Informatica — Universiteit van Amsterdam

Bachelor Informatica

UvA Universiteit van Amsterdam

Redesigning a Linux Jailing System

Indan Zupančič

August 19, 2009

Supervisor: Guido van 't Noordende

Signed: Andy D. Pimentel

Abstract

System call interception based jailing is a well-known method for confining (sandboxing) untrusted binary applications. This report is about improving one such jailing implementation.

The insight obtained by analysing the code is used to avoid most complexity and code repetition found in the current jailer, resulting in a small and simple system call interception based jailer.

A new mechanism to inject system calls is used to simplify shared read-only memory mapping, which is needed for a novel way of avoiding race conditions introduced by the original jailer.

Abstracting system calls to resource accesses is used to drastically reduce the amount of code and make the jailer easier to port to other platforms.

Contents

CHAPTER 1

Introduction

In computer science, a jail is a securely separated environment in which untrusted applications can be safely run without having unrestricted access to the rest of system. The software implementing such jail is called a *jailer* and the processes running within *prisoners*.

The jailer starts a process in a jail and that process can do anything it wants as long as it does not affect anything outside the jail. This means it can create threads or fork other processes and communicate with them. The same rules apply to these new processes as for the first process. The jailer keeps track of everything and contains all the prisoners. It does confinement per whole jail instead of per process. All that needs to be configured is which files and directories the prisoners have read or write access to. Access to other resources that normally would fall outside the jail, such as Internet addresses, can be granted with exception rules in the policy file.

All the system calls made by the prisoners are intercepted and audited. For a process making system calls is the only way to have interaction with the outside world, hence auditing all system calls is sufficient to contain a process.

System call interception on Linux can be done with `ptrace(2)`, which gives the chance to intercept system call events of a traced process. Pre syscall events are generated just before the system call is actually executed by the kernel. Similarly post system call events are generated just before a system call returns. In both cases the tracer can read and modify the process' registers and memory. The traced process is suspended till the tracer tells it to continue, in the case of jailers usually after it has checked whether the system call should be allowed or not.

One system call interception based jailer is designed and written by Guido van 't Noordende and Rutger Hofman [van 't Noordende et al., 2007]. What makes this jailer appealing is its simplicity: There is no need for kernel modifications, super user privileges or complicated configurations. Certain race conditions are avoided in a novel way by using shared read-only memory. This jailer consists of an operating-system specific interception layer and a portable policy layer. Both layers use a memory manager module which manages the shared read-only memory (ShRO) of the prisoner processes in the jail. The interception layer drives the jailer by

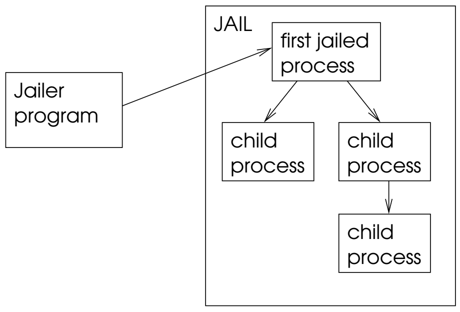

Figure 1.1: A jail's process hierarchy. Attempts to access resources outside the jail, such as files, unjailed processes or the network, are audited and only allowed when the jailing policy says so.

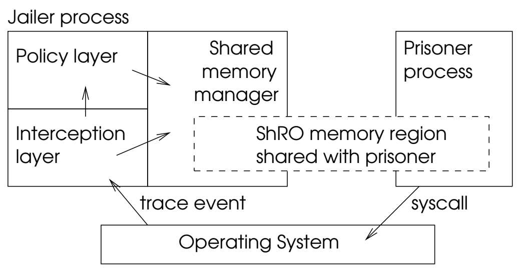

Figure 1.2: The jailer's internal architecture. Note that a trace layer can be separated from the intercept layer, which isn't shown here.

handling trace events from the operating system and by calling the memory manager and policy layer accordingly. See figure 1.2.

The initial goal was to port this jailer to the x86_64 cpu architecture and to do a security audit. When doing this a few things became clear: However simple and elegant the jailer is from a conceptual view, the actual code is not. Partly because implementing a secure jailer is hard and more tricky than expected, but also because the current design is suboptimal. It has grown into a big pile of tricky subtle code, around 25 thousand lines of code. The inherent complexity of a jailer cannot be avoided, but the rest can be much improved. Too much subtle interaction between different parts of the code can be avoided as well.

Because of the sheer size and repetitive code it is also hard to audit the code for known possible exploits, as for each a lot of code needs to be checked. This makes the jailer practically unverifiable, and hence it can't be trusted to be secure. Multiple exploitable bugs are found which could all be fixed one by one, but as said, fixing a problem in one spot is not sufficient, a lot of other code needs to be checked for the same problem. This is partly caused by lack of structure and abstraction in the code itself.

A more systematic approach is needed. After days of bug hunting it became clear that it might be a good idea to go back to basic and start from scratch with a solid base. It won't have the full functionality of the original jailer, but considering that many of the more complex features are currently insecure because they can be bypassed one way or the other, this is a chosen trade-off, preferring security above functionality.

The new goal is to improve the current jailer design. The insight gained from working with the current jailer and analysing its code is used to redesign the jailer and write a new jailer that tries to keep the good parts of the original jailer, without making the same mistakes. This should result in a smaller, simpler, more secure and portable jailer.

CHAPTER 2

Trace Layer

2.1 Introduction

The trace layer is responsible for tracing processes. It intercepts system calls and notifies the intercept layer who does the further handling. The trace layer generates pre and post system call events and provides low-level functionality that is trace layer dependent. This includes things like changing system call number or argument values, as well as providing a way of denying a system call from happening. In the old jailer the trace layer also keeps track of the processes and threads within the jail.

The current jailer is written on top of Strace [Str,], a system call tracing program used for debugging and diagnostics. The jailer's trace layer consist of a modified Strace version. For a proof-of-concept implementation that is a sensible approach, as Strace does part of what is needed and hooking into Strace's current code is a relatively quick and easy way of getting something that works. Another upside is that Strace is very portable, supporting many operating systems and CPU architectures.

2.2 Problems with using Strace

Using Strace also has its downsides. Strace itself is a big chunk of code which was not written with security in mind. It does not expect the traced processes to try to escape the jail or to crash the tracer. Hence a security audit of Strace's code is needed before it can be trusted. For process containment to work, it must be guaranteed that processes can't escape the jail and that all system calls are intercepted. Any mistake in the trace layer can render the whole jailer ineffective.

One problem is that Strace's assumptions are not documented and thus are hard to verify. This also makes auditing the code for correctness more difficult than it already is. The code has grown over the years, supporting more and more architectures and operating systems, resulting in complex and unclear code. Strace consists of about 29 thousand lines of code, more than the current jailer code itself. This is mostly because it implements functionality not needed for a jailer. Ignoring unused parts of the code then still ten thousand lines or more need to be checked.

A more practical problem is the dependency on an outside project. Strace has its own build system which has to be hooked into or extended to built the jailer too. It was chosen to hook into it, using a different build system for the jailer code. This results in two build systems for one project.

Using Strace results in a jailer that can't be really trusted, or if it can, in a lot of extra auditing work in an area the jailer authors aren't familiar with. One hard to track bug in the Strace code eroded the trust in the Strace code for jailing purposes. After all, it is used for a totally different purpose and functions important for a jailer may not be tested much, as they are not used by a normal Strace.

All in all there is enough motivation to replace Strace with a custom thin trace layer that only does what is needed for a jailer, and nothing more.

2.3 New Trace Layer

Strace uses a special system call named ptrace(2), or a similar interface like /proc/ on Solaris. Ptrace is also used by debuggers to hook into a running process. Unfortunately, the details can differ a lot between operating systems, though they all have the basic functionality needed. The new trace layer's task is to provide a portable API that hides away all the operating system and trace layer's implementation details, while still being small and simple.

A design goal was to keep the trace layer easily portable to different CPU architectures. This is achieved by abstracting away all architecture specific things and keeping the trace layer system call agnostic as much as possible: It has no system call specific knowledge at all. Another goal is to keep it as simple as possible, that is why the trace layer is stateless.

As the rest of the jailer only supports Linux on x86, the first version was written for Linux. The Linux kernel 2.6 has a few ptrace features that are very handy in keeping all created processes within the jail as well as tracking process exits easily, by automatically tracing new child processes and generating an exit event when a traced process exits. Sadly kernel 2.4 ptrace does not support tracking process and thread creation and exits. This is also true for other operating systems like BSD. The consequence is that keeping the trace layer totally system call agnostic is not always possible. Kludges around exit(2), fork(2), etc. are still needed sometimes.

The old jailer used Strace as a basis. Strace provided the start-up code and main program loop, making upcalls into the jailer's intercept layer. The new trace layer is implemented more as a library, moving the program control loop elsewhere.

An unfortunate property of ptrace(2) is that it does not tell whether an event is a system call pre or post event. As the new trace layer is stateless the task of keeping track of this moves to the intercept layer, which now also needs to do process tracking. This makes the trace API less elegant to use than hoped. To determine the event type the intercept layer needs to remember whether the last system call event was a pre or post one. The very first system call event is always a pre, all other events alternate between pre and post.

2.4 Implementation

The trace layer API is attached as appendix A. The trace API is generic, all trace module implementation specific information is elsewhere. This way the architecture and operating system details are clearly separated from the rest. Although ptrace's characteristics had a big influence on the interface, the API is still flexible enough for other kind of tracing mechanisms to fit in.

The jailer calls `trace_fork()` and does an execve to start and contain the first prisoner. In the program main control loop `trace_event()` is called to get the next trace event. EVENT_SYSCALL events are passed to the intercept layer, the other event types are handled directly.

When the upper layers are done with the system call event the intercept layer calls one of `event_allow()`, `event_deny()` or `event_kill()`. Then the trace layer continues the intercepted system call in the requested way.

The Linux trace module implementation has very little CPU architecture specific code. This is all code needed to support x86:

```
#define NUM_REGS        17

#define REG_ORIG_SCNR   11      // used to read the syscall nr.
#define REG_SCNR        6       // used to change the syscall nr.
#define REG_ARG1        0
#define REG_ARG2        1
#define REG_ARG3        2
#define REG_ARG4        3
#define REG_ARG5        4
```

```
#define REG_ARG6        5
#define REG_RET         6
#define REG_IP          12

/* Luckily int 80, sysenter and syscall are all 2 bytes */
#define SYSCALL_INSTR_SIZE 2
```

The main purpose of this list is to tell in which registers required information can be found. To be supported by the Linux trace module, an architecture needs to have a similar list of defines. The Linux trace module is for the rest architecture independent.

It is more complicated when 32 bits prisoners can run in a 64 bits jail. This happens when a CPU supports running processes in different modes and the jailer is in a different mode than some prisoners, for instance on a x86_64 system.

In addition to the above mentioned architecture specific information, for x86_64 the only code needed to support running 32 bits prisoners under a 64 bits jailer is the following:

```
#define COMPAT32

static inline int compat32(const long* regs)
{
        long cs = regs[17];

        if (cs & 0x23)
                return 1;
        return 0;
}
```

`compat32()` returns true if the event originates from a 32 bits prisoner running in a 64 bits jail. This function must be defined to support such mixed modes for an architecture.

The rest of the code uses the COMPAT32 define or the compat32() function to decide what to do. This is rarely needed though. One example is when the intercept layer gets a new event and it wants to find out which system call it is. The system call numbers are different for 32 and 64 bit processes. For instance, getpid(2) is number 20 for 32 bits and number 39 for 64 bits system calls. The trace layer uses compat32() to figure out which look-up table to consult.

2.5 Discussion

The result is a tiny trace layer consisting of about 500 lines of code. The new trace layer supports Linux 2.6 kernel on X86 and X86_64, as well as jailing 32 bits programs within a 64 bits jail, if the platform supports it. Adding support for a new CPU architecture in Linux should be not much more work than adding some defines that map the system call argument numbers to the right CPU registers. Porting the trace layer to another operating system is more work, depending on the ptrace-like functionality provided by the system. Adding support for BSD isn't that much different than Linux 2.4, as they both need mostly the same kludges.

CHAPTER 3

Intercept Layer

3.1 Introduction

The intercept layer is the core of the jailer where most work is done, it is two-third of the original jailer code, not counting Strace. The intercept layer's task is slightly unclear because it does much more than merely intercepting system calls and consulting the policy layer for what to do with them.

Most importantly the intercept layer makes the whole into a hopefully inescapable and secure jailing environment. It is called by the trace layer and asks the policy layer whether system calls should be allowed or not. It prevents race conditions that otherwise would render the jail useless. It keeps track of processes, threads and other resources such as file descriptors and shared read-only memory. In practice it does everything that Strace doesn't and is not part of the policy layer. It is big, complex and unfortunately not easy to port.

That the intercept layer does not really do what it originally was supposed to do, translating raw system calls into a known set of POSIX-like pseudo system calls, does not make things clearer. It only lets Strace decode some Linux specific system calls like ipc(2) and socketcall(2), but it does not normalise system call arguments and return values. An alternative approach which was implemented in the new jailer is described in the *Resource-Centric Policy Layer* chapter.

3.2 Shared Read-Only Memory

A big problem of trace based jailers is the *Time of Check to Time of Use* race [Garfinkel, 2003]. This is solved in a novel way [van 't Noordende et al., 2007]: Sensitive system call arguments, namely the data referenced by pointers, are copied to shared read-only memory by the jailer before the policy check is done and the system call is executed. This assures that data pointed

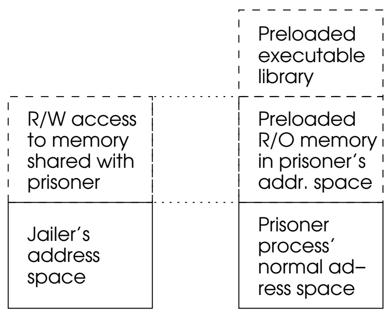
Preloaded executable library
R/W access to memory shared with prisoner
Preloaded R/O memory in prisoner's addr. space
Jailer's address space
Prisoner process' normal ad-ress space

Figure 3.1: The shared read-only memory solution for avoiding user-level multithreading and shared memory race conditions. The shared memory region is mapped in transparently at prisoner startup time, after the executable is loaded with an execve(2) call.

to by arguments cannot be modified by any prisoner after the check. On Linux, arguments themselves are in registers and are safe. On BSD the arguments are passed via the stack, so that part of the stack should be copied as well (and the stack pointer adjusted).

This copying to shared read-only memory is, for instance, needed for all file paths. Otherwise an attacker could modify the path from another thread or process after the jailer checked it, but before the kernel uses the path. This would render the whole jailer insecure because the prisoners would be able to open arbitrary files.

Shared read-only memory is writable by the jailer but read-only for prisoners. This memory can be either a SYSV shared memory segment or a shared file mapping. See figure 3.1.

Creating the writable mapping is simple for the jailer, but letting the prisoners map the shared memory too is a lot harder. Prisoners are arbitrary programs and we don't have direct control over them.

3.3 Problems with Preloading

In the original design the read-only memory was mapped by using library preloading, a feature provided by the dynamic linker. At every prisoner program start-up the libraries listed in the environment variables LD_PRELOAD are loaded first. This was used to load a small library which did the necessary setup in its initialisation routine. Library preloading was used as a clever trick to let prisoners execute code provided by the jailer, this way shared read-only memory can be setup. This happens after each execve(2) call, when the process memory is replaced and a new program is loaded by the dynamic loader.

One obvious limitation is that this trick only works with dynamically linked libraries and not for statically linked ones. How big of a problem this is depends on whether people want to run statically linked programs within the jailer: Either it doesn't matter or the jailer is unusable.

The main problem is that a lot happens before the preloaded code is run, namely everything the linker and libc do. This can't be handled like all other system calls because the shared read-only mapping isn't in place yet. That is why the jailer has a special pre-mapped state for new processes and runs them in a mini-jail with its own rules. This adds complexity and a heap of extra source code, decreasing overall security.

Most potential problems are absent when assuming that malicious programs can't do anything until the preloaded library is done loading and the new program runs in the full jail. Combined with us trusting the loader as well as libc and all seems fine. It would be true if there is no way to run untrusted code before the mapping is done.

Unfortunately running malicious code in the mini-jail is easier than it may seem. For instance by using an exploitable bug in the loader, which doesn't expect to load malicious libraries, or by using tricks like changing the library search path before calling execve(2), so that a custom libc is loaded instead of the system one. The latter would work with the original jailer.

For the mini-jail to be secure, choosing what may and what may not be done should be done very carefully. All race conditions must be prevented from happening in another way than with read-only shared memory. All slightly dangerous system calls must be disallowed. At the very least any multithreading must be disallowed, but care should also be taken that no memory is shared with other processes. The original jailer didn't prevent either from happening, with as result that race conditions are possible in the mini-jail environment.

The mini-jail, preload library with assembly code, the interaction and communication between jailer and preload library, and all the extra tricks and checks, like rewriting the prisoner's environment, make up a great part of the intercept layer's code and complexity.

Although the current jailer is mostly secure, though not quite, that security is achieved by adding a lot complex and security sensitive code. It is a bad sign when to make things more secure, more and more checks and tricks need to be added. Basically security is achieved by plugging all known exploits, this is not security by design. Any new trick can cause everything to fall apart. All this trouble is not worth it if a better way can be found.

3.4 System Call Injection

The whole mini-jail and all the careful checking and tricks can be avoided if we can ensure that prisoners have setup the memory mapping before trying to make any system call. Then there are no special cases and everything happens in the regular jailing code.

The solution is to let the prisoner do a mmap(2) or shmat(2) to get the shared read-only mapping, somewhere between the execve return and the first system call. This ensures the mapping is present before even the dynamic loader or libc made any system calls. We achieve this by using a technique called *system call injection*

Combining system call replacement with system call restarting gives the ability to inject arbitrary system calls into a prisoner.

We can already replace system calls done by prisoners by changing the system call number and arguments. Changing the syscall number to something harmless is how the jailer denies system calls not passing its policy. Modifying system call arguments to point to shared read-only memory is the main mean of preventing race conditions, so the jailer must be able to change system call arguments too. Together that makes it possible to replace a whole system call.

All we need now is a way to let the prisoner make one system call more than it usually would. Then we can replace the first one with the system call setting up the mapping, without losing the original call, which would now become the second system call. This second call is caused by modifying the prisoner's instruction pointer.

After execution of the injected system call, we need to set the instruction pointer to the start of the CPU instruction triggering the original system call. For this we need to know the size of that instruction, because the instruction pointer points just after it when a system call is intercepted. This is also how the Linux kernel restarts system calls that were interrupted by a signal handler and need to be restarted.

On x86 and x86_64 system calls are done by triggering a software interrupt with a `int 0x80`, `sysenter` (Intel) or `syscall` (AMD) instruction. Luckily all three of them are two bytes long, which means that to restart a system call we have to subtract two from the instruction pointer. For other CPU architectures there is commonly only one instruction used to enter the kernel, making restarting trivial for them as well.

To implement shared read-only memory, a file is created and mapped writable at jailer start-up. The file is opened in read-only mode to obtain a safe file descriptor. This fd is used by prisoners to get the shared read-only mapping. They can do this because file descriptors, like stdin, are inherited by child processes and shared between threads.

The mapping does not disappear after fork-like calls, only after execve(2) calls or when it gets unmapped. The unmapping is prevented by the jailer. However, when a new task is created the jailer needs to know the address of the shared read-only mapping. That address is the same as the task creator's address, because it didn't change.

After a successful execve call the jailer clears the stored mapping address because it became invalid. At the start of pre system call event handling the jailer checks if there is a valid shared read-only memory address. If not, a mmap(2) call is injected.

Appendix B shows what state is tracked for each task.

Getting rid of library preloading avoids the need to modify prisoners' environments, the mini-jail, all the special careful handling and the preloaded library itself, including its assembly code. This makes the jailer as a whole much simpler and more robust. System call injection is also faster than library preloading and works with static binaries.

CHAPTER 4

Resource-Centric Policy Layer

4.1 Introduction

The policy layer has as task to decide whether a prisoner system call should be allowed or denied. To make this decision the policy layer consults the user configuration file and follows a simple jailing model, which allows all read and write operations within the jail and denies all actions that would affect anything outside of it and isn't explicitly allowed in the user policy file. The intercept layer asks for the decision and carries it out.

The policy layer does some other things as well, most importantly it makes sure that sensitive prisoner data, like file paths, get copied to shared read-only memory. The shared read-only memory handling itself is taken care of by the shared memory module.

4.2 Problem of Doing Things per System Call

Observed was a lack of abstraction in the original jailer implementation. Some problems caused by doing everything per system call are: Code duplication and scattered policy and some decision making. This results in making code auditing a laborious task.

Doing things per system call is also not portable, but Operating system and CPU architecture dependent, more than expected considering most system calls follow the POSIX standard. What is defined by POSIX is the application programming interface (API), not the binary interface (ABI), which for normal applications is mostly hidden behind libc. The jailer works directly with the raw system call interface, but is written as if working with libc's API. This mostly works, but it assumes that each system call has exactly the same interface on all platforms. It also ignores duplicate system calls: In e.g. Linux when an interface turns out to be too limiting in some way and needs to be changed, a new system call is added implementing the extended interface. The old one is left to keep binary compatibility with old applications. This results in things like mmap(2) and mmap2(2), something that is normally hidden away by libc, but is totally ignored by the original jailer, although not by design. As most jailer code does things per system call most code isn't portable. The bad thing is that the differences between platforms are often subtle.

The original idea was to map all platform system calls to pseudo POSIX like system calls with a well defined ABI, having one mapping for each platform. This would solve the portability problem and is simple to implement. However, it would not solve the code duplication problem and it is still a lot of work. The current code does no remapping at all and is already huge, adding explicit remapping would only increase the code size and complexity. The mapping would re-implement what libc does: Mapping the raw system call interface to a POSIX-like ABI, though the other way round than libc does. Avoiding all this was the motivation to find a better, more elegant solution.

4.3 Resource Access Abstraction

The jailer is not interested in what prisoners do, it wants to know what resources are accessed instead. This means that the jailer is often more interested in some arguments supplied to a system call than which system call it exactly is.

The chosen solution is to abstract all system calls into resource accesses. This is done by consulting a look-up table which describes for each system call what needs to be done to handle it. The table stores for each system call argument what type of resource it is and, if necessary, also if it's used for a read or write access.

All other system call specific information is put in that table too, keeping system call specific information in one place. This gives a way to handle most system calls by their resource accesses, but also has enough flexibility to do system call specific things when needed.

There is one table for each operating system and CPU architecture combination, as the kernel system call interfaces differ between all variants.

The new Policy layer is a lot simpler than the old one because it is system call agnostic and works with resource accesses, which often map directly to policy rules.

One advantage of this design is that it degenerates into the original jailer design for when system calls can't be abstracted. This means that even for cases where this approach doesn't work at all, the end result isn't worse than it used to be.

4.4 Implementation

The look-up table was inspired by Strace's system call table and the original jailer's rule table, which was used by a perl script to generate a huge switch statement. Everything that is really system call specific is put into the new look-up table, keeping all other source code system call agnostic.

At least this is the goal, some system calls like open(2), ipc(2) and socketcall(2) can do too many different things depending on their arguments and may need to be demultiplexed before being properly abstracted into resource accesses. These are decoded into something more usable.

Other system calls might be just too exotic to be easily abstracted or need unique handling for other reasons. Those can be marked to be handled specially by the intercept layer. For example, after a *execve* we need to inject a mmap(2) to map the shared read-only memory segment, and after a $fork$ we need to do some memory bookkeeping. The power of the look-up table approach is that the same mark can be added to *clone* and $vfork$, handling those without adding any new code.

The goal is to minimize the amount of special code and to abstract as many system calls away as possible.

The look-up table is an array of `struct syscall`. This array is indexed by the system call number to get to the corresponding syscall's information, hence the order is very important.

The complete abstraction can be found in appendix C, and appendix D shows how it is applied for Linux x86's look-up table, or at least the first part of it.

As can be seen in appendix C, `struct syscall` is defined as:

```
struct syscall
{
        const char*     name;
        sc_flags_t      flags;
        arg_t           args[MAX_ARGS];
};
```

`name` is the system call name, such as "uname", "olduname" or "oldolduname", to pick an example that illustrates why abstracting system calls away is a good idea.

`flags` is used to store system call specific information like which resources are accessed, if and how it needs decoding, and so on.

One noteworthy flag is ALWAYS_ALLOW: It is used for system calls we are not interested in and do not access sensitive resources, like *getpid*(2). Similarly a system call with no flags at all set is always denied, like *ptrace*(2).

In args the resource type of each argument is stored, and sometimes also what access type: A read or a write operation.

The original policy layer had pre and post policy functions which were called by the intercept layer. Those functions used the system call number to figure out what to decide. The new policy layer has pre and, if needed, post policy functions for each resource type. Currently there are file_pre(struct task*), pid_pre(struct task*), fd_pre(struct task*) and mem_pre(struct task*). They check the system call arguments to decide what kind of resource access is done and whether it should be allowed.

A common work flow for a resource policy function would be the following:

1. Check struct task->current_syscall, which was filled in by the intercept layer, to see if this system call does anything this policy module is interested in. Commonly that is done by having a C_MY_RESOURCE flag and checking it to know quickly if the system call access the resource type the policy module tries to handle. If not interested return NONE as verdict given.

2. If interested go through all the system call arguments and check its type. If the arguments are independent then they can be checked one by one, like with file descriptors. But sometimes the policy module is interested in multiple arguments. Then it first goes through all the arguments to collect the information it needs, before checking whether the resource may be accessed. The memory module does this, because it wants to know both the start address as well as the length of the affected memory segment.

3. Check if the resource access(es) should be allowed according to the jailing policy.

4. If the system call is allowed and it has sensitive argument data, copy that data to shared read-only memory. It will be automatically freed by the intercept layer when the system call is done.

5. Return the final verdict, in general ALLOW or DENY.

4.5 Resource Types

4.5.1 Memory

In general the jailer does not care about what prisoners do with memory. The only exception is that the shared read-only memory segment should be protected. If that could be bypassed then the whole read-only mechanism is ineffective. This means that system calls like *mmap* and *munmap* should be checked.

Look-up table example:

```
{"munmap",       C_MEM,  {MEM_ADDR, MEM_LEN          }}, /* 91 */
```

(All examples are taken from the Linux i386 table.)

With the system call abstracted into resource accesses by introducing MEM_ADDR and MEM_LEN, seven system calls can be handled in a generic way. Adding support for new system calls later on is easy too, as well as adding support for other platforms.

4.5.2 File Descriptor

The new jailer does not track file descriptors because it does all checking at fd creation time. In the original jailer file descriptors were tracked to implement the user-space firewall. The only fd which needs special care is the one belonging to the shared read-only memory file. If it is closed then mapping the shared read-only memory would fail in subsequent child processes that call *execve*.

Examples:

```
{"close",  C_FD,       {FD                         }}, /* 6 */

{"mmap2",  C_MEM|C_FD, {MEM_ADDR, MEM_LEN, 0, 0, FD}}, /* 192 */
```

By introducing the FD type nine system calls can be generically handled with about 20 lines of code.

For *mmap*2 both the memory addresses as file descriptor are checked. As the file descriptor has the file open in read-only mode, the mmap should be safe, but prisoners are never expected to use the fd at all. The injected mmap(2) call by-passes the policy.

To illustrate how simple policy checking like this works, here the file descriptor policy check function:

```
int fd_pre(struct task* t)
{
        int i;
        int fd;
        const struct syscall* s = t->current_syscall;

        if (!(s->flags & C_FD))
                return NONE;
        for (i = 0; i < MAX_ARGS; ++i){
                if (s->args[i] != FD)
                        continue;
                fd = t->current_event.arg[i];
                fprintf(stderr, "%d ", fd);
                if (fd == memro_fd){
                        return DENY;
                }
        }
        return ALLOW;
}
```

4.5.3 File System

Protecting the file system against unauthorized accesses by prisoners is probably the most important task of the whole jailer. It is also the most difficult one, because of the potential race conditions [Garfinkel, 2003]. Ignoring those, the policy layer needs to check whether read or write access should be allowed to a certain path. Thwarting the race conditions is very system call independent, so less interesting from the policy's layer point of view.

A simple proof-of-concept file policy module is implemented which doesn't handle racy chdir(2) calls, symlinks or embedded "/../" path components. Those could be handled by expanding the paths to absolute ones with realpath(3) and system call serialisation, but that isn't done yet. It does copy paths to shared read-only though. This module should consult a user policy configuration file to make its decisions. This feature isn't implemented yet because of time constraints, combined with the little effect it has on the design and working of the rest of the jailer. Instead a default policy is used where write access is allowed to the jaildir and /tmp/, and read-only access to system directories like /lib/ and /bin. The jaildir is the directory where the jailer was originally started.

The important part is that making the jailer secure by closing those holes shouldn't be hampered by the resource abstraction. In case of file paths this abstraction actually helps, because there are a lot system calls using paths who all need to be handled the same path by path. Some system calls need to be serialized to avoid path race conditions, but that can be done by adding a serialisation flags to the system calls involved.

Examples:

```
{"rename",      C_FS,         {FILE_RW, FILE_RW   }}, /* 38 */
{"mkdir",       C_FS,         {FILE_W             }}, /* 39 */
```

open(2) needs to be decoded to figure out what kind of access is intended, for this the access mode flags need to be checked. This is a system call specific thing, that is why it was chosen to decode open(2) to `FILE_R`, `FILE_W` and `FILE_RW` types. An alternative approach would have been to introduce `FILE` and `MODE` types and check `MODE` when a `FILE` argument is encountered. Decoding open(2) seemed simpler though.

There are 22 file related system calls that are handled in a similar way.

4.5.4 Process ID

Deciding the policy for process identifiers is simple. Prisoners may for instance not kill processes outside the jail. All actions on PIDs that are known to the jailer are allowed, the rest aren't.

All system calls for x86:

```
{"kill",              C_PID,         {PID           }}, /* 37 */
{"setpgid",           C_PID,         {PID, PID      }}, /* 57 */
{"rt_sigqueueinfo",   C_PID,         {PID           }}, /* 178 */
{"tkill",             C_PID,         {PID           }}, /* 238 */
{"tgkill",            C_PID,         {PID, PID      }}, /* 270 */
{"move_pages",        C_PID,         {PID           }}, /* 317 */
```

There are only six system calls affecting PIDs, but every little bit helps to keep the code base small.

4.5.5 Inter-Process Communication

Another kind of resources are inter-process communication mechanisms: Message queues, SYSV shared memory and semaphores. For those it is needed to check whether the resource they try to get access to was created within the jail by a prisoner or originates from elsewhere. The user policy may allow exceptions as well.

This is not implemented yet because, although simple, it is tedious specific work. Specific because the Linux i386 specific *ipc*(2) system call needs to be decoded first and its interface isn't documented in the man page, which makes it tedious work. All in all it won't be more than one or two hundred lines of code to add this functionality.

4.5.6 Network

The original jailer implemented a user-space firewall. To do that it introduced a lot of code complexity with doubtful gain, as it failed to protect unwanted UDP data traffic. There are also race conditions between file descriptor modifications and checking, making the TCP side of the whole also vulnerable. Both problems can be solved, but solving the code complexity is harder. Most of that complexity comes from trying to restrict incoming access, as for those the peer address can only be known after the system call is finished. Restricting outgoing access is a lot easier because for that only the destination address needs to be checked, which is simple.

Network access control is not implemented yet. Checking outgoing access is easy and will be done, but incoming access needs more thought before it will be implemented.

CHAPTER 5

Discussion

5.1 Code Size

Not all functionality is implemented. For some that was known beforehand, because a different trade-off was made, like the user space firewall. For others, like IPC and configurability, it was because of circumstances, of which time was one factor. Sadly, writing an extensive test-suite would have taken more diligence than was available. It would have been nice if the jailer was ported to more platforms to prove its portability. What hampered that most was the hardware for proper testing, as the porting itself doesn't take much time. At least when porting the Linux version to a different CPU architecture. A less rudimentary file policy module would make testing easier too.

All in all some things are lacking, but in the end a functioning, basic but solid jailer was implemented.

A lot of effort was put into write as little code as possible. The output of `wc` can be used to give an idea of how tiny the resulting jailer really is (comments added afterwards):

```
$ wc *.c include/*
   42   110   859 decode.c        # decodes open().
  154   456  3112 file.c          # File policy module.
   42   102   697 list.c          # generic doubly linked list.
  119   386  2727 main.c          # Initialisation and main loop.
  313   990  6705 memro.c         # Everything ShRO, fd and mem policy.
  209   645  4943 syscall.c       # Main chunk of intercept layer.
  192   577  3986 task.c          # Task tracking, PID policy.
  411  1395  9550 trace.c         # Trace layer.
   13    36   253 include/decode.h
   12    17   138 include/file.h
  324  1837 11929 include/linux_i386.h   # Look-up table for i386
  122   399  2573 include/linux_ptrace.h # Linux and ptrace specific
   61   291  1648 include/list.h
   74   294  1736 include/memro.h
  118   491  2805 include/syscall.h
   91   379  2260 include/task.h
  131   644  3927 include/trace.h        # Platform independent
 2428  9049 59848 total
```

The 29 thousand lines of Strace code were replaced with a custom trace layer of 650 lines of code: 400 lines of C code and 250 lines of header files. The trace layer used to be more than half the code, now it's a quarter of the total size.

The 26 thousand lines of jailer code were replaced with 2.4 thousand lines for the redesigned jailer.

The 16 thousand lines of the intercept layer code were replaced with about 800 lines, or 1200 when including the shared read-only memory module and even 1500 when counting the look-up table.

The 7 thousand lines of policy layer code were replaced with only 240 lines for the file system module and other resource policy functions. This is admittedly too little, proper race-free file access checking, if implemented with readlink, should take a couple hundred lines more. IPC and outgoing network access checking should take a couple hundred lines as well. The configuration file parser can be big though.

5.2 Performance

A quick performance comparison to Strace shows that the new jailer is faster. Below the test results for running Strace configure. All output was redirected to `/dev/null`, otherwise the overhead of the trace printing would dominate too much:

```
$ time strace -fF ./configure &>/dev/null
real    0m16.258s
user    0m8.853s
sys     0m6.152s

$ time jailer ./configure &>/dev/null
real    0m13.967s
user    0m7.160s
sys     0m5.280s

$ time ./configure &>/dev/null
real    0m9.072s
user    0m5.820s
sys     0m2.128s
```

Test results for the original jailer could not be obtained because on this machine Strace stopped compiling.

In previous test results [van 't Noordende et al., 2007, p421] configure showed to be one of the worst cases for trace based intercepting. As can be seen running configure through normal Strace has a slowdown of 79%, while the jailer has a slowdown of 54%. This advantage in performance is consistent with other quick tests. The above test was run cache hot and multiple times in different orders, with similar results every time.

Note that it are the worst case results because both Strace and the jailer print out all system calls and arguments, which causes a lot overhead for both. The main cost is in the context switches to and from the jailer though. The jailer also copies file paths to shared read-only memory, but that seems to be very fast.

The difference could be attributed to Strace reading more argument data, and calling ptrace more often and having a less efficient way of tracing child processes. The jailer reads all registers with one ptrace call, Strace read the registers one by one when it needs it. The jailer only read file paths, but no other data. Strace reads the results of system calls and for that it often need to read prisoner's memory, which is slow. Lastly the jailer uses Linux 2.6 specific flags to let the kernel automatically trace new processes. Strace probably uses a less efficient mechanism, causing more context switches.

5.3 Portability

The Trace layer is tiny but platform and operating system specific. The Intercept layer is portable, but might need some tweaks for different operating systems. The same is true for the Policy layer, though it's less likely that tweaks are needed there. Mostly it should be a matter of creating a new look-up table for the new architecture, which is just tedious and careful work, but quite simple.

Running 32 bits prisoners in a 64 bits jail was not an afterthought, but something the design kept in mind from the beginning. This makes it almost trivial.

The portability of the whole jailer is improved by handling system calls in a generic way and avoiding system call specific code as much as possible.

CHAPTER 6

Conclusion

When starting a long journey it is not always clear which path is best to take. Sometimes, before scaling the next cliff, it helps to look back to see where a wrong turn was taken, and around to see where a better path may lay. But before that better path can be found a mountain needs to be climbed first.

The two biggest problems of the original jailer were complexity escalation and code duplication, resulting in a massive code size. Both problems are convincingly solved, but at the price of functionality.

Adding the most important missing pieces should be relatively easy and not too much work, depending on which path is chosen next. The new jailer is smaller, simpler and more secure, if only because it does not implement some features insecurely. The jailer is also more portable than the old one, not only because of its smaller size, but also because it was written with portability in mind.

Even running 32 bits prisoners under a 64 bits jail should just work, something that was totally hopeless to implement for the original jailer, and one of the motivations for this redesign.

APPENDIX A

trace.h

```
/*
 * Public trace module API. Trace implementation independent.
 */
#ifndef TRACE_H
#define TRACE_H

#ifdef NEW_OS
 #include "NEW_OS_*.h"
#elif defined(linux)
 #include "linux_ptrace.h"
#else
 #error "Unsupported operating system, please write a port! See doc/porting.txt"
#endif

typedef enum event_type
{
        EVENT_NONE,     /* No event. Usually because all prisoners exited. */
        EVENT_INTR,     /* The tracer was interrupted. */
        EVENT_SIG,      /* Prisoner is receiving a signal. */
        EVENT_SYSCALL,  /* System call enter or exit */
        EVENT_EXIT,     /* Prisoner exited one way or the other. */
} event_t;

/*
 * The tracer checks for changes to struct event and does the necessary
 * system call number, return code or argument values updates.
 */
struct event
{
        /* Read-only, don't change */
        event_t type;           /* Invalid when EVENT_NONE */
        int pid;                /* The traced task generating the event */
        int num;                /* syscall or signal number */
        int compat32;           /* True for 32bit prisoner in 64bit jailer */

        /* Changes to these will affect what happens */
        long syscall_nr;        /* If changed, expect a post for the new nr. */
        long result;            /* Setting this makes only sense in post. */
        long arg[MAX_ARGS];     /* Beware of (no) sign extension */
        long instruction_pointer;

        /* for internal use only, do not touch */
        event_state _p;
};
```

```
/*
 * Just like fork(), but child process is being traced.
 */
int trace_fork(void);

/*
 * Get the next trace event, either syscall or signal, filled in 'e'.
 * Returns the event type on success, < 0 on error (-errno).
 * If EVENT_NONE is returned then there are no traced processes left.
 */
event_t trace_event(struct event* e);

/*
 * Allow or deny the current syscall or signal delivery. Prisoner is stopped
 * until either one or event_kill() is called. After the call 'e' is invalid
 * for further use, until it's filled in again by a trace_event() call.
 */
void event_allow(struct event* e);
void event_deny(struct event* e);

/*
 * Kill and stop tracing a process.
 * (A kill(2) is asynchronous and might generate more events.)
 */
void event_kill(struct event* e);

/*
 * Copy 'len' data from prisoner memory address 'src' to jail address 'dst'.
 *
 * Returns the number of bytes read. If an error occurs that can be less than
 * asked for.
 */
int read_prisoner_mem(struct event* e, void* dst, long src, int len);

/*
 * Copy a string from 'src' into 'dst', but no more than 'len' bytes.
 *
 * Returns 0 when a complete string is copied over, errno otherwise.
 * ENOMEM is returned if the source string is longer than 'len'.
 */
int read_prisoner_str(struct event* e, char* dst, long src, int len);

/*
 * Write to prisoner's memory. Returns the number of bytes written.
 */
int write_prisoner_mem(struct event* e, long dst, void* src, int len);

/*
 * Restarts a system call by adjusting the instruction pointer to point
 * to just before the syscall cpu instruction.
 */
static inline void restart_syscall(struct event* e)
{
        e->instruction_pointer -= SYSCALL_INSTR_SIZE;
}

#endif
```

APPENDIX B

Task State

```
/*
 * 'addr' is the prisoner's shared read-only memory address as returned
 * by mmap(2). The original system call number is kept to restore the
 * original system call. The arguments are already restored at post.
 */
struct memro {
        unsigned long addr;
        long original_syscall_nr;
};

/*
 * Separate syscall state independent of the task state,
 * as that is a more realistic view of a task's systemcall states.
 */
typedef enum TASK_STATE {
        USER = 0,                       // Running user code (must be zero)
        PRE,                            // pre-system call: At start of a syscall
        SYS,                            // Doing the actual system call
        POST,                           // post-system call: At end of a syscall
} task_state_t;

/*
 * All per-task state in here.
 * A task can be a process or a thread.
 */
struct task {
        int pid;                        // On Linux it's the thread ID for threads.
        struct task* next;              // for handling hash collision

        task_state_t syscall_state;
        const struct syscall* current_syscall;

        struct event current_event;
        long original_args[MAX_ARGS];   // restore args at post syscall.
        void* shm_arg[MAX_ARGS];        //The shmRO address of each arg, NULL for none.

        struct memro memro;
        struct list queue;              // for serialisation
};
```

APPENDIX C

syscall.h

```
#ifndef SYSCALL_H
#define SYSCALL_H

#include "trace.h"
/*
 * Per system call flags to be used with struct syscall.flags.
 */
enum SC_FLAGS {
        ALWAYS_ALLOW    = 1 << 0,       /* It’s harmless, let it be */
        DECODE          = 1 << 1,       /* Demultiplex syscalls like open(2) */
        D_OPEN          = 1 << 2,       /* Needs to be decoded */
        D_SOCKETCALL    = 1 << 3,       /* Needs to be decoded */
        D_IPC           = 1 << 4,       /* Needs to be decoded */
        EXEC            = 1 << 5,       /* execve() is being called */
        FORK            = 1 << 6,       /* fork/clone/vfork is being called */
        SERIALIZE       = 1 << 7,       /* Check C_* to see how to serialize? */
        SERIALIZE_R     = 1 << 8,       /* Serialize only with writers */
        C_PID           = 1 << 9,       /* Check for PID arguments */
        C_FD            = 1 << 10,      /* Check for FD arguments */
        C_MEM           = 1 << 11,      /* Protect shared read-only memory */
        C_FS            = 1 << 12,      /* Check file paths */
        C_IPC           = 1 << 13,      /* Check IPC access */
        C_NET           = 1 << 14,      /* Check Network access */
};
typedef unsigned short sc_flags_t;

/*
 * This is the main way to abstract system calls into resource accesses.
 * Give argument resource and, if needed, access type.
 */
enum ARG_TYPES {
        ARG_IGNORE      = 0,
        PID,            /* Process ID */
        FD,             /* File desciptor */
        MEM_ADDR,       /* Memory address */
        MEM_LEN,        /* Memory address length */
        FILE_R,         /* File read path */
        FILE_W,         /* File write path */
        FILE_RW,        /* File read and write path */
        IPC_KEY,
        NET_ADDRESS,
};

typedef unsigned char arg_t;
```

```
/*
 * All system call specific information.
 *
 * 'name' is the name of the system call.
 * 'flags' tells how this system call should be handled.
 * 'args' stores argument type information.
 *
 * The look-up tables are arrays of these syscall structures.
 */
struct syscall
{
        const char*     name;   // syscall name, useful for logging.
        sc_flags_t      flags;
        arg_t           args[MAX_ARGS];
};

struct task;

/*
 * Given a system call number, return the corresponding struct syscall.
 * Returns NULL for invalid or unknown system calls.
 */
const struct syscall* get_syscall(struct task* t, int syscall_number, int compat32);

/*
 * Called for each system call event.
 * Should call event_allow, event_deny, event_kill, or queue it up.
 */
void handle_syscall(struct event* e);

/*
 * For a system call to be allowed, ALLOW must be set and DENY not set.
 */
typedef enum V {
        NONE            = 0,
        ALLOW           = 1 << 0,
        DENY            = 1 << 1,
        QUEQUE          = 1 << 2,
        KILL            = 1 << 3,
} verdict_t;

#endif
```

APPENDIX D

syscall_table[] for x86

```
/*
 * The system call number is the index into this array for the same system call.
 * E.g. the "read" system call number is 3 and hence syscall_table[3] points to
 * its information.
 */
static const struct syscall syscall_table[] = {
{"restart_syscall",     ALWAYS_ALLOW                       }, /* 0 */
{"_exit",       ALWAYS_ALLOW                               }, /* 1 */
{"fork",        ALWAYS_ALLOW|FORK                          }, /* 2 */
{"read",        ALWAYS_ALLOW                               }, /* 3 */
{"write",       ALWAYS_ALLOW                               }, /* 4 */
{"open",        DECODE|D_OPEN,  {                          }}, /* 5 */
{"close",       C_FD,           {FD                        }}, /* 6 */
{"waitpid",     ALWAYS_ALLOW                               }, /* 7 */
{"creat",       C_FS,           {FILE_W                    }}, /* 8 */
{"link",        C_FS,           {FILE_RW, FILE_RW          }}, /* 9 */
{"unlink",      C_FS,           {FILE_W                    }}, /* 10 */
{"execve",      EXEC|C_FS,      {FILE_R                    }}, /* 11 */
{"chdir",       C_FS,           {FILE_R                    }}, /* 12 */
{"time",        ALWAYS_ALLOW                               }, /* 13 */
{"mknod",                                                  }, /* 14 */
{"chmod",       C_FS,           {FILE_R                    }}, /* 15 TODO: check mode?*/
{"lchown",      ALWAYS_ALLOW                               }, /* 16 idem dito */
{}, /* name,    flags, args[6] */
{"oldstat",     C_FS,           {FILE_R                    }}, /* 18 */
{"lseek",       ALWAYS_ALLOW                               }, /* 19 */
{"getpid",      ALWAYS_ALLOW                               }, /* 20 */
{"mount",                                                  }, /* 21 */
{"oldumount",                                              }, /* 22 */
{"setuid",                                                 }, /* 23 */
{"getuid",      ALWAYS_ALLOW                               }, /* 24 */
{"stime",                                                  }, /* 25 */
{"ptrace",                                                 }, /* 26 */
{"alarm",       ALWAYS_ALLOW                               }, /* 27 */
{"oldfstat",    C_FS,           {FILE_R                    }}, /* 28 */
{"pause",       ALWAYS_ALLOW                               }, /* 29 */
{"utime",       C_FS,           {FILE_RW                   }}, /* 30 */
{},
{},
{"access",      C_FS,           {FILE_R                    }}, /* 33 */
{"nice",                                                   }, /* 34 */
{}, /* name,    flags,  args[6] */
{"sync",                                                   }, /* 36 */
{"kill",        C_PID,          {PID                       }}, /* 37 */
{"rename",      C_FS,           {FILE_RW, FILE_RW       }}, /* 38 */
{"mkdir",       C_FS,           {FILE_W                 }}, /* 39 */
{"rmdir",       C_FS,           {FILE_W                 }}, /* 40 */
{"dup",         C_FD,           {FD                     }}, /* 41 */
{"pipe",        ALWAYS_ALLOW                            }, /* 42 */
{"times",       ALWAYS_ALLOW                            }, /* 43 */
{},
{"brk",         ALWAYS_ALLOW                            }, /* 45 */
{"setgid",                                              }, /* 46 */
{"getgid",      ALWAYS_ALLOW                            }, /* 47 */
{"signal",      ALWAYS_ALLOW                            }, /* 48 */
{"geteuid",     ALWAYS_ALLOW                            }, /* 49 */
{"getegid",     ALWAYS_ALLOW                            }, /* 50 */
{"acct",                                                }, /* 51 */
{"umount",                                              }, /* 52 */
{},
{"ioctl",       ALWAYS_ALLOW                            }, /* 54 */
{"fcntl",       ALWAYS_ALLOW                            }, /* 55 */
{},
{"setpgid",     C_PID,          {PID, PID               }}, /* 57 */
{}, /* name,    flags,  args[6] */
{"oldolduname", ALWAYS_ALLOW                            }, /* 59 */
{"umask",       ALWAYS_ALLOW                            }, /* 60 TODO: mask the mask */
{"chroot",                                              }, /* 61 */
{"ustat",                                               }, /* 62 */
{"dup2",        C_FD,           {FD                     }}, /* 63 */
{"getppid",     ALWAYS_ALLOW                            }, /* 64 */
{"getpgrp",     ALWAYS_ALLOW                            }, /* 65 */
{"setsid",                                              }, /* 66 */
{"sigaction",   ALWAYS_ALLOW                            }, /* 67 */
{"siggetmask",  ALWAYS_ALLOW                            }, /* 68 */
{"sigsetmask",  ALWAYS_ALLOW                            }, /* 69 */
{"setreuid",                                            }, /* 70 */
{"setregid",                                            }, /* 71 */
{"sigsuspend",  ALWAYS_ALLOW                            }, /* 72 */
{"sigpending",  ALWAYS_ALLOW                            }, /* 73 */
{"sethostname",                                         }, /* 74 */
{"setrlimit",                                           }, /* 75 */
{"old_getrlimit",       ALWAYS_ALLOW                    }, /* 76 */
{"getrusage",           ALWAYS_ALLOW                    }, /* 77 */
{"gettimeofday",        ALWAYS_ALLOW                    }, /* 78 */
{"settimeofday",                                        }, /* 79 */
{"getgroups",   ALWAYS_ALLOW                            }, /* 80 */
{"setgroups",                                           }, /* 81 */
{"oldselect",   ALWAYS_ALLOW                            }, /* 82 */
{"symlink",     C_FS,           {FILE_R, FILE_W         }}, /* 83 TODO */
{"oldlstat",    C_FS,           {FILE_R                 }}, /* 84 */
{"readlink",    C_FS,           {FILE_R                 }}, /* 85 */
{"uselib",                                              }, /* 86 */
{"swapon",                                              }, /* 87 */
{"reboot",                                              }, /* 88 */
{"readdir",     ALWAYS_ALLOW                            }, /* 89 */
{"old_mmap",    C_MEM|C_FD,     {MEM_ADDR, MEM_LEN, 0, 0, FD    }}, /* 90 */
{"munmap",      C_MEM,  {MEM_ADDR, MEM_LEN              }}, /* 91 */
{"truncate",    C_FS,           {FILE_W                 }}, /* 92 */
...
{"eventfd",             ALWAYS_ALLOW                    }, /* 323 */
};
```

Bibliography

[Str,] Strace program. *http://sourceforge.net/projects/strace/*.

[Garfinkel, 2003] 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.

[van 't Noordende et al., 2007] van 't Noordende, G., Balogh, A., Hofman, R., Brazier, F., and Tanenbaum, A. (2007). A secure jailing system for confining untrusted applications. *Proc. 2nd International Conference on Security and Cryptography (SECRYPT), Barcelona, Spain*, pages 414–423.