this problem is essentially the same as that for web browser helper applications, above. As before, we could edit the `.mailcap` configuration file to replace each helper application with a Janus-wrapped version. In the case of MIME-aware mailers, one small improvement is possible, since mailers typically execute the `metamail` program and let it dispatch the data to the appropriate helper application. Therefore, we may simply wrap the `metamail` application; Janus will confine not only `metamail` but also all the helper applications it spawns. Wrapping `metamail` is probably preferable in any case, since it is rather poorly written, and has been known to have security holes in the past [22, 29].

9.3 Protected web browsing

The arguments that leave us suspicious of helper applications also apply to web browsers: in both cases, large complex programs are interpreting data that may be under adversarial control, and this indicates a propensity for security holes. For example, a buffer overrun bug was found in an earlier version of the Netscape browser [30].

A natural extension of the work on confining helper applications involves running web browsers under the Janus secure environment. The recursive tracing of child processes ensures that running a browser under Janus will protect all spawned helper applications as well.

The main challenge is that browsers legitimately require many more privileges; for instance, most manage configuration files, data caches, long-term state stored in the filesystem, and make numerous network connections. Nonetheless, these can still be confined to a relatively small "browsing" subsystem.

We have implemented a simple prototype Janus configuration intended to protect the integrity of the rest of the system from a potentially compromised browser. We allow the browser (Netscape Navigator, in our tests) full access to everything under `$HOME/.netscape`. File downloads are restricted to a special "downloads" sandbox directory; after downloading an interesting file, it is the user's responsibility to check it over for safety and copy it out of the sandbox to its end destination. The browser is run inside a filtering X proxy (`Xnest`) to prevent it from manipulating other windows on the same X display. Using `Xnest` has some user-interface implications—it runs inside another window, and one cannot cut-and-paste between Netscape and other X applications—but these limitations could be corrected by moving to a better X11 proxy (e.g. [44]).

The resulting prototype works well enough, in our experience, for day-to-day use, so long as we remember not to try to save anything outside the special "downloads" directory. However, our prototype has seen only limited testing, because Janus is not available on our preferred platform (Linux). Still, our experience suggests that a production-quality system could probably be built with little additional effort.

9.4 Sendmail

The `sendmail` mail agent is infamous within the security community as a favorite point of attack for hackers. It was one of the security holes exploited by the 1988 Internet Worm; and in years since, various versions have had dozens of other security vulnerabilities discovered and exploited. Nonetheless, `sendmail` remains the *de facto* deployed standard for Internet mail communications: one 1996 survey found that "80% of the reachable SMTP servers—more than half a million IP addresses—were running sendmail" and "no SMTP package other than sendmail is running on more than 2% of the reachable SMTP servers." In this sense, `sendmail` represents a trial-by-fire: if a security mechanism can help improve `sendmail`'s security, that mechanism is potentially interesting.

To give the flavor of a typical `sendmail` hole and motivate our approach to the problem, we first describe the vulnerability (also known as "the wizard hole") exploited by the 1988 Internet Worm. [2]

The wizard mode feature was a special privileged debugging mode; network access to the privileged mode was protected by a password specified in the configuration file. To avoid parsing the configuration file every time `sendmail` starts up, as an optimization `sendmail` would save a pre-parsed frozen configuration. This was implemented by merely writing out the entire `bss` and `heap` segments to a file. (Recall that the `bss` segment holds those C program variables that aren't explicitly initialized in the source code; the OS zeroes out that segment at program loading time.) Therefore, the `sendmail` implementor merely had to ensure that all parsed data be contained in the `bss` or `heap` segment for the frozen configuration optimization to work.

However, one implementation of `sendmail` wizard mode made the mistake of declaring the pointer to the parsed password as

```
char *wizpw = NULL;
```

in the source (instead of "`char *wizpw;`"), which meant that it was explicitly initialized and thus contained in the `data` segment. Therefore, the parsed password was not written to the frozen file, and when sendmail was started from a frozen configuration, it acted as though no protective password was enabled: anyone could connect to the SMTP port, begin wizard mode, and get privileged access to an interactive debugging shell without needing a password. The moral is that complex code (such as that found in `sendmail`) is subtle and hard to get right; no matter how many times we test our applications, we can never be sure whether there is one more hidden bug lurking in the code.

Certainly `sendmail`'s status as legacy code has contributed (at least in part) to its security problems. `Sendmail` was adopted as a *de facto* standard at a time when security was less of a concern. As a result, the code is modular and well-engineered, but the module boundaries are not based on a security-oriented decomposition; instead, security-critical code can be found throughout the program. This certainly contributes at least in part to `sendmail`'s history of longstanding security problems; but it is not the whole story.

A much more important factor is `sendmail`'s status as an *evolving* legacy application. Since `sendmail`'s early days, much has changed. As was already mentioned, the

[2] Thanks to Steve Bellovin for this war story [9].

threat model changed during sendmail's lifetime: where the net was once comprised of researchers, today it is teeming with corporations (with assets that are important to protect) and hackers (with the skill and will to mount sophisticated attacks). More subtly, the underlying infrastructure has also changed. For instance, a hostname query used to be a lookup in a local database (/etc/hosts), which thus had no security implications. However, as the net grew, systems moved to a distributed name service (DNS) where hostname queries involved a remote network lookup and thus could not be trusted. But this change was transparent to sendmail. All that changed was the semantics of some library functions, but that was enough: sendmail code that was once secure suddenly became a potential avenue of attack. And in fact, at least one security hole can be directly attributed to this silent evolution of the underlying infrastructure. How can anyone blame sendmail for these security problems, when the semantics of the programming model changed out from under it invisibly? Clearly this class of security failures is hard to anticipate and protect against at design time.

Because of these factors, large numbers of security vulnerabilities have been found in sendmail over the years, and the Unix community has been forced into a *plug-and-patch* methodology, where implementors end up fixing bugs one at a time as they appear. This has been a losing battle [1, 2, 3, 4, 13, 14, 15, 16, 17, 18, 19, 21]. Eric Allman (sendmail's creator) has worked extremely hard to make the best of this unpalatable situation, but this is a fundamentally difficult task.

In short, the history of sendmail is checkered with serious security problems. Because sendmail handles email that could potentially come from anywhere on the net, often the effect of a new sendmail hole is an extremely broad exposure: attacks could come from any of the millions of Internet users. Furthermore, in most cases penetration of sendmail would compromise the entire system it runs on, even though sendmail legitimately needs access only to a very small subset of the system.

These crucial facts dictate our approach to the problem. We consider it prudent to assume that sendmail may be invisibly penetrated at any time by parties unknown. Rather than trying to limit the probability of compromise, we focus on reducing the harmful effects of a potential compromise, and in this way aim to limit and manage the risks as best as possible. Our solution, then, uses a custom Janus configuration to confine sendmail in a minimalist environment, under the assumption that it may be acting with hostile intent at any time; we allow sendmail access only to those parts of the system legitimately necessary for mail delivery, and no more. Sendmail still runs as root, but many dangerous operations are prevented by Janus's system-call filtering.

This allows us to drastically limit the impact of a future sendmail compromise on the rest of the system. Of course, in the event that sendmail is penetrated, the integrity and confidentiality of the mail subsystem will be compromised—Janus cannot help in that regard—but at least we have some assurance that the rest of the system will not fall with it.

Our security policy for sendmail is our most complicated policy developed to date. First, we allow unlimited access to the mail queue (where mail waiting to be processed is stored) and to the mail spool directory tree (where users' mail files are kept). Second, we allow several forms of network access: sendmail may bind to the SMTP port and may connect to SMTP ports on other hosts. Also, sendmail is allowed to perform DNS queries. (Our approach involves allowing sending to UDP port 53 and binding to high-numbered local UDP sockets.) Third, we allow read-only access to certain relevant configuration files (e.g. /etc/sendmail.cf). Fourth, we also allow read-only access to /etc/passwd, so that sendmail may extract GECOS information. (Note that this is potentially dangerous if shadow passwords are not in use, since a compromised sendmail could leak encrypted password entries to a dictionary search engine; for now, we insist that the site use shadow passwords, since sendmail very strongly wants other information contained in the password file.) Finally, any other access not explicitly allowed in the list above is prohibited. (For example, in our prototype, sendmail cannot access users' .forward files; this policy could be easily changed with a slight modification to the configuration file.)

This draws a tight (though admittedly somewhat complicated) boundary around sendmail. The key point is that sendmail is only permitted access to the mail subsystem, so that a compromised sendmail daemon cannot adversely affect the other subsystems. This is just the principle of least privilege applied to mail handling. Because we are using the Janus tool, our boundaries are specified in terms of the interface between sendmail and the OS: namely, as a list of the system calls to be allowed.

Our prototype successfully protects stock Solaris 2.5.1's sendmail version SMI-8.6.[3] We disabled NIS, because of its poor security properties. We have not performed extensive stress-testing of all sendmail functionality; however, we have informally verified that it works for mail transport, forwarding, and delivery.[4]

For concreteness, we describe the harmful actions a compromised sendmail can take when our security protection is in force[5]. It can selectively prevent messages from reaching their destination, or delete messages from the mail spool files. It can perform denial of service attacks on SMTP or DNS servers, but then anyone with network access can already do that. It could snoop on all email passing through the site, but of course, anyone with a sniffer on your local network can often do the same. It could forge email, or modify email in transit through the site, but then, anyone already can forge email. So in sum the possible harmful effects are very limited, and most are already a threat anyway. This limited worst-case impact is a sign that we have successfully compartmentalized the mail subsystem.

This form of protection for sendmail can have huge security benefits. In recent years, sendmail has been the source of a tremendous number of security holes, and we

[3]In future work we will port the custom configuration to work with Eric Allman's sendmail version 8.8.8.

[4]We expect that our prototype may deny certain obscure but legitimate operations from sendmail. However, this is from an inadvertent lack of knowledge of every corner of sendmail functionality, rather than from any fundamental limitation in our technique. For instance, we know that currently our security policy does not make .forward files available to sendmail, though that would be very easy to change. In other words, this is an indication of the immaturity of our sendmail prototype, rather than a fundamental flaw in our approach.

[5]It is hard to guarantee with 100% reliability that this list is complete, because the list of harmful actions is given at a different level of abstraction than the policy (system calls vs. application semantics). Still, this problem is no more acute for Janus than it is for firewalls, and we can make a pretty good assessment of the risks. The basic approach is very simple: for each policy rule which allows some set of actions, we estimate what a malicious application could do with those actions. Fortunately, the set of allowed actions in our policy for sendmail is quite small, and it is this property which makes our task feasible.

can be sure that there will be more in the future. Normally, an attacker who successfully compromises `sendmail` with one of these holes will have the run of the system. The benefit of our security mechanism is that the successful attacker will not be able to corrupt the rest of the system—only the mail subsystem will be at risk. In other words, we have added a robust second line of defense that prevents catastrophic security failures.

One attractive property of this direction to securing `sendmail` is that it complements other approaches nicely. We know that, at worst, even if Janus fails to do its job, our configuration will be no less secure than running without Janus; of course, in practice we do expect Janus to reduce risks significantly. Furthermore, even if other techniques are developed to reduce the incidence of bugs in `sendmail` (e.g. perhaps through careful auditing, code restructuring, formal methods, or other as-yet-unidentified methods), they compose nicely with Janus so that one could get the best of both worlds.

Our customized solution based on Janus has a number of advantages. First, development time was relatively small (especially as compared to the difficulty of securing `sendmail` by traditional means). We were able to re-use the Janus tool with only a few modifications needed. Second, the solution's complexity is very modest. Third, our solution with Janus provides a way to introduce a security-oriented code decomposition, where all security-critical code is collected in a small module, to legacy code not originally structured in this way. Finally, because of its simplicity and orthogonality, assurance is likely to be quite high; certainly there are no guarantees, but our chance of correctness is significantly higher than with just the current plug-and-patch methodology for `sendmail`. In short, the sandboxing approach yields a solid high-assurance tool for securing `sendmail`; Janus is a good fit for this application.

9.5 Other system daemons

Just as Janus was used to secure `sendmail` in case of penetration, we expect that other security-critical system daemons could be protected in a similar way. Potential candidates include `httpd` (a web server), `bind`, `inetd`, etc.

Securing a web server would perhaps be the most compelling application. Like `sendmail`, web servers are well-known to be very high security risks, yet by their very nature web servers typically require only very limited access to the system. `inetd` also makes for an interesting possibility, because it usually spawns many other net daemons that also need protection. Since Janus recursively confines all children of a confined process, wrapping `inetd` in a Janus wrapper would be an attractive technique to secure many network services at once.

Our expectation is that Janus could be used as a general-purpose tool to help manage the risk of security vulnerabilities in many critical network daemons. However, we do not have implementation experience with such a configuration, so we can only suggest this as a promising area for further study.

9.6 CGI scripts

We speculate that Janus may prove useful at sandboxing user-written CGI scripts at sensitive sites (e.g. Internet service providers). As a system administrator, one of the biggest risks with allowing users to write their own CGI scripts is that one of the scripts may inadvertently open up a hole that allows remote adversaries full access to the site. Of course, most users are neither very experienced at writing security-critical programs, nor very knowledgeable about security issues, so this is a real concern. Running all user-written CGI scripts inside a sandbox enforced by Janus might help alleviate some of the security concerns. However, we have no experience with this, so we will leave it to other researchers to explore the space of possible solutions.

10 Summary

This section enumerated many appealing applications for Janus, and gave implementation experience with a number of them. These includes providing security for mobile code and mobile agents, securing legacy systems, and protecting ourselves from future failures of critical system daemons such as `sendmail`. This illustrates the potential benefits from confinement-based mechanisms, shows that our basic approach is workable, and demonstrates the success of Janus as a general-purpose tool for confinement.

Chapter 4

OS support for Janus

Janus cannot exist in a vacuum. The Janus architecture relies heavily on the ability to mediate all interactions between the sandboxed application and its environment, and these mediation facilities can only come from the operating system. Therefore, we require some level of support from the OS if Janus is to do its job.

It is then natural to wonder exactly what level of OS support is required. Operating system designers may wish to decide whether to implement special support for Janus—what are the costs? what tack do current commodity OS's take? is it worth the effort?—and if so, which techniques are appropriate? In this chapter, we explore various levels of OS support, evaluate their value, examine the tradeoffs, and conclude with a recommended approach for operating system architects.

We argue that our research provides strong evidence that the operating system designer can support Janus with little additional cost, if some care is taken during the design of the system's process-tracing facility. This means that OS designers can offer significant security benefits to the user at little extra cost. We describe here our experience with implementing Janus on top of several existing process-tracing facilities, and develop a list of specific features that Janus needs from the OS to work well. Roughly speaking, the core requirement is the ability to implement system-call interposition; with that proviso, we can implement the Janus architecture, and the corresponding security benefits will accrue. Fortunately, most operating systems already implement some form of process tracing, and process tracing takes you most of the way towards system-call interposition, so not much extra functionality is needed from the OS kernel. There are a few subtle traps for the unwary, but our work enables us to offer a set of guidelines that should help OS builders negotiate the pitfalls with ease. Therefore, we claim that folks who build new operating systems should endeavor to support the Janus security tool.

This chapter is organized as follows. First, we argue that, to be viable, Janus needs some sort of support for system-call interposition from the operating system. Next, we describe three existing process-tracing facilities, analyze what functionality Janus needs to extract from them, and use that analysis to provide a detailed evaluation of each of the three tracing facilities. After that, we describe a tracing primitive that we designed and implemented in Linux, and describe what went wrong with that effort. Finally, we present a list of guidelines on the design of tracing primitives that summarize many of the lessons we learned from our work.

Implementors should also see [39] for more insights and implementation experience regarding the design of interposition mechanisms.

11 Theory and foundations

Janus's goal is quite aggressive: it must transparently protect a large legacy system from an untrusted pre-existing application. Recall that Janus is unprivileged, which means that it cannot—by itself—control the untrusted application's interactions with its environment. In short, Janus requires outside help to accomplish this task. However, most other system entities will not provide any help. Neither the untrusted application nor its trusted environment are likely to provide special support for Janus. In practice, they almost always consist of legacy code, and usually will not even be aware of Janus: Janus must operate transparently without assistance from them. The only remaining entity Janus can look to for help is the operating system.

This argument demonstrates that Janus needs OS support to be viable. Specifically, we see two crucial requirements:

1. **Complete mediation:** The operating system must have the power to mediate all potentially unsafe interactions between the untrusted application and its environment, and

2. **Extensibility:** The operating system must allow us to further limit the set of allowed operations. (The OS need not allow us to replace its pre-existing access controls, only to supplement them with additional checks.)

These two properties are sufficient to build arbitrarily powerful user-level confinement mechanisms.

The requirement for **complete mediation** essentially says that the OS must support some form of protection. In order to mediate interactions between an application and its environment, the OS must first be able to distinguish between applications and their environments: we require a well-defined boundary separating an application from all other system components. Furthermore, we insist that the boundary be drawn so that the OS controls (or can control) all interactions across that boundary. In today's operating systems, an application is most naturally viewed as a process along with its associated address space. In this model, applications cannot interfere with each other (or with the OS kernel) except by invoking an operating system primitive.

Most modern operating systems that are even minimally security-conscious implement process-level protection, and so **complete mediation** should essentially come for free. One important exception is Microsoft's Windows 95 (but not Windows NT), which provides no protection: all applications have full access to all of the machine's memory as well as unmediated access to most of the hardware. In other words, Windows 95 fails the first requirement on the OS. In principle one might be to work around this by limiting the untrusted application's access to memory and hardware with software fault isolation (SFI) [71], but it is not clear how effective SFI+Janus would be in practice.

The second requirement, **extensibility**, means that a user-level process, such as Janus, must be able to extend the operating system's default reference monitor in a way that makes it more restrictive. (A reference monitor is an abstraction of the mechanisms the OS uses to validate applications' requests and enforce protection [32].) This is most naturally accomplished through interposition, where Janus gets to monitor and filter the application's requests to the operating system.

The most straightforward way to implement interposition is by a callback: each time the untrusted application invokes a OS primitive, Janus is notified and offered the chance to veto (or cancel) unsafe requests. In most of today's operating systems, the sole interface to OS primitives is through "system calls"[1]; therefore, it suffices to lend Janus the ability to monitor and filter system calls issued by untrusted applications.

Of course, for security reasons, access by unprivileged applications to system-call monitoring/filtering functionality should be carefully controlled and supervised. For instance, Janus should not be allowed to extend the OS's reference monitor in a way that makes it more permissive.

In the remainder of this section, we explore the available OS support for system-call interposition in detail.

12 Real implementations

Most operating systems already offer some kind of support for monitoring the system calls invoked by a target process, in the form of a process-tracing facility. Though these facilities were not intended for use as a system-call interposition mechanism, often they can be adapted to this end, if they are sufficiently general. But not all process-tracing facilities are made equal—they span a whole range of sophistication and flexibility, and not all are adequate for system-call interposition. We list here three notable approaches, ordered from the least powerful to the most powerful:

- `ptrace` is a very common (though minimal) primitive available in most Unix-based operating systems; it is widely used in debugging tools such as `gdb` and `strace`.

- Solaris 2.4's `/proc` filesystem offers very flexible support for fine-grained user-level system-call tracing, with some additional capabilities over `ptrace`. Solaris maintains backwards-compatibility with the older `ptrace` interface by emulating `ptrace` calls using the `/proc` filesystem's newer, more powerful functionality.

- SLIC [39] is a technique for building extensible operating systems via interposition on the system-call interface: instead of receiving a callback from the kernel, with SLIC one may download trusted code into the kernel to be executed before each system call is processed. SLIC offers a powerful OS-independent toolkit to ease the implementation of system-call interposition and free implementors from the painful aspects of writing kernel code. As such, SLIC can be easily used to implement the `ptrace` or `/proc`

[1] Still, one must actually verify that there are no other unrestricted ways—such as a divide-by-zero hardware trap—to invoke OS functionality.

primitives, or can be used to deploy an efficient, customized solution targeted to Janus's needs.

We listed the approaches in order of their expressive power, but this ordering also reflects the relative costs[2] of the three primitives, from least expensive to most costly. It behooves us to pick the least powerful (and thus least costly) primitive that will suffice. We concentrate our attention on these three levels of OS support, with an eye towards how well they meet the needs of Janus.

12.1 Which features Janus needs

First, we examine what OS support Janus needs to be effective. We list the minimum required capabilities here:

- **Complete monitoring powers:** We group several related requirements into a single category. First, Janus must be notified whenever the untrusted application attempts to execute a system call. Furthermore, Janus needs to be able to view the arguments of the system call (and to examine the application's data space, if any arguments live there). Last, Janus must be able to view the return value of the system call. These are all essential if Janus is to monitor the untrusted application effectively.

  All three process-tracing facilities support this fully. Both the `ptrace` and `/proc` interfaces implement a callback from the kernel to the tracing process. `ptrace` uses the Unix signals facility: at each tracing event, the tracee blocks with a `SIGTRAP` signal, and the tracer may detect this by calling `wait`. `/proc` uses `ioctl`s on a special file under the `/proc` filesystem: the tracer issues an `ioctl` that blocks until a tracing event is available at the tracee. SLIC offers many different ways of supporting this requirement.

- **Fine-grained control:** Janus should be able to specify which system calls to receive a callback on and which system calls should be allowed to execute without delay. This greatly improves performance, because we can apply the following major optimization. We trace only `open` system calls, and allow `read` and `write` to execute without tracing; a similar technique can be applied to a number of other system calls. This optimization lets `read` and `write` calls, a common case, proceed at full speed, while still safely controlling all file accesses.

  Only the `/proc` filesystem and SLIC support this requirement. In fact, SLIC goes even further, and lets us optimize the performance of Janus by downloading some security checks directly into the kernel.

- **Ability to prevent selected system-call invocations from executing:** Merely monitoring the actions of the untrusted application is not enough—Janus also needs to be able to take forceful action against mis-behaving applications. In particular, when the untrusted application calls a system call with arguments that are potentially

[2] We picture cost being measured in some natural measure of interest to the OS designer, such as the amount of code one must write to implement the primitive, or the amount of interdependencies with the rest of the kernel the primitive introduces.

unsafe—e.g. `open("/etc/passwd",...)`—Janus must be able to abort the system-call request and prevent its execution.

Only the `/proc` filesystem and SLIC support this requirement; `ptrace` is insufficient.

- **Follow all children of the traced application if it forks:** Clearly Janus's efforts would be for naught if a malicious application could merely `fork` and let its child execute unmonitored. Therefore, the operating system must preserve tracing across forks.

  The `proc` and SLIC primitives do support this. In contrast, `ptrace` does not provide adequate support for this requirement, due to subtle technical issues[3].

These features are the bare minimum required to support Janus.

Next, we list a number of capabilities that would be convenient, though not strictly necessary:

- **Multiplexing:** The ability to monitor many traced processes at once is useful. In our first naive implementation, Janus forked each time the untrusted application did, to maintain the invariant that each Janus process only monitors one traced process. However, this causes process table bloat. Worse, it incurs a large penalty in frequent context switches, and frustrates OS attempts to take advantage of locality. These costs can be mitigated by slightly increasing the complexity of Janus so one Janus process can handle many traced processes at once. Of course, the operating system must support multiplexing for traced processes if we are to achieve this improvement in performance.

  On the other hand, multiplexing does have some tradeoffs. In particular, adding multiplexing to Janus adds complexity, which in turn decreases its assurance level somewhat, so multiplexing might not be appropriate in all environments.

  `ptrace`, `/proc`, and SLIC all support multiplexing.

- **Kill on tracer-death:** With many tracing subsystems, if the tracer dies unexpectedly (perhaps because of a bug in its code), the traced process is no longer traced and thereafter executes without interruption. This is clearly an undesirable situation; we would prefer to be able to set a flag so that the traced process (the untrusted application) is killed if the tracer dies unexpectedly. This capability increases the robustness of Janus in case the implementation has minor bugs.

[3]Note that, with `ptrace`, one can observe when the application has forked. By checking the return value from the `fork` system call, one can deduce the process id of the child. At this point, one could enable tracing on the child. However, this approach fails due to a serious race condition. By the time we enable tracing on the child, it may already be too late—the child may have already executed an unsafe system call.

The `strace(1)` utility attempts to remedy this with a messy trick. When `strace` observes that the tracee is calling `fork`, `strace` sets a *breakpoint* at the next instruction after the `fork` invocation, so that both the parent and the child halt temporarily after the `fork` returns. Then `strace` enables tracing on the child, removes the breakpoint, and resumes execution of both processes. However, this is a painful and unreliable hack. It introduces complexity, race conditions, and failure modes that are unacceptable for a security-critical program such as Janus, and so we reject this approach.

Bugs in the OS are also a concern. One bug has recently surfaced in many Unix kernels whereby one process can bypass the usual protection checks and kill any other process. In this case, a traced program might be able to kill the tracer, which would be a serious problem if the traced program subsequently continued to run untraced. Patches to fix the bug are available, but there will always be legacy systems with old versions of the kernel.

More problematically, this brings up the possibility that other such bugs may be found in the future (perhaps even by the hacker community before they are found in the security community). Of course Janus cannot guarantee security in case the OS kernel is buggy, but it seems valuable to add protection against this particular class of bugs. In the interests of robustness, "kill on tracer-death" seems important.

`/proc` supports this feature; `ptrace` unfortunately does not. The SLIC prototype in [39] does not currently implement this feature [38].

- **Setuid executables forbidden:** We want to prevent the untrusted application from raising its privilege level. In Unix, one gains privilege by executing a setuid (or setgid) program; therefore, it is important that Janus be able to deny the execution of setuid programs (and setgid programs) by traced processes.

  If the OS does not support this feature, we can admittedly work around the omission at some cost in complexity and assurance, as follows. First, whenever an untrusted application attempts to `exec` a new program, Janus must check that the referenced program is not a setuid (or setgid) executable. This leaves open a race condition: a local process could change permissions on the file (or, more likely, make a symbolic link pointing to a pre-existing setuid executable, and move the symlink around) between the time when Janus checks the program's permissions and when the OS kernel executes the `exec` system call. At first glance, this race condition does not seem to affect security. In the Janus threat model, we assume that the traced untrusted application does not have any local untraced co-conspirators. However, there is a subtlety: the untrusted application might fork and try to exploit the race condition by using the child to change a symlink while the parent `exec`'s the symlink. Therefore, if the OS doesn't prevent execution of setuid executables directly, Janus would also have to prevent creation of links and modifications to file permissions that would enable this race condition.

  We see that Janus can work around a lack of OS support for forbidding access to setuid executables, but the workaround is complex, which makes it potentially susceptible to security holes. In short, the setuid check rightfully belongs in the kernel where it can be made atomic. Therefore, it would be extremely useful for the OS tracing subsystem to support this capability in the kernel.

  All three tracing subsystems provide this support.

- **Flexible error codes:** In practice, Janus will often deny many system calls from well-intentioned non-malicious applications, merely because the application unwittingly tried to overstep its bounds. In such cases, we want the application to recover gracefully and continue execution. Most applications do, if we simply abort execution of the system call. However, some applications have problems if the aborted system call returns an error code other than `EPERM`. For instance, on Solaris aborted system calls return `EINTR`. (The `EINTR` return code was originally intended for system calls that are interrupted by a signal and should be restarted by the application.) This occasionally causes problems: applications that dutifully restart such system calls will get into an infinite loop, each time retrying the same system call and each time having it denied by Janus. `EPERM` is a more natural error code that reflects the true cause of the failure (namely, a permission check failed) and thus causes fewer problems. Therefore, to best handle the widest possible array of applications, it would be convenient if the operating system allowed denied system calls to return `EPERM`.

`ptrace` does not provide the ability to abort system calls, and so control over the error code is moot. SLIC easily handles `EPERM` return codes for aborted system calls. As for the `/proc` tracing subsystem, our original implementation using `/proc` did not have control over the error code returned; however, it is now possible to force a `EPERM` return code under the `/proc` filesystem by using a special trick.

These optional features enhance the performance, assurance, and transparency of Janus.

Lest we lose all sense of perspective in a maze of wish lists and required features, we should take care not to forget the original reason for enumerating the desired OS support. The intent is to analyze the suitability of the three primary tracing primitives (`ptrace`, `/proc`, and SLIC) with a two-step process: first, decide what functionality we need from a good tracing primitive, and then evaluate how well the available primitives meet those needs. Having now completed the first step in great detail, we move on to evaluate the existing primitives against this checklist.

12.2 Evaluating SLIC

As the most powerful mechanism, SLIC is the easiest to evaluate. The SLIC primitive can support all of the capabilities we need and all of the convenient features we want except "kill on tracer-death." Even better, SLIC's powerful features make it easy to implement the support Janus needs with a minimum of fuss and without much additional complexity; simplicity is a compelling advantage, because it helps wards off bugs in Janus, thus maximizing the chances that Janus does correctly confine untrusted code.

Moreover, SLIC can help with security and performance by moving permission checks into the kernel when appropriate. File permissions, for example, can be checked in the kernel to avoid race conditions and to avoid duplicating the existing file permissions kernel code.

Duplicating the kernel's standard file permission checks in user-space is dangerous: preventing race conditions can be tricky, and it is difficult to ensure consistency between the kernel and user-space implementations. In some cases, duplication is impossible: for instance, user-space code cannot get direct access to inode data and so some checks are impossible. Therefore, we feel that duplication of access control checks in both kernel and user-level code is best avoided wherever possible, and SLIC makes it possible to avoid this type of duplication.

Security is not the only consideration; SLIC can also help with performance in ways that the other primitives cannot. With SLIC, one can optimize away hot spots by downloading code into the kernel to check commonly used system calls. Remember that the principle cost of Janus lies in the extra context switch incurred on each potentially-unsafe system call. If one finds a bottleneck in execution performance—suppose one application uses the `sendto` system call very frequently, say—then one can eliminate the extra context switch for `sendto`, and thus the performance bottleneck, by moving the `sendto` security checks into the kernel. This ability to tune performance to eliminate hot spots is a valuable advantage.

In summary, the power of SLIC provides very strong support for Janus, significantly enhancing performance and assurance levels. The only drawback is that SLIC does add considerable complexity to the OS.

12.3 Evaluating `/proc`

The Solaris `/proc` filesystem is nearly as powerful as SLIC. It supports all the capabilities Janus needs; it also supports all the convenient features listed earlier. The first implementation of Janus was built using the `/proc` tracing mechanism, and the combination has performed beautifully. (We will see later some areas where `ptrace` fails due to subtle pitfalls while `/proc` excels.) In short, the `/proc` tracing primitive is powerful, elegant, and well-engineered—it is an excellent choice for use with Janus, where available.

12.4 Evaluating `ptrace`

In contrast, the `ptrace` primitive has a number of serious failings that make it simply unsuitable for our purposes. First of all, fine-grained control over which system calls get traced is not available—with `ptrace`, it's all or nothing. This is very bad for performance. Secondly, it is impossible to follow `fork`s with `ptrace` securely, so Janus would not be able to confine applications that legitimately need to spawn children.

Far worse, `ptrace` doesn't offer the ability to abort system-call requests that fail Janus's security checks. Without OS help, the only alternative is to kill the traced process brutally before a dangerous system call is executed. This response is far too precipitous. In practice, most large applications will occasionally issue system-call requests that Janus denies; but those requests are typically not critical, and if they are denied the application usually recovers gracefully. In contrast, if we were to kill the application in response to such syscalls, graceful recovery would be impossible, and Janus would render most non-trivial applications unusable. To drive the nail in the coffin, it is possible that there might be a race condition: the system call might be executed before the traced process is killed.

Therefore, we find that `ptrace` is totally inadequate: it is too slow and far too limited to support Janus's needs.

12.5 `ptrace++`

The inadequacy of `ptrace` is very unfortunate, for `ptrace` is a sort of "least-common-denominator" standard for process tracing. Most operating systems implement the ptrace primitive. In particular, it is the only tracing primitive supported on Linux, our preferred platform. However, Linux has a powerful advantage: it comes with source. This makes Linux especially well-suited to experimentation with potential OS extensions. Therefore, we decided to test our analysis of what OS support Janus needs by extending ptrace. We set out to support Janus more effectively by implementing a more sophisticated tracing primitive, which we call ptrace++.

The new ptrace++ interface implements a few simple new features over and above the standard ptrace feature-set. First, ptrace++ allows system-call requests to be aborted before they are executed. The error code is EPERM by default, though any other code can be easily specified. Also, there are two new features that may be enabled on a per-process basis: tracing features are inherited across forks when follow-fork is enabled, and fine-grained control over which system calls are traced is available. Thus, ptrace++ is backwards-compatible with ptrace, but the new extended functionality may be invoked by any process (such as Janus) that understands the new extensions to the interface.

Implementing ptrace++ on Linux was not very difficult. The primary challenge lay in handling the new requirement for per-process state. The vanilla ptrace interface was stateless, but ptrace++ needs to keep state for which features are enabled and which system calls to trace. One natural approach for managing state would be to add a new entry to the Linux process structure; however, the organization of the Linux source makes this change very unwieldy[4]. Instead, we followed a more modular and somewhat less preferable approach: a separate data structure containing per-process tracing state. Our implementation uses a linked list in static kernel memory, indexed by process id, with an entry for each traced process with extended ptrace++ functionality enabled. With this strategy, one has to do some extra work to avoid memory leaks. In particular, one must ensure that a process's tracing state is freed after the process exits.

In the end, our implementation of ptrace++ added about 185 lines of C to a 415-line file containing the core of the ptrace kernel code. Most of the new code was dedicated to managing the per-process state. The attempt to confine the kernel modifications was largely successful: apart from the changes to the core ptrace code, only two minor additions—one affecting fork and one to exit—were required elsewhere in the Linux kernel.

We then extended Janus to use the ptrace++ interface. The development process was lengthy and painful. Most of the effort was dedicated to identifying, isolating, and working around obscure idiosyncrasies in the ptrace++ primitive. Most of these speed-bumps can be attributed to ptrace's design: it overloads the Unix signals mechanism to provide upcalls on each system-call event, and the tracer receives the "upcall" by waiting for a SIGTRAP signal from the tracee. In general, it is well-known in object-oriented circles that operator overloading tends to be rife with subtle pitfalls. In ptrace's case, the overloading of the signals mechanism creates a number of thorny problems for users of ptrace.

We list a few of the less-obvious implementation pitfalls here:

- **ptrace breaks wait:** One serious problem with overloading the signals mechanism

[4] The size of the process structure is hard-coded in a number of places—for example, in some assembly files—and adding a new entry would require changes in a number of other places. It wasn't clear just how large an avalanche this would create, and we weren't anxious to find out, so we avoided changing the process structure and instead built a system of shadow structures.

is that some signals-related functionality breaks in certain circumstances, ruining the transparency of tracing. In particular, if both child and parent are traced, and the parent `wait`s for a signal from the child, things will break. Due to interference from the tracing, the parent's `wait` may return early, perhaps with notice of a *tracing event* from the child (which is only supposed to be received by the tracing application, not the traced parent). In practice, this interference usually confuses the parent enough that it fails to recover and dies. Because of fundamental flaws in the `ptrace` architecture, it is not possible to fix this in the kernel without drastic measures.

We attempt to work around this failure mode in the user-level Janus code; unfortunately, heroic efforts are required. Janus traps `wait` calls by all traced parent processes that have traced children. When such a trap occurs, Janus manually suspends the parent (i.e., refrains from resuming the process and leaves the process in a stopped state). Janus doesn't allow the parent to continue execution until one of its children enters a state that might (legitimately) cause the parent's `wait` to return; then Janus manually restarts the parent and allows its `wait` request to execute and return. This hack has worked well enough, so far. However, we are not happy with it. It adds complexity—about 40 (highly subtle) lines of C code of a 380 line adaption layer. It also leaves us with some nagging doubts about whether the extra code opens up new security holes.

- **Can't transfer control securely:** One obscure limitation of `ptrace` is that a traced process may have at most one tracer at any time. This makes it impossible to transfer control of an untrusted application from one Janus process to another atomically[5]. This, in turn, requires that when a Janus instantiation controls an untrusted application, it must also control all its children[6] (since it originally has control of each child created by a `fork` and cannot give away this control securely).

  In fact, the one-tracer-per-tracee limitation of `ptrace` can ultimately be blamed on `ptrace`'s signals-based upcall mechanism. Because the tracer `waits` for a `SIGTRAP` signal from the tracee, and because the semantics of `wait` are that a process can only `wait` on its children, the tracer must be the "parent" of the tracee; but a process can only have one "parent", giving rise to the one-tracer limitation. Thus we see that `ptrace`'s architecture fundamentally prevents secure transfer of control, and no extension to `ptrace` can remedy this shortcoming.

[5]The obvious approach would have the prior owner detach the untrusted application—causing it to resume execution without any tracing in effect—and then signal the new owner to attach immediately. However, this has a race condition: it leaves open a short window during which the untrusted application may issue an unsafe system call.

[6]It is worth pointing out that, even if it were possible to transfer control of an untrusted application securely, the `wait` workaround described above would still provide strong pressure for a single Janus process to control not only the untrusted application but also all children it spawns. The reason is obscure: if a Janus process controls a traced parent but not one of its children, the child's controller will need to communicate frequently with the parent's controller so that the parent's controller knows when to let `wait` system calls complete. It is better to avoid the need for inter-process communication and synchronization between Janus processes (which is tricky to get right) by maintaining a one-tracer-many-tracee relationship. The `fork` complication mentioned below would also provide some pressure to maintain a one-tracer-many-tracee relationship, because (again) IPC would otherwise be needed to handle a relatively rare race condition.

In contrast, our original Janus implementation—which never had any difficulty transferring control of an untrusted application securely, because it used `/proc`—maintained a one-to-one relationship between tracing and traced processes by forking a new Janus process and transferring control of the traced child to the new Janus process each time the untrusted application forked a new child. This invariant simplified the implementation of Janus somewhat.

- **Kill on tracer-death not supported:** It would be desirable for the tracee to be automatically killed by the kernel if the tracer dies unexpectedly. However, we have not implemented this wish list item; it would require adding too much new complexity to the kernel, and we are concerned that a coding error in our kernel code might open a security hole that allows an attacker to assassinate innocent applications.

- **Race condition in fork:** Due to a design oversight on our part, `ptrace++` does not handle `fork` system calls very effectively. The issue is that the tracer does not receive a tracing event when the `fork` returns in the newly created child process; instead, the tracer receives tracing events only on the child's subsequent system calls. This leaves open a race condition: after observing a `fork` syscall entry, the tracer knows to expect a new child process, but will not learn the process id of the child until the `fork` syscall exits in the parent; yet the child may issue system calls before the `fork` returns in the parent. When relying on `ptrace++`, Janus needs to learn the child's process id and connect it to the parent, because Janus keeps track of the full process tree structure[7]. This opens up a race condition: if two different traced processes fork, and in both cases the `fork` system call returns in the child before returning in the parent, and a child issues another system call before the `fork` returns in the parent, Janus will not be able to establish the process tree structure. We note that Solaris's `/proc` tracing primitive does not have these problems.

  We work around the race condition with some heuristic approaches that work most of the time. When they don't work, the application may fail to work properly, but security won't be breached.

  This is the only `ptrace++` flaw that can be attributed to our extensions, rather than being a fundamental consequence of the `ptrace` architecture. We know how to fix this flaw, but the fix appears to require some changes to the Linux system-call entry/exit sequence, which is coded in assembly because it is such a commonly used and performance-critical execution path. We are loath to tamper with code this critical, especially to fix such a minor flaw. Still, integrity requires that we describe the unfixed weakness in our implementation.

As much as we hated to do it, we were forced to add substantial complexity to the user-level Janus code to work around these pitfalls in `ptrace++`. We will detail below how much this affects the assurance level of our implementation.

Our implementation of Janus relies on a thin adaptation layer (which provides a simple interface to the tracing functionality Janus needs) to allow Janus to work with

[7]Janus needs to maintain the full process tree structure to apply the `wait` workaround, for instance. This is an example where two annoying shortcomings of `ptrace` interact to create even more annoyance.

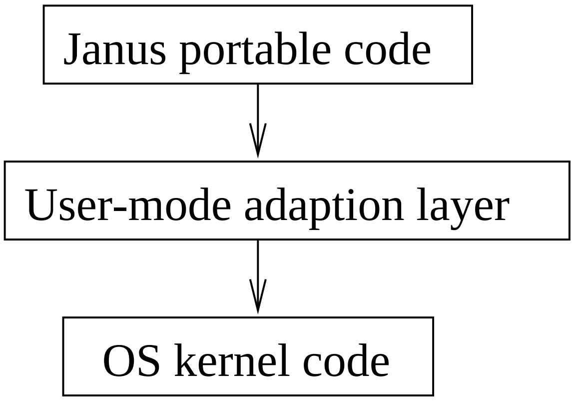

Figure 4.1: The high-level organization of Janus.

whatever tracing primitive may be available. This simplifies the task of porting Janus to a new operating system, as well as simplifying subsequent configuration control issues. We summarize the organization of Janus in Figure 4.1.

By comparing the adaption code for `/proc` and for `ptrace++`, we can meaningfully and fairly compare the suitability of the `/proc` and `ptrace++` mechanisms. Figure 4.2 gives a detailed look at the structure of the Janus implementation, listing the code size for each key component. We can see that the adaption layer for `/proc` is significantly smaller—and, incidentally, simpler as well—than the corresponding code needed to support `ptrace++`. Most of the `/proc` code is legitimately dedicated to adaption (e.g. translation of interface formats), while the `ptrace++` bloat is explained by the need to work around `ptrace++`'s many idiosyncrasies and oddities with extra user-level adaption code. Indeed, our `/proc` adaption layer *just worked* the first time without any debugging needed (apart from correcting a few typos that the compiler detected immediately), while our `ptrace++` adaption code took a significant amount of effort to debug and fix.

In short, while `ptrace++` is minimally adequate, it has a number of severe drawbacks. We found that the `/proc` mechanism provides a cleaner, simpler, and more powerful interface to the tracing functionality Janus demands from the operating system, and we would encourage folks to avoid the `ptrace++` approach when possible.

Our implementation experience with `ptrace++` taught us that it is trickier to design and implement a well-engineered tracing primitive than we had originally thought. Previous paragraphs enumerated a number of pitfalls in using `ptrace++` that unexpectedly[8] reared their head during the development of Janus's `ptrace++` adaption layer.

Most of the pitfalls in `ptrace++` were fundamental in `ptrace`'s original design—the primary culprit was the overloading of the Unix signals mechanisms—and could not have been avoided by any mechanism that attempted to extend `ptrace`. Still, we were surprised by the idiosyncrasies and pitfalls associated with `ptrace`-based tracing mechanisms. According to the "principle of least surprise", this is a strong indication of a ill-chosen

[8] If we had studied the `strace` source code, perhaps we would have been less sanguine about our chances.

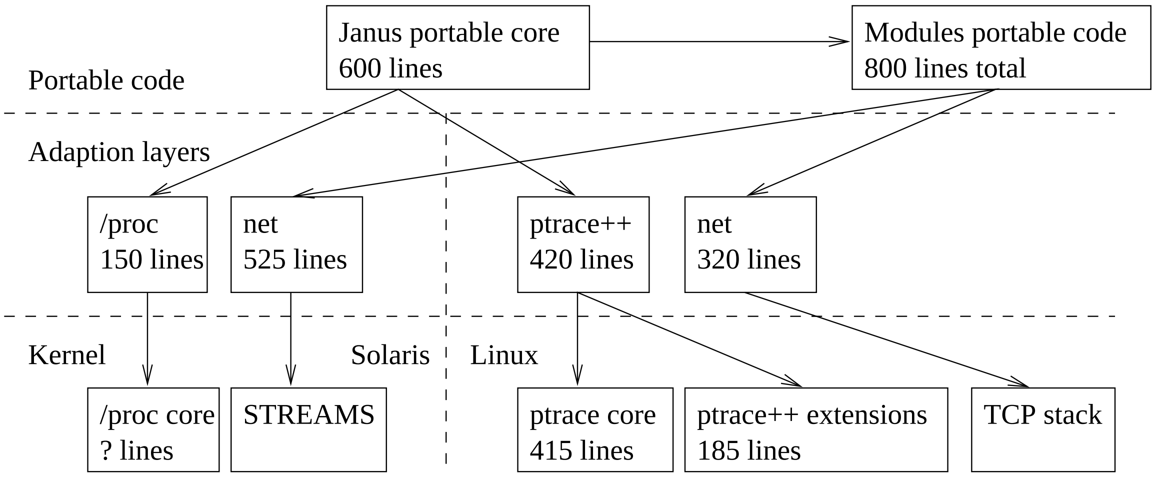

Figure 4.2: The code size of key Janus components. The figure is divided vertically into portable code, adaption code, and kernel code; the latter two categories are divided horizontally according to whether they are built for Solaris or Linux.

primitive.

There were two core failures in the `ptrace++` architecture. First, `ptrace` and `ptrace++` violate the transparency of tracing: when tracing is initiated on a child process, the shape of the process tree is modified slightly, and the semantics of the Unix signals interface are overloaded to communicate tracing events. Much user-level adaption code in Janus is dedicated to restoring the lost transparency. Better event notification primitives could have helped prevent these failures. Second, `ptrace` and `ptrace++` violate extensibility in a subtle way: although it is possible for a single Janus process to extend the system reference monitor by tracing the targeted application, once the reference monitor has been extended in this way it cannot be extended any further, because of the one-tracer-per-tracee limitation. As a result, it is impossible to transfer control securely from one Janus process to another.

Most of `ptrace++`'s problems could have been avoided had it fully obeyed both the transparency and extensibility principles. Full extensibility would have allowed for secure control transfers. Full transparency would have prevented `ptrace++` from breaking `wait`, thereby obviating the need for the workaround to keep track of the process tree structure and thus rendering the `fork` race condition irrelevant. The few remaining implementation challenges could have been eliminated with some additional support from the OS for extending kernel state such as the process data structures.

We hope that future implementors will be able to benefit from our experience and avoid the worst of the subtle traps we encountered. As a result of our experiences with `ptrace++`, we have identified two additional principles for the design of system-call interposition: provide transparency, and plan for multiple extensions. Also, we suggest that operating systems should provide better event notification primitives and better ways to extend internal kernel state.

13 Principles for designing interposition facilities

We hope that the account of our struggles and the analysis of which features Janus needs, as presented above, will help the OS designer steer away from the thorniest pitfalls. We also offer a few principles to help OS designers select an architecture for a system-call interposition primitive:

1. **Be general**: OS designers should commit to a certain level of generality of mechanism. This means building a facility for system-call interposition, not a facility for debugging. In particular, the OS designer should target security and extensible reference monitors as one important application.

   Other benefits of generalized interposition mechanisms are documented in papers describing new filesystems [5, 39, 43, 42], transparent result caching [67], emulation of other operating systems [43, 42], transactional software environments [43, 42], and so on. Because these applications can be handled with very similar mechanisms, we argue that one generalized mechanism could handle them all with little additional cost.

   On the other hand, security considerations weigh against including unnecessary features, as suggested in the next principle.

2. **Respect economy of mechanism**: Implement the least powerful interposition mechanism that will suffice; simpler mechanisms are more likely to be correct (and thus secure). The least powerful tracing mechanism will usually also be the one that is the least costly, in terms of design effort, implementation complexity, code bloat, and kernel modularity. Our analysis of which features Janus absolutely needs should help the OS designer maintain economy of mechanism. Also, our analysis of the value of additional features (in terms of added security robustness, savings in user-level complexity, and performance benefits) should help decide when to add extra functionality.

3. **Transparency is critical**: It is vital that the interposition mechanism be transparent to the application. In particular, the mechanism should not interfere with the traced application's execution environment or overload the meaning of existing interfaces in a way that is visible to the traced application. `ptrace` and `ptrace++` do affect the execution environment slightly by modifying the form of the process tree when tracing is initiated on a child process. Also, to communicate tracing events, `ptrace` and `ptrace++` overload the semantics of the Unix signals interface. Both of these abstraction violations ruined the illusion of transparency in certain cases, and substantial extra code was required in Janus to work around these failures. OS developers should learn from our mistakes; this painful experience is well worth avoiding.

4. **Provide better event notification primitives**: Event notification and upcalls into user-level applications are not trivial to implement. Had the OS provided better event notification primitives, there would have been no temptation to overload the Unix signals interface to notify tracers of tracing events. This overloading of the semantics of existing interfaces is exactly what led to the violation of transparency

in `ptrace` and `ptrace++`. Event notification is one area where OS implementors can help enable security mechanisms based on system-call interposition.

5. **Plan for multiple extensions**: Where two or three objects of some type can exist, make it easy for outsiders to add a fourth or fifth[9]. Applied narrowly, this just suggests that users ought to be able to extend the system reference monitor by making it more restrictive. However, this also applies more generally. It is not enough to allow a single process to contribute extra access control checks to the extensible reference monitor; the system should allow multiple processes to extend the reference monitor simultaneously. `ptrace` and `ptrace++` do not allow multiple processes to trace a target application, and this has the subtle side effect that transferring control of a traced application atomically is impossible in `ptrace` and `ptrace++`. Once again, OS designers are advised to take note of our mistakes.

6. **Allow interposing on internal state**: Kernels should make it easy (or at least possible) to extend some internal data structures. Since we were not able to add to the Linux process structure, in `ptrace++` we were forced to construct shadow copies of the process structures; and since we were not able to interpose on process death events or learn when internal data structures were deallocated, to prevent memory leaks from shadow data structures that are never freed we were forced to poll the official kernel data structures regularly. Perhaps if the Linux kernel had been implemented in an object-oriented language such as Java it might have been easier to extend the kernel process structure by, for example, subclassing the `Process` class.

   Also, on a related note, if the kernel could somehow export more of its internal data structures to user-level reference monitors, this would be useful. For instance, direct access to inode structures, the process tree structure, and pathname lookup functions would greatly decrease the amount of duplication of state and code in Janus, and thus increase Janus's assurance level significantly.

7. **Simplify the system-call interface**: The principles above have all concentrated on the tracing facility's feature set. However, OS developers should be aware that other factors also affect the effectiveness of Janus. In particular, the structure of the system-call interface can significantly impact Janus.

   One of the central insights that we derive from this work is the importance of simplicity in OS interfaces. We took advantage of a powerful methodology for implementing confinement mechanisms:

   (a) Pick a simple, narrow, clean interface. Ensure that all interactions between the untrusted system element and its environment must pass through this interface.

   (b) Enforce the security policy by interposing extra permission checks on this interface.

[9] This excellent piece of advice is due to Steve Bellovin [69], and we are grateful to be able to repeat his design principle here.

On most modern operating systems, the system-call interface is very convenient for our purposes: it is already relatively simple, narrow, and clean; and all privileged system interactions pass through it.

However, the suitability of the system-call interface does vary from system to system. Here are a number of lessons we learned from experience with several operating systems' system-call interfaces:

- We learned that it should be simple and clean, to avoid cluttering the Janus implementation with all sorts of code for parsing system calls.
- Also, the interface should be narrow and minimal, rather than rich and complex, to avoid complicated unforeseen interactions between different features of the interface.
- Special cases should be avoided (because they are easy to overlook when implementing Janus).
- Message-passing interfaces should be avoided, because interposing on them typically requires interception and scanning of all data flow, including non-dangerous communication (which is the majority of all data flow); it is much better to explicitly distinguish cross-domain function invocation (e.g. system calls) from communication.
- It is helpful to separate potentially unsafe operations (which must be subjected to permission checks) from frequently used safe operations. For instance, the `open` and `read` system calls are very well chosen in this regard: all permission checks are done at `open` time, and then `reads` on a valid file descriptor may always proceed unchecked. This allows Janus to trap only `open` system calls (which are relatively rare) and not trap `reads`, so `reads` can proceed at full speed.

  The basic paradigm here is the use of early binding: all access checks are done when the resource is bound, and then the application is allowed fast direct access. This paradigm is very well suited to Janus, since it allows permission checks to be separated from operations that must be fast.
- Requests should be *context-free*—checking them (and thus executing them) should not require much implicit state—because the more security-critical state Janus relies on, the greater the chance for mistakes and security holes. For example, the Berkeley sockets network interface suffers in this regard: to filter network connections, one must match up `socket` calls to `connect` calls (since the `socket` call specifies whether to use TCP or UDP, and the `connect` call specifies the rest of the destination address). The end result is that Janus ends up saving security-critical state associated with each socket file descriptor. Furthermore, this state is a duplication of information that is already managed by the OS kernel, so any error in handling the state, or any mismatch between Janus's state and the kernel's internal state, could lead to a security compromise.

All of the above criteria are geared towards making Janus as effective and secure as possible; other considerations may be more important in some cases, of course.

Our experience with Solaris and Linux showed us the strengths and weaknesses of their respective OS interfaces. We found that `ioctls` were a serious nuisance on both OS's, because they implement a great variety of functionality that is often simultaneously poorly documented and security-critical. Fortunately, `ioctls` are a Unix idiosyncrasy.

However, the majority of our problems with the two system-call interfaces were found in the interface for accessing network functionality. The Solaris STREAMS-based network interface was very poorly suited to monitoring by Janus. The Solaris kernel implements network protocols as a STREAMS module; user-level code interacts with the STREAMS modules by a protocol-independent message-passing interface. A user-level shared library emulates the Berkeley sockets interface by sending messages to the STREAMS modules. Any one socket call may translate into multiple system calls: usually a `putmsg` or two, and occasionally an `open` or several `ioctls`. This required a lot of security-critical state in Janus to track the progress of the STREAMS module, and match up multiple system calls that were associated with one socket call; also, it was quite a pain to implement, as we had to monitor and parse *all* messages sent by `putmsg`. (Fortunately for us, the message-passing interface was implemented with `putmsg`: if it had been implemented with `write`, we would have had to monitor *all* `write` calls, with disastrous results for performance.) Linux's networking interface is much better, as it simply offers the Berkeley sockets calls directly as syscalls. There was still the issue of matching `socket` calls to `connects`, but this was relatively minor.

In any case, during our studies it became clear to us that Janus is dependent upon the OS designer to pick a simple, clean interface for system calls; any failure on this point forces Janus to implement complex and potentially untrustworthy workarounds.

8. **Other security-related events**: OS designers might want to consider support for interposition on other security-related events. Currently Janus interposes only on the system-call interface, on the assumption that system calls are the only security-critical ways for untrusted applications to interact with the OS. However, other OS interfaces—such as floating point exceptions, virtual memory events, hardware interrupts, or other traps—might conceivably be relevant to some future application.

14 Summary

The core contribution of this work towards operating systems research is the realization that OS designers can enable powerful new security tools with just a little extra effort. Supporting Janus is cheap: implementing process tracing takes you most of the way towards Janus's requirements, and most systems already support some form of process tracing; furthermore, we gave above some experience and guidelines to help reduce the cost of supporting Janus even further. At the same time, this support provides great security benefits: as we showed in other sections, Janus can be used to secure `sendmail`, Java, Netscape and its helper applications, as well as other vital-yet-dangerous elements of the system. In short, OS designers should strongly consider providing support for Janus, as this provides great leverage from simple mechanisms.

Currently, many in the security community feel that the operating system is becoming increasing irrelevant to computer security, especially with the advent of firewalls, the shift in emphasis to the security of large network applications (such as `sendmail`, Netscape, web servers, etc.) instead of the security of the OS kernel, and the growing importance of language-based techniques for securing mobile code. However, our work shows that simple new OS primitives for system-call interposition can provide powerful solutions for securing large network applications and for confining mobile code. By suggesting that OS research still has a lot to offer, we suggest this work ought to form a wake-up call to the security community.

Chapter 5

Other work

This chapter describes some other work relevant to Janus. First we survey some related work and how it influenced our design; next we discuss some limitations of our tool, and how they might be fixed; and then we list some possible areas for future work.

15 Related work

There is a significant body of work in the OS community on interposition. SLIC [39] considers interposition as a technique for low-cost extensibility for commodity operating systems; it allows users to interpose on selected OS interfaces by downloading code into the kernel. The SLIC work studies a number of sample applications, with our work on restricted environments for helper applications [40] motivating one of them.

Also, Jones studied interposition in the Mach kernel as a tool for implementing user-level extensions [43, 42]. His work demonstrated that OS support for interposition on the system-call interface can enable significant new functionality. However, his toolkit as presented is not particularly useful for confinement, as Mach's support for interposition does not allow one to protect the enforcer (the interposed code) from the enforcee (the untrusted application).

After our initial work on secure environments for helper applications [40] was published, Alexandrov *et al.* [5] showed how to implement a user-level filesystem extension with system-call interposition. Like Janus, their implementation used the Solaris `/proc` system-call tracing facility to intercept system calls and simulate the semantics of the extended interface. Their work provides further evidence for our assertion that OS support for user-level system-call interposition would enable many other potential applications as well as help improve site security.

Other recent work on using system call interposition for security wrappers has built on the approach we described in [40]. One paper [53] provides an implementation of system call interposition using Linux kernel modules. They build a tool to limit the file accesses an untrusted process may make and demonstrate their approach by using it to protect the Netscape web browser. (They do not consider access control for other resources, such as the network.) A more recent paper [37] proposes a wrapper definition language to reduce the development cost of building wrappers that use system call interposition. One key contribution of their work is the notion of tagging related system calls, so that the wrapper developer can specify policy at a high level; the wrapper compiler translates this into a platform-dependent form. They validate their approach with implementations for FreeBSD and Solaris using loadable kernel modules, and they build several applications, including confinement of untrusted applications and intrusion detection based on recognizing sequences of system calls.

Other researchers have previously advocated the use of interposition to improve system security. Bump-in-the-stack implementations of IPSEC interpose on the interface between the TCP/IP stack and the network driver to add encryption and authentication to the IP protocol [69]. `securelib` is a shared library that replaces the C `accept`, `recvfrom`, and `recvmsg` library calls by a version that performs address-based authentication; it is intended to protect security-critical Unix system daemons [50]. Also, [11] briefly describes an implementation of dynamic detection and prevention of security-relevant race conditions that works by interposing on dynamically-linked libraries. Other research that also takes advantage of custom shared libraries for non-security purposes can be found in [45, 34]. However, interposition on the library interface is not usable for confinement, since a hostile application could bypass this access control by simply issuing the dangerous system call directly without invoking any library calls. Therefore, we consider simple replacement of dangerous C library calls with a safe wrapper insufficient in our extended context of untrusted and possibly hostile applications.

Fernandez and Allen [33] extend the filesystem protection mechanism with per-user access control lists. Lai and Gray [46] describe an approach that protects against Trojan horses and viruses by limiting filesystem access: their OS extension confines user processes to the minimal filesystem privileges needed, relying on hints from the command line and (when necessary) run-time user input. TRON [10] discourages Trojan horses by adding per-process capabilities support to the filesystem discretionary access controls. These works all suffer from two major limitations: they require kernel modifications, and they do not address issues such as control over process and network resources.

Domain and Type Enforcement (DTE) is a way to extend the OS protection mechanisms to let system administrators specify fine-grained mandatory access controls over the interaction between security-relevant subjects and objects. A research group at TIS has amassed considerable experience with DTE and its practical application to Unix systems [7, 8, 63, 65]. DTE is an attractive and broadly applicable approach to mandatory access control, but its main disadvantage is that it requires kernel modifications; we aimed instead for user-level protection.

More recently, Schneieder has given an automata-theoretic treatment of interposition as a implementation technique for security, yielding a theory of implementable enforcement mechanisms [62, 61]. His work suggests that interposition is a very powerful technique for implementing a large class of security policies.

The confinement problem was first identified by Lampson in [48]. His formulation dealt primarily with confidentiality protection, but modern versions of the confinement problem are typically concerned with system integrity and availability as well as the confidentiality of secret information. His work spawned a small subfield of research on covert channels and information leakage; in this thesis, we have sidestepped the theoretical difficulties posed by covert channels by ensuring that confined programs never get access to confidential information in the first place. The Orange Book [32] tried to address a number of the problems with discretionary access control schemes as solutions to the confinement problem by introducing mandatory access control, but this effort failed in commercial systems because its confidentiality protections were too strong and its integrity protections too weak. For an example of a more modern treatment of the confinement problem and its theory, see [12].

Also, [24, 25] gives practical experience on the benefits of confining untrusted processes to a sandboxed “jail”. The authors used `chroot()` for confinement, which worked well for their purposes. Today, the rising importance of network security means that `chroot()`'s inability to handle resources other than the filesystem is a serious limitation; this motivates our work on more general mechanisms for confining untrusted applications.

To achieve security, we relied heavily on the concept of sandboxing, first introduced by Wahbe *et al.* in the context of software fault isolation [71]. However, those authors were actually solving a somewhat different problem. What they achieved was memory safety for untrusted modules running in the same address space as trusted modules. They ignored the problem of system-level security; conversely, we do not attempt to provide safety. They also use binary-rewriting technology to accomplish their goals, which makes it very challenging to run arbitrarily general pre-existing applications robustly and efficiently[1].

The same basic idea—isolating certain operations for extra scrutiny—may be found in both software fault isolation and Janus. Thus, we could probably have used software fault isolation instead of process tracing to isolate system calls and interpose on them, at least for those binaries which software fault isolation can be applied to. However, such an approach has a serious disadvantage: memory safety must also be guaranteed to protect the enforcement mechanism from the untrusted application, whereas in our implementation the enforcement mechanism uses a separate address space and thus gains dependable protection for free. In any case, no matter which mechanism is used for system call interposition (process tracing, software fault isolation, SLIC [39], or something else entirely), our work on Janus will still be relevant for what it says about how to check the system calls issued by untrusted code.

In a more recent approach, proof-carrying code [54, 56, 59, 55, 57, 58], the compiler embeds security checks at each unsafe operation and emits a proof that the resulting code satisfies the security policy; at runtime, a verifier can quickly check the safety proof. Proof-carrying code allows one to check very sophisticated and fine-grained security policies, and thus can be much more flexible than Janus. (The only challenge is to formalize the security policy in the logical framework used by the prover and verifier.) In particular, it would presumably be possible to build a compiler that embeds Janus security-checking code in the application and can prove to a verifier that the application would pass all checks Janus would make, thereby avoiding the need for a separate Janus process. However, Janus does provide some extra benefits because it is *orthogonal* to the protected application: we can specify the security policy at runtime rather than at compile time, and we can protect

[1] Consider, e.g., self-modifying code, indirect jumps, variable-length instructions, dynamic linking, trampolines, instruction atomicity, exception handlers, and the complex addressing modes found in today's CISC architectures to get some idea of the implementation challenges [49, 64].

pre-existing, pre-compiled legacy code.

Java [41] is a comprehensive system that addresses, among other things, both safety and security, although it achieves security by a different approach from ours. Java cannot secure pre-existing programs, because it requires use of a new language. We do not have this problem; our design will run any application, and so is more versatile in this respect. However, Java offers many other advantages that we do not address; for instance, Java provides architecture independence, while Janus only applies to native code and provides no help with portability. Another important difference is that Java provides support for high-level and fine-grained security policies, while we focus on low-level, coarse-grained (system call based) policies that are less expressive but simpler to get right.

OmniWare [28] takes advantage of software fault isolation techniques and compiler support to guarantee memory safety for untrusted code. However, as we show in this work, security for untrusted code requires much more than just memory safety: access to all resources must be mediated, and [28] says nothing about mediation for other resources. Like Java, OmniWare offers architecture-independence, extensibility, and efficiency as important goals; Janus ignores these issues. Another major difference between Janus and OmniWare is that OmniWare cannot provide security for legacy code, while Janus takes this on as a primary goal.

We note two important differences between the Java approach and the Janus philosophy. The Java protection mechanism is much more complex, and is closely intertwined with the rest of Java's other functionality. In contrast, we have more limited goals, we explicitly aim for extreme simplicity, and we keep the security mechanism orthogonal from the primary functionality.

16 Limitations

One inherent limitation of the Janus implementation is that we can only successfully run applications that do not legitimately need many privileges. Our approach will easily accommodate any program that only requires simple privileges, such as access to a preferences file. Application developers may want to keep this in mind and not assume, for example, that their applications will be able to access the whole filesystem[2].

We have followed one simple direction in our prototype implementation, but others are possible as well. One could consider using specialized Unix system calls to revoke certain privileges. The two major contenders are `chroot()`, to confine the application within a safe directory structure, and `setuid()`, to change to a limited-privilege account such as `nobody`. Unfortunately, programs need superuser privileges to use these features; since we were committed to a user-level implementation, we decided to ignore them. In retrospect it may be prudent to reconsider this design choice, especially when using Janus to confine system daemons (e.g. `sendmail`) that already require superuser privilege to execute. (But note that this shortcoming is easy to remedy, thanks to the Unix tool philosophy, by `exec`ing Janus from a tiny setuid program that simply performs a `chroot()` and `setuid()`.) Other

[2]The Athena X file open widget is a good example of what not to do: it assumes that it has access to the whole filesystem so that it can identify the absolute pathname of all files, and it doesn't handle failure well at all.

security policies (such as mandatory audit logs) may also be more appropriate in some environments.

The most fundamental limitation of our implementation, however, stems from its specialization for a single operating system. Each OS to which Janus might be ported requires a separate security analysis of its system-call interface. Also, a basic assumption of Janus is that the operating system provides multiple address spaces, allows trapping of system calls, and makes it feasible to interpose proxies where necessary. Solaris 2.4 has the most convenient support for these mechanisms; we believe our approach may also apply to some other Unix systems. On the other hand, platforms without support for these services cannot directly benefit from our techniques. In particular, our approach cannot be applied to PCs running MS-DOS or Microsoft Windows. The utility of these confinement techniques, then, will be determined by the underlying operating system's support for user-level security primitives.

17 Future work

A possibility for future enhancement of our implementation is to offer extensive auditing. It would be conceptually simple to add logging and accounting capabilities to our prototype. The difficult task is figuring out what to do with the audit logs after you have them; this seems to require some significant work on developing an intrusion detection back-end, and so we do not currently have any plans to implement auditing extensions.

We believe that interposition of filtering proxies is a promising approach for improving control over network accesses. By taking advantage of earlier work in application proxying firewalls, we were able to integrate a safe X proxy into our prototype easily. We suspect that one can achieve enhanced control over many other network communications of interest by leveraging existing application-level proxies developed by the firewall community. This would enable our techniques to be used in broader contexts. The overlap with research into firewalls lends hope that these problems can be solved satisfactorily.

One issue is how to interpose proxies forcibly upon untrusted and uncooperative applications. We currently use environment variables as hints—for instance, we change the `DISPLAY` variable to point to a proxy X server, and disallow access to any other X display—but this only works for well-behaved applications that consult environment variables consistently. One might consider implementing the hints also with a shared library that replaces network library calls with a safe call to a secure proxy. Other tricks are also possible.

A more aggressive area for future work lies in shedding code from the Janus implementation to increase its assurance level. One particularly troubling aspect of Janus is that, though we have tried to keep duplication to a minimum, Janus ends up duplicating a number of privilege checks already found in the kernel, which leaves room for inconsistencies and potential bugs. We suspect that this would probably require increased support from the operating system to access internal data structures; a mechanism like SLIC might help substantially in this regard.

Finally, more experience with some of the applications suggested in Chapter 3 (or other applications as appropriate) would help to better understand the advantages and limitations of the Janus approach.

Chapter 6

Conclusions

We have showed how confinement can be used to secure systems that include untrusted and untrustworthy code. We have demonstrated that interposition on the OS interface is a powerful technique to achieve this goal, and furthermore we have shown that existing process tracing primitives can enable the construction of high-performance, user-level, general-purpose confinement tools. Our prototype, Janus, proved to be well-suited to a number of compelling applications, including security for mobile code and mobile agents as well as security for potentially vulnerable daemons like `sendmail`. This shows that Janus is a powerful tool with broad applicability. Finally, we analyzed the implications for OS designers, noting that Janus requires only minimal support from the operating system and arguing that OS designers ought to include this support.

Bibliography

[1] [8lgm]-Advisory-16.UNIX.sendmail-6-Dec-1994, December 1994.

[2] [8lgm]-Advisory-17.UNIX.sendmailV5-2-May-1995, May 1995.

[3] [8lgm]-Advisory-17.UNIX.sendmailV5.22-Aug-1995, August 1995.

[4] [8lgm]-Advisory-20.UNIX.sendmailV5.1-Aug-1995, August 1995.

[5] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. Extending the operating system at the user-level: the UFO global file system. In *Proc. 1997 Annual USENIX Technical Conference*, January 1997.

[6] J. P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, ESD/AFSC, Hanscom AFB, Bedford, Mass., October 1972. (NTIS AD-758 206).

[7] Lee Badger, Daniel F. Sterne, David L. Sherman, and Kenneth M. Walker. A domain and type enforcement UNIX prototype. *USENIX Computing Systems*, 9(1):47–83, Winter 1996.

[8] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. Practical domain and type enforcement for UNIX. In *Proc. 1995 IEEE Symposium on Security and Privacy*, 1995.

[9] Steven M. Bellovin. Re: sendmail wizard thing..., February 1995. Post to `bugtraq` mailing list. `http://geek-girl.com/bugtraq/1995_1/0350.html`.

[10] Andrew Berman, Virgil Bourassa, and Erik Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Proc. 1995 USENIX Winter Technical Conference*, pages 165–175. USENIX Assoc., 1995.

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

[12] W.E. Boebert and R.Y. Kain. A further note on the confinement problem. In *Proc. IEEE 30th Annual 1996 International Carnahan Conference on Security Technology*, pages 198–202, 1995.

[13] CERT advisory CA-88:01, 1988.

[14] CERT advisory CA-90:01, January 1990.

[15] CERT advisory CA-93:15, October 1993.

[16] CERT advisory CA-93:16, November 1993.

[17] CERT advisory CA-94:12, July 1994.

[18] CERT advisory CA-95:05, February 1995.

[19] CERT advisory CA-95:08, August 1995.

[20] CERT advisory CA-95:10, August 1995.

[21] CERT advisory CA-95:11, September 1995.

[22] CERT advisory CA-97.14.metamail, 1997.

[23] Bill Cheswick and Steven M. Bellovin. A DNS filter and switch for packet-filtering gateways. In *Proc. 1996 USENIX Security Symposium*. USENIX Assoc., 1996.

[24] William R. Cheswick. An evening with Berferd, in which a cracker is lured, endured, and studied. In *Proc. of the Winter USENIX Conf.*, 1992.

[25] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994.

[26] Frederick Cohen. Personal communication.

[27] Frederick Cohen. Internet holes. *Network Security Magazine*, January 1996.

[28] Colusa Software. OmniWare technical overview, 1995.

[29] Alan Cox. Vulnerability in `metamail`, October 1997. Post to `bugtraq` mailing list. `http://www.dhp.com/~fyodor/sploits/metamail.inappropriate.helpers.html`.

[30] Ray Cromwell. Buffer overflow, September 1995. Announced on the Internet. `http://www.c2.net/hacknetscape/`.

[31] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: From HotJava to Netscape and beyond. In *Proc. of the 1996 IEEE Symposium on Security and Privacy*, 1996.

[32] DoD trusted computer system evaluation criteria. DoD 5200.28-STD, DoD Computer Security Center, 1985.

[33] G. Fernandez and L. Allen. Extending the Unix protection model with access control lists. In *Proc. Summer 1988 USENIX Conference*, pages 119–132. USENIX Assoc., 1988.

[34] Glenn S. Fowler, Yennun Huang, David G. Korn, and Herman Rao. A user-level replicated file system. In *Summer 1993 USENIX Conference Proceedings*, pages 279–290. USENIX Assoc., 1993.

[35] Armando Fox and Eric A. Brewer. Reducing WWW latency and bandwidth requirements via real-time distillation. In *Proc. Fifth International World Wide Web Conference*, May 1996.

[36] Armando Fox, Steven D. Gribble, Yatin Chawathe, and Eric A. Brewer. Cluster-based scalable network services. In *Proc. 1997 Symp. Operating System Principles (SOSP-16)*, October 1997.

[37] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS software with generic software wrappers. In *Proc. 1999 IEEE Symp. Security & Privacy*, 1999.

[38] Douglas P. Ghormley, April 1998. Personal communication.

[39] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Thomas E. Anderson. SLIC: An extensibility system for commodity operating systems. In *Proc. 1998 USENIX Technical Conference*. USENIX Assoc., 1998.

[40] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *Proc. 1996 USENIX Security Symposium*. USENIX Assoc., 1996.

[41] James Gosling and Henry McGilton. The Java language environment: A white paper, 1995. `http://www.javasoft.com/whitePaper/javawhitepaper_1.html`.

[42] Michael B. Jones. Transparently interposing user code at the system interface. Technical Report CMU-CS-92-170, Carnegie Mellon University, September 1992. PhD thesis.

[43] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proc. 14th ACM Symp. on Operating System Principles*, pages 80–93, December 1993.

[44] Brian L. Kahn. Safe use of X window system protocol across a firewall. In *Proc. of the 5th USENIX UNIX Security Symposium*, 1995.

[45] David G. Korn and Eduardo Krell. The 3-D file system. In *Summer 1989 USENIX Conference Proceedings*, pages 147–156. USENIX Assoc., 1989.

[46] Nick Lai and Terence Gray. Strengthening discretionary access controls to inhibit Trojan horses and computer viruses. In *Proc. Summer 1988 USENIX Conference*, pages 275–286. USENIX Assoc., 1988.

[47] Butler Lampson. Hints for computer system design. In *Proceedings of the 9th ACM Symposium on Operating Systems Review*, volume 17:5, pages 33–48. Bretton Woods, 1983.

[48] B.W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[49] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software—Practice and Experience*, 24(2):197–218, February 1994.

[50] William LeFebvre. Restricting network access to system daemons under SunOS. In *UNIX Security Symposium III Proceedings*, pages 93–103. USENIX Assoc., 1992.

[51] Davor Matic. Xnest. Available in the X11R6 source. Also `ftp://ftp.cs.umass.edu/pub/rcf/exp/X11R6/xc/programs/Xserver/hw/xnest`.

[52] Marshall Kirck McKusick. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, 1995.

[53] Terrence Mitchem, Raymond Lu, and Richard O'Brien. Using kernel hypervisors to secure applications. In *Annual Computer Security Application Conference (ACSAC'97)*, 1997.

[54] George C. Necula. Proof-carrying code. In *POPL'97*, 1997.

[55] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *OSDI'96*, 1996.

[56] George C. Necula and Peter Lee. Research on proof-carrying code for untrusted-code security. In *Proc. 1997 IEEE Symp. Security & Privacy*, 1997.

[57] George C. Necula and Peter Lee. Research on proof-carrying code on mobile-code security. In *Proc. Workshop on Foundations of Mobile Code Security*, 1997.

[58] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In *Mobile Agent Security*. Springer-Verlag, October 1997. LNCS 1419.

[59] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *PLDI'98*, 1998.

[60] Marcus J. Ranum. Thinking about firewalls. In *Proc. 2nd Conf. on System Administration, Networking and Security*, 1993.

[61] Fred B. Schneider. Enforceable security policies. Technical Report TR98-1664, Dept. of Computer Science, Cornell Univ., January 1998.

[62] Fred B. Schneider. Towards fault-tolerant and secure agentry. In *Proc. 11th Intl. Workshop on Distributed Algorithms*, 1998. Also available as Tech. report TR94-1568, Cornell computer science dept.

[63] David L. Sherman, Daniel F. Sterne, Lee Badger, and S. Murphy. Controlling network communication with domain and type enforcement. Technical Report 523, TIS, March 1995.

[64] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation (for the Alpha AXP architecture). *Communications of the ACM*, 36(2):69–81, February 1993.

[65] Daniel F. Sterne, Terry V. Benzel, Lee Badger, Kenneth M. Walker, Karen A. Oostendorp, David L. Sherman, and Michael J. Petkac. Browsing the web safely with domain and type enforcement. In *1996 IEEE Symposium on Security and Privacy*, 1996. Research abstract.

[66] Jeff Uphoff. Re: Guidelines on cgi-bin scripts, August 1995. Post to `bugtraq` mailing list. `http://www.eecs.nwu.edu/cgi-bin/mfs/files2/jmyers/public_html/bugtraq/0166.html?30#mfs`.

[67] Amin Vahdat and Thomas Anderson. Transparent result caching. In *Proc. 1998 USENIX Technical Conference*, 1998.

[68] David Wagner. Secure worker uploading for TACC. Unpublished manuscript. `http://www.cs.berkeley.edu/~daw/classes/inet-svcs/writeup.ps`.

[69] David Wagner and Steven M. Bellovin. A "bump in the stack" encryptor for MS-DOS systems. In *Proc. 1996 ISOC Symposium on Network and Distributed System Security*, 1996.

[70] David Wagner, Ian Goldberg, and Eric A. Brewer. Orthogonal security, 1997. Unpublished manuscript.

[71] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proc. of the Symp. on Operating System Principles*, 1993.

[72] Christian Wettergren. Re: Mime question..., March 1995. Post to `bugtraq` mailing list. `http://www.eecs.nwu.edu/cgi-bin/mfs/files2/jmyers/public_html/bugtraq/1995a/0759.html?30#mfs`.