**Introduction**

Ptrace is a system call provided by Linux and other UNIX-like operating systems. It was designed mainly for debugging, but it has been used for tracing purposes as well. Ptrace allows a process to observe and control the execution of another process; the first process is referred to as the tracer or monitor process and the latter as the tracee or target process.

The ptrace interface is:

**long** ptrace ( **enum** \_\_ptrace\_request request, **pid\_t** pid, **void** \*addr, **void** \*data)

The tracee needs to be attached to the tracer by issuing a request with *ptrace*; the details regarding setting up the tracing mechanism are introduced in [section 1]. All ptrace commands are per thread, thus in a multithread application each of its threads must be attached individually. This situation might be a source of potential race condition if not handled carefully. The [Tracing multithread application] section surveys different problems regarding tracing multithread application and provides an effective solution. In this chapter, the term multithread process/application refers to an application/process whose threads have been created using clone [ref clone] with the CLONE\_THREAD flag.

Once the tracing mechanism has been correctly initialized by the kernel, the tracee will stop each time a signal is delivered. The tracer will be notified via a wait system call and then it can control the execution of the tracee calling ptrace with the following requests:

* PTRACE\_SYSCALL : continues execution until next entry or exit from a system call.
* PTRACE\_CONTINUE : continues execution until the tracee receives a signal.
* PTRACE\_SINGLESTEP : continues execution until the next tracee instruction.

While the tracee is in a stopped state, the tracer is allowed to examine and change its memory and registers. Ptrace provides the following requests to perform these operations:

* PTRACE\_GETREGS/PTRACE\_SETREGS : respectively get or set the values of the general purpose register of the tracee process.
* PTRACE\_PEEKDATA /PTRACE\_POKEDATA: respectively write or read a word from the memory address space of the tracee process.

Accessing the tracee memory is necessary for retrieving the system call id (contained in the EAX register) and its arguments. This may be used in applications for auditing purposes such as strace [ref to strace], where all system calls and their arguments are recorded or for security purpose such as sandbox application [references] where each system call is tested against a policy specifying which arguments are allowed and which are not. Although this interface is easy to use, it provides very low bandwidth only 4 bytes per call can be retrieved. Each call implies two context switches to the kernel space and back, introducing an enormous overhead in the case of large buffers. This is one of the main limitations of ptrace and different techniques have been developed during the recent years to mitigate it. These are presented in the [Accessing tracee memory] section.

Ptrace has been successfully used to implement a user-space system call interceptor in numerous and different software, for example Orchestra (MVEC) [ref], Systrace (Sandbox) [ref], User Mode Linux (Software Virtualization), Goanna (Virtual File System). However, all these approaches suffer from different limitations due to the fact that ptrace has been designed for debugging single thread applications and not for tracing applications. An important requirement for an application such as a sanbox is the possibility to deny the execution of a system call when it does not satisfy the policy. This is also the base for building up a system call virtualization mechanism [ref UML], where all actual system calls must be nullified; see section [System call virtualisation section]. While other operating system such Solaris provides a way of aborting a system call, ptrace does not offer any. In the section [aborting system call] different methods to overcome this shortcoming are introduced.

Finally, another requirement missed by ptrace is the possibility to trace only a sub set of the system calls provided by the operating system and leave others to be executed without any overhead.

This reduces the tracing overhead in several different cases, for example, in a sandbox application where only the system calls which might affect the security of the Operating system should be intercepted. In [ref ?] all system calls which open or change a file descriptor (open, socket, bind) are intercepted, while those which use it (read, lseek, write) are not.

The major disadvantage of the ptrace approach is that the performance may be noticeably reduced in a program with uses intensively system calls. However, an interceptor mechanism can be used for really different purposes and therefore different criterion can be used for choosing it. In the case a performance is not a fundamental requirement such as in the debugging or security, ptrace is a valid solution. It provides an easy way to set up the tracing mechanism compared with kernel based mechanism and it allows to easy transport the interception infrastructure between different operating systems.

The remainder of the chapter introduces two different mechanisms for setting a system call interception using ptrace. The first consist of tracing all system calls made by a program, while the second is based on virtualizing all system calls. The rest of the chapter presents different approaches to the limitations introduced before.

**2.1 Ptrace tracing mechanism**

Although ptrace has been mainly designed for debugging purposes, it can be used successfully to build up a system call interceptor. To implement an effective system call interceptor, the tracer process must be able to intercept each system call made by the tracee process and its child processes.

There are two different ways to install the trace mechanism. The tracer might invoke the fork system call to create a child process which notifies the kernel its willingness to be traced by calling ptrace with the argument PTRACE\_TRACEME. This request causes the kernel to set a traced flag (PT\_TRACED) in the target process descriptor and stop it by setting its state to non-interruptible sleep. This approach is usually used before the tracing software initiates a new program invoking an exec system call. Alternatively, the monitor process can make a request to the kernel for tracing a running process via ptrace specifying the argument PTRACE\_ATTACH and the target process pid. In this case, there are three conditions to be checked. If the privileges of the tracer process allow it to trace the trace process and if the tracee process is being already traced and finally if the target process belongs to a set of process that cannot be traced ( init and itself). If all previous controls are satisfied, the monitor process becomes the father of the traced process and the traced flag is set on its process descriptor. Then, as the previous case, the tracee is put in a non-interruptible sleep state. This approach is useful when a process, which has been created by a different process from the tracer, has to be traced. This case happens each time the tracee spawns a new process or thread which must be traced as well.

It is worth noticing that in both cases the tracee becomes the father of the tracee process. In Linux a process can have at maximum one process father. Consequentially, this limits ptrace to track only one process per time making it a no-efficient mechanism for tracing multithread applications.

When the tracing mechanism has been correctly installed, all system calls made by the tracee are intercepted by the tracer. The tracee process is put in a no-interruptible sleep state by the kernel each time it receives a signal or attempts to invoke a system call. Once the tracee has been stopped, the tracer is notified via a signal SIGCHILD and then it is allowed to access the tracee address space. The tracer may deal with this signal either through a wait family-system call or installing a dedicated handler.

In the case of system call interceptor, the only relevant notifications are those regarding the attempt of invoking a system call. Precisely, the tracer should be notified twice, at the entry point before that the system calls is executed and at the exit point before the traced process is resumed.

The cause of the stop in the tracee can be identified through the status variable returned by the wait primitive or the integer argument in the handler function. The <signal.h> interface provides different macros to easily deal with this signal status variable [ref signal]. The tracee’s stop due to a system call invocation can be identified as follows:

WSTOPSIG(status) &(WITSTOPPED(status)== SIGTRAP)

Unfortunately, the status variable will assume the same value also when a SIGTRAP signal is delivered to the tracee for a different reason from invoking a system call. (WHICH?)

This ambiguity may be solved in two different ways. Further information about the source of the signal might be retrieved by issuing a ptrace request with PTRACE\_GETSIGINFO. Although this will introduce a further system call for each SIGTRAP signal delivered to the trace, which in the case of a traced process, are the most common. This performance decrease can be avoided using ptrace with the PTRACE\_O\_TRACESYSGOOD option. It ensures that when a signal trap is generated because the tracee is trying to make a system call the 7th bit of signal status variable is set. Therefore the tracer can easily identify this situation slightly changing the previous code to:

WSTOPSIG(status) &(WITSTOPPED(status)== SIGTRAP | 0x80)

Now, the tracer can skip all false notifications and analyses only those regarding a system call. The tracee can be resumed by calling ptrace with SYS\_SYSCALL as argument, its execution will continue until the next system call. The sequence of event triggered when the traced process invokes a system call is shown is Figure 1.

While the tracee process is a sleep state, its memory and registers can be accessed by the tracer process. The tracee memory is accessed, before the system call is executed, to retrieve the system call identifier and arguments. This is used in software such as sandbox [systrace, jail, c++] to check whether the system call invocation satisfies the requirements specified in a policy file or for auditing purposes in [systrace]. Furthermore, there is also the possibility to access the tracee memory when the process is stopped after the execution of the system call. This gives a means of analysing the system call return value, which is important for applying post policy rules in sandbox application [jailer]. Post policy rules are used for those system call where a sensitive value is known only after the call is executed such as accpet, recv.

Accessing the memory introduces always an overhead which has to be taken in account when performance is an important aspect of the application. Different approaches for accessing the memory of the tracee process are discussed in detail in [memory section].

**System virtualization:**

Virtualisation has become increasingly popular and a new demand for Linux is to act as a hypervisor. Different virtualization technologies have been developed to cope this increasing demand, such as instruction emulators QEMU, full or partial hardware emulation.

System call virtualization is a technique to emulate the execution of a system call made by a process.

Ptrace can be used to set up a system call virtualization mechanism in Linux. This approach has been followed by Jeff Dikk [ two papers] developing UML whose goals was to run Linux kernel in user space for easing the debugging of kernel.

A system call virtualisation needs a way to nullify system calls so that they would execute in such way as to cause no effects on the host. However, ptrace did not provide a means of aborting a system call [annulling a system call]. This shortcoming has been fixed introducing the possibility to change to change the actual system call. Annulling a system call can be done by substituting the actual system call with a getpid, which introduces a small overhead due to its execution.

When a system call is virtualised, there is no need to intercept the exit point because the system call has been nullified. This prevents the execution of two context switches between kernel space and user space which yields to a 50% improvement in the performance. All these ptrace enhancements have been introduced in the kernel mainline and they can be used by calling ptrace with the parameter SYSEMU [ptrace documentation]. As it has already been said, this prevents the system call to be executed and notifies the tracing parent only at system call entry.

The tracing mechanism can be installed in a similar manner has described in the previous chapter. The only difference is that when the traced software is resumed issuing ptrace with PTRACE\_SYSEMU parameter instead of PTRACE\_SYSCALL.

System call virtualization is implemented by the tracing thread intercepting and redirecting process system calls to the system call handler. It reads out the system call and its arguments, then annuls this in the host kernel and executes the virtual system call instead. When the emulated system call is finished; the return values are stored in the memory of the traced process.

This process is depicted in the figure.

An example of this approach can be found in [GOANA], where ptrace has been used to create a user-level file system development environment. The bulk of the application consists of a monitor called GOANA which intercepts all system calls made by the traced process. Once the system call has been intercepted this is substituted with a prototype function. This provides an easy way to test a file system prototype because we do not need to deal with massive body of kernel-level code and due to the fact it runs on user space the system can be analysed using a powerful debugger such as gdb which would have been impossible to use in kernel mode.

The performance is still the main problem of this approach as well. Even though by using SYSEMU we reduced the context switch between User Space and Kernel space to 2. There is still the necessity to access the memory at the system call entry point to retrieve its arguments and after the execution of the system call is finished to store the return values. This is a critical part of each monitor process, this problem as in the previous case will be treat in detail in the next section.

**Memory access:**

There are two kinds of arguments direct and indirect, the former is contained in the general purpose register, while the latter is contained in the process memory and its address can be found on the registers. The maximum argument number for a system call is six, so they fit in the register and we don t need to check the stack. What in case a system call with more arguments is added?

The monitor process is executed in user mode, it thus is not allowed to write or read the memory space of the traced process. The monitor needs to read the memory of the traced process in order to retrieve the contents of the indirect parameters such as strings or pointers. Also, it needs to write its memory address, for instance, when the system call is emulated and the outcome of the computation need to be stored in the traced memory space.

One method to access the memory of the tracee process is provided by ptrace. The monitor process may read from the monitor process calling ptrace with PTRACE\_PEEKDATA when the target process is suspended. The writing procedure can be performed in a similar manner by using PTRACE\_POKEDATA. Unfortunately, ptrace transfers only 4 bytes per time, therefore it has to be called many times to read or write a large amount of memory. Every call to ptrace requires a context switch from the monitor to the kernel and back, which decreases its performance since to make it, in fact, not a feasible way with a large amount of data.

In order to improve the performance in accessing the traced memory space different approach can be taken.

-easy way to access to the tracee registers

**Shared memory Orchestra and Jailer**

Attaching the shared memory to the target process, as well as reading from and writing to it is not within the application which is being traced. Therefore the monitor process has to ensure that the target process performs these operations. In [Jailer] the shared memory is set up using a preload library technique. The preload library uses mmap() to read-only map the memory region in the target’s process space , in addition, it loads some code routine. Another technique presented in [Orchestra] consists of replacing the first system call made by the target process with an appropriate one such as shmget, ipc after making a backup of the values contained in the registers. This makes the target to run the new system call and attaching the shared memory to the target process. After performing this operation, the original system call is restored by using the register values previously saved. In both cases a small code is injected to the space memory which copies the content of a buffer to another one. When the monitor process needs to access to an indirect arguments, it can retrieve the base address from the registers by using ptrace and then use this routine to copy all the buffer in the share memory. In the Jailer system, the values of registers are modified to point to this memory area for security reasons [See race condition].

-excellent for accessing big buffer

**FIFO**

Another approach proposed in [Orchestra] is to use FIFO structures. FIFO is inter-process communication which allows the monitor process to communicate to the target process by writing to and reading from the FIFO. The pipe can be created using the system call mkfifo() [ref] and then it can be managed via I/O system calls usually used to deal with files (in fact the FIFO is a file in the file system).

The target process can easily open the FIFO by using the FIFO’s name as parameters calling *open*. While, if the target process has been spawned by the monitor process, it inheritances the file descriptor corresponding to the FIFO open by monitor process. However, there is still need to use a routine to copy the values from the target process space to the pipe. Other ipc mechanisms such message queue [jailer] can be used in a similar fashion.

Shared memory has better performance.

**/Proc/$pid/mem**

This is the most important because it provides a good solution.

**Multi thread applications**

Nowadays, most the applications are multithread. For example, a web server usually creates a new thread for each incoming request and let this newly created thread to accomplish it. This family of applications must be taken into account as possible target process for a system call interceptor and this requires a thoroughly analysis. As we already said in the introduction part, one of the most import requirements of a system call interceptor is to provide a means of intercepting all processes spawned by the traced process. Ptrace misses this aspect because it provides a tracing mechanism for single thread [ptrace documentation]; where each thread must be attached individually to the tracer process issuing a request with PTRACE\_ATTACH. Moreover, a child process does not inherit the trace flag from the parent, so it can run untraced until the father attaches to it. These flaws make tracking multithreaded application difficult and error prone [race condition paper], so that some applications [jailer] decided to not support this kind of applications.

Nevertheless, ptrace offers a solution, though it is quite elementary. Setting the option PTRACE\_O\_TRACECLONE [insert a note for other arguments], the tracee will be stopped at the next clone() function and the tracer will start tracing the newly cloned process. This solution, in fact, is not a feasible way to trace multithread application when performance are important because each time a signal is delivered to a traced thread, all threads in the traced poll enter in a stop state.

This introduces an unnecessary latency because the other threads (those that have not received any signal) must wait until the event is processed by the tracer and the entire application is resumed before being able to carry on their tasks. However, in application where performance is not important, for instance debuggers, this represents a valid solution. In fact, this approach is used by GDB. Another issue related with this approach is that ptrace does not always guarantee that the forked processes are always automatically traced. Linux allows setting a flag on clone() (CLONE\_UNTRACED) , which determines whether the child process can be traced or not. However, this can be easily solved noticing that the clone function will be intercepted by the monitor program and it can modify its arguments changing that flag to CLONE\_PTRACE which makes the process traceable.

The common approach to deal with multithread application is to create a new tracer process [jain and sekar, jailer, orchestra, systrace,strace] for each new tracee process. Unfortunately, this solution raises a race condition which might yield to some system calls to escape from being intercepted. The new monitor is not allowed to trace the newly thread until the parent monitor detaches from it. When the parent monitors detaches from it the kernel sends a message to the tracee process and allows it to continue its execution. This leaves a small temporal window within some system calls might escape. This is critical problem in application such as sandbox, where all system calls must be intercepted for obvious security reason. A completely different approach is taken by strace where this problem is not tackled because a missed system will not comprise the entire application, while solving it will complicate the code, and the temporal window is rather small and the probability of missing a system call or more should be quite low.

A possible kernel enhancement, which solves this, would be to notify to the tracer when the newly thread has been created but it is not yet running. This would give the opportunity to safely attach a new tracer to the newly created thread without losing any call. Instead the tracer receives a tracing event only on the child’s subsequent system call.

This problem has been addressed successfully in [Orchestra] as follows. When the tracee process spawns a new thread, automatically it is stopped by the kernel and the monitor start monitoring the newly created thread. At its first system call invocation, the monitor makes a backup of all new tracee process registers and then it substitutes the requested system call with a pause system call and finally lets it continue its execution. When the new process is resumed, it will enter in a pause state due to the injected system call. While this process is in the sleep state, the monitor is allowed to detach from it and the new monitor can safely attach to it. Once the new monitor has successfully attached, it restores the previous system call so that the execution of the new tracee process can continue. Although this solution works well, it introduces a small time overhead due to the execution of the pause system. Moreover, a communication between the new monitor and its father is needed to retrieve the backup information regarding the execution context of the new thread in order to restore the previous system call.

Race condition linked to the multithreads application; at least I have to insert the reference to the papers.

**Aborting a system call**

Having the possibility to abort a system call is an important feature. Let’s consider the following scenario. A large application such as database has been sandboxed for security reason. It is possible that due to the nature of the application and the high number of system call requests, some policy checks fail. When this happen the system call should be nullified, but if this feature is not provided the only solution is to terminate the application even if there is not any real threat.

Unfortunately Linux doesn't provide any means of aborting a system call. This lack has been one of the main reason because ptrace was not considerate a feasible way to implement a system call interceptor [Janus thesis].

Different solutions have been propose [C++ sandbox] and [Goanna], but this problem has been solved by [Uml Guy] who implemented a patch for the Linux kernel. This patch allows the tracer to substitute the invoked system call with another one by inserting a different system call id in the EAX register. Leveraging on the enhancement introduced for UML, the tracee’s system call can be substituted with the low-overhead system call getpid.

This technique for aborting a system call raises another problem, which value should be returned?

One possibility is EINTR which represent the error code for system call interrupted. However, this is not a good choice because some application may be coded in such a way that, if they receive EINTR error they will retry the system call. If this happens in a loop the application gets stuck and the only way to break that loop is to kill the entire application. A better error code is EPERM, which stands for operation not permitted.

Goanna nullifies a system call by setting EAX=-1

**Conclusion**