A Secure Environment for Untrusted Helper Applications

Confining the Wily Hacker

Ian Goldberg David Wagner Randi Thomas Eric A. Brewer
{iang,daw,randit,brewer}@cs.berkeley.edu
University of California, Berkeley

Abstract

Many popular programs, such as Netscape, use untrusted helper applications to process data from the network. Unfortunately, the unauthenticated network data they interpret could well have been created by an adversary, and the helper applications are usually too complex to be bug-free. This raises significant security concerns. Therefore, it is desirable to create a secure environment to contain untrusted helper applications. We propose to reduce the risk of a security breach by restricting the program's access to the operating system. In particular, we intercept and filter dangerous system calls via the Solaris process tracing facility. This enabled us to build a simple, clean, user-mode implementation of a secure environment for untrusted helper applications. Our implementation has negligible performance impact, and can protect pre-existing applications.

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. Since the software and data exchanged on the Internet is very often unauthenticated, it could easily have been created by an adversary.

Web browsers are an increasingly popular tool for retrieving data from the Internet. They often rely on helper applications to process various kinds of information. These helper applications are security-critical, as they handle untrusted data, but they are not particularly trustworthy themselves. Older versions of `ghostscript`, for example, allowed malicious programs to spawn processes and to read or write an unsuspecting user's files [15, 18, 19, 34, 36]. What is needed in this new environment, then, is protection for all resources on a user's system from this threat.

Our aim is to confine the untrusted software and data by monitoring and restricting the system calls it performs. We built Janus[1], a secure environment for untrusted helper applications, by taking advantage of the Solaris process tracing facility. Our primary goals for the prototype implementation include security, versatility, and configurability. Our prototype is meant to serve as a proof-of-concept, and we believe our techniques may have a wider application.

2 Motivation

2.1 The threat model

Before we can discuss possible approaches to the problem, we need to start by clarifying the threat model. Web browsers and `.mailcap` files make it convenient for users to view information in a wide variety of formats by de-multiplexing 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`. 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. Therefore we consider helper applications untrusted, and wish to place them outside the host's trust perimeter.

[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.

We believe that this is a prudent level of paranoia. 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. Worse still, these programs are generally big and bloated, and large complex programs are notoriously insecure.[2] Security vulnerabilities have been exposed in these applications [15, 18, 19, 34, 36].

2.2 The difficulties

What security requirements are demanded from a successful protection mechanism? Simply put, an outsider who has control over the helper 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 helper 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 helper 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 helper 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 several possible approaches.

Building security directly into each helper application: Taking things to the extreme, we could insist all helper applications be rewritten in a simple, secure form. We reject this as completely unrealistic; it is simply too much work to re-implement them. More practically, 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, 8, 9, 10, 11, 12, 13, 14, 16]. In any event, attempts to build security directly into the many helper applications would require each program to be considered separately—not an easy approach to get right. For now, we are stuck with many useful programs which offer only minimal assurances of security; therefore what we require is a general, external protection mechanism.

Adding new protection features into the OS: We reject this design 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 too high. Better to find a user-level mechanism so that users can protect themselves, and so that pre-existing access controls can serve as a backup; even in the worst case, security cannot decrease.

The pre-existing reference monitor: The traditional operating system's monolithic reference monitor cannot protect against attacks on helper applications directly. At most, it could prevent a penetration from spreading to new accounts once the browser user's account has been compromised, but by then the damage has already been done. In practice, 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 in rapid succession.

The conventional network firewall: Packet filters cannot distinguish between different types of HTTP traffic, let alone analyze the data for security threats. A proxy could, but it would be 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.

We therefore see the need for a new, simple, and general user-level protection mechanism that does not require modification of existing helper applications or operating systems. The usual techniques and conventional paradigms do not work well in this situation. We hope that the difficulty of the problem

[2] For instance, `ghostscript` is more than 60,000 lines of C; and `mpeg_play` is more than 20,000 lines long.

and the potential utility of a solution should help to motivate interest in our project.

3 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.

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. We use the term *sandboxing* to describe the concept of confining a helper application to a restricted environment, within which it has free reign. This term was first introduced, in a slightly different setting, in [35].

To achieve security, a slogan we kept in mind was "keep it simple" [29]. Simple programs are more likely to be secure; simplicity helps to avoid bugs, and makes it easier to find those which creep in [17, Theorem 1]. We would like to keep our program 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 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.

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.

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.

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 a function which validates the arguments of the call *before* the call is executed by the operating system.[3] The function can then use this information to optionally 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 which 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. 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.

In designing the framework we aimed for simplicity and versatility as much as possible, though these goals often conflict. One can imagine more versatile and sophisticated algorithms to dispatch system calls, but they would come at a great cost to simplicity.

4 Implementation

4.1 Choice of operating system

In order to implement our design, we needed to find an operating system that allowed one 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. Most operating systems offer a program called `trace(1)`, `strace(1)`, or `truss(1)` which can observe the system calls performed by another process as well as their return values. This is often implemented with a special system call. `ptrace(2)`, which allows the tracer to register a callback that is executed whenever the tracee 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 disadvantage of the `ptrace(2)` interface is that many OS implementations provide no way for a tracing process to abort a system call without killing the traced process entirely.

Some more modern operating systems, such as Solaris 2.4 and OSF/1, however, offer a better process-tracing facility through the `/proc` virtual filesystem. This interface allows direct control of the traced process's memory. Furthermore, it has fine-grained control: we can request callbacks on a per-system call basis.

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.

4.2 The policy modules

4.2.1 Overview

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 establish TCP connections. In addition, the traced application is prevented from performing certain system calls, as described below. The provided modules offer considerable flexibility themselves, so that may configure them simply by editing their parameters in the configuration file. However, if different modules

[3] 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 helper application. This function can examine the arguments to the system call, as well as the return value, and update the module's local state.

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 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.

4.2.2 Sample security policy

We implemented a sample security policy to test our ideas, as a proof of concept.

Helper applications are allowed to `fork` children, we then recursively trace. Traced processes can only send signals to themselves or to their children, and never to an untraced application. Environment variables are initially sanitized, and resource usage is carefully limited.

In our policy, access to the filesystem is severely limited. A helper application is placed in a particular directory; it cannot `chdir` out of this directory. We allow it full access to files in or below this directory; to prevent escape from this sandbox directory, access to paths containing `..` are always denied. The untrusted application is allowed 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 allows us a performance optimization further down the line; see Section 4.4.

Of course, protecting the filesystem alone is not enough. Nearly any practical helper application will require access to network resources. For example, all of the programs we considered need to open a window on the X11 display to present document contents. In our security policy, network access must be carefully controlled: we allow network connections only to the X display, and this access is allowed only through a safe X proxy.

X11 does not itself provide the security services we require (X access control is all-or-nothing). A rogue X client has full access to all other clients on the same server, so an otherwise confined helper application could compromise other applications if it were allowed 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 [26, 31]. We integrated our Janus prototype with `Xnest` [31], which lets us run another complete instance of the X protocol under `Xnest`. `Xnest` acts as a server to its clients (e.g. untrusted helper applications), but its 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. `Xnest` is not ideal—it is not as small or simple as we would like—but further advances in X protocol filtering are likely to improve the situation.

4.2.3 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 which are not explicitly mentioned are unset. The special parameter `display` causes the helper application to inherit the parent's `DISPLAY`. The `tcpconnect` module allows us to restrict TCP connections 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 accesses 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.

Our sample configuration file for this policy can be seen in Figure 2 in the Appendix.

4.3 The framework

4.3.1 Reading the configuration file

The framework starts by reading the configuration file, the location of which can be specified on the command line. This configuration file consists of lines like those shown in Figure 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.

4.3.2 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 4.4, 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 directory is created in `/tmp` with a random name, but the `SANDBOX_DIR` environment variable can be used to override this choice.

4.3.3 Running the traced process

The application runs until it performs a system call. At this point, it is put to sleep, and the tracing process wakes up. The tracing process 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, the tracing process simply wakes up the application, which proceeds to complete the system call. 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.

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 which 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, except in one special case.

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 application, and begins tracing the application's child. 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. Furthermore, traced programs cannot turn off their own tracing.

4.4 The optimizer

Our program has the potential to add a non-trivial amount of overhead to the traced application whenever it intercepts a system call. In order to keep this overhead down, we obviously want to intercept as few system calls as possible—or at least, as few of the common ones 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 before the untrusted helper application executes. 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 allowed; so 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. By eliminating the need to trace common system calls such as `read` and `write`, we can greatly speed up the common case.

5 Evaluation

The general population is more interested in efficiency and convenience than in security, so any security product intended for general use must address these concerns. For this reason, we evaluate our prototype implementation by a number of criteria, including security, applicability, and ease of use, in addition to performance.

5.1 Ease of use

The secure environment is relatively easy to install. All that is needed is to protect the invocation of any helper application with our environment. The most convenient solution is to specify our `janus` program in a `mailcap` file, which could look like

```
image/*; janus xv %s
application/postscript; janus ghostview %s
video/mpeg; janus mpeg_play %s
video/*; janus xanim %s
```

With little effort, a system administrator could set up the in-house security policy by listing `janus` in the default global `mailcap` file; then the secure environment would be transparent to all the users on the system. Similarly, users could protect themselves by doing the same to their personal `.mailcap` file.

5.2 Applicability

Users will want to run our secure environment with pre-existing helper applications. We tested a number of programs under our secure environment, including `ghostview`, `mpeg_play`, `xdvi`, `xv`, and `xanim`. Though we followed the Principle of Least Privilege and were very restrictive in our security policy, we found that each of the applications had sufficient privilege, and we had not unduly restricted the applications from doing their legitimate intended jobs.

In addition, we ran the shells `sh` and `bash` under our secure environment. Unless the user explicitly tries to violate the security policy (e.g., by writing to `.rhosts`), there is no indication of the restricted nature of the shell. Attempts to violate the security policy are rewarded with a shell error message.

5.3 Security

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 much smaller chance of having an undetected security hole.

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.

5.4 Performance

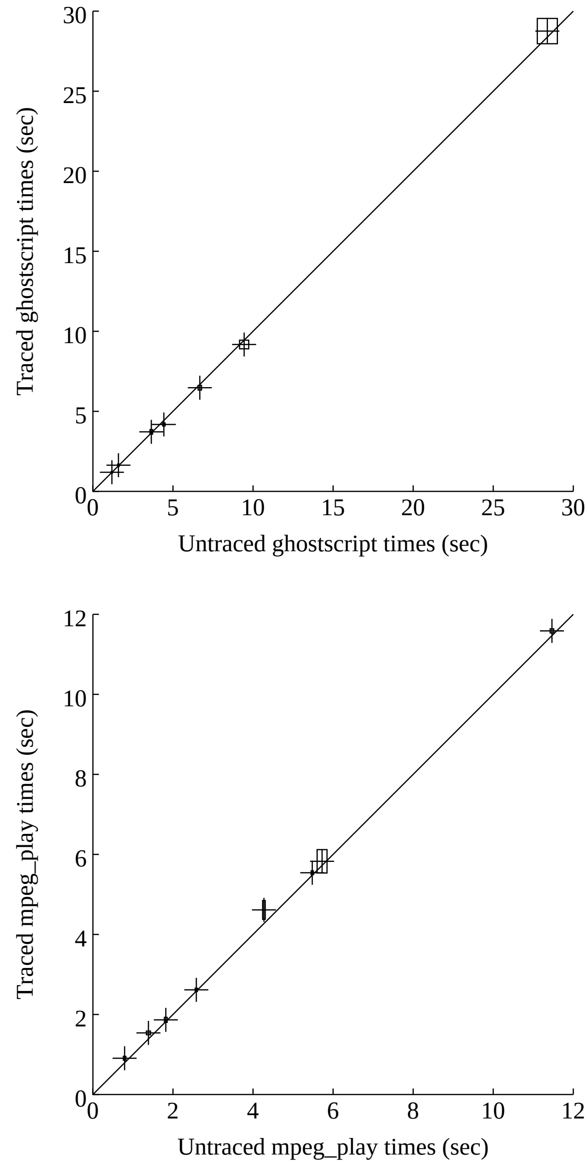
Since our design potentially adds time-consuming context switches for every system call the untrusted application makes, the obvious performance metric to evaluate is time. We measured the peak performance of `ghostscript` and `mpeg_play`, two large commonly used helper applications, under our secure environment. Note that `mpeg_play` in particular is performance critical.

`mpeg_play` was used to display nine mpeg movies ranging in size from 53 KB (18 frames) to 740 KB (400 frames). `ghostscript` was used to display seven Postscript files ranging in size from 9 KB to 1.7 MB. `ghostscript` was run non-interactively, so that all the pages in the Postscript file were displayed in succession with no user intervention. We took 100 measurements for each file, 50 traced under our secure environment and 50 untraced, calculating the mean and standard deviation for each set. The measurements were done using an unloaded single-processor SPARCstation 20 workstation running Solaris 2.4. The `Xnest` X windows proxy [31] was used with the secure environment, but not with the untraced measurements.

The results are displayed in Figure 1. For each set, we plotted the traced time against the untraced time.[4] The boxes around the data points indicate one standard deviation. The diagonal line shows the ideal result of no statistically significant performance overhead. In the best possible case, the error boxes will all intersect the ideal line. Boxes entirely above the line indicate statistically significant overhead. As can be seen, the secure environment imposes no significant performance penalty.

[4] Similar results were obtained when measuring frame rate per second for `mpeg_play`.

Figure 1: Performance data for `ghostscript` and `mpeg_play`



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. We first note that system calls are already so time-consuming that the additional overhead of the Janus process filtering is insignificant. 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.

6 Related work

Due to the accelerated development of communication technology, the security and protection problems inherent in an open and free communication environment, such as the Internet, are relatively new ones to solve. Consequently, much of the work addressing security for this environment is still being developed.

To achieve security, we use the concept of sandboxing, first introduced by Wahbe et al. in the context of software fault isolation [35]. However, they were actually solving a different problem. What they achieved was safety for trusted modules running in the same address space as untrusted 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 prevents them from running arbitrarily general pre-existing applications.

Java [25] is an 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.

OmniWare [20] takes advantage of software fault isolation techniques and compiler support to safely execute untrusted code. Like Java, it also has architecture-independence, extensibility, and efficiency as important goals.

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, explicitly aim for extreme simplicity, and keep the security mechanism orthogonal from the non-security-critical functionality.

`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 [30]. Other research that also takes advantage of shared libraries can be found in [27, 24]. We note that simple replacement of dangerous C library calls with a safe wrapper is insufficient in our extended context of untrusted and possibly hostile applications; a hostile application could bypass this access control by simply issuing the dangerous system call directly without invoking any library calls.

Fernandez and Allen [23] extend the filesystem protection mechanism with per-user access control lists. Lai and Gray [28] describe an approach which 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 [7] discourages Trojan horses by adding per-process capabilities support to the filesystem discretionary access controls. These works all suffer two major disadvantages: 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 [5, 6, 32, 33]. 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.

7 Limitations and future work

7.1 Limitations of the prototype

One inherent limitation of the Janus implementation is that we can only successfully run helper applications which 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.

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. However, this design choice could be reconsidered. Other 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 calls. 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 that do not support 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.

7.2 Future work

In this paper, we have limited discussion to the topic of protecting untrusted helper applications. It would also be interesting to explore how these techniques might be extended to a more ambitious scope.

One exciting area for further research involves Java applet security. Java [25] is seeing widespread deployment, but several implementation bugs [22] have started to shake confidence in its security model. For more protection, one could run Java applets within a secure environment built from techniques described in this paper. This approach provides defense in depth: if the Java applet security mechanism is compromised, there is still a second line of defense. We are experimenting with this approach; more work is needed.

Another natural extension of this work is to run web browsers under the Janus secure environment. The recursive tracing of child processes would ensure that running a browser under Janus would protect all spawned helper applications as well. The arguments which leave us suspicious of helper applications also apply to web browsers: they are large, complex programs that interpret untrusted network data. For example, a buffer overrun bug was found in an earlier version of the Netscape browser [21]. The main challenge is that browsers legitimately require many more privileges; for instance, most manage configuration files, data caches, and network connections. Of these, the broader network access seems to pose the most difficulties.

We believe proxies are a promising approach for improving control over network accesses. By taking advantage of earlier work in firewalls, we were able to easily integrate a safe X proxy into our prototype. We have shown that one can guard access to system calls with a reference monitor constructed from process-tracing facilities; we suspect that one can effectively and flexibly guard access to the outside network with existing proxies developed by the firewall community. 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 such hints with a shared library that replaces network library calls with a safe call to a secure proxy.

So far we have followed the policy that a helper application should not be able to communicate with the outside network, since there are several subtle security issues with address-based authentication, trust perimeters, and covert channels [22]. Integration with filtering proxies and fine-grained control over access to other network services, such as domain nameservers and remote web servers, 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.

8 Conclusion

We designed and implemented a secure environment for untrusted helper applications. We restrict an untrusted program's access to the operating system by using the process tracing facility available in Solaris 2.4. In this way, we have demonstrated the feasibility of building and enforcing practical security for untrusted helper applications.

The Janus approach has two main advantages:

- The Janus protection mechanism is *orthogonal* from other application functionality, so our user-mode implementation is simple and clean. This makes it more likely to be secure, and allows our approach to be broadly applicable to all applications.
- We can protect existing applications with little performance penalty.

We feel that this effort is a valuable step toward security for the World Wide Web.

9 Availability

The Web page
`http://www.cs.berkeley.edu/~daw/janus/`
will contain more information on availability of the Janus software described in this paper.

10 Acknowledgements

We would like to thank Steven Bellovin, David Oppenheimer, Armando Fox, Steve Gribble, and the anonymous reviewers for their helpful comments.

References

[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] 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.

[6] 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.

[7] 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.

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

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

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

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

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

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

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

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

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

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

[18] Frederick Cohen. Personal communication.

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

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

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

[22] 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.

[23] 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.

[24] 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.

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

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

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

[28] 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.

[29] 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.

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

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

[32] 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.

[33] 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.

[34] 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`.

[35] 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.

[36] 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`.

Note: CERT advisories are available on the Internet from `ftp://info.cert.org/pub/cert_advisories/`.

8lgm advisories can be obtained from `http://www.8lgm.org/advisories/`.

Figure 2: Sample configuration file

```
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

tcpconnect allow 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.
# note that the wildcard `*' can match anything, including /
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/servic
es /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.
path allow read,write /dev/tcp /dev/ticotsord

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

# these are here so you can read the files placed by Netscape and Mosaic
path allow read /var/tmp/* /tmp/*

# 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/*
```