Secure and Portable Confinement of Untrusted Programs

Guido van 't Noordende, Ádám Balogh*, Rutger Hofman, Frances Brazier, Andrew Tanenbaum
Department of Computer Science, Faculty of Sciences
Vrije Universiteit, Amsterdam, The Netherlands
*Department of Algorithms and their Applications
Eötvös Loránd University, Budapest, Hungary
{guido,rutger,frances,ast}@cs.vu.nl, bas@elte.hu

Abstract

We present a secure system call interception based system that allows users to protect their resources from untrusted programs by jailing them. A jailed program may, by default, only access the directory in which it was started up and its subdirectories. A user-editable policy file is provided using which a user can specify which additional (read-only or read-write) directories or network addresses the jailed program may access. The jailing model is specifically aimed at running previously unknown and completely untrusted programs in a secure way, such that the user’s resources (e.g., files) are protected from inspection or tampering by these programs. An important contribution of this system is that it defines a clear jailing model which ensures that jailed programs can make use of most of the UNIX system call API inside their jail, but that programs running in different jails cannot leak information to each other through, for example, shared directories on the file system or (normally unprotected) IPC channels.

The jailing system is implemented using a portable jailing infrastructure which runs in user space and can be ported to most UNIX systems without modifications to the operating system or system administrator intervention. We present a novel way for implementing the system such that it avoids certain race conditions which rendered earlier attempts to build jailing systems in user space insecure. This solution can be implemented even if the operating system only provides rudimentary (debugging) support for tracing the system calls that are made by a program. This paper describes and provides performance measurements of two different implementations that were built for the Linux operating system.

1. INTRODUCTION

Operating systems currently do not provide sufficiently fine-grained protection mechanisms for protecting a user against the programs that he or she executes. The UNIX protection model is based on a discretionary access-control model, where all programs executed by a user inherit the user’s permissions with regard to accessing resources, such as files. However, not all programs can be trusted. For example, programs can be downloaded from the Internet or obtained in some other way, and even normally trusted programs can contain backdoors or trojan horses that can be used by an intruder to, for example, send information over the network, or abuse vulnerabilities that allow the intruder to gain root access. Therefore, the most common use of jailing systems is for protecting systems against untrusted (downloaded) executable or interpreted code [1, 2, 3, 4]. All jailing systems feature some kind of policy language that allows a user to express what an application is allowed to do, and the jailing system should make sure that the application does not gain access to resources not allowed by this policy [1]. However, policy generation is a nontrivial task, as it requires detailed knowledge from the user about many system calls, and awareness of all possible side-effects of allowing a particular system call. As security of the system depends on the policy used, creating a policy that is sufficiently secure for every (unknown) program is a very hard if not undoable task. The jailing system itself should therefore provide sufficient structure and built-in support to make sure that a user cannot easily define a policy that provides jailed programs with, for example, a way to leak information or bypass the confinement rules intended by the user.

The jailing system presented in this paper distinguishes itself from earlier systems in that it provides a clear jailing model, designed with confinement in mind. The system is based on system call interception to allow programs to be jailed irrespective of the language in which they are written. Our jailing system allows for confining previously unknown, completely untrusted programs such that these programs cannot access resources, unless the user has explicitly granted access to them. The default jailing model is sufficiently strong to allow for enforcement of information flow control policies, by preventing unchecked communication between a program in one jail to a program in another jail.

The jailing model distinguishes system calls made by a program within the jail from actions which influence the outside world, such as writing to a file outside the jail. Within a jail, an application is allowed access to most calls of the UNIX API; interactions with the outside world are subject to a user-defined policy. The policy used by our jailer is high-level and simple. It can be used to define a set of files or directories which are read-only or read-write accessible to the jailed program, and a set of hostnames or IPC channels that the jailed program may connect to or accept connections from. A jailed program cannot escape its jail, either via file access, or by directly communicating with another (untrusted) process outside the jail, except when explicitly allowed by the jail’s policy. We implemented our jailer using a novel, portable approach that allows the jailing system to run securely on any UNIX system without modification to the operating system’s code.

This paper is organized as follows. First, we describe the application domain and design goals of our jailer, and discuss a number of practical issues that make system call interception systems difficult to implement securely. Next, we discuss the jailing model and policy in some detail. Then we discuss how we implemented our

Copyright 2002 ACM X-XXXXX-XX-X/XX/XX ...$5.00.
user-level system and discuss aspects regarding portability. Finally, we provide performance measurements and conclude.

2. APPLICATION DOMAINS OF JAILING SYSTEMS

The application domain that our jailing system is initially targeted for is executing untrusted mobile agents in a secure way. Mobile agents are autonomous software programs which can migrate from machine to machine in order to achieve some goal on behalf of their users [6, 7]. By nature, agents consist of unknown code. An important advantage of using a mobile agent is that its author can customize the code to suit his or her needs, depending on the application in which the agent operates. A particular advantage of mobile agents is that the code can migrate to the machine where the data resides, avoiding the transfer of potentially large amounts of data to a client machine for local inspection (most of which may be discarded afterwards). Moreover, this data may be sensitive and a data owner may not want all its data to be shipped to a client’s machine for inspection [6]. Multiple agents may be executed at the same time, even by the same user, so special care has to be taken to ensure that cooperating agents on the same machine cannot bypass an agent system’s control mechanisms.

In Mansion, a mobile agent system which we designed [6], mobile agents are managed by a specially written middleware system, which receives agents and starts each of them up in a different jail. Agents can communicate with each other and access resources, but only by making RPC calls over carefully controlled IPC channels set up between the agent and the (trusted) middleware system, or via read-only access to shared data (e.g., files) to which access is allowed by the jail’s policy file. In Mansion, agents can run in a special Confined Room. A room is a logical container of objects and agents. An agent in a confined room can communicate with other agents and invoke objects in its confined room, but it is not allowed to interact with agents or objects outside this room. Before an agent leaves the confined room, it can give the information it wants to take out to a special guardian agent in the room, which determines if this information may be taken out of the room or not, depending on application-specific rules defined by the room’s owner. When an agent leaves the confined room, it is restarted without any recollection of what it did or saw in the room, and it or its owner should contact the room’s guardian agent to collect the information that the agent was allowed to export from the room. As multiple agents (in different rooms) may run simultaneously on the same machine, the jailing system that is used to execute the agents in must take care that the agents cannot bypass the confinement rules imposed by the application. Our jailing system was defined with this requirement in mind.

Although our system is initially targeted at executing (Mansion) mobile agents, it is designed and implemented as a stand-alone jailing system that can be used to securely execute any program. A default policy is provided which allows users to run most programs directly in a secure way. If required, policy modification (for example, to adjust it according to the system’s particular directory structure) is straightforward. The jailing model by default provides strong confinement of the jail program, yet it allows the jailed program to use most of the system call API. This makes it possible to run most modern programs in a jail directly without modification. We have tested our jailing system with a range of regular programs (see section 12), which ran directly in the jail with no or almost no modification of the default policy.

3. DESIGN GOALS

Although the design of our jailing system was motivated by a mobile agent system, we implemented the jailing system as a stand-alone program that can be used for confining any untrusted application.

The main design requirements for our jailing system are:

- The system should be simple to use. In particular, it should be simple to specify a sufficiently secure policy to confine an untrusted application, and preferably it should be possible to run an application with a default policy that ensures that basic security constraints are enforced.
- The system should be portable to any UNIX system, and runnable by any user without requiring root privileges. As we use the system in a distributed mobile agent system, we would not want to be restricted to using a single operating system only, and we do not want to require system administrator intervention, e.g., to patch or reconfigure the operating system before regular users can use the system.
- We want to impose as few constraints as possible on the applications: they could be as wide-ranging as shell-scripts spawning many processes that communicate via pipes or other forms of IPC, interpreted programs, heavily multithreaded programs such as a JVM, and more. In short, we would like most programs to run out of the box. Nevertheless, the system should be completely secure, even in view of race conditions or coordinated attacks.
- Strong confinement of processes should be possible, such that application-specific information flow policies can be enforced even in view of multiple programs that cooperate to bypass these rules.

These requirements are not trivial to meet for a secure system call interception system. For example, the second requirement implies that the system should be runnable using standard operating system primitives, such as standard ptrace() debugging support. How this requirement is addressed in a secure and portable way is discussed in section 8 and later. The first requirement implies that specifying a jailing policy should be a straightforward task. To address this aspect, we defined a simple, high-level policy language which can easily be adapted to, in particular, the local file system’s directory structure. In addition, runtime parameters can be specified on the jailer program’s command line (e.g., by a user or a program that executes the jail) such that the jailed program can, for example, connect to one or more local TCP ports to access a service or other sort of program, such as the Mansion middleware program. The way in which jailing policies and runtime arguments are used is discussed in section 7.

4. OVERVIEW OF TERMINOLOGY AND TECHNOLOGY

The terminology used in this paper is the following:

- The jailer is a trusted process that monitors an untrusted application and enforces a policy on the user’s behalf.
- A prisoner is an untrusted application that is being monitored by a jailer and must [and is forced to] adhere to a predefined jailing policy.
- The tracer is the interface offered by the operating system for debugging / tracing an application. Every major UNIX system to date provides one or more tracing interfaces, such as ptrace() or System-V’s /proc interface.
The jailer (step 2), the jailer has to make a decision on whether to suspend the invoking thread and reflects the system call to the jailer (step 3), which results in the system call being continued or an error being returned to the prisoner. Step 4 and 5 repeat step 2 and 3 after the system call has been made, so that the jailer can inspect the result of the system call before returning control to the prisoner.

The basic idea of system call interception is demonstrated in figure 1. Most if not all current UNIX systems provide some form of debugging support that allows for catching and inspecting the system calls that an application makes, allowing for inspection of the system call’s arguments. The primary example, which is rudimentary but still often used (e.g., by gdb), is the ptrace() system call. Ptrace is available on many UNIX systems, even when more advanced system call tracing mechanisms are available as well. We will assume ptrace() as the underlying system call tracing interface, as it is the most primitive tracing system currently available and demonstrates the minimal requirements of a user-level jailing system.

Ptrace is a system call that allows a parent to monitor its child’s behaviour. When a traced process makes a system call, this call is automatically trapped by the operating system and reflected to the parent process, which can then inspect the child’s current register state and system call arguments. Based on this, the jailer can decide whether to let the system call proceed or whether it should return an error without being executed. Ptrace allows the jailer to change the value of registers that contain the system call’s arguments, before letting the kernel execute the call. Ptrace catches every system call just before it is executed by the kernel, and right after executing it. Only after the jailer agrees to let the process continue on both the pre and post system call event, the prisoner thread is resumed. More advanced tracing systems (such as /proc) allow for specifying the specific system calls which are to be reflected to the jailer. However, ptrace() does not allow for this.

5. THREATS AND VULNERABILITIES IN SYSTEM CALL-INTERCEPTION BASED JAILING SYSTEMS

Figure 1 illustrates a significant problem in all system call interception based jailing systems that support multithreaded applications or processes that use shared memory. When the operating system suspends the invoking thread and reflects the system call to the jailer (step 2), the jailer has to make a decision on whether to allow the system call based on its arguments. However, between the time that an invocation was made (step 1/2) and the decision has been passed back to the operating system (step 3), a different thread of the prisoner (or a process which has access to the prisoner’s address space) could have modified the argument in the original thread’s address space. In this case, the system call would end up using the modified system call argument rather than the system call argument as used by the jailer for checking the policy. This race condition is often referred to as a Time of Check to Time of Use (TOCTOU) race, and it is a realistic threat for all jailing systems that intend to support multithreaded programs or programs that use shared memory. This threat applies to system calls that take a file name as an argument, but also to a connect call that takes an IP address as an argument, for example.

Several solutions for the shared memory TOCTOU race have been proposed for existing system call interception based jailing systems [8, 4, 1, 2]. The most secure approach among current systems is to let the kernel create a safe, normalized, copy of the arguments before reflecting that to a user-level policy enforcement module [1], to make sure that the kernel will use the same arguments as those passed to the policy module when it executes the call.

The race condition is harder to solve for systems that rely on existing tracing mechanisms such as ptrace() or /proc, which provide no protection of the arguments of a system call at the time of inspection. The approach closest to solving the shared memory race outlined above is described in [3]. This solution is based on relocating a system call’s argument to a random location on the caller’s stack before checking it, so that another thread in the child’s address space is unlikely to be able to find and replace this argument. However, it is certainly not impossible for another thread to find such a relocated argument and replace it, given the speed of current hardware.

Other jailing systems simply completely disallow thread creation, or suspend all threads of a jailed process while a system call is being evaluated [9]. Both approaches significantly limit the applicability of such systems for executing modern thread-based applications.

Certain file system race conditions have also been documented for system call interception systems [8] and for operating systems in general [10]. These race conditions are again caused by a lack of atomicity between the argument checking and system call invocation steps. Between the time that a system call’s filename argument is verified by the jailer and the time that the system call is executed in the kernel, another prisoner thread (or even an altogether different process) may have substituted a part of the underlying filesystem path for a symbolic link to a directory outside the paths that are allowed by the policy, without the jailer being able to detect this. Intermittent changes to the current working directory can evoke similar cause race conditions. An excellent overview of these and other vulnerabilities in system call interception based systems is provided in [8].

A novel threat was identified when we designed the Mansion mobile agent system [6]. Mansion provides a simple but effective mandatory access control model for controlling data flow. Agents in Mansion run in separate jails executed by the same user. To prevent information flow from a prisoner in one jail to a prisoner in another jail, it is important that a jail does not allow its prisoners to appear to be different.
to directly or indirectly communicate with other processes outside the jail without explicit permission. One way in which information can escape from a jail is via file system access, as outlined above. Another way using which prisoners can escape their confinement, is when they pre-agree on certain IPC tokens prior to being started up in a jail. IPC tokens are used for identifying shared memory (shmem) segments, message queues, mutexes and semaphores in the kernel. IPC tokens have user permissions to ward off other users on the same machine, but processes executed by the same user (in different jails) can bind to a shared IPC channel using a 32-bit token. A process associates the token with the IPC object itself when it creates it. When the jail system does not take special measures, a prisoner in one jail can bind to an IPC channel created in another jail or by another user using a pre-agreed token. Mobile agents can pre-agree on a token (hardwired in their code or initialization data) prior to migrating to a particular machine. Using this token, an agent executed in one jail can create a new IPC channel and an agent in another jail can bind to this channel using the pre-agreed token. Using this IPC channel, an agent in one jail can pass information to an agent in another jail, thus potentially bypassing information flow control rules imposed by the application.

A general weakness of most existing jailing systems is that the only way they can handle system calls is to either always allow or deny them altogether, or to conditionally allow or deny the system call based on its argument using a user-provided or user-generated policy file. Conditional decisions are made by, for example, comparing a filename with a set of allowed filenames in the policy file. However, it is not possible for a user to specify allowed arguments for conditional allow/deny decisions for every system call of the UNIX API. In particular, some system calls (such as IPC calls or the kill system call used for sending signals) take arguments which are determined at runtime by the jailed program. For this reason, the jailing system itself should deal with the issue of conditionally allowing or denying these system calls according to a built-in model if it wants to allow these system calls in such a way that the prisoner cannot escape its confinement. Current systems lack such a model. Therefore, these systems cannot effectively enforce confinement rules, except by completely denying usage of, for example, IPC channels or signals altogether, which would break many applications. For example, Java Virtual Machines make heavy use of signals internally, so it will not run in a jail which denies the JVM access to the kill system call used for sending signals. Our system provides a clear model for handling these issues in such a way that most system calls can be readily used by jailed programs, while still preventing that these processes can escape their jail’s confinement rules.

6. THE JAILING MODEL

To address the requirements and threats outlined above, the jailing model distinguishes an application’s allowed actions within a jail from the actions it can take that influence the world outside the jail. Within a jail, the jailer allows almost the full UNIX API, including IPC mechanisms such as shared memory (but no root privileged calls) to be used by all processes within the jail. The jailing system keeps track of all IPC channels created within the jail to make sure that a prisoner cannot escape the jail by, for example, using a pre-agreed IPC token. We solve this particular issue by keeping track of the IPC tokens used in one jail, and making sure that prisoners cannot bind to an IPC channel that was not created in this jail. We also make sure that a prisoner cannot create or write to an arbitrary file outside the jail, or set up socket connections to arbitrary processes outside the jail. Jailed programs can send signals, but only to processes running in the same jail.

Every jail provides its prisoners with a private directory structure in which they can read and write files. This directory is protected from other users using standard UNIX protection mechanisms. A jailed process is by default started up in a (normally empty) scratch directory. Different jails are not allowed to have read/write access to common directories shared with other jails. This is an important constraint, as it allows information flow between untrusted processes running in different jails to be controlled, an important design criterion for our jailing system. Typically, a jail’s scratch directory is created by a script. In this initial working directory, prisoners can write/read files and create and access subdirectories. An important assumption of the jailer is that a user will jail all untrusted programs, to make sure that untrusted programs cannot read and write in a jail’s private directory structure, for example. Every unjailed process run by the same user that started up a jail, is able to access the files and processes within this jail directly. The user can also send signals to processes in the jail, for example to kill a prisoner.

A jail always starts with a single program, but this program may (modulo policy) fork or execve() other programs or create new threads. Child processes are executed in the same jail under the same policy as their parent. The jailing process hierarchy is shown in figure 2. According to the jailing model, a jailed process and its children are free to communicate with other processes in the jail using sockets or IPC channels (including signals), or by writing

---

4 The jailing model defines that prisoners may use IPC channels for inter-process communication within a jail.

5 We are helped by the fact that the jailer can set a flag on every call for creating a new IPC channel, which indicates that the IPC channel indicated by this token must not yet exist (i.e., was not already created outside the jail). In addition, we set the user permissions to 700 at creation time, to prevent processes created by another user from binding to a prisoner’s IPC channel.
and reading files in the jail’s directory structure, but they are not allowed to communicate with processes outside the jail using sockets or IPC, or write to or read from files outside the jail without explicit policy permission.

By keeping track of, among other things, which IPC tokens were created in a jail, and of which processes reside in a jail, the jailer can effectively ward off attacks using pre-agreed IPC tokens, signals, and other globally usable communication mechanisms, that would otherwise allow prisoners to easily channel information to processes outside their jail. Avoiding this would have been very hard if not impossible without a strict jailing model.

7. THE JAILING POLICY

The jailing model simplifies the procedure for specifying a secure policy by providing default rules which are aimed at providing strict confinement of the jailed programs. This simplifies policy specification, as the policy mainly deals with interactions with the outside world, such as reading or writing files in specific directories outside the jail. As the jailer’s default rules handle most system calls of a jailed program appropriately, the policy can remain high-level and simple, and because of that, the user is less likely to make errors during the policy specification process.

Each jail is associated with a single policy, specified at jail startup as a user-editable policy file. Every jail potentially has a different policy, although in most cases the default policy will suffice for most applications. The policy file contains some variables which are filled in by the jailer program at runtime, such as the jail’s private working directory, as specified on the commandline. This keeps the policy file independent of a particular jail’s settings.

The policy file allows for specification of specific directories available for reading from, and for change-rooting the program’s visible directory structure. Every policy must contain an explicit list of read-only and read-write accessible directories, called RPATH and RWPATH. Directories not in either of these lists are not accessible to prisoners. As an example, a prisoner will normally be allowed read-only access to certain other directories (e.g., /lib or /usr/bin), so that it can load in standard libraries or execute regular programs (e.g., a perl interpreter or a JVM). As these programs are executed by a jailed program, they are run in the same jail as the process that executed them.

Many programs (e.g., the Java Virtual Machine) require write access to /tmp. However, allowing every jail to use the /tmp directory for writing / reading files, would allow jailed processes of the same user but in different jails to communicate easily through files in this shared directory. To avoid this, we transparently map /tmp to a directory used exclusively by this jail using a virtual change-root system built into the jailer. Note that our virtual change-root system does not use the UNIX chroot call: virtual change-rooting is implemented in the jailer using a simple path-prefix substitution algorithm. Change-rooting adheres to (default) CHDIR rules specified in the policy file. For example, when a prisoner specifies a path starting with /, this can be substituted by the jailer to /home/guido/jails/jailroot/. However, our system is more flexible than a traditional chroot environment, as it allows multiple CHDIR paths to be applied simultaneously. For example, in addition to the above /change-root, /usr can be explicitly substituted back to the actual /usr directory, to override the /change-root in

The change-rooted /tmp directory could be a subdirectory of the real /tmp directory. Another approach is to simply change-root the agent’s initial working directory (the jailering directory) to look like the root (/) directory, and to provide an empty /tmp directory in it at prison startup time. All change-root directories, including /tmp, are defined in the jailer’s policy file, and can be changed if required.

case the prisoner opens a file relative to the /usr directory. This avoids that regularly used files or libraries have to be copied into the jail’s change-rooted directory structure before it can be used. Thus, the jailer’s change-root system more resembles a virtual mountpoint structure than a traditional, less flexible, chroot environment.

In addition to accessible directories or files, the policy allows a user to specify network addresses that the program is allowed to connect to or from which connections may be accepted. In addition to the policy, certain “IPC escapes” can be defined on the jailer program’s command line, making it possible to change some parameters for the jail at runtime without changing the policy file, for example if the program needs access to a DNS resolver or an X server.

The policy language is high-level and simple, such that a user can edit it in a straightforward way, for example to adapt it to the directory structure used on a specific UNIX system. Once a basic policy is defined which allows the prisoners read-only access to the most important system directories (for example those which contain shared libraries), most programs can be readily executed using this basic policy file. In some cases, one or more commandline IPC escapes (e.g., to allow it to connect to a process listening on a specific port on the local or on a remote machine) have to be specified to allow a prisoner to do its job inside its jail. When a prisoner needs to write files, for example containing results, it can do so in its (private) jail directory or in its virtual change-rooted /tmp directory. The user (or invoking program or script) can access or copy these files from the jailering directory while the prisoner is doing its work, or after the prisoner has done its job and it and its jailer program have exited. An example policy is shown in Appendix A. This policy is for somewhat simplified for brevity, but it is in principle identical to the policy we used for running the benchmark tests described in section 12.

8. WINNING THE SHARED MEMORY RACE

The most important implementation issue to solve is how to secure the arguments of a system call in view of the shared memory race conditions outlined in section 5. We describe the Linux solu-
The basic idea implemented in our system is shown in figure 3. When a new prisoner process is executed, it is provided with a preloaded region of memory which is shared between the prisoner and the jailer. The prisoner has only read-only access to this region; the jailer can write into it. We call this shared memory region Shared Read-Only memory, or ShRO in short. Once a system call is made, the system call is reflected to the jailer using standard techniques, as explained in section 4. After this, the jailer fetches the arguments from the kernel using a standard (ptrace) mechanism. An argument can be a filename or an IP address, for example - depending on the system call. The jailer makes a safe copy of the arguments pointed to in ShRO and adjusts the argument values (i.e., registers pointing to the arguments) in the Linux kernel to point to these copied arguments. Then the jailer does a policy check using the safe copy of the arguments (e.g., filename or IP address) in ShRO, and informs the kernel of its decision to let the system call proceed or not. If the system call is to proceed, the kernel now uses the updated system call registers, i.e., pointing to the safe copy of the arguments, to execute the system call. No other thread can touch the registers or the safe copy of their arguments. Note that this provides an effective solution to the shared filesystem attacks.

In Linux, the system call arguments are stored in registers at the time of invocation, and immediately copied to the operating system’s process table when the system call is made. These registers are secure from modification as the jailer inspects the system call’s arguments directly from their copy in the operating system’s internal process table. These arguments cannot be changed by another thread of the invoking process, as these threads cannot alter the register copies in the kernel. However, this is completely operating system specific. Many UNIX operating systems simply leave the system call arguments on the stack and the only things which are securely passed to the operating system are the invoking thread’s program counter (i.e., return address) and the stack pointer, which is used by the operating system to fetch the system call number and arguments from the stack. To secure system call arguments when these are on the stack, the jailer copies the stackframe containing the system call arguments to a safe region in ShRO memory, and modifies the stack pointer in the kernel such that it will use the safe (possibly modified) stackframe to execute the system call. After this, the stack pointer is again modified by the jailer to the appropriate location in the prisoner’s address space.

In addition to ShRO, a prisoner is also preloaded with a small (read-only) executable library containing some custom post systemcall processing routines written by us, which are required by the jailer to manage certain calls whose arguments are only known after execution of the system call. This library and its use are discussed in section 9.

### 9. JAILER ARCHITECTURE

The architectural design of our jailer is such that it separates generic functionality (i.e., policy enforcement) from platform-specific functionality. The jailer is split up in two layers, which both share an interface to a standard (portable) memory management module. The jailer architecture is shown in figure 4.

The lowest layer is called the **interception layer**. This layer interfaces with the underlying system call tracing interface, e.g., ptrace() or /proc. We have currently implemented two interception layers, one that uses ptrace() and one that uses a specially-built in-kernel system call interception interface, called kernel jailer in short. Above the interception layer lies the **policy layer**, which takes care of enforcing the jailing policy. Both layers have access to a shared memory manager module, using which the ShRO memory region is managed. It has a high-level interface which allows the interceptor layer and the policy layer to allocate memory for writing system call arguments or stack frames into when required.

The interception layer handles the tracer-specific mechanism for attaching to a prisoner process, and it sets up the shared memory region that the user is allowed to access. The interception layer also normalizes and expands symbolic links in filename arguments, such that it uses an absolute pathname for comparison with the policy, or for applying pathname substitution to implement a possible virtual changeroot environment defined in the policy file. The policy layer informs the interception layer of any changed arguments (e.g., pointing to a new ShRO region where an expanded filename is located).

The policy layer handles copying of the system call arguments to the shared memory region, after which it passes control to the policy layer. The policy makes a decision on whether to allow or deny the system call depending on the arguments. The policy layer also normalizes and expands symbolic links in filename arguments, such that it uses an absolute pathname for comparison with the policy, or for applying pathname substitution to implement a possible virtual changeroot environment defined in the policy file. The policy layer informs the interception layer of any changed arguments (e.g., pointing to a new ShRO region where an expanded filename is located).

The policy layer is unaware of the system call convention used in a specific OS: it will receive the system call arguments in a system-independent format from the interception layer. System call numbers are normalized by the interception layer before calling the policy layer, such that it sees the same value on every operating system. If necessary, the interception layer creates a safe copy of the
arguments in ShRO, to which the policy layer may make modifications if required. After the policy layer agreed with the system call (determined by the return value of the policy layer upcall), the interceptor will let the system call proceed using the possibly modified argument values, possibly after updating some registers in the operating system if required. How these updates are made is hidden from the policy layer inside the interceptor layer, as this is operating system dependent.

The policy layer does not have to worry about fetching the system call arguments, which is to some extent platform specific. The policy layer can simply base its decision on the safe copy of the arguments in ShRO, as provided by the interceptor layer. The policy layer may replace any argument (e.g., by prefix substitution for a virtual changeroot environment) according to the policy. To do so, the policy layer can allocate a piece of ShRO memory, write the substitute argument in this piece of memory, and modify the original arguments to point to the rewritten argument in ShRO memory. The interception layer makes sure that the safe, possibly rewritten, arguments are used to execute the system call, rather than the original arguments in the prisoner’s address space. Note that filename expansion is sufficient to protect against attacks based on changing the current working directory, which may be done by any prisoner thread using the chdir system call.

Not shown in the figure is that the interceptor layer also maintains an action table internally in the case where the tracing mechanism is ptrace(). The policy layer in some cases decides that a system call is always allowed or always denied. It notifies the interceptor layer of this by returning an appropriate returnvalue to the interceptor layer (see section 10). The ptrace() based interceptor layer stores this returnvalue in its internal action table, such that if the same system call is made by a prisoner, it is again allowed or denied, depending on the policy layer’s decision. If denied, the policy layer specifies an errno (returnvalue of the system call) to be reported to the prisoner (implementation is discussed below). If the in-kernel tracing mechanism supports this, it can maintain an action table in the operating system, such that the jailer will not be notified of all system calls. Some system calls can then proceed (or be denied) instantly, which can improve efficiency significantly. For example, read or write calls are almost always safe, as they use a file descriptor that was returned earlier by a successful verified open() or similar system call, e.g., connect or accept⁷. Such an in-kernel action table is maintained in our own kernel jailer implementation, which is described in the following section.

10. IMPLEMENTATION

We implemented a user-level jailer under Linux using the architecture outlined above. We modified the strace [11] source as the basis for our ptrace() interceptor layer. Strace is a program that is normally used for debugging purposes to display the system calls that a process makes. The strace program provides code for interfacing with a number of tracing systems (such as ptrace and /proc)⁸.

⁷PTRACE is rather inefficient at reading data from a prisoner, as it allows only one word to be read at a time. However, most operating systems provide more efficient mechanisms (e.g., Linux has a /proc/mem device) for reading from a child process’s address space. When this is not the case, a poor-man’s /proc/mem device could be implemented by mapping in the address space of a jailed process at fork/exec time using shared memory primitives.

⁸There is an issue with using read or write on an UDP socket, which may have to be checked if the policy specifies a limited set of peers from which datagrams may be received - then all recv() and read() calls on such a socket must be checked by the jailer individually. However, for most policies this will not apply (see section 11), and read and write calls will therefore generally be allowed instantly.

A second interceptor layer has been implemented which makes use of a modified Linux system V in-kernel action table. The kernel jailer is implemented as an extra Linux system call. This kernel jailer automatically jails all children of a traced process. System call events are exchanged between the kernel jailer and the user-level jailer program via messages over a standard (System-V) message queue. The kernel jailer provides a mechanism for fetching arguments (registers and dereferenced arguments such as filenames) from a traced process’s address space using a request/reply mechanism that uses a pair of standard System-V IPC msgsnd/msgrecv system calls. Except for short arguments, this is more efficient than the ptrace mechanism that reads one word per system call from a traced process’s address space. Similar to ptrace, the kernel jailer allows for updating system call argument registers in the Linux kernel. The kernel jailer has been integrated in the user-level jailer code by writing a new interceptor layer. Other than this layer, nothing was changed to the user-level jailing system’s code. In particular, the mechanisms for setting up a preload environment and handling system call arguments in a secure way is identical for both systems.

No policy decisions are hardwired in the kernel: the first time a process does a particular system call, the call is always passed to the policy layer in the user-level process which makes a decision, possibly to always allow the call in future events by this process. As the kernel does not have to implement a mechanism for securing system call arguments, its implementation can be kept minimal. The main part of our in-kernel jailer extension currently consists of 390 lines of code. We will focus on the Linux ptrace() based jailer code for the remainder of this section. We will describe the performance of the kernel jailer, as compared with the ptrace-based jailer, in section 12.

A number of implementation issues had to be resolved in the ptrace() based interceptor layer, which are not unique to our system. For example, ptrace does not always guarantee that forked children of a prisoner are automatically traced. Linux allows setting a flag on the Linux variant of fork, clone(), which determines if the child is also traced. The interceptor layer simply sets this flag for each clone call by a prisoner. For other systems we use a solution described in an earlier paper [3], which consists of placing a breakpoint just after fork(). This gives the jailer the time to attach to the forked process using a ptrace primitive, after which the jailer removes the breakpoint and the child can continue execution.

A new ShRO region must be preloaded at the time that an execute() is done. We do this by modifying the arguments of the execve() call such that the loader forces preload of the ShRO environment. When a process forks, the preloaded shared memory is automatically shared with the child, so no further work is required in this case. If a thread is created, this thread shares the ShRO region with all other threads of this process. The jailer makes sure that this region is safe in view of concurrent access by multiple prisoner threads.

The policy layer is called by the interceptor layer using a very simple interface. The policy layer provides a syscallPre and a syscallPost method. These methods take a pointer to a normalized syscall argument buffer as an argument. This buffer is filled in by the interceptor layer with a (normalized) copy of the arguments (i.e., the system call’s register set in Linux) of the system call. In addition, it contains the (normalized) system call number and the caller’s process-ID and thread ID (if applicable). Based on the nor-
malized system call number, the policy layer can deduce the meaning of the arguments of the system call. If required, the policy layer can request the interceptor layer to dereference a specific system call argument (e.g., a pointer to a filename or a network address) from the prisoner’s address space, which it does using mechanisms provided by ptrace or the kernel jailer. If required, checked and possibly modified arguments can be frozen in ShRO (section 8) on request of the policy layer, with its corresponding argument register changed and updated in the kernel correspondingly by the interceptor layer. This way, fetching arguments and freezing them in ShRO is only done when required, depending on the system call that was made. Whether syscall_pre or syscall_post is called is determined by the interceptor layer dependent on the call that was made, and in some cases on earlier results of calling the policy layer; for example, the syscall_pre method may return a code using which the policy layer indicates that, in future cases, it is only interested in post-system call notification for this particular system call.

Based on the information obtained from the interceptor layer, the policy layer makes a decision on whether it allows or denies the call, possibly after rewriting the argument, e.g., when a CHRDIR directive (see section 7) in the policy applies to a filename argument. Except for the standard allow/deny decision, a policy layer can also return an always_allow or always_deny decision to the interceptor layer. This sets a bit in the action table in the interceptor layer or in the kernel, such that the call can be allowed or denied instantly without bothering the policy layer. If an in-kernel action table is used, always decisions can avoid the overhead of switching to the user-level jailer program for these calls.

When a system call is denied, the policy layer returns this decision and a normalized error code (errno) to the interceptor layer. Ptrace, unfortunately, does not provide a straightforward mechanism for denying a system call [3]. Therefore, the ptrace() based interceptor layer executes a harmless getpid() call instead of the original call, and substitutes this call’s returnvalue for the errorcode specified by the policy layer before resuming the invoking thread. An interceptor layer that uses a more sophisticated tracing system can typically specify an error code and truly deny the call using a tracer primitive.

11. POST-SYSTEM CALL POLICY EVALUATION

The ShRO region provides an efficient and simple security measure for arguments that are specified by a prisoner before making a system call. However, there are a few system calls for which a potentially policy sensitive argument is only known after the system call has been made. This applies in particular to TCP accept() calls and UDP recv( )/recvmsg( )/recvfrom( ) primitives: the peer address is only known to the kernel after accept or recvfrom took place. The problem here is that the result of the call is written into the caller’s address space by the kernel, and between the time that the result (e.g., peeraddr) is returned and the jailer checks this, another thread of the prisoner could have modified the returned value such that it passes the jailer’s policy check. Also note that the operating system has already created the file descriptor as the result of executing the accept() call. For this reason, keeping the invoking thread from resuming until checking is done is not a feasible idea either, as the file descriptor can be easily guessed and used by another thread to send out information even before the jailer has had a chance to check the peer’s address against its policy.

To handle this issue we use an approach first introduced in the Ostia system [4]. The Ostia approach is based on a delegation mechanism, where every sensitive call (such as open() or accept()) is executed by the jailer instead of the prisoner. As the jailer is the process that executes the system call, it can be sure that the prisoner has no possibility of modifying the system call arguments or using the file descriptor before a peer’s address has been verified. Most sensitive system calls return a file descriptor, which the jailer can forward to the prisoner over a UNIX domain socket, after which it is usable by the prisoner in the normal way. To get this working, the authors of the Ostia paper implemented a kernel extension (implemented as a loadable kernel module) to handle transferring the file descriptors from the jailer to the prisoner. This solution is not usable in our system, as it is at odds with our requirement of being able to run on unmodified UNIX systems without system administrator intervention.

In our jailer, post-system call processing is implemented using a trampoline construction. When a post-system call routine has to be invoked by the prisoner after a system call has been made, the jailer sets the return address (program counter) of the invoking prisoner thread to an address of a dispatcher routine in the preloaded executable library. When the jailer tells the kernel to proceed execution of the call, the operating system will resume the calling thread at the specified return address after executing the system call. As a result, the dispatcher routine is run, which calls an appropriate handler routine, which does its job and then returns to the original prisoner’s return address\(^9\). The dispatcher routine is written in assembly to handle certain architecture specific things, such as saving/restoring registers according to the i386 convention. Except for these 20 lines of assembly code, all the jailer code is written in portable C.

The trampoline construction is only invoked for a few calls. Two calls are not security critical, but important for consistency. The readdir() and getcwd() call’s returned directory names need to be modified by the jailer before return to the calling thread, such that the returned directory names are consistent with the virtual change-root environment applied to the prisoner. The accept() call is actually invoked in the jailer, which has to check the peer’s address after the call was made. After that, the jailer passes the file descriptor to the prisoner using the technique described above. Recvmsg() on an UDP socket requires similar handling in the jailer when the policy specifies a limited set of peers that may send datagrams to the prisoner. In this case, the jailer first PEEKs the socket\(^10\) to check the sender’s address, and if not allowed discards the data gram by reading it from the socket and discarding it before reading the next datagram. If a message is found whose sender is allowed, the prisoner’s thread is resumed, so it can read the datagram from the socket. The prisoner is unaware of any failed connect() calls or discarded datagrams. Listen must also be post-processed, such that the jailer obtains a copy of the allocated file descriptor (via a UNIX domain socket) so that it can later do an accept() using this file descriptor. The jailer has to do some bookkeeping in order to correctly handle read and write (or recv and send) calls on socket descriptors. In particular, it has to keep track of the type of a socket, i.e., whether it is a TCP socket or an UDP socket to be able to handle calls on these sockets correctly.

Note that the delegation approach as outlined above is only re-

\(^9\)Note that in case that the jailer executes the delegated call, the original call of the prisoner must be aborted to make sure that the prisoner does not do an actual (unverified) accept. In the ptrace()-based jailer, this is achieved by replacing the original system call number for that of the harmless getpid() call. The post-syscall processing mechanism makes sure that these mechanisms are completely transparent to the prisoner.

\(^10\)PEEK leaves the datagram in the socket’s queue so it can be read again.
required when the policy specifies that only some senders are allowed to connect or send datagrams to a prisoner. Such conditional policy statements are probably rare. Normally, a server process will be allowed to accept connections from any party, and processes that are only allowed to communicate with specific parties will typically use connect only and will simply not be allowed to accept. The delegation technique outlined above imposes some extra overhead due to the system calls required for file descriptor passing, and is therefore bypassed in case of a policy that does not specify conditional peer addresses for TCP accept or UCP recvfrom calls.

12. PERFORMANCE

In this section we show performance results for both jailing systems that we implemented, the ptrace()-based jailer and the kernel jailer. We use micro-benchmarks to investigate and analyse the overhead of some representative system calls, and present the performance of three applications whose performance is dominated by system calls, so they represent a "hard case" for jailers.

All experiments were conducted on an Athlon 64 3200+ with a Linux 2.6.13.2 kernel, compiled with our kernel jailing patches. We present benchmark measurements for the ptrace() jailer and the kernel jailer. For comparison we present the same benchmarks run outside the jail, and run under control of our space policy engine. Its performance is therefore comparable to the unjailed case.

Table 1: Microbenchmarks of selected system calls. Time is in \(\mu s\) per system call

<table>
<thead>
<tr>
<th>Syscall</th>
<th>Unjailed</th>
<th>Ptrace jail</th>
<th>Ptrace</th>
<th>Kernel jail</th>
<th>calls in loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>geteuid</td>
<td>0.07</td>
<td>5.1</td>
<td>6.2</td>
<td>0.08</td>
<td>100000</td>
</tr>
<tr>
<td>stat</td>
<td>0.85</td>
<td>7.2</td>
<td>14.0</td>
<td>14.3</td>
<td>10000</td>
</tr>
<tr>
<td>getcwd</td>
<td>0.51</td>
<td>6.4</td>
<td>12.7</td>
<td>9.5</td>
<td>10000</td>
</tr>
<tr>
<td>accept</td>
<td>91</td>
<td>169</td>
<td>537</td>
<td>466</td>
<td>1000</td>
</tr>
<tr>
<td>connect</td>
<td>98</td>
<td>178</td>
<td>508</td>
<td>466</td>
<td>1000</td>
</tr>
</tbody>
</table>

Table 2: Breakdown of the time spent by the jailer for a stat system call, in microseconds (\(\mu s\), averaged over 10000 runs.)

<table>
<thead>
<tr>
<th>Jailer</th>
<th>ptrace</th>
<th>kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>intercept bookeeping</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>read syscall args</td>
<td>1.02</td>
<td>2.35</td>
</tr>
<tr>
<td>canonicalize args</td>
<td>0.86</td>
<td>0.81</td>
</tr>
<tr>
<td>check pathname</td>
<td>0.52</td>
<td>0.54</td>
</tr>
<tr>
<td>update kernel registers</td>
<td>0.27</td>
<td>0.28</td>
</tr>
<tr>
<td>microtimers, various</td>
<td>0.69</td>
<td>0.54</td>
</tr>
<tr>
<td><strong>total jailer costs</strong></td>
<td><strong>4.46</strong></td>
<td><strong>5.93</strong></td>
</tr>
<tr>
<td>basic tracer overhead</td>
<td>6.35</td>
<td>unknown</td>
</tr>
<tr>
<td>system call</td>
<td>0.85</td>
<td>0.85</td>
</tr>
<tr>
<td>kernel extra</td>
<td>2.34</td>
<td>unknown</td>
</tr>
<tr>
<td><strong>total time</strong></td>
<td><strong>14.0</strong></td>
<td><strong>14.3</strong></td>
</tr>
</tbody>
</table>

Getcwd returns the prisoner's current working directory. It requires a rewrite of the returned directory name in the prisoner's address space using a post-syscall processing routine when the prisoner runs in a change-rooted directory. In this case, only the post-syscall routine is called, but no rewriting is necessary. The ptrace jailer requires seven ptrace system calls to retrieve the 5-byte file name from the prisoner, whereas the kernel jailer requires a msgrcv call. However, the latter calls are each more expensive than a ptrace system call. The contribution to the overhead of various parts of the jailer is analysed below.

Accept and connect show that these calls are relatively expensive even outside a jail. For these benchmarks, a client and a server program were run on the same machine, one in a jail and one free. Each connection setup therefore already requires some process switches between benchmark processes. Accept is done by the jailer, and the resulting descriptor is returned to the prisoner over a Unix domain socket. Connect also requires freezing the argument in ShRO. Both in the ptrace and the kernel jailer, accept causes more overhead than connect, compared to the unjailed case. However, for the ptrace jailer, the difference between accept and connect is larger than with the kernel jailer. We attribute this to a peculiarity of the ptrace implementation, which allows only the main jailer thread to make ptrace calls. In contrast, the kernel jailer allows all threads in the jailer to make controlling jailing system calls. The delegation mechanism (section 11) requires the use of different threads in the jailer. This means, in the ptrace jailer, that the helper thread that is involved in delegation (i.e., when it does accept itself) must wake up the main jailer thread via an

Stat takes a filename "junk" as an argument and returns this file's status. It requires securing of the filename in ShRO and updating the register for this argument in the kernel. The user-level jailer program keeps track of the prisoner's current working directory, and uses this information to convert the relative pathname to an absolute pathname\(^{11}\). The ptrace jailer requires two ptrace system calls to retrieve the 5-byte file name from the prisoner, where the kernel jailer requires a msgrcv and a msgrcv call. However, the latter calls are each more expensive than a ptrace system call. The contribution to the overhead of various parts of the jailer is analysed below.

---

\(^{11}\) Note that there may seem to be a race in that another thread of the prisoner may change the current working directory at the time that the filename is absolutilized, which could be prevented by temporarily blocking the chdir system call until stat has been handled by the jailer. However, this is not necessary, as from the jailer’s perspective the only importance is that the expanded filename is secured and allowed by policy; the expanded, secured and checked copy of the filename in ShRO is used by the stat system call.
expensive operating system signal.

Table 2 shows a breakdown of the various parts of the jailer code for a `stat` system call, measured by nanosecond timers inserted into the jailer code. As we find from the `getuid` call, bookkeeping in the jailer costs 1.1μs. Reading the file name from the prisoner address space is implemented by reading a word at a time with the `ptrace` system call, and this is the largest of the jailer costs; for the kernel jailer, an even more expensive pair of System-V IPC `msgsnd/msgrec` calls is done. Calculation of the canonical path name and checking this against the policy paths is also a considerable contribution. Another `ptrace` or kernel jailer system call is involved in copying the pointer that points to the immutable copy of the file name (in ShRO) into the prisoner’s register set. For the `ptrace` jailer, the time spent in the jailer should equal the difference between a jailed system call and an unjailed, ptraced system call. However, 2.34μs are unexplained. We found that this difference must be attributed to kernel peculiarities: timers indicate that the time the kernel takes to actually perform the system call for the prisoner takes approximately this much longer for a jailed prisoner than for a prisoner that is merely controlled by trace. Apparently, the kernel must perform more bookkeeping when the jailer interferes with the prisoner’s address space or register set. For the kernel jailer, we cannot make a detailed analysis, since there is currently no implementation that traces system calls without invoking the jail.

### 12.2 Macrobenchmarks

To measure the overall performance of the jailing system, we ran three macrobenchmarks which emphasize different aspects of the jailing system. All macrobenchmarks are non-trivial for a jailing system, as they do a relatively large number of system calls compared to the time used for doing computations, as shown by the columns for system time and user time in table 3. Many applications will require fewer system calls than the benchmarks presented here.

The first macrobenchmark is to run a `configure` shell script for the source code tree [11]. This script executes a number of programs which try out availability of required functionality on the operating system. It presents a worst-case scenario for the jail system: `execve` calls imply preloading and setting up a new ShRO region for the new process.

The second macrobenchmark is a build of this jail system itself using make. To find out dependencies and compile accordingly, make and its spawned subprocesses must open many files and generate many new files. This benchmark is dominated less by system call time than the configure script.

The third macrobenchmark is a Java build system (ant) which compiles a large Java source tree. The sources consist of 1005 Java source files of a total length of 181073 lines (5554227 bytes), which are compiled to Java bytecode using the IBM 1.4 Java compiler and virtual machine. Ant is a multithreaded Java program. Considerable time is spent both in compiling the source code and in reading and writing files.

What these benchmarks show is that it is possible to run nontrivial, multitthreaded or multiprocess applications within a jail with reasonable performance. The measured applications make a large number of system calls. Despite that, the overhead imposed by the ptrace-based jailer is no more than 113% for configure or 88% for make. The columns “jailer upcalls” in table 3 shows the number of times that the user-level jailer process is consulted for making a decision in both the ptrace and the kernel jailer. In the ptrace-based jailer, the user-level jailer process is consulted for every system call, whereas the kernel jailer consults this process only for a subset of the system calls. With configure and make, the kernel jailer is capable of deciding the system call verdict immediately from its action table, without dispatching to the jailer process, for about half the number of system calls made by the prisoner. As a result, the jail overhead drops to 75% for configure, and 64% for make for the kernel jailer. A significant part of the jail overhead in these cases, although more so for configure than for make, is caused by the expensive `execve` call.

The Ant Java build system (a nontrivial Java program) incurs significantly less ptrace jailer overhead (47%) than configure and make; most of the overhead in this benchmark is caused by ptrace overhead. This is due to the fact that the system calls made by Ant are generally less expensive to handle than the system calls (such as `execve`) made by configure and make. When using the kernel jailer, the user-level jailing program is only consulted for one tenth of all system calls. The majority of system calls that is immediately allowed is for manipulation of thread signal masks, which Java appears to do very frequently. As most of the jail overhead is incurred by switching to the jailer process, the kernel jailer provides significant performance gain compared to the ptrace jailer. In conformance with the relative time spent in user mode and system mode, the total overhead for jail Ant is much smaller than for the other two benchmarks, in both jail systems.

For the many applications that spend the majority of their time in user mode, we expect that performance will be close to ptrace()-performance for a ptrace()-based jail, and better for a kernel jail. Indeed, we verified with some applications (e.g., gzip of large files, results not shown) that jail overhead drops to nearly zero for both jailers if the application spends only a tiny fraction of its time in system mode.

### 13. RELATED WORK

There exist a number of different designs that address system-call interception based jailing of untrusted programs [1, 2, 3, 4, 12]. A number of these systems [1, 4, 12, 13] depend on specially implemented modifications to the operating system to function. Jail- ing systems which require changes to the operating system have obvious deployment drawbacks, as few system administrators are willing to modify their operating system kernel, if this is possible at all. Also, changes to the operating system can lead to the intro-
duction of vulnerabilities in the operating system’s code. As operating systems are highly critical pieces of software, this is a significant risk, especially when the in-kernel jailing system is complex. Systrace [1] is a notable exception in that it has reached significant deployment: systrace is part of a number of open-source BSD UNIX systems. Systrace requires manual policy generation for every program. It is primarily aimed at generating policies for known programs, such that these cannot exceed their normally required permissions, e.g., after an intrusion took place. However, like most jailing systems, systrace cannot deal effectively with confinement issues due to the lack of a jailing model that deals with runtime-determined system call arguments such as IPC tokens, as explained in section 5. This makes the systrace system less suitable for confining completely unknown, untrusted programs such as agents in a secure way than our jailing system.

A number of jailing systems [2, 3, 14] were built that ran on unmodified UNIX systems using standard debugging support such as ptrace() or /proc. However, as outlined earlier in this paper, these systems suffered from a number of race conditions that rendered these systems insecure for the majority of modern (e.g., multithreaded) applications [8]. Consh [14] provided a virtualized environment for applications. In consh, an untrusted application’s resources (e.g., file system) are mapped onto local or remote resources, giving the user some control over, for example, the directories that a prisoner can be access. Consh was based on the original Janus [2] /proc based jailer, and as such it suffered from the race conditions that made Janus and other /proc or ptrace) based jailing systems insecure. None of the above described jailing systems provided a model that dealt with confinement and runtime argument based policy decision issues as our system does.

An alternative to system-call level jailing is language-based sandboxing such as provided by, for example, Java or Safe-Tcl [15]. Compared to language-based systems, system call interception based jailing has the important advantage of being language-independent. Also, language-based sandboxing systems are generally much more complex than system call interception based systems, as these generally require monitoring of a much larger and generally less stable set of primitives than the set of available system calls, and getting a language’s security model right is far from easy [5]. As a result of this, more bugs can find their way in the language’s security enforcement code [16]. System call interception based jailing systems is independent of, and can effectively safeguard the user from vulnerabilities in, any language’s security enforcement mechanism.

Various operating system level techniques have been proposed to achieve better security, flexibility, or software fault isolation for different concurrently executing applications. Notable examples are [17, 18, 10]. In addition, operating system security enhancements were proposed or implemented which allow for enforcement of mandatory access control policies, such as DTE [19] or Security Enhanced Linux [20]. Several of these designs could increase security or software fault isolation. Contrary to these approaches, we aim at supporting secure confinement of applications on top of existing, standard UNIX platforms.

Most jailing systems do not provide a clear jailing model which defines precisely what a jailed process can and cannot do, both inside and outside the jail. Making a distinction between a prisoner’s allowed actions within a jail and outside it, allows is to run complex, multithreaded programs or multiple programs that use IPC inside a jail, while still being able to constrain these program’s interactions with the outside world in such a way that we can securely enforce, in particular, information flow policies. Having a clear jailing model that offers strong confinement by default helps security in practice as it avoids that the jailing environment is escaped using, for example, global IPC tokens or some other mechanism that was either not considered by the jailing system or by the user when a policy was defined.

14. CONCLUSION

The jailing system presented in this paper provides a simple but effective jailing model that allows users to run untrusted programs securely. The solutions for protecting the jailer against a number of shared memory based race-conditions are portable to most UNIX operating system which provide minimal debugging support, for example through the ptrace() system call.

We have shown that we can safely execute programs (also non-trivial, multithreaded programs that make a relatively large number of system calls) in our jailing system. Our jailing code adds unavoidable performance overhead to system calls that require evaluation of its arguments by the user-level jailer program, and thus require a context switch to this jailer program before the system call can proceed. In the case of the ptrace jailer, the jailing costs are comparable to the basic system call tracing (ptrace) overhead for the majority of system calls, although a few system calls (such as execve and open) cause more jailing overhead. Using ptrace, a worst-case program (configure) causes 113% overhead compared to not jailing this program; a less system-call intensive but non-trivial Java program causes 47% overhead when running it in the ptrace jailer. Many programs in practice will probably suffer much less jailing overhead than the benchmarks we chose. By using a kernel jailer with an in-kernel action table, upcalls to the user-level jailing program can be avoided for many system calls (such as read and write) which are unconditionally allowed or denied. Measurements show that using a kernel jailer indeed leads to a significant performance improvement compared to the ptrace based jailer for the programs that we tested.

Our solution is the first that presents an effective and secure solution for alleviating shared memory and file system race conditions, without requiring special in-kernel support for securing system call arguments. This solution is based on compiling sensitive system call arguments to a user-level shared memory region to which the prisoner only has read-only access, before allowing the system call to continue. This solution allows the jailer to be ported to any UNIX system, even if it has only rudimentary system call tracing support such as provided by the ptrace() or /proc system call tracing interface. The obvious advantage of this is that the jailer can be ported to any UNIX system, allowing execution of untrusted programs in a secure way without any changes to the operating system, and without system administrator intervention.

Our jailing model makes a clear distinction between actions within a jail and actions that influence the outside world. Within a single jail, prisoners are granted access to most of the system call API and can freely exchange information, but actions that influence the outside world (such as writing files or setting up socket or IPC connections) are guarded by a simple, user-defined policy. The jailing system provides sufficient confinement of untrusted programs by default, even when information control policies need to be enforced by the application: the jailing model has been designed to prevent untrusted applications from using UNIX system calls for exporting information to parties outside the jail, except when permitted by policy. The jailing system presented in this paper is the first that was designed for confinement of untrusted applications in this way.

15. REFERENCES


Appendix A

An example policy file.

# JAILER POLICY
# The policy file uses similar syntax as
# a shell script, including the way
# that variables are used. Some variables
# are known prior to parsing the policy
# file, such as $USER and the jail’s jailing
# directory, ($JAILDIR) which is specified
# on the command line.

# We can set variables in the prisoner’s
# environment:

ENVIRON=HOME=$JAILDIR
ENVIRON=PATH=$HOME:/usr/bin:/bin/
ENVIRON=JAVA_HOME=/usr/local/j2sdk1.5.0

# A synonym for JAILDIR is JAILID.

JAILID=$JAILDIR

######## FILE SYSTEM PROTECTION ########

# This part of the policy deals with
# protecting parts of the file system.
#

# Changeroot paths:
#
# Syntax:
# CHRDIR=ORIG-PREFIX:REPLACEMENT-PREFIX
#
# CHRDIR, as well as ROPATH and RWPATH,
# can be declared multiple times.

# This policy makes the prisoner believe that
# / is its private directory.
# Note that / must be in RWPATH for / to be
# usable to the agent. An alternative option
# is to place / in ROPATH and to create a
# $JAILDIR/home directory and place that
# in RWPATH.

CHRDIR=/:$JAILDIR

# Normally, the / CHRDIR is not really
# required (as / will not be in ROPATH
# or RWPATH anyway) unless one wants to
# prevent, for example, a prisoner to
# learn about the user’s login name and
# home directory. We defined the / CHDIR
# here for demonstration purposes.

# The /usr and /lib directories are escaped,
# and map onto the real /usr and /lib
# directories.

CHRDIR=/usr:/usr
CHRDIR=/lib:/lib

# /tmp is moved to a private directory in
# /tmp, which is pre-setup by the a jail
# script.

CHRDIR=/tmp:/$USER/jails/$JAILID

# /etc is used commonly by binaries to find
# out about the host environment. To avoid
# prisoners learning too much about the system,
# the user can copy a selected number
# of files to a prisoner’s fake /etc
# directory, to which the prisoner’s /etc
# directory is changerooted.
# An example use is for copying a fake
# /etc/passwd file containing only the
# user’s entry, or a modified version
# of the user’s entry.

CHRDIR=/etc:$JAILDIR/etc/

###
### ROPATHs and RWPATHs
###

# ROPATH and RWPATH automatically
# include subdirectories. Care has to
# be taken if subdirectories must not
# be accessible. Then, the subdirectories
# which are accessible must be
# explicitly included, avoiding inclusion
# of the top-level directory. This is
# not done here for simplicity.
# It is also possible to include a
# filename in this list, as a means to
# avoid recursive accessibility of a
# directory: then only this file
# is accessible, not its directory.
# Note that the policy is only checked
# against absolute, expanded pathnames.

ROPATH=/usr/local/bin:/opt/sfw/bin:\
/lib:/usr/bin:/usr/lib:/usr/lib32:\
/usr/lib64:/usr/local/lib

# Some other (safe) and sometimes
# required paths. Note that a prisoner
# is not allowed to access another
# process’s information through /proc.

ROPATH=/proc/cpuinfo
ROPATH=/proc/filesystems
ROPATH=/proc/meminfo
ROPATH=/proc/mounts
ROPATH=/proc/net/if_inet6
ROPATH=/proc/stat
ROPATH=/proc/version

# X support:
# ROPATH=/usr/X11R6/bin
# ROPATH=/usr/X11R6/lib
# ROPATH=/usr/share/icons
# ROPATH=$HOME/.Xauthority

# Writable paths:
# Note: these are the paths
# as the prisoner sees them,
# i.e., before CHRDIR is
# applied.

RWPATH=$JAILDIR:/tmp:/dev/null

#### IPC / NETWORK PROTECTION ####
#
# This part of the policy deals with
# protecting the network and allowing
# IPC channels to fine-tune the
# default jailing model.
#
# ________________________________

# Named IPC channels (i.e., UNIX
# domain sockets) may only be
# in the jail’s RWPATH.

IPC_STREAM_BIND=$RWPATH # or DENY

# By default, if ALLOW, IPC streams
# (i.e., unix domain sockets) are only
# allowed within the jail, or to specific
# endpoints specified as an IPC escape on
# the commandline. But it is possible to
# specify a set of colon-separated
# filenames that the prisoner may connect
# to (unix domain sockets have a file
# name).

IPC_STREAM_CONNECT=ALLOW
# or DENY (always) or a list
# of IP-addresses / hostnames
# that tcp-sockets are allowed
# from.

INET_STREAM_ACCEPT=DENY
# or ALLOW (always) or a list
# of IP-addresses / hostnames
# that tcp-sockets are allowed
# from.

# In this example, we allow
# the prisoner to send datagrams
# to the local DNS resolver.

INET_DGRAM_RECVFROM=ALLOW;

INET_DGRAM_SENDTO=LOCALHOST/53

# This option defines which
# TCP sockets a prisoner may

INET_STREAM_CONNECT=localhost/53

# connect to. In this case, only
# the local DNS resolver.

INET_STREAM_ACCEPT=DENY
# or ALLOW (always) or a list
# of IP-addresses / hostnames
# that tcp-sockets are allowed
# from.

# In this example, we allow
# the prisoner to send datagrams
# to the local DNS resolver.

INET_DGRAM_SENDTO=LOCALHOST/53