A SECURE JAILING SYSTEM FOR CONFINING UNTRUSTED APPLICATIONS

Guido van 't Noordende, Ádám Balogh*, Rutger Hofman, Frances M. T. Brazier and Andrew S. Tanenbaum
Department of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
{guido,rutger,frances,ast}@cs.vu.nl

**Department of Algorithms and their Applications, Eötvös Loránd University, Budapest, Hungary*
bas@elte.hu

Keywords: System Call Interception, Application Confinement, Jailing.

Abstract: System call interception based jailing is a well-known method for confining (sandboxing) untrusted binary applications. Existing systems that are implemented using standard UNIX debugging mechanisms are rendered insecure by several race conditions. This paper gives an overview of the most important threats to jailing systems, and presents novel mechanisms for implementing jailing securely on standard UNIX systems. We implemented these solutions on Linux, and achieve competitive performance compared to existing jailing systems. Performance results are provided for this implementation, and for an implementation that uses a special-purpose extension to the Linux kernel designed to improve performance of the jailing 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. To safely execute untrusted programs on UNIX systems, system-call interception based jailing systems can be used which protect the system and the user's resources, and which allow a user to configure network addresses with which a jailed program is allowed to communicate. System-call interception based jailing systems (Goldberg et al., 1996; Alexandrov et al., 1999; Jain and Sekar, 2000; Provos, 2003; Garfinkel et al., 2004) are based on a kernel-level tracing mechanism (e.g., `ptrace`) that allows a trusted jailer to intercept all system calls of its child process(es), and accept, deny, or modify arguments of the system calls made by an untrusted process before the kernel proceeds with executing the system call. All jailing systems make use of a policy file which describes which parts of the local file system may be accessed, and which network addresses are reachable by the jailed processes.

A number of jailing systems require modifications to the operating system to function securely. Jailing systems which are implemented in user-mode, using the ptrace() or /proc debugging facilities offered on standard UNIX, also exist. However, existing systems suffer from several race conditions, which allow an attacker to bypass the jailer's control mechanisms (Garfinkel, 2003). This paper describes novel solutions to these race condition problems. These solutions allows complex programs, including multi-threaded programs that make use of IPC mechanisms and signals, to be jailed effectively. The jailing system presented in this paper provides sufficient control to allow for effective confinement of untrusted programs using standard system call tracing mechanisms available on most UNIX systems.

2 TERMINOLOGY

This paper uses the following terminology:

- 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 is forced to adhere

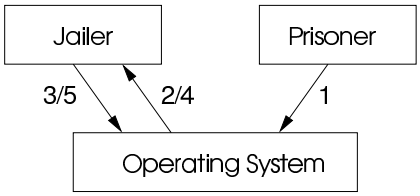

Figure 1: General positioning of a system call interception system showing a jailer and a traced prisoner. When a prisoner makes a system call (step 1), the operating system suspends the invoking thread and reflects the system call to the jailer (step 2). The jailer inspects the system call's arguments and decides if it allows the system call or not. It informs the operating system of its decision (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.

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 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 and its arguments. Linux provides the `ptrace()` system call tracing interface, which is rudimentary but still often used (e.g., by gdb). We used this interface to implement our jailing system. Since ptrace is about the most primitive tracing interface possible, it demonstrates the minimal requirements for implementing a user-level jailing system well.

When a ptraced process makes a system call, this call is trapped by the operating system and reflected to the parent process, which can then inspect the child's register set and system call arguments. Based on this, the parent (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 intercepts 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.

3 THREATS AND VULNERABILITIES

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. These arguments often contain a pointer to a string (e.g., a filename) in the prisoner's address space, which must be derefenced and checked by the jailer. 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 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 argument checked by the jailer. This race condition is called 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 all system calls that take an argument residing in the prisoner's address space, such as a filename or an IP address.

Several solutions for the shared memory TOCTOU race have been proposed (Garfinkel, 2003; Garfinkel et al., 2004; Provos, 2003; Goldberg et al., 1996). The most secure among current approaches is to let the kernel create a safe copy of the arguments before reflecting that to a user-level policy enforcement module (Provos, 2003).

TOCTOU race conditions are 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 is described in (Jain and Sekar, 2000). 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 find and replace this argument. However, it is certainly not impossible for another thread to find such a relocated argument and replace it[1]. Other jailing systems simply completely disallow thread creation, or suspend all threads of a jailed process while a system call is being evaluated[2].

[1] Winning this race is not as far-fetched as it may seem, since the prisoner knows the argument which it originally specified itself and can tell another thread to search for it in its address space.

[2] http://www.subterfugue.org

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 (Garfinkel, 2003). These race conditions are again caused by a lack of atomicity of argument checking and system call invocation. 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 a different process running in the jail) 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[3]. Intermittent changes to the current working directory can evoke similar race conditions.

A weakness of most existing jailing systems is that the only way they can handle system calls automatically is to either always allow or deny them altogether, or to conditionally allow or deny the system call by comparing its argument with, for example, a set of filenames or network addresses in a user-provided policy file. Another solution is to make a callback to the user (Provos, 2003). However, it is a hard task for a user to understand the meaning of the arguments of every system call of the UNIX API and the potential side-effects of allowing the call. For example, some IPC calls or the kill system call take arguments which indicate some kernel object of which a user cannot easily determine whether access to it should be allowed or not. This approach is also not feasible when a large number of jailed processes are running simultaneously on a system.

4 THE JAILING MODEL

To address the issues outlined above, our system provides a clear *jailing model*, which distinguishes an application's allowed actions *within* a jail from actions that influence the world *outside* the jail. A jail always starts with a single program, but this program may (modulo policy) `fork` or `execve` other programs or create new threads[4]. Child processes run under the same policy as their parent (fig. 2).

[3]For example, the prisoner may invoke open with /tmp/user/temp/passwd in the allowed path /tmp/user/temp, and substitute the temp component for a symbolic link to /etc, hoping that the system call will result in /etc/passwd to be opened.

[4]Multiple processes may run in a jail, for example a set of processes executed by a shell script which communicate via pipes.

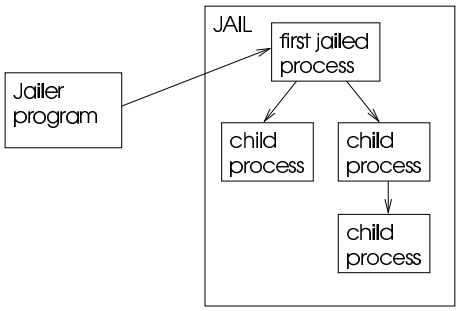

Figure 2: A jail's process hierarchy. The jailer starts the first jailed process in its own jail, and controls this jail by enforcing the jailing model and the jail's policy. A jailed process can create child processes (using fork and execve). These processes are now in the same jail as their parent and execute under the same policy as their parent. All processes within a jail can communicate with each other using UNIX IPC primitives (e.g., using pipes or shared memory primitives like shmem or mmap) or signals, or by writing files in their jailing directory. Communication with processes outside the jail is controlled by the jailer's policy.

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 same jail. Actions which may influence the outside world, such as accessing the file system or connecting to a network endpoint, are only allowed when allowed by the jailer's policy file. The importance of this model is that it allows the jailer for most system calls to automatically determine whether they are allowed or not, even when these system calls take arguments which are determined at runtime (see sec. 3). The jailer keeps track of which communication endpoints or IPC channels were created inside the jail, and only allows access to internal endpoints[5] or IPC channels. The jailer makes sure that a prisoner cannot set up connections to arbitrary processes outside the jail. Jailed programs can send signals, but only to processes running in the same jail.

The jail concept is quite suitable to describe whether a set of processes may communicate with each other or not: processes within a single jail may freely communicate with each other, but communication with the outside world is not allowed unless explicitly permitted by policy. Because the jailer allows most UNIX calls to be used freely within a jail, it allows for execution of the majority of programs, even modern multithreaded and multiprocess applications, within the confinement rules of the jail.

A jailed process is by default started up in a (nor-

[5]The jailer also controls who may connect to a communication endpoint created in a jail, to prevent external processes to initiate a connection to a jailed process.

mally empty) scratch directory, which is read-write accessible and not shared with other jailed processes. Each jail has a simple policy using which the user can define which parts of the local file system a jailed program may access. Examples are /usr/bin and /usr/lib, which contain files that may be used by programs[6]. Any part of the file system may be marked read-only or read-write accessible. The policy also allows for 'mounting', and for change-rooting parts of the local file system into the prisoner's jailing directory. This avoids that parts of the local file system have to be copied to the prisoner's jailing directory. More details on the policy file are given in (van 't Noordende et al., 2006). Except for the policy file, specific escapes from the default confinement model and the policy file can be specified on the jailer's commandline. Examples are IPC escapes, using which a user can specify TCP addresses or UNIX domain sockets which are reachable from a jail. In our experience, it is generally not necessary to override the default policy, as most applications and the libraries they use simply run as expected within a jail[7].

5 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 3.

The basic idea implemented in our system is shown in figure 3. When a new prisoner process is executed, it is provided with a region of shared memory which is shared between the prisoner and the jailer. The prisoner has 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. The shared memory is set up in the prisoner's address space using a library preloading technique. The preload library uses mmap() to read-only map the memory region in the prisoner's address space. The preload library also contains some code which is required for certain post-system call processing tasks which are explained in section 8. Preloading avoids patching the prisoner binary.

[6]The jailing model allows a prisoner to execute programs from, e.g., /bin/, such as sh, perl, sed, or awk, given that policy allows.

[7]Libc turns out to make many calls which are denied, e.g., to files in /etc. However, libc turns out to be quite resilient to denied system calls, and most programs run as expected in a jail.

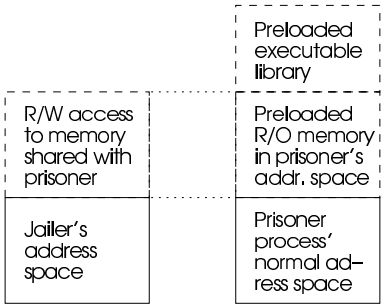

Figure 3: The Shared Read-Only (ShRO) memory solution for avoiding user-level multithreading and shared memory race conditions. The shared memory region is mapped in transparently at prisoner startup time. The prisoner program itself is unmodified and normally not even aware of the preloaded ShRO memory region and library in its address space.

Once a system call is made, the jailer fetches the arguments from the kernel using standard ptrace calls (sec 2). The argument can be a filename or an IP address, for example. The jailer makes a safe copy of the arguments in ShRO and adjusts the argument registers (which are stored in the Linux kernel) to point to the copied arguments. Then the jailer does a policy check using the safe copy of the arguments 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 uses the safe argument copies to execute the system call. No prisoner thread can modify the registers or the safe copy of the arguments in ShRO.

The ShRO solution alone is not sufficient to prevent all race conditions. For example, there is a window of oppertunity for a malicious program to substitute a file in its read-write accessible jailing directory (which has its canonical filename stored safely in ShRO) for a symlink to a file in a directory which is normally not accessible, between the time where the filename is stored in ShRO, and the time that the system call is executed. We prevent this by *serializing* all system calls that modify an object in a read-writeable path in the jail while an open call is in progress. Serializing is implemented by a readers/writers lock around the open and modify/create/write-type system calls. This provides an effective solution to the shared file system TOCTOU race outlined in section 3.

6 JAILER ARCHITECTURE

The architectural design of our jailer separates generic functionality (i.e., policy enforcement) from platform-specific functionality. The jailer is split into two layers. The lowest layer is called the *interception*

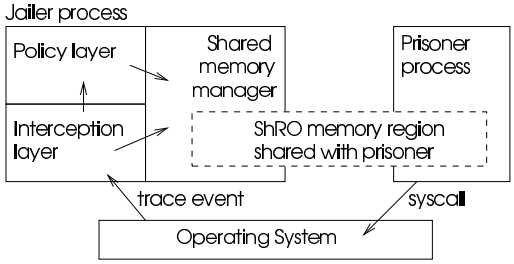

Figure 4: The jailer's internal architecture. The jailer consists of an operating-system specific interception layer and a portable policy layer. Both layers use a portable shared memory manager module which manages the ShRO memory region(s) of the prisoner process(es) in the jail. The interception layer drives the jailer by handling trace events from the operating system and by calling the memory manager and policy layer accordingly. The interception layer also takes care of resuming the prisoner's calling thread after policy evalutation is done, using an OS tracing primitive.

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 in Linux, called *kernel jailer*. Above the interception layer lies the *policy layer*, which enforces the jailing policy. Both layers can access a *shared memory manager* module, which manages the ShRO memory region. It has an interface which allows the interception layer and the policy layer to allocate memory to write system call arguments or stack frames into when required. The jailer architecture is shown in fig. 4.

The interception layer handles the tracer-specific mechanism for attaching to a prisoner process, and it sets up the shared memory between the prisoner and the jailer. There is no time when prisoner code runs uncontrolled. The first prisoner is created by having the jailer fork, and the forked child attaches to the jailer before it exec's the prisoner code. After the attach, the child lets the ELF dynamic loader (ld.so) execute our jailing preload library (sec. 5). The preloaded library sets up the ShRO region, and connects to the jailer using a UNIX domain socket. Preloading takes place in a mini jail environment, in which the prisoner is only allowed to make those system calls necessary to initialize the preload library. The way in which the ShRO region is set up is completely controlled by the jailer, which checks the correctness of the system calls and arguments made during the preload phase. The prisoner's own code (main) is invoked only after preloading and ShRO setup is complete.

The interception layer handles copying of the system call arguments to ShRO, after which it passes control to the policy layer. The policy layer decides whether to allow or deny the system call depending on the arguments. The policy layer also expands symbolic links in filename arguments, such that it uses an absolute pathname for comparison with the policy. If the system call is to be denied, this is specified by returning a negative error code (corresponding to *errno*) from the policy layer. Then, control is returned to the interceptor layer to let the system call proceed. How the arguments are fetched[8] from the prisoner's address space, and how registers are modified are platform-specific issues which are hidden inside the interceptor layer.

7 IMPLEMENTATION

We implemented a user-level jailer under Linux using the architecture outlined above. For the interception layer, we used a modified version of the open-source program strace[9]. Strace is a program that is used to display all system calls that a process makes, e.g. for debugging purposes. Strace provides interfacing with a number of tracing systems (such as ptrace and /proc), and it provides mechanisms for system call normalisation for different platforms. This helped our implementation effort and will probably make porting the jailer to other platforms simpler.

A second interception layer has been implemented which makes use of a modified Linux kernel. The modification is called *kernel jailer*, and consists of an extra tracing system call. The kernel jailer differs from ptrace in that it allows for efficiently fetching system call arguments (registers and dereferenced arguments such as file names) from a prisoner's address space, and in that it caches the policy evaluation results for certain system calls. This prevents repeated upcalls to the user-level jailing process for system calls which are always allowed or always denied. The *kernel jailer* automatically jails all children of a traced process. System call events are exchanged between the kernel jailer and the user-level jailer using a System-V message queue. Except for very short arguments, this is more efficient than ptrace, which can only read one word at a time from a traced process's address space. Similar to ptrace, the kernel

[8]Ptrace 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, which can be used by the interception layer.

[9]http://www.liacs.nl/~wichert/strace/

jailer allows for updating system call argument registers in the Linux kernel. We wrote a new interceptor layer for integrating the kernel jailer; other than that, nothing was changed to the user-level jailing system's code. Because the kernel does not have to implement a mechanism for securing system call arguments as most other jailing systems must do, adding the kernel tracer adds only about 390 lines of code to the Linux kernel.

A number of implementation issues had to be resolved in the ptrace() based interception layer, which are not unique to our system. For example, ptrace does not always guarantee that forked children of a prisoner are automatically traced (which *is* the case in the kernel jailer). Linux allows setting a flag on the Linux variant of fork, clone(), which determines whether the child is also traced. The interception layer simply sets this flag for each clone call by a prisoner. For other systems we can use a solution described in (Jain and Sekar, 2000), 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[10].

A new ShRO region must be preloaded at the time that an execve() 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 creates a thread (i.e., calls clone on Linux), this thread shares the ShRO region with all other threads of this process, so no further work is required; this also applies to `fork`. The jailer makes sure that ShRO is safe in view of concurrent access by multiple prisoner threads within a single jail.

The policy layer is called by the interception layer using a very simple interface. The policy layer provides a *syscall_pre* and a *syscall_post* method, which are called at the system call entry point, and at the time that the system call returns, respectively. Whether syscall_pre or syscall_post is called depends on earlier results of calling the policy layer, depending on the call that was made. For example, the syscall_pre method may return a code which indicates that the policy layer is only interested in post-system call notification for this particular system call on future events.

No policy decisions are hardwired in the interceptor layer: the first time a prisoner makes a particular system call, the call is always passed to the policy layer in the user-level process which makes a decision. The policy layer in some cases decides that a system call is always allowed or always denied. It notifies the interception layer of this by returning an appropriate return value to the interception layer. The ptrace() based interception layer stores this return value in its internal *action table*, such that if the same system call is made again, the system call is immediately allowed or denied. Similarly, the kernel jailer maintains an action table in the kernel, to avoid making upcalls (i.e., expensive context switches) to the user-level jailer process for system calls which are always allowed or denied. This improves efficiency significantly. For example, read or write calls are almost always safe, as they use a file descriptor that was returned earlier by a succesful verified `open` or similar system call, e.g., `connect` or `accept`.

When a system call is denied, the policy layer returns this decision and a normalized error code (*errno*) to the interception layer. Ptrace, unfortunately, does not provide a straightforward mechanism for denying a system call. Therefore, the ptrace() interceptor substitues a harmless getpid() call for the original call, and replaces this call's return value with the errorcode specified by the policy layer.

8 POST-SYSTEM CALL POLICY EVALUATION

The ShRO region provides an efficient 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 known only *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 to make it it pass the jailer's policy check. Worse, in the case of accept the new socket may already be used by another thread of the prisoner, before the jailer even looked at the peeraddr, because the operating system has already created the file descriptor as the result of executing the call[11].

[10] Note that this is a potentially vulnerable solution, as care must be taken that no process in the same jailer can access the part of the prisoner's address space where the breakpoint resides (e.g., using mmap()), and remove it prematurely. This can be mediated by suspending other prisoners during execution of a fork. Fortunately, we can avoid this issue in the Linux implementation.

[11] Keeping the invoking thread from resuming until checking is done is not feasible either, as the file descriptor can be easily guessed and used by another thread.

To handle this issue we use the *delegation* mechanism first introduced in the Ostia system (Garfinkel et al., 2004), 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 using the file descriptor before a peer's address has been verified. Most sensitive system calls return a file descriptor. The jailer can pass this file descriptor to the prisoner over a UNIX domain socket, after which the prisoner can use it in the normal way. Ostia makes use of a loadable kernel module to facilitate file descriptor passing from the jailer to the prisoner. This solution is not usable in our system, as our jailer program must run on unmodified UNIX systems without system administrator intervention.

Post-system call processing is required for a few more system calls. The jailer may need to modify the directory names returned by `readdir()` and `getcwd()` to make them consistent with a modified file system view (sec. 4) defined in the jailer's policy.

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 the address of a dispatcher routine in the preloaded executable library. When the jailer tells the kernel to proceed with execution of the call, the operating system will resume the calling thread at the modified return address after executing the system call. As a result, the dispatcher routine is run, which calls an appropriate handler routine. This is used to modify the string returned by getcwd(), for example. The dispatcher routine then returns to the original prisoner's return address. The dispatcher routine contains 20 lines of assembly language to handle certain architecture specific things, such as saving/restoring registers according to the i386 convention. For the rest, all the jailer code is written in C.

The trampoline construction is also invoked for delegated calls. The `accept()` call is invoked by the jailer, which checks the peer's address after the call was made[12]. After that, the jailer passes the file descriptor to the prisoner. To implement this transparently, the jailer lets the prisoner invoke a harmless getpid() call, followed by a trampoline instruction that reads the file descriptor from the UNIX domain socket. `recvmsg()` on a UDP socket also requires handling in the jailer, but only if the policy specifies a limited set of peers that may send datagrams to the prisoner.

[12]In order for accept() delegation to be implementable, 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()` on this file descriptor.

9 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 microbenchmarks 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 "bad 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 *strace*, modified to intercept all system calls but to generate no tracing output. This latter comparison exactly shows the overhead incurred by the ptrace() mechanism, and the time spent in the jailer can be deduced from it.

We also evaluate the effects of the optimizations offered by the kernel jailer.

9.1 Microbenchmarks

Table 1 presents the performance of microbenchmarks that each invoke one system call in a tight loop. The time presented is the average time for one system call. Measurements are presented for unjailed benchmarks, benchmarks run under the control of *strace* with output disabled, and under control of our ptrace and kernel jailers.

`Geteuid` is a system call that takes no argument and is always allowed by the jailer. This benchmark shows that the impact of ptrace() intervention is considerable in comparison to system call times; the overhead is dominated by context switching between the benchmark process, the kernel and the jailer. This benchmark does not require any argument fetching or rewriting, or nontrivial policy logic by the jailer; accordingly, the extra time added by the jailing part is small, 1μs. Because this system call is always allowed, the kernel jailer immediately allows it without consulting the user-space policy engine. Its performance is therefore comparable to the unjailed case.

`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

Table 1: Microbenchmarks of selected system calls. Time is in μs per system call

Syscall	Unjailed	Ptrace	Ptrace jail	Kernel jail	calls in loop
`geteuid`	0.07	5.1	6.2	0.08	100000
`stat`	0.85	7.2	14.0	14.3	10000
`getcwd`	0.51	6.4	12.7	9.5	10000
`accept`	91	169	537	466	1000
`connect`	98	178	508	466	1000

argument in the kernel. The jailer keeps track of the prisoner's current directory, and uses this to convert the relative pathname to an absolute pathname. The ptrace jailer requires two `ptrace` system calls to retrieve the 5-byte file name from the prisoner, where the kernel jailer requires a `msgsnd` 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.

`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 directory name, whereas the kernel jailer requires only one pair of system calls. This difference makes the kernel jailer substantially faster.

`Accept` and `connect` show that these calls are 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` uses delegation, so the resulting descriptor is returned to the prisoner over a Unix domain socket. `Connect` 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. The delegation mechanism (section 8) requires the use of different threads in the jailer. For the ptrace jailer, the difference between accept and connect is larger than with the kernel jailer. We attribute this difference to threading peculiarities of the ptrace mechanism.

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 `geteuid` 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/msgrcv` calls is done. Calculation of the canonical path name and checking this against the policy paths is also a noticeable 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 remain unexplained. We found that this difference must be attributed to kernel peculiarities.

Table 2: Breakdown of the time spent by the jailer for `stat` (in μs, averaged over 10000 runs.)

jailer	ptrace	kernel
intercept bookkeeping	1.1	1.1
read syscall args	1.02	2.35
canonicalize args	0.86	0.81
check pathname	0.52	0.54
update kernel registers	0.27	0.58
microtimers, various	0.69	0.54
total jailer costs	**4.46**	**5.93**
basic tracer overhead	6.35	unknown
system call	0.85	0.85
kernel extra	2.34	unknown
total time	**14.0**	**14.3**

9.2 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 nontrivial for a jailing system, as they do a large number of system calls compared to the time used for doing computations, as shown in table 3. Many applications will require far fewer system calls than the benchmarks presented here.

The first macrobenchmark is a configure shell script for the strace source code tree. 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 jailing system: `execve` calls imply preloading and setting up a new ShRO region for the new process.

The second macrobenchmark is a build of this jailer 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, con-

Table 3: Results of an strace configure script, a make build, and a build of a large Java source tree using ant. Times in seconds, between brackets the percentages overhead imposed by jailing.

	Unjailed			Ptrace	Ptrace jail		Kernel jail	
	system	user	total	total	total	jailer upcalls	total	jailer upcalls
configure	2.2	3.4	**6.7**	**9.14** (36%)	**14.3** (113%)	367,320	**11.7** (75%)	147,947
make build	3.5	10	**14.6**	**19.0** (30%)	**27.4** (88%)	598,770	**24.0** (64%)	264,815
ant build	1.2	15.5	**16.7**	**22.4** (34%)	**24.5** (47%)	669,557	**18.1** (8%)	62,562

sisting of 1005 Java source files of a total length of 181073 lines (5554227 bytes). These are 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, multithreaded 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%. The columns "jailer upcalls" in table 3 show the number of times that the user-level jailer process is consulted for a policy decision. 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 jailing overhead drops to 75% for make for the kernel jailer. A significant part of the jailing overhead in these cases, although more so for configure than for make, is caused by the expensive `execve` call.

The Ant Java build system incurs significantly less ptrace jailer overhead (47%), and most of this is consumed by ptrace: Ant does relatively few expensive system calls. With 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. This leads to a significant performance gain for the kernel jailer. In conformance with the relative time spent in user mode and system mode, the total overhead for jailing Ant is much smaller than for the other two benchmarks, in both jailing systems.

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

10 RELATED WORK

There exist a number of system call interception based jailing systems which depend on operating system modifications (Garfinkel et al., 2004; Peterson et al., 2002). These solutions have obvious deployment drawbacks. Systrace (Provos, 2003) is an exception in that it is deployed in several open-source BSD UNIX systems. Systrace requires manual policy generation for each program, which limits its usability for confining previously unknown programs automatically. Another class of jailing systems were built that run on unmodified UNIX systems using standard debugging support such as ptrace() or /proc (Goldberg et al., 1996; Jain and Sekar, 2000; Alexandrov et al., 1999; Liang et al., 2003). However, these systems suffered from a number of race conditions that rendered these systems insecure for the majority of modern multithreaded applications (Garfinkel, 2003). Our system is the first that solves these issues securely in user mode.

An alternative to system-call level jailing is language-based sandboxing such as provided by, for example, Java or Safe-Tcl (Ousterhout et al., 1997). Compared to language-based systems, system call interception based jailing has the important advantage of being language-independent. Also, getting a language's security model right is far from easy (Back and Hsieh, 1999; Wallach et al., 1997). Jailing can effectively safeguard users from vulnerabilities in any language's security enforcement mechanism.

Various operating system level techniques or new oprating systems have been proposed to achieve better security, flexibility, or software fault isolation for different concurrently executing applications. Notable examples are (Ghormley et al., 1998; Engler et al., 1995; Mazières and Kaashoek, 1997; Efstathopoulos et al., 2005). Several of these designs could increase security or software fault isolation. Contrary to these approaches, we aim at supporting secure confinement on top of existing, standard UNIX platforms.

Virtual machine approaches such as FreeBSD jail (Kamp and Watson, 2000), VMware or Xen (Dragovic et al., 2003) can also be used to confine applications. However, virtual machines are relatively

heavy-weight, which makes them unsuitable for isolating a very large number of concurrently executing programs individually.

11 CONCLUSION

The jailing system presented in this paper provides a simple but effective jailing model that allows users to run untrusted programs securely. Our solution is the first that presents an effective and secure solution for alleviating shared memory and file system race conditions, without requiring kernel support for securing system call arguments. This solution is based on copying sensitive system call arguments to a user-level shared memory region to which the prisoner has read-only access, before allowing the system call to continue. This solution is in principle portable to any (POSIX compliant) UNIX system, given that it has rudimentary system call tracing support such as the ptrace() or /proc system call tracing interfaces.

By differentiating between resources created inside and outside the jail, our jailing system has a clear model for deciding which system calls to accept or deny, even when a system call takes a runtime-determined kernel object as an argument. Actions that influence the outside world are guarded by a simple, user-defined policy. Policy modification, in particular to adapt the policy to the local system's directory structure, is straightforward, and generally required only once. We have found that we can safely execute many programs (also nontrivial, multithreaded programs that make a large number of system calls, or programs executed from a script) in our jailing system using the default policy. The overhead of our jailing system is acceptable, although this depends on the type of system calls made by a prisoner. Performance is competitive compared to existing user-mode jailing systems.

REFERENCES

Alexandrov, A., Kmiec, P., and Schauser, K. (1999). Consh: Confined execution environment for internet computations. *http://www.cs.ucsb.edu/~berto/papers/99-usenix-consh.ps*.

Back, G. and Hsieh, W. (1999). Drawing the red line in java. *Workshop on Hot Topics in Operating Systems (HotOS VII)*. pp. 116-121.

Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Pratt, I., Warfield, A., Barham, P., and Neugebauer., R. (2003). Xen and the art of virtualization. *Proc. ACM Symposium on Operating Systems Principles (SOSP)*.

Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F., and Morris, R. (2005). Labels and event processes in the asbestos operating system. *Proc. 20th Symposium on Operating Systems Principles (SOSP), Brighton, United Kingdom*.

Engler, D., Kaashoek, M., and O'Toole Jr., J. (1995). Exokernel: an operating system architecture for application-specific resource management. *Proc. Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*. pp. 251-266.

Garfinkel, T. (2003). Traps and pitfalls: Practical problems in system call interception based security tools. *Proc. Symposium on Network and Distributed System Security (NDSS)*. pp. 163-176.

Garfinkel, T., Pfaff, B., and Rosenblum, M. (2004). Ostia: A delegating architecture for secure system call interposition. *Proc. ISOC Network and Distributed System Security Symposium (NDSS)*.

Ghormley, D., Rodrigues, S., Petrou, D., and Anderson., T. (1998). Slic: An extensibility system for commodity operating systems. *USENIX 1998 Annual Technical Conference*.

Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. (1996). A secure environment for untrusted helper applications - confining the wily hacker. *Proc. 6th Usenix Security Symposium*. San Jose, CA, USA.

Jain, K. and Sekar, R. (2000). User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. *ISOC Network and Distributed System Security Symposium (NDSS)*. pp. 19-34.

Kamp, P. and Watson, R. (2000). Jails: Confining the omnipotent root. *Proc. 2nd Intl. SANE Conference*.

Liang, Z., Venkatakrishnan, V., and Sekar, R. (2003). Isolated program execution: An application transparent approach for executing untrusted programs. *19th Annual Computer Security Applications Conference (ACSAC), Las Vegas, Nevada*.

Mazières, D. and Kaashoek, M. (1997). Secure applications need flexible operating systems. *Workshop on Hot Topics in Operating Systems (HotOS)*.

Ousterhout, J., Levy, J., and Welch., B. (1997). The safe-tcl security model. *Sun Microsystems Laboratories Technical Report TR-97-60*.

Peterson, D., Bishop, M., and Pandey, R. (2002). A flexible containment mechanism for executing untrusted code. *Usenix Security Symposium*.

Provos, N. (2003). Improving host security with system call policies. *Proc. 12th USENIX Security Symposium*. pp. 257-272.

van 't Noordende, G., Balogh, A., Hofman, R., Brazier, F., and Tanenbaum, A. (2006). A secure and portable jailing system. *Technical report IR-CS-025, Vrije Universiteit*.

Wallach, D., Balfanz, D., Dean, D., and Felten, E. (1997). Extensible security architectures for java. *16th ACM Symposium on Operating Systems Principles*. pp. 116-128.