CITI Technical Report 02-3

Improving Host Security with System Call Policies

Niels Provos
`provos@citi.umich.edu`

Abstract

We introduce a system that eliminates the need to run programs in privileged process contexts. Using our system, programs run unprivileged but may execute certain operations with elevated privileges as determined by a configurable policy eliminating the need for suid or sgid binaries. We present the design and analysis of the "Systrace" facility which supports fine grained process confinement, intrusion detection, auditing and privilege elevation. It also facilitates the often difficult process of policy generation. With Systrace, it is possible to generate policies automatically in a training session or generate them interactively during program execution. The policies describe the desired behavior of services or user applications on a system call level and are enforced to prevent operations that are not explicitly permitted. We show that Systrace is efficient and does not impose significant performance penalties.

November 11, 2002

Center for Information Technology Integration
University of Michigan
535 West William Street
Ann Arbor, MI 48103-4943

Improving Host Security with System Call Policies

Niels Provos *Center for Information Technology Integration*
University of Michigan
provos@citi.umich.edu

1 Introduction

Computer security is increasing in importance as more business is conducted over the Internet. Despite decades of research and experience, we are still unable to make secure computer systems or even measure their security.

We take for granted that applications will always contain exploitable bugs that may lead to unauthorized access [5]. There are several venues that an adversary may choose to abuse vulnerabilities, both locally or remotely. To improve the security of a computer system, we try to layer different security mechanisms on top of each other in the hope that one of them will be able to fend off a malicious attack. These layers may include firewalls to restrict network access, operating system primitives like non-executable stacks or application level protections like privilege separation [28] . In theory and practice, security increases with the number of layers that need to be circumvented for an attack to be successful.

To recover quickly from a security breach, it is important to detect intrusions and to keep audit trails for post-mortem analysis. While there are many intrusion detection systems that analyse network traffic [26] or host system activity [20] to infer attacks, it is often possible for a careful intruder to evade them [29, 32].

Firewalls can prevent remote login and restrict access, for example to a web server only [11]. However, an adversary who successfully exploits a bug in the web server, gains its privileges and may possibly use them in subsequent attacks to gain even more privileges. With local access to a system, an adversary may obtain *root* privileges, *e.g.*, by exploiting *setuid* programs [6, 10], using *localhost* network access or special system calls [9].

Instead of detecting intrusions, we may try to confine the adversary and limit the damage she can cause. For filesystems, access control lists [14, 30] allow us to limit who may read or write files. Even though ACLs are more versatile than the usual Unix access model, they do not allow complete confinement of an adversary and are difficult to configure.

We observe that the only way to make persistent changes to the system is through *system calls*. They are the gateway to privileged kernel operations. By monitoring and restricting system calls, an application may be prevented from causing harm. Solutions based on system call interposition have been developed in the past [19, 23]. System call interposition allows these systems to detect intrusions as policy violations and prevent them while they are happening. However, the problem of specifying an accurate policy still remains.

This paper presents *Systrace*, a solution that efficiently confines multiple applications, supports multiple policies, interactive policy generation, intrusion detection and prevention, and can be used to generate audit logs. Furthermore, we present a novel approach called *privilege elevation* that eliminates the need for *setuid* or *setgid* binaries. We discuss the design and implementation of Systrace and show that it is an extensible and efficient solution to the host security problem.

The remainder of the paper is organized as follows. Section 2 discusses related work. In Section 3, we provide further motivation for our work. Section 4 presents the design of Systrace and Section 5 discusses its implementation. We present an analysis of the system in Section 6. In Section 7, we present a detailed performance analysis of our system. We conclude in Section 8.

2 Related Work

While capabilities [25] and access control lists [14, 30] extend the traditional Unix access model to provide finer-grained controls, they do not prevent untrusted applications from causing damage.

Instead, we may use mechanisms based on system call interception or system call interposition to prevent damage from successful intrusions.

Janus by Goldberg *et al.* [19] is one of the first system call interception tools. It uses the *ptrace* and */proc* mechanisms. Wagner states that *ptrace* is not a suitable interface for system call interception, *e.g.*, race conditions in the interface allow an adversary to completely escape the sandbox [33]. The original Janus implementation has several drawbacks: Applications are not allowed to change their working directory or call *chroot* because Janus can not keep track of the application's changed state. Janus has evolved significantly over time and its latest version uses a hybrid approach similar to Systrace to get direct control of system call processing in the operating system [17].

One particularly difficult problem in application confinement are symlinks, which can be used to redirect filesystem access almost arbitrarily. Garfinkel introduces safe calling sequences that do not follow any symlinks. The approach uses an extension to the *open* system call that is specific to the Linux operating system but breaks any application that accesses filenames containing symlinks. Systrace solves this problem using filename normalization and argument replacement. Currently, Janus does not address intrusion detection, auditing or policy generation.

Jain and Sekar [23] offer another fairly complete treatment of system call interposition. On some systems their implementation is based on *ptrace* and suffers the problems mentioned above. Furthermore, they do not address the problem of naming ambiguities that may result in policy circumvention. Because C++ is used as their policy language, creating comprehensive policies is difficult. Systrace, on the other hand, supports automatic and interactive policy generation which allows us to create policies quickly even in very complex environments.

Other systems that use mechanisms like system call interception are Cerb [13], Consh [2], MAPbox [1] and Subterfugue [12].

Peterson *et al.* present a general-purpose system call API for confinement of untrusted programs [27]. The API is flexible but has no provisions for recording audit trails or intrusion detection. Furthermore, specifying security policies is labor intensive as the sandbox needs to be programmed into applications.

Domain Type Enforcement [4, 34] is a kernel-level approach to restrict system access for all processes depending on their individual domains. A complete DTE implementation requires extensive changes to the operating system and does not automatically extend to new subsystems. Because policies are locked down on system start, users may not create individual policies. In contrast to Systrace, DTE domains do not differentiate between users. We feel that system call interposition offers higher flexibility as it allows us to design and create a simple system that also addresses policy generation, audit trails, intrusion detection, etc.

The security architecture for the Flask microkernel emphasizes *policy flexibility* and rejects the system call interception mechanism claiming inherent limitations that restrict policy flexibility [31]. Instead, the Flask system assigns security identifiers to every object and employs a security server for policy decisions and an object server for policy enforcement. However, Flask does not consider application confinement and provides no support for auditing or intrusion detection.

SubOS [22] takes a similar approach based on object labeling to restrict access to the system. Depending on their origin, objects are assigned sub-user identifiers. A process that accesses an object inherits its sub-user id and corresponding restrictions. As a result, a process subverted by a malicious object may cause only limited damage. In practice, there are only a few applications that can be subverted that way and enforcing security policies for these applications is sufficient to prevent malicious data from causing damage.

Forrest *et al.* analyze system call sequences to discriminate between processes [15]. Their is extended by Hofmeyer *et al.* to achieve intrusion detection by recording the system calls that an application executes and comparing the recorded sequences against a database of good sequences [20]. Abnormal sequences indicate an ongoing intrusion. The training process that collects good system call sequences is similar to the automatic policy generation in Systrace. While analyzing system call sequences is an effective mechanism to detect intrusions, it does not help to prevent them. Recent research also shows that *mimicry* attacks can evade the intrusion detection system [32]. Systrace not only detects intrusions, it can also prevent them or at least limit the damage they can cause. Furthermore, evasion attacks are not possible as we discuss in Section 6.

3 Motivation and Threat Model

Most applications that run on computer systems are too complex and complicated to trust: web browsers, name servers, etc. Even with access to the source code, it is difficult to reason about the security of these applications. They might harbor malicious code or contain bugs that are exploitable by carefully crafted input.

Because it is not possible to find all vulnerabilities, we assume the existence of programming errors known to the adversary that she can use to gain unauthorized access to the system.

We aim to limit the impact an adversary can have on the system by restricting the operations an application is allowed to execute. The observation that security relevant changes are performed via *system calls* makes the enforcement of restrictions at the system call level a natural choice.

An application is confined by a set of restrictions which are expressed by a security policy. Defining a correct policy is difficult and not possible without knowing all possible code paths that an uncompromised application may take. Therefore we require the policy language to be intuitive while still expressive, and it should be possible to generate policies without complete knowledge of an application.

We may use the security policy as specification that describes the expected behavior of an application. When monitoring the operations an application attempts to execute, any deviation from the specified policy indicates a security compromise [24]. To further facilitate forensic analysis of an intrusion, we also wish to generate an audit log of previous operations related to the application.

Experience shows that adversaries escalate their privileges by abusing *setuid* or *setgid* programs [6]. These programs are executed by the operating system with different privileges than the user starting them. While increasing privileges is often necessary for correct operation, the *setuid* model is too coarse grained. We aim to provide a fine-grained model that eliminates the need for *setuid* binaries and integrates a method to elevate privilege into a policy language.

Systrace realizes these goals and is an effective improvement of host security that limits the damage an adversary can cause by exploiting application vulnerabilities. The next section discusses the design of Systrace.

4 Design

There are several approaches for implementing system call interposition. We may use existing interception mechanisms to create an implementation completely in user space, we may implement the system entirely at the kernel-level or choose a hybrid of both. A user space implementation is often more portable but may suffer a larger performance impact. Furthermore, the interception mechanism may not provide the required security guarantees or make it difficult to keep track of operating system state like processes exiting and forking. A notable exception is SLIC [18], a mechanism to create extensible operating systems via system call interposition. Unfortunately, it is not portable and adds significant complexity to the operating system.

On the other hand, an implementation completely at the kernel-level is likely to be fast but less portable and also causes a significant increase in the complexity of the operating system.

We choose a hybrid approach to implement a small part of the system at the kernel-level. The kernel-level part supports a fast path for system calls that should always be allowed or denied. That case should incur almost no performance penality because it does not require a context switch to ask a user space policy daemon for a decision.

Some control in the kernel also allows us to make the system *fail-safe*, *i.e.*, no application can escape its sandbox even if there are unforeseen errors that might cause the monitor itself to terminate. When the sandboxing process terminates, the kernel terminates all processes that it was monitoring. Additionally, the kernel keeps track of new processes being created and of processes that exit. Child processes inherit the policy of their parents.

If the kernel can not use the fast path for a system call, it needs to ask the policy daemon in user space for a policy decision. In that case, the process is blocked until the daemon returns with an answer to permit the system call or to deny it with a certain error code. Information is exported from the kernel to user space via a simple yet comprehensive interface.

The user space policy daemon uses the interface to the kernel to start monitoring processes and to get information about pending policy decisions or state changes. The state changes may be process creation, processes exiting, processes changing their uid or gid, and other state changes.

The daemon may also request information about the result of a system call. This allows us to know, for example if the *execve* system call has succeeded in replacing the current process image with a new application. This event can be used to install a new policy from the policy database.

System call interception does not provide atomicity between the time a policy decision is made and the time a system call is executed. As a result, an adversary can change the system call before it is executed but after the policy daemon has inspected it. For example, two processes that share parts of their address space may cooperate to present one set of system call arguments to the policy daemon and another one to the kernel. When the kernel suspends the first process to consult the policy daemon, the second process is still running and may change the system call arguments of the first process after they have been inspected by the daemon. This lack of atomicity may result in the execution of unchecked arguments allowing an adversary to escape the sandbox.

We avoid these race conditions by replacing the system call arguments in the kernel with the arguments that were resolved and evaluated by Systrace. This ensures that the kernel executes only system calls that passed the policy check.

Using argument replacement, it is possible to change the way an application interacts with the operating system. By changing filename arguments, it is possible to present a virtual filesystem layout to the application. We may also rewrite the addresses an application attempts to access on the network. This allows us to redirect network traffic to different hosts. A more detailed analysis of this feature will be the subject of future work.

Before making a policy decision, the system call and its arguments are translated into a system independent human-readable format. The policy language operates on that translation and does not need to be aware of system call specific semantics.

4.1 Policy

Existing frameworks for making policy decisions propose generic policy languages [7, 8] that provide policy evaluation methods but are more complex than necessary in our case.

We use an ordered list of policy statements per system call. A policy statement is a boolean expression B combined with an action clause: B then *action*. Valid actions are either *deny* or *permit* plus optional flags. If the boolean expression evaluates to *true*, the specified action is taken.

The variables X_n of a boolean expression are tuples of the form ($subject$ *op* $data$), where $subject$ is the translated name of a system call argument, *data* is a string argument, and *op* a function with boolean return value that takes $subject$ and *data* as arguments.

The set of all lists forms the security policy. For a given system call, policy evaluation starts at the beginning of the system call specific list and terminates with the first boolean expression that is true. The action from that expression determines if the system call is denied or allowed.

If no boolean expression becomes true, the policy decision is forwarded to the user of the application or automatically denied depending on the configuration. Section 4.2 explains in more detail how this mechanism is used to generate policies interactively or automatically. When denying a system call, it is possible to specify which error code is passed to the monitored application.

To create comprehensive policies that apply to different users, policy statements may carry predicates. A policy statement is evaluated only if its predicate matches and ignored otherwise. Using predicates, it is possible to restrict the actions of certain users or be more permissive with others, for example system administrators. Predicates are appended to the policy statement and are of the form $if\ \{user, group\}$ *op* *data*, where *op* is either equality or inequality and *data* a user or group name.

The *log* modifier may be added to a policy statement to record matching system calls. Every time a system call matches this policy statement, the operating system records all information about the system call and the resulting policy decision. This allows us to create arbitrarily fine-grained audit trails.

4.2 Policy Generation

Creating policies is usually relegated to the user who wishes to sandbox applications. Policy generation is not an easy task as some policy languages resemble complicated programming languages [23]. While those languages are very expressive, the difficulty of creating good policies increases with the complexity of the policy language.

We designed our policy language to be simple and stateless. Each policy statement can be evaluated by itself, thus it is possible to extend a policy by appending new policy statements. The major benefit of this approach is that a policy can be generated iteratively.

We create policies automatically by running an application and recording the system calls that it executes. We translate the system call arguments and canonically transform them into policy statements for the corresponding system calls. When an application attempts to execute a system call during the training run, it is checked against the existing policy and if not covered by it, a new policy statement that permits this system call is appended to the policy.

On subsequent runs of the application, the automatically created policy is used. For some applications that create random file names, it is necessary to edit the policies by hand to account for nondeterminism.

When generating policies automatically, we assume that the application itself does not contain malicious code and that it operates only with benign data. Otherwise, the resulting policies might permit undesirable actions.

To address cases for which our assumptions do not hold or for which it is impossible to exercise all code paths in a training run, we use interactive policy generation. Interactivity implies a the user needs to make policy decisions when the current policy does not cover the attempted system call. When a policy decision is required by the user, she is presented with a graphical notification that contains all relevant information. She

```
Policy: /usr/sbin/named, Emulation: native
     native-__sysctl: permit
     native-accept: permit
     native-bind: sockaddr match "inet-*:53" then permit
     native-break: permit
     native-chdir: filename eq "/" then permit
     native-chdir: filename eq "/namedb" then permit
     native-chroot: filename eq "/var/named" then permit
     native-close: permit
     native-connect: sockaddr eq "/dev/log" then permit
                 ...
```

Figure 1: Partial policy for the name daemon. Policies can be improved iteratively by appending new policy statements. The policy statement for *bind* allows the daemon to listen for DNS requests on any interface.

then either improves the current policy by appending a policy statement that covers the current system call or terminates the application.

The system assists the user by offering generic policy templates that can be used as starting point. Once an initial policy has been created, policy notifications appear only when an attempted operation is not covered by the configured policy. This might indicate that a new code path is being exercised, or that a security compromise is happening. The user may either permit the operation or deny and investigate it.

Once a security policy for an application has been finalized, automatic policy enforcement may be employed. In that case, the user is not asked for a policy decision when an application attempts to execute a system call that is not covered by the policy. Instead, the system call is denied and an error code returned to the application. The errant attempt is logged by the operating system.

4.3 Privilege Elevation

Beyond restricting an application to its expected behavior, there are situations in which we would like to increase its privileges. In Unix, there are many system services and applications that require *root* privileges to operate. Often, higher privileges are required only for a few operations. Instead of running the whole application with special privileges, we extend the policy language to assign the desired privileges to a single system call. The kernel raises the privileges just before the system call is executed and lowers them directly afterwards.

As every user may run their own policy daemon, privilege elevation is available only when the Systrace policy daemon runs as *root*. Otherwise, it would be possible for an adversary to obtain unauthorized privileges by creating her own policies. Identifying the privileged operations of *setuid* or *setgid* applications allows us to create policies that elevate privileges of those operations without the need to run the whole application privileged. As a result, an adversary who manages to seize control of a vulnerable application receives only very limited additional capabilities instead of full privileges.

The *ping* program, for example is a *setuid* application as it requires special privileges to operate correctly. To send and receive ICMP packets, *ping* creates a raw socket which is a privileged operation in Unix. With privilege elevation, we execute *ping* without special privileges and use a policy that contains a statement granting *ping* the privilege to create a raw socket.

Unix allows an application to discard privileges by changing the *uid* and *gid* of a process. The change is permanent and the process can not recover those privileges later. If an application occasionally needs special privileges throughout its lifetime dropping privileges is not an option. In this case, privilege elevation becomes especially useful. For example, the *ntpd* daemon synchronizes the system clock. Changing system time is a privileged operation and *ntpd* retains *root* privileges for its whole lifetime. A recent remote root vulnerability [16] could have been prevented with single system call privilege elevation.

5 Implementation

We now give an overview of the Systrace implementation. Systrace is available for Linux, NetBSD, and OpenBSD; we concentrate on the OpenBSD implementation.

To help reason about the security of our implementation, simplicity is one of our primary goals. We keep the implementation simple by introducing abstractions

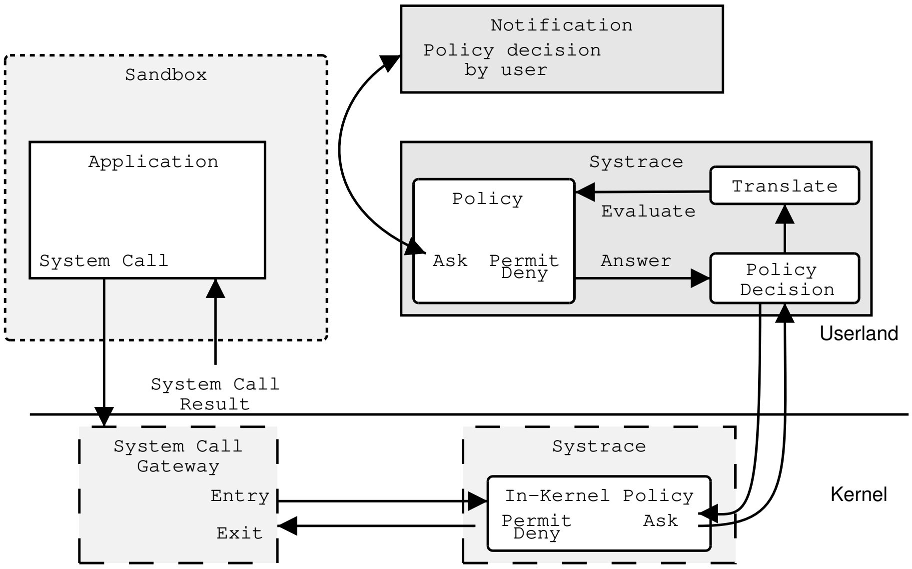

Figure 2: Overview of system call interception and policy decision. For an application executing in the sandbox, the system call gateway requests a policy decision from Systrace for every system call. The in-kernel policy provides a fast path to permit or deny system calls without checking their arguments. For more complex policy decisions, the kernel consults a user space policy daemon. If the policy daemon can not find a matching policy statement, it has the option to request a refined policy from the user.

that separate different functionalities into their own components. A conceptual overview of the system call interception architecture is shown in Figure 2.

When a monitored application executes a system call, the kernel consults a small in-kernel policy database to check if the system call should be denied or permitted without asking the user space daemon. At this point, policy decisions are made without inspecting any of the system call arguments. Usually, system calls like *read* or *write* are always permitted. The kernel communicates via the `/dev/systrace` device to request policy decisions from the daemon.

While processes may have different policies, the initial policy for all system calls defers policy decisions to a corresponding user space daemon. When the kernel is waiting for an answer, it suspends the process that requires a policy decision. If the process is awakened by a signal before a policy decision has been received, the kernel denies the current system call and returns an error. To enforce synchronization, each message from the kernel carries a sequence number so that answers from user space can be matched against the correct message. The sequence number ensures that a user space policy decision is not applied to a system call other than the one that caused the message.

When the user space policy daemon receives a request for a decision, it looks up the policy associated with the process and translates the system call arguments. To translate them, we register translators for each argument in a system call. The translation of the

```
socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
```

system call takes the following form:

```
socket: sockdom: AF_INET, socktype: SOCK_RAW
```

While many argument translators are fairly simple, translating filenames is more complicated. Filenames in Unix are relative to the current working directory of a process. In order to translate a filename into an unambiguous absolute path name, we need to know the current working directory of the monitored application even if it is working in a *chroot* environment. Additionally, all symbolic links in components of the file name need to be resolved so that access restrictions imposed by policy can not be circumvented by an adversary.

The translators also act as argument normalizers. The argument replacement framework is used to replace the original arguments with their translation. As the kernel sees only normalized arguments, an adversary can not use misleading arguments to circumvent a security policy. The kernel installs the rewritten arguments on the stack of the monitored process before resuming execution of the system call.

A policy statement that permits the creation of raw sockets might look like this:

```
socket:  socktype eq "SOCK_RAW" then permit
```

The operators in the boolean expression use the translated human-readable strings as input arguments. We currently support `eq, match, re` and `sub` as operators:

- The `eq` operator evaluates to *true* only if the system call argument matches the text string in the policy statement exactly.
- The `match` operator performs file name globbing as found in the Unix shell. It can be used to match files in directories for file name arguments.
- The `re` operator uses regular expressions to match system call arguments. It is very versatile but more expensive to evaluate than other operators.
- The `sub` operator evaluates to *true* only if the system call argument contains the specified substring.

If evaluating the policy for the current system call results in either *deny* or *permit*, the policy daemon returns the answer to the kernel which then awakens the sleeping process. Otherwise, the user monitoring the applications is asked for a policy decision. The notification mechanism can be implemented independently from the rest of the system and is currently either a graphical user interface or a text prompt on the terminal. At this point, the user can add new policy statements to the policy.

Policies for system calls accessing the filesystems tend to be similar. For example, the *access*, *stat*, and *lstat* system calls all fulfill similar functionality. In order to avoid duplication of policy, we introduce *system call aliasing* to map system calls with similar functionality into a single virtual system call which is then used for policy evaluation. Currently, *fsread* is used for system calls that read from the filesystem, and *fswrite* for system calls that cause change in the filesystem. The *open* system call is mapped to *fsread* or *fswrite* depending on the kind of filesystem access that is indicated by its arguments. System call aliasing reduces the size of policies and simplifies policy generation.

It is possible to make policies more flexible by using predicates. Policy statements are only evaluated if their predicate matches. For example, to prevent *root* access via the SSH daemon, a policy statement that permits the execution of a shell could be predicated so that it applies only to non-root users. In order to keep track of a process' *uid* and *gid*, the kernel sends informational messages to the policy daemon when those values change.

The *execve* system call is treated specially. When a process executes another application, its in-memory image is replaced with the one of the executed program. To support more fine-grained policies, we can set a new policy for the process. The policy is obtained from the name of the executed application. As a result, one Systrace daemon may concurrently enforce multiple policies for multiple processes.

Policies for different applications are stored in a policy directory as separate files. Users may store their own policies in a user specific policy directory. The system administrator may also provide global policies for all users.

6 Analysis

An adversary who takes control of a sandboxed application may try to escape the sandbox by confusing the policy enforcement tool and tricking it into allowing actions that violate policy. While many sandboxing tools share common problems, we present novel solutions to some of them and discuss inherent limitations of policy systems based on system call interposition.

6.1 Security Analysis

To enforce security policies effectively by system call interposition, we need to resolve the following challenges: incorrectly replicating OS semantics, resource aliasing, lack of atomicity, and side effects of denying system calls [17, 31, 33].

We briefly explain their nature and discuss how we address them.

The sandboxing tool needs to track operating system state in order to reach policy decisions. Systrace, for example, has to keep track of process *uid*s and the filename of the program binary the monitored process is executing. In order to avoid incorrectly replicating OS semantics, our kernel-level implementation informs the Systrace daemon about all relevant state changes.

Resource aliasing provides multiple means to address and access the same operating system resource. For example, on some Unix systems, it is possible to gain access to files by communicating with a system service or by using symbolic links in the filesystem to create different names for the same file. An adversary may use these indirections to circumvent policy and obtain unauthorized access. The system call interposition mechanism is unaware of system services that allow proxy access to operating system resources. When creating policies that allow a sandboxed application to contact such system services, we need to be aware of the consequences. However, we can prevent aliasing via symbolic links or relative pathnames as discussed below.

Another problem is the lack of atomicity that may cause the mapping of names to resource to change between policy decision and system call execution. An adversary may cause such a state change that allows a process to access a different resource than the one originally approved, for example a cooperating process sharing memory may rewrite system call arguments between policy check and execution.

Systrace solves both aliasing and atomicity problems by *normalizing* the system call arguments. We provide the *normalized* values to the operating system in such a way that the name to resource mapping can not be changed by an adversary. For filenames this includes resolving all symbolic links and all relative paths.

Side effects of denying system calls need to be taken into consideration when system calls are being denied. If we assume correct security policy, system calls are denied only if an application attempts to do something that it should not. As the behavior of many applications depends on the error code returned to them, we can specify the error code as part of the Systrace policy. Every system call has its own set of valid return codes which do not always include `EINTR` or `EPERM`. To avoid confusing applications, we allow policies to set their own error code instead of mandating a fixed value. For example, we let the kernel return `EACCESS` for the *stat* system call if the application should think that it is not permitted to access a certain file. On the other hand, returning `ENOENT` causes the application to think that the file does not exist.

Furthermore, we address secure process detaching and policy switching, problems that are often overlooked. When an application executes a new program, the operating system replaces the code that the process is running with the executed program. If the new program is trusted, we may wish to stop monitoring the process that runs it. On the other hand, a new program also implies new functionality that could be confined better with a different, more suitable policy. If requested, Systrace reports the return value of a system call to indicate if it was successfully executed or not. In the case of *execve*, success indicates that the monitored process is running a new program and we allow the policy to specify if we should detach from the process or allow a different policy to take effect. After these changes take effect, the execution of the process is resumed.

Because the security of our system relies on the integrity of the filesystem, we assume that it is secure. If an adversary can control the filesystem, she may modify the policies that determine the permissible operations for monitored applications or replace trusted programs with malicious code. We may relax this assumption if other security mechanisms are in place such as a secure bootstrap architecture [3] that prevent an adversary from corrupting the filesystem.

Audit trails may be generated by adding the *log* modifier to policy statements. An an example, for an audit trail of all commands a user executes, it is sufficient to Systrace her shell and log all the executions of *exexve*.

The benefit of privilege elevation is the reduction of privileges an application requires for its execution. Applications that formerly required *root* privileges for their entire lifetime now execute only specific system calls with elevated privileges. Other system calls are executed with the privileges of the user who invoked the application. The semantics of *setuid* prevent a user from debugging privileged applications via *ptrace*. We apply the same semantics when policy elevates an application's privileges.

6.2 Policy generation

Policy generation is an often neglected problem. In order for a sandbox to function correctly, it requires a policy that restricts an application to a minimal set of operations without breaking its functionality. To facilitate policy generation, our policy language allows policies to be improved iteratively by appending new policy statements.

We can generate policies automatically by executing applications and recording their normal behavior. Each time we encounter a system call that is not part of the existing policy, we append a new policy statement that matches the current translated system call.

The resulting policy covers the executed code path of the application. For applications that randomize arguments, we post process the policy to make it independent of arguments with random components.

For example, when `mkstemp("/tmp/confXXXXXX")` creates the file `/tmp/confJ31A69`, automatic policy generation appends a corresponding policy statement:

```
fswrite: filename eq "/tmp/confJ31A69" then permit
```

Post processing changes the policy statement so that it is independent of the randomness and thus applies to subsequent executions of the application:

```
fswrite: filename match "/tmp/conf*" then permit
```

Automatic policy generation and the process of profiling normal application behavior by Hofmeyr *et al.*[20] face similar problems. We need to make sure that no abnormal behavior occurs during policy training and try to exhaust all possible code paths. However, interactive and automatic policy generation go hand in hand. We do not require a complete policy to sandbox an application because we may request a policy decision from the user if an operation is not covered by the existing policy.

The feasibility of our approach is demonstrated by *monkey.org*, a Unix shell provider in Ann Arbor, who uses Systrace to sandbox over two hundred users. They generated separate policies for approximately 250 applications.

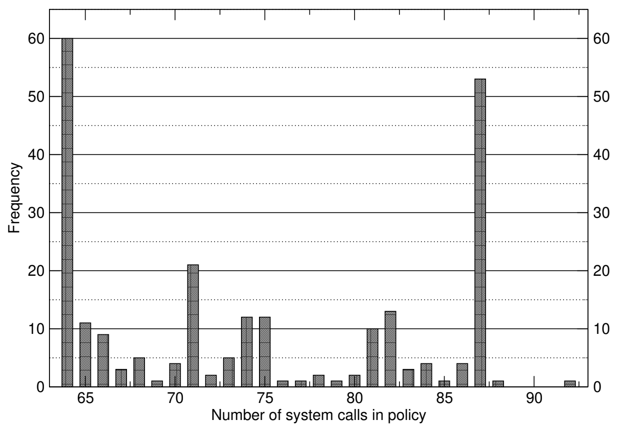

Figure 3: Analysis of the number of system calls that applications are allowed to execute. Most applications use only sixty to ninety different system calls. As average Unix systems support several hundred system calls, we disallow the execution of all other system calls to prevent an adversary from using them to cause damage. Note that the abscissa origin is not zero.

An analysis of the policies shows that applications are allowed to call 71 different system calls on average; see Figure 3. Usually Unix systems support several hundred system calls. When an adversary gains control over an application, she may attempt to obtain higher privileges by using all possible system calls. By limiting the adversary to only those system calls required by the application, we reduce her potential to cause damage.

We notice two peaks, one at sixty four system calls and the other one at eighty seven. The first peak is caused by policies for standard Unix utilities like *chmod*, *cat*, *rmdir* or *diff* all of which have similar policies. The second peak is caused by identical policies for the different utilities in the MH message system, which require more system calls for establishing network connections and creating files in the filesystem.

Most of the policy statements specify access to the filesystem: 24% of them control *read* access, 6% *write* access and 5% the execution of other programs.

6.3 Intrusion Detection and Prevention

The capability for intrusion detection and prevention follows automatically from our design. System calls that violate the policy are denied and recorded by the operating system. This prevents an adversary from causing damage and creates an alert that contains the restricted operation.

A correct policy restricts an application to only those operations required for its functionality. While this prevents an adversary from harming the operating system arbitrarily, she may still abuse an application's functionality to cause damage. We employ audit trails to log malicious activity not prevented by policy.

6.4 Limitations

Although powerful, policy enforcement at the system call level has inherent limitations. Monitoring the sequence of system calls does not give complete information about an application's internal state. For example, some system services change the privileges of a process on successful authentication but deny extra privileges if authentication fails. A sandboxing tool at system call level can not account for such state changes. However, it is still possible to enforce global restrictions that, for example, state that *root* should never be allowed to login. This is possible because those restrictions do not depend on an application's internal state.

To increase the security of authentication services like SSH, it is possible to use a combination of *privilege separation* [28] and *system call policy enforcement*. With privilege separation, the majority of an application is executed in an unprivileged process context. Vulnerability in the unprivileged code path should not lead to privilege escalation. However, in a Unix system an unprivileged process can still execute system calls that allow local network access. Using Systrace to sandbox the application, we can prevent the unprivileged process from executing any system calls that are not necessary for its functionality.

6.5 Other Applications

Systrace may also be used for quality assurance by injecting random faults into a running application. This allows us to introduce error conditions that are not normally triggered and to observe if the application recovers correctly from them. For example, we may simulate resource starvation such as a full filesystem or out-of-memory conditions.

7 Performance

To determine the performance impact of Systrace, we measured its overhead on the execution time of single system calls and on several applications. All measurements were repeated several times on a 1.14 GHz Pentium III running OpenBSD. The results are displayed as averages with corresponding standard deviation.

Mode	Real time in μsec	User time in μsec	System time in μsec
Normal	0.35 ± 0.00	0.14 ± 0.03	0.22 ± 0.03
In-kernel	0.46 ± 0.01	0.17 ± 0.04	0.28 ± 0.04
User space	37.71 ± 0.18	0.30 ± 0.07	5.60 ± 0.61

Figure 4: A microbenchmark to compare the overhead of a single *geteuid* system call for an unmonitored process and for process confinement with different policies. Making a policy decision in the kernel is considerably faster than requesting a policy decision from the user space policy daemon.

We conduct the microbenchmarks of a single system call by repeating the system call several hundred thousand times and measuring the real, system, and user time. The execution time of the system call is the time average for a single iteration.

As a baseline, we measure the time for a single *geteuid* system call without monitoring the application. We compare the result with execution times obtained by running the application under Systrace with two different policies. The first policy permits the *geteuid* via the in-kernel policy table. For the second policy, the kernel consults the user space policy daemon for a decision. We see that the *in-kernel* policy evaluation increases the execution time by $31\% \pm 3\%$ and that slightly more time is spent in the kernel. When the kernel has to ask the user space daemon for a policy decision, executing a single system call takes much longer, mostly due to two context switches required for every policy decision. The results are shown in Figure 4.

The *open* system call requires more work in the kernel than *getuid*. A microbenchmark shows that the in-kernel evaluation of the policy increases the execution time by $7\% \pm 0.6\%$. The execution time for a user space policy decision depends on the depth of the file in the directory tree. When the path to the filename has only one component, the increase in execution time is over 25-fold. Each directory component in the path adds approximately thirty microseconds to the execution time due to filename normalization, as shown in

Mode	Real time in μsec	User time in μsec	System time in μsec
Normal	5.52 ± 0.01	0.34 ± 0.20	5.08 ± 0.16
In-kernel	5.88 ± 0.03	0.31 ± 0.22	5.55 ± 0.22
1-deep	139.20 ± 0.09	0.56 ± 0.12	15.80 ± 1.01
2-deep	167.72 ± 0.41	0.64 ± 0.18	15.84 ± 1.10
3-deep	198.34 ± 0.67	0.40 ± 0.17	18.28 ± 0.38
4-deep	231.121 ± 0.27	0.43 ± 0.13	19.40 ± 1.39

Figure 5: A microbenchmark to compare the overhead of the *open* system call. Due to filename normalization, the time to make a policy decision in user space depends on the number of components in the filename. Every component adds a about 30 μsec.

Mode	Real time in μsec	User time in μsec	System time in μsec
Normal	37.61 ± 0.03	0.11 ± 0.11	37.34 ± 0.10
In-kernel	37.61 ± 0.03	0.14 ± 0.16	37.45 ± 0.21

Figure 6: A microbenchmark to compare the overhead of the *read* system call when reading a 1 kbyte buffer from /dev/arandom. In this case, there is no measurable performance penality for the in-kernel policy decision.

Figure 5.

The last microbenchmark measures the overhead of using the *read* system call to read a 1 kbyte buffer from `/dev/arandom`, which outputs random data created by a fast stream cipher. There is no noticeable difference in execution time and system time increases by less than 1% for in-kernel policy evaluation. We omit measurement of user space because *read* requires no user space policy decision. The results are shown in Figure 6.

Enforcing system call policies adds overhead to an application's execution time, but the overall increase is small.

Figure 7 compares the runtime of *gzip* for different file sizes from 500 kByte to 5 MByte. Gzip executes thirty system calls per second on average, most of them *read* and *write*. In this case, the execution time is not significantly effected by Systrace, because the application spends most of its time computing, and executes relatively few system calls.

To assess the performance penality for applications that frequently access the filesystem, we created a benchmark similar to the Andrew benchmark [21]. It consists of copying a tar archive of the Systrace sources, untarring it, running configure, compiling the sources and then deleting all files in the source code subdirectory.

File size in MByte	Normal	Systrace	Increase in percent
0.5	0.88 ± 0.04	0.92 ± 0.07	4.5 ± 9.3
1.4	2.51 ± 0.01	2.52 ± 0.01	0.4 ± 0.6
2.3	4.15 ± 0.01	4.17 ± 0.01	0.5 ± 0.3
3.2	5.62 ± 0.01	5.64 ± 0.01	0.4 ± 0.3
4.0	7.18 ± 0.03	7.18 ± 0.03	0.0 ± 0.6
4.9	8.55 ± 0.01	8.57 ± 0.02	0.2 ± 0.3

Figure 7: A macrobenchmark comparing the runtime of an unmonitored gzip process to gzip running under Systrace. Because this benchmark is computationally intensive, policy enforcement does not add a significant overhead.

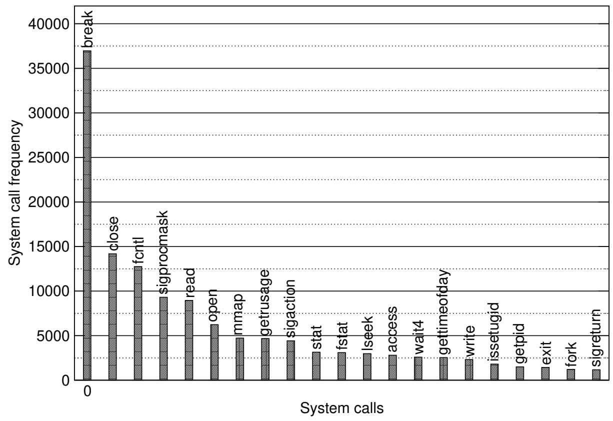

Figure 8: Histogram of system call frequency for compilation benchmark. The performance impact of application confinement depends mostly on the number of system calls that require a policy decision by the user space daemon. The histogram shows that the most frequent system calls can be handled by the in-kernel policy.

During the benchmark, forty four application programs are executed. We use Systrace to generate policies automatically, then improve the policies that result with a simple script. The benchmark executes approximately 137,000 system calls. A decomposition into the most frequent system calls is shown in Figure 8. The system call with the highest frequency is *break* which is used to allocate memory. System calls that access the filesystem are also prominent.

A direct comparison between the execution times is shown in Figure 9. Under Systrace, we notice an increase in running time by $31\% \pm 1.4\%$. The number of system calls increases to approximately 726,000 because filename normalization requires the *getcwd* function, which causes frequent calls to *lstat* and *fstat*. Running the same benchmark under NetBSD 1.6I shows a significantly smaller increase in system calls because it

Benchmark	Normal in sec	Systrace in sec	Increase in percent
Compile	10.44 ± 0.09	13.71 ± 0.09	31 ± 1.4
Crawler	0.84 ± 0.03	0.88 ± 0.03	4.8 ± 5.2
Gzip-4.9M	8.55 ± 0.01	8.57 ± 0.02	0.2 ± 0.3

Figure 9: Overview of different macrobenchmarks comparing the execution time of an unmonitored run with the execution time running under Systrace. The compilation benchmark incurs the highest performance penality. On the other hand, it is very complex, consisting of more than forty applications and still shows acceptable performance. Running the other benchmarks with Systrace incurs only small performance penalties.

implements a *getcwd* system call.

A second macrobenchmark measures the runtime of a web crawler that downloads files from a local web server. The crawler retrieves approximately one hundred forty files with an average throughput of two megabytes per second. For this macrobenchmark, the running time under Systrace increases only by $4.8\% \pm 5.2\%$; see Figure 9.

The additional cost of Systrace, although noticeable is not prohibitive, especially for interactive applications like web browsers, in which there is no observable performance decrease for the end user.

8 Conclusion

This paper presents a new approach for application confinement that supports automatic and interactive policy generation, auditing, intrusion detection and privilege elevation and applies to both system services and user applications. We argued that system call interception is a flexible and appropriate mechanism for intrusion prevention. Our hybrid implementation enables fail-safe operation and low performance overhead while maintaining good portability. This paper addresses important issues not addressed by previous research. The translation of system call arguments into human-readable strings allows us to design a simple policy language. It enables our system to generate fine grained policies both automatically and interactively. The resulting policies restrict applications without breaking their functionality.

Privilege elevation in conjunction with application confinement allows us to reduce significantly the privileges required by system services. Using privilege elevation, we assign fine-grained privileges to applications without requiring the *root* user. Instead of retaining *root* privileges throughout an application's lifetime, an application may run without special privileges and receives elevated privileges as determined by policy.

Our security analysis discusses how we overcome problems common to system call interception tools and how our design allows for further functionality such as intrusion detection and prevention.

We analyzed the performance of Systrace and showed that additional performance overhead is acceptable and often not observable by the user of a sandboxed application.

9 Acknowledgments

I would like to thank Peter Honeyman, Perry Metzger, Terrence Kelly and Jose Nazario for careful reviews. I also thank Perry Metzger, Dug Song, Markus Watts and Marius Eriksen for helpful discussions on this topic.

References

[1] Anurag Acharya and Mandar Raje. MAPbox: Using Parameterized Behavior Classes to Confine Applications. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[2] Albert Alexandrov, Paul Kmiec, and Klaus Schauser. Consh: Confined Execution Environment for Internet Computations, 1998.

[3] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A Secure and Reliable Bootstrap Architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 65–71, May 1997.

[4] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. A Domain and Type Enforcement UNIX Prototype. In *Proceedings of the 5th USENIX Security Symposium*, pages 127–140, June 1995.

[5] Steven M. Bellovin. Computer Security - An End State? *Communications of the ACM*, 44(3), March 2001.

[6] Matt Bishop. How to write a setuid program. *;login;*, 12(1):5–11, 1987.

[7] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos Keromytis. The KeyNote trust-management system version 2. RFC 2704, September 1999.

[8] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized Trust Management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173, May 1996.

[9] CERT. OpenBSD contains buffer overflow in "select" call. Vulnerability Note VU#259787, August 2002. `http://www.kb.cert.org/vuls/id/259787`.

[10] Hao Chen, David Wagner, and Drew Dean. Setuid Demystified. In *Proceedings of the 11th Usenix Security Symposium*, August 2002.

[11] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security Repelling the Willy Hacker.* Addison-Wesley Publishing Company, 1994.

[12] M. Coleman. Subterfogue: A Framework for Oberserving and Playing with Reality of Software. `http://subterfugure.org/`.

[13] Pawl J. Dawidek. Cerb: System Firewall Mechanism. `http://cerber.sourceforge.net/`.

[14] G. Fernandez and L. Allen. Extending the UNIX Protection Model with Access Control Lists. In *Proceedings of the Summer 1988 USENIX Conference*, pages 119–132, 1988.

[15] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128, 1996.

[16] Przemyslaw Frasunek. ntpd $\leq$ 4.0.99k remote buffer overflow. Bugtraq, April 2001. CVE-2001-0414.

[17] Tal Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, 2003. To appear.

[18] Douglas P. Ghormley, Steven H. Rodrigues, David Petrou, and Thomas E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 39–52, June 1998.

[19] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 6th Usenix Security Symposium*, July 1996.

[20] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[21] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[22] Sotiris Ioannidis, Steven M. Bellovin, and Jonathan M. Smith. Sub-Operating Systems: A New Approach to Application Security. In *Proceedings of the SIGOPS European Workshop*, September 2002.

[23] K. Jain and R. Sekar. User-Level Infrastructure for System Call Interposition: A Platform for Intrustion Detection and Confinement. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, February 2000.

[24] Calvin Ko, George Fink, and Karl Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, December 1994.

[25] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984. `http://www.cs.washington.edu/homes/levy/capabook/`.

[26] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[27] David S. Peterson, Matt Bishop, and Raju Pandey. A Flexible Containment Mechanism for Executing Untrusted Code. In *Proceedings of the 11th USENIX Security Symposium*, pages 207–225, August 2002.

[28] Niels Provos. Preventing Privilege Escalation. Technical Report CITI 02-2, University of Michigan, August 2002.

[29] Thomas Ptacek and Timothy Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Secure Networks Whitepaper, August 1998.

[30] Jerome H. Saltzer. Protection and the Control of Information in Multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[31] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of the 8th Usenix Security Symposium*, pages 123–139, August 1999.

[32] David Wagner and Paolo Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, November 2002.

[33] David A. Wagner. Janus: an Approach for Confinement of Untrusted Applications. Technical Report CSD-99-1056, 12, 1999.

[34] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Shermann, and Karen A. Oostendorp. Confining Root Programs with Domain and Type Enforcement (DTE). In *Proceedings of the 6th Usenix Security Symposium*, July 1996.