Tracy: UNIX system call tracing

Jurriaan Bremer Merlijn Wajer Bas Weelinck

March 11, 2015

Abstract

Presented is a uniform interface to trace the behaviour of programs by means of the system calls they perform. Tracing by the user is done without regard to kernel version, operating system or processor architecture. The interface, called Tracy, provides a means to watch, modify, augment and restrict program execution in a controlled environment.

Contents

Chapter 1

Introduction

Programmers spend rather a large part of their time finding mistakes in their computer programs. They usually refer to this routine as debugging and they employ various tools to aid them in debugging programs.

Several kinds of debugging exist. The most common debugging method is focussed around crashes in a program. Other types of debugging include performance analysis and analysis of system calls performed by the program. A system call is a method for a program to request a service from an operating system's kernel. The kernel of an operating system is the main component of most operating systems; it is a bridge between the hardware and programs.

Inspecting (and modifying) the system calls performed by a program is done programmatically using the **ptrace** system call. A program inspecting the system calls of another program is said be to be "tracing" the other program. The program that is being traced is called the "tracee" and the program tracing the "tracee" is called the "tracer". Tracing is a limited definition in the sense that the term does not mention the possibility to modify the program that is being traced. Using **ptrace** the tracer can also modify the tracee by reading and writing to the memory of the tracee or even changing the system calls made by the tracee.

1.1 Motivation

The ability to modify the program memory and registers [1] allows for a great variety of applications. A possible application is the so called "jailing" of programs - only allow system calls that satisfy certain requirements and modify the arguments to some system calls to effectively restrict the access of a program to the system.

Other applications are transparently routing or inspecting all the network traffic of a program. Being able to modify the system call arguments and return value also allows for extensive stress testing of programs, one could fake being out of available memory for example; or even randomly reject or fail system calls.

Tracing programs is done using the **ptrace** system call on most[2] UNIX

[1] Changing cpu registers allows for system call modification

[2] If the UNIX system supports ptrace.

systems. Using ptrace directly to trace a program has several downsides: the ptrace interface is not very programmer friendly: the interface is not standardised (ptrace is not part of the POSIX[5] standard) and ptrace is not architecture agnostic, meaning a tracer requires architecture specific code.

The Tracy research project has as its aim the the production of a cross architecture, programmer friendly system call tracing and modification framework for the Linux kernel. During this research, we put emphasis on a few key areas:

- Safely tracing a program. A program should not be able execute system calls that undetected.
- Creation of an API that is simple to use; yet provides very powerful functionality such as the injection of system calls.
- Aforementioned API should be cpu architecture agnostic.
- Aforementioned API should be operating system agnostic, with the only limitation that the operating system is a modern, UNIX-like operating system with support for the ptrace system call.

1.1.1 Problems With Ptrace

First of all, we wanted to solve problems introduced by utilising ptrace directly. As we have previously noted, a particular problem with ptrace is that the ptrace interface to the programmer differs per platform. This means that ptrace code written for Linux will most likely not work on another UNIX operating system such as FreeBSD.

Different UNIX platforms also support different ptrace features. FreeBSD offers a ptrace option to quickly read or write a large amount of memory of the tracee, whereas Linux does not support reading more than a processor word at a time. On the other hand, Linux has an ptrace option to automatically trace any child processes created by a tracee; FreeBSD currently does not offer such an option (Section 3.8.2).

The system call invocation differs per computer architecture. The assembly instruction to invoke a system call usually differs. Kernels like Linux and FreeBSD sometimes even support several ways to invoke system calls. ptrace does not provide a way for the tracer to detect how the tracee invoked the system call. This is problematic because the meaning of system call numbers differs per system call invocation thus leading to inevitable amiguity about the current system call being performed (Section 5.1).

The fact is that ptrace is a very low-level system call, which operates on assembly level, therefore it is not possible to simply "change the return value" of a system call. Each instruction set has its own registers and on top of that, operating systems often have different uses for each register.

One would have to find out what register this is on each platform, keep track if the current ptrace event describes the start or end of a system call[3], let alone perform some extra architecture-specific calls to ensure that the result is properly set.[4]

[3] If the process is about to execute a system call or has just executed a system call

[4] ARM requires the programmer to call ptrace with the *PTRACE_SET_SYSCALL* command to make the changes to the system call register permanent.

Finally, ptrace offers no mechanism to recieve events of specific system calls, rather than on every system call. This leads to performance issues (Section 5.5).

1.1.2 LD_PRELOAD

LD_PRELOAD is a way to prioritize loading of certain libraries while creating and loading a process. The *LD_PRELOAD* environment variable controls which libraries are loaded into a process before all other libraries are loaded.

One can use *LD_PRELOAD* to load a shared object (a library) into a process, before any other libraries (such as glibc) are loaded; this makes it possible to (transparently) override functions typically provided by other libraries.

This mechanism can be used (amongst other) to create a proxyfier. Such an "injected" library could provide its own "read" and "write" functions which will be called by the program in favour of the default read and write functions.

One common proxyfier, *tsocks*, is an application that loads other applications with their own library that overwrites specific network functionality, thus allowing transparent network routing.

LD_PRELOAD relies on the dynamic loader and will only have any effect if the programs have to make use of the libraries being loaded.

A downside to this approach is that *LD_PRELOAD* simply does not work on all programs; some programs do not use glibc methods and instead perform system calls directly, for example by directly using assembly in their program. Other programs are simply statically linked; which means libraries were linked into the program at the time of the programs creation rather than when the program is loaded; this makes *LD_PRELOAD* useless in this case. Theoretically it should be able to prioritize library loading even with statically linked libraries, but this is beyond the scope of this research.

There are some upcoming languages that do not even use glibc at all - languages like Go [4] talk to the kernel directly and thus perform their system calls by calling the kernel directly instead of relying on (g)libc functions which would in turn call the kernel.

LD_PRELOAD is not a viable solution when one wants to transparently capture all (specific) system calls of a process, independent on what kind of userspace libraries the process uses.

1.2 Structure of This Document

In Chapter 2 we introduce Tracy, our ptrace based system call tracing and modification library. An in depth explanation on how we implemented Tracy and worked around ptrace issues is presented in Chapter 3. In Chapter 4, we present an application of Tracy called Soxy: a transparent SOCKS5 (Section 4.2.1 proxifier Chapter 5 presents an overview of features that Tracy currently lacks and features we would like to see implemented in Tracy. We complete the paper with recap of the research, with a focus on goals set by the Tracy research project.

1.3 Terminology

We introduce a few terms that the reader will encounter throughout the document. We assume that the reader is a least a bit familiar with the UNIX family of operating systems.

- ABI: Application Binary Interface. Describes the low-level interface between computer programs and the kernel.
- Tracer: The program tracing other programs.
- Tracee: A program that is being traced by a Tracer.

Throughout the document, all system calls are identified by bold typesetting. ptrace options are typically italic and can their description can be found in Appendix B or in the ptrace manual page.

Chapter 2

Tracy and Ptrace

This chapter provides a theoretical introduction to ptrace and its problems for cross platform and cross architecture programming.

2.1 Introduction to Tracy

Several UNIX and UNIX-like systems support the **ptrace** system call, as introduced in Section 1.1.

Tracy is a library that uses **ptrace** (See Section **??**) to trace the system calls of a process. Tracy can inspect, modify and even inject system calls. Being able to modify system calls gives the controlling process the ability to change arguments of system calls before system calls are executed - and even the ability to "deny" system calls by changing the system call to a harmless one such as **getpid**.

Obvious use cases for modifying system calls is a process called "sandboxing": intercepting and changing special system calls like **open** and prepend (if not already in place) a specific path to the arguments such that the tracee cannot open files outside a specified path.

ptrace is very platform and operating system dependent. The Tracy API strives to be platform and operating system independent by applying architecture and operating system dependent hacks and fixes "behind the scenes". We have tried to design Tracy to be as portable as possible - meaning that code for computer architectures such as ARM and x86 should be nearly (if not completely) the same. Tracy elegantly works around most architecture specific issues and implements functionality required to trace programs on several operating systems.

Tracy introduces an API that allows a programmer to hook into specific system calls, rather than every system call as is the case with ptrace; ptrace does not differentiate per system call and delivers an event on each and every system call.[1] Tracy also has both a synchronous and asynchronous injection API (Section 3.4) and Tracy keeps track of the state of a system call: if the event is an event generated before a system call is executed; or after a system call is executed.

[1] Of course, under the hood Tracy will have to handle every system call, whether the system call is hooked or not. We propose a fix to this problem in Section 5.5

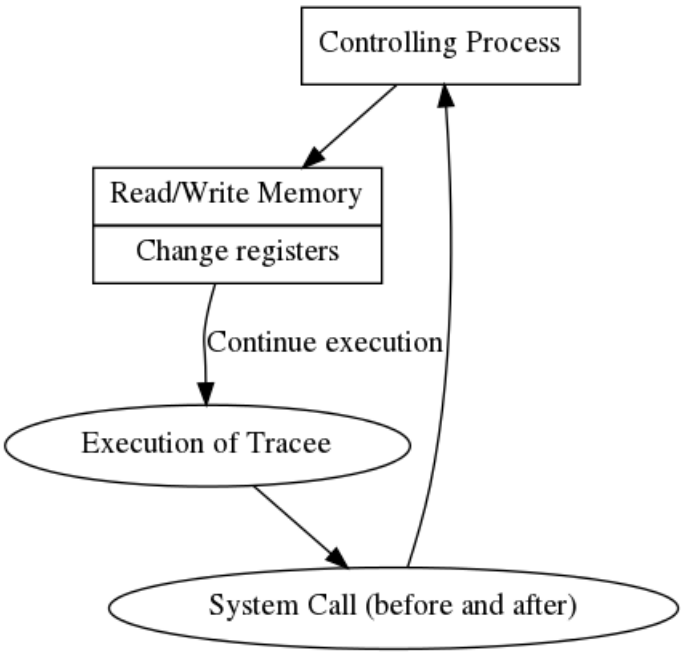


Figure 2.1: Ptrace control flow

It can be useful to inspect and modify data pointed to by a system call argument of tracee. To transparently and safely change these additional hacks are required: both changing the contents directly as well as validating the value when the child can still change the value are generally bad ideas: the first interferes with the normal execution of the process (although this may be exactly what the programmer wants) and the second is sensitive to race conditions. A solution is presented in [8] and the feature in Tracy is planned, see Section 5.2).

Tracy has functionality to allocate memory in the tracee which the tracee cannot write but only read; thus allowing the tracer to copy arguments to that memory space, validate and change them as necessary; finally change the pointer in the arguments to point to the memory space and continue the system call as normal.

2.2 System Call Tracing, Modification and Injection

ptrace suspends the tracee and signals the tracer right before the tracee executes a system call as well as just after the tracee has executed a system call. When the tracee is stopped before a system call, we say that the tracee is in *pre* system call state; if the system call has executed and the tracee has once again been suspended we say that the tracee is in *post* system call state. While the tracer is inspecting the tracee, the execution of the tracee is paused. (Figure 2.1)

When the tracee is suspended, the tracer can read and modify registers

as well as read and write to the memory of the tracee. Combining these two features, the tracer can:

- Modify the system call (number) that is executed.
- Modify the arguments to system calls of the tracee.
- Modify the instruction pointer (commonly called program counter), allowing the tracer to resume the executing of the tracee at an entirely different set of instructions.
- Inject system calls, by modifying the instruction pointer and changing the system call number.
- Modify the return value of the system call.

2.3 Memory Access

To be able to fully modify a tracee, the tracer needs to be able to read and write the tracee's memory. If the tracer is to change arguments that point to a specific page in the tracee's memory - like a string, the tracer needs to be able to write to the tracee's memory - or share a small part of the tracers own memory with the tracee.

Changing the memory of the tracee is usually not a good idea, as that could interfere with the tracee's execution. Instead, it would be a better idea to allocate a few pages in the child and copy the data to those pages, change the argument to point to the new pages and continue the system call. After the system call has completed, the pages can be freed again. This way the child's original arguments are left untouched, and for a reason: the child may want to re-use the memory later on.

Allocating pages in the tracee as described above is problematic if one wants to make sure the tracee does not change the memory contents. ptrace suspends only one process and each thread on Linux is a seperate process. Threads share memory with other threads in the thread group and can thus write to the same memory. If one thread is suspended, another thread can still write to the newly allocated pages and change the data stored in the pages just before the system call is executed and after the tracer has written data to the page. To safely share memory, see Section 5.2.

2.4 Tracing Children

A tracee can spawn children by calling the **fork**[2] system call. **ptrace** does not automatically trace children of a tracee.

In certain cases, one may want to trace all the children of a tracee. To do this, Linux supports specific **ptrace** options to automatically trace the tree of children spawned by the tracee as well (See Section 3.2).

However, other platforms do not support these options and tracing children of the tracee can be somewhat problematic. strace, a ptrace-based system and

[2]among other system calls: vfork, rfork (BSD), clone (Linux)

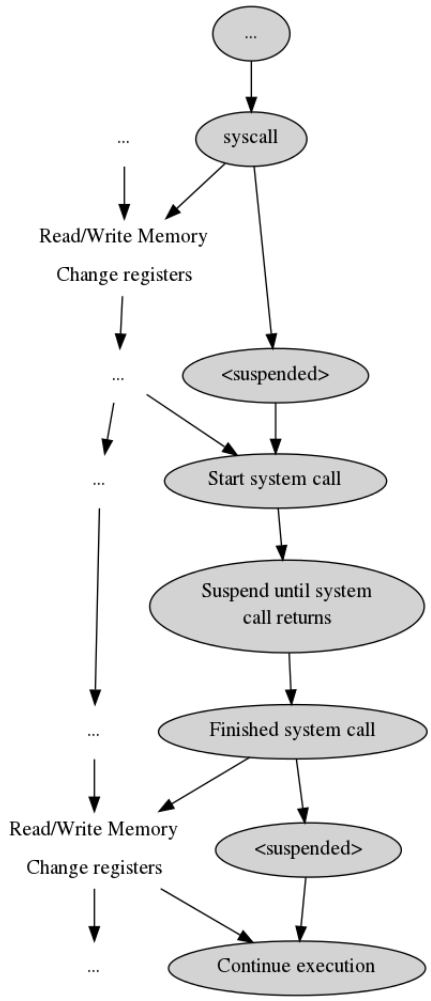


Figure 2.2: One system call with ptrace

signal tracer does not trace every child immediately on creation, from the strace manual page:

> On non-Linux platforms the new process is attached to as soon as its pid is known (through the return value of fork in the parent process). This means that such children may run uncontrolled for a while (especially in the case of a vfork), until the parent is scheduled again to complete its (v)fork call.

We provide a solution to this in Tracy such that any children spawned by the tracee will not run uncontrolled on any platform in Section 3.8.2.

Chapter 3

Tracy Implementation

This chapter covers the implementation of Tracy using the ptrace api. Provided in Appendix B is an overview of most of the ptrace api. We will discuss problems we ran into while implementing Tracy, with a focus on ptrace limitations and caveats.

3.1 Tracing a Process

Tracing a process can be done in two different ways. A process can either attach to another running process, or create a child process and consequently use **ptrace** in the child to make the parent (the original process, soon to be the *tracer*) trace the child after which the child will call **execve** to run another program.

3.1.1 Fork and Trace

As explained in the previous section, the parent issues a **fork** call. The child then performs a **ptrace** call, with the *PTRACE_TRACEME* argument.

The child then (by Tracy design) sends itself a *SIGTRAP* signal; and because the child is being traced by its parent, the child is suspended until the parent performs a **wait** followed by a call to ptrace to continue the process with the *PTRACE_SYSCALL* argument to **ptrace**. The parent can inspect and change the child before it issues the continue call. The code demonstrating this can be found in Appendix A.

3.1.2 Attaching

Attaching to an existing process is done using the *PTRACE_ATTACH* mechanism, which will attach a running process specified by the process id, granted that the user has sufficient permissions to trace the process.

3.2 System Call Tracing

Tracing system calls with **ptrace** is done by issuing ptrace on the tracee with the *PTRACE_SYSCALL* option, this will make the tracee suspend on entry

(pre) and exit (post) of the next system call. Resuming a suspended tracee is done with the same option. Tracy uses this option on all the tracees.

Tracy keeps track of each child; it has to store information about the state of the child: will the next stop be a system call entry or exit, or are we currently injecting a system call? Aside from those two states, Tracy also has to do some other internal bookkeeping such as store the number of the system call that is being denied.

3.3 System Call Modification

Modifying system calls can be done at two points: before and after the system call is executed. Changing the system call "after" it has been executed does not undo the effect of said system call, but it allows the tracer to change result value of the system call. [1] Changing values before the system call is executed is more exciting, as it allows the tracer to actually *change* the system call. For example, by modifying the register that stores the system call number, it is possible to execute a completely different system call. This serves two purposes: "denying" system calls by changing the system call register to a harmless system call such as getpid and "injecting" system calls by changing the system call number and later on the instruction pointer (See Section refsyscal-inject).

3.3.1 System Call Hooks

Users of the Tracy library can hook specific and even all system calls. For each system call, the user provides a callback which will be called when the related system call occurs. The callback is provided with full access to event. The return value of the callback determines the action that Tracy will take. A callback for signals also exists as well a callback default callback that is called for all the system calls that are not hooked.

3.4 System Call Injection

Tracy supports injecting system calls into any process that is being traced. The injection of a system call is the process of executing a system call in a specified tracee, without requiring said tracee to explicitly call the system call.

Theoretically a system call can be injected at any point during the execution (as discussed in Section 5.4), Tracy however can currently inject system calls in a pre and post system call state. In other words, one can currently only inject system calls when a tracee is stopped due to the tracee executing a system call; not if the tracee is recieving a signal.

Transparent injection of system calls is not possible, in the most strict sense. As most system calls indirectly affect the tracee (with the exception of system calls like **getpid** and **fstat**), the tracee will usually be left in a modified state. However, to perform system call injection as transparently as possible, we ensured that the registers before the injection are identical to the registers after the injection.

[1] The tracer can of course change every register and the memory of the tracee at any point when the tracee is suspended.

Injecting system calls relies on that fact that the program counter (or instruction pointer) of the tracee can be modified. If the length of the system call instruction is known, it is possible to "jump back" to the system call instruction, as can be observed in Listing 3.1. Once the "int 80h" has finished; the system call has completed; if we put the process instruction pointer back 2 bytes (the size of the "int 80h" instruction), the program will execute another system call. The system call is determined from the contents of the *eax* register (which we can change), therefore we can execute any system call by simply changing the contents of the *eax* registers and is required the registers that hold the system call arguments.

Listing 3.1: Intel x86 Assembly that performs an "exit" system call. Taken from `http://www.fizik.itu.edu.tr/turhan/asm/mnasm.html`

```
1       mov eax,1               ; 'exit' system call
2       mov ebx,0               ; exit with error code 0
3       int 80h                 ; call the kernel
```

While the process of injecting a system call varies for each system call state - pre or post - the idea is the same. Figure 3.1 shows the injection process for both pre and post system call states. "Modify IP/PC" represents jumping back to the system call instruction. Note that this instruction and the size of the instruction differ per platform.

3.4.1 Asynchronous Injection in Tracy

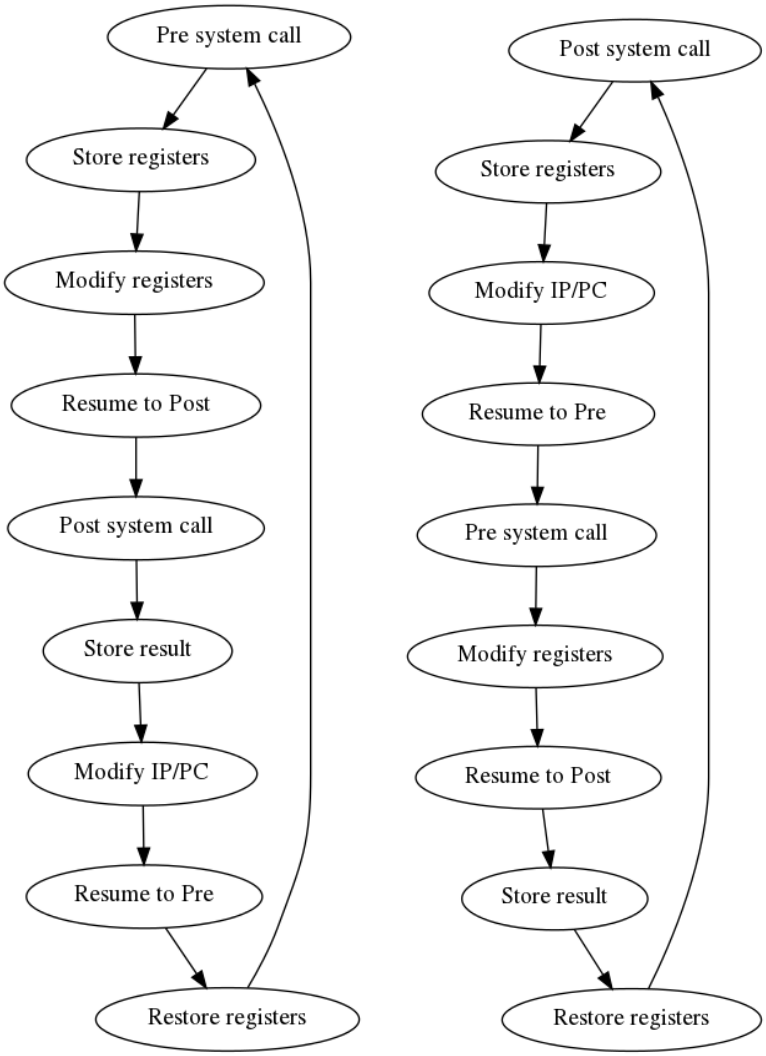
Tracy supports asynchronous injection. Typically, when injecting a system call in a tracee, Tracy waits for the system call that is being injected to complete and then restores the tracee to its previous state. This process can be split into two stages, readying the tracee to perform a system call and restoring the tracee to its original state. No system call returns instantly and it may be favourable to perform other tasks rather than waiting for the system call to complete.

Asynchronous injection is a way to inject system calls that allows Tracy to handle system calls by other processes while Tracy is waiting for the injected system call to finish, this can be useful as some system calls can take quite some time. Once process A has finished its (injected) system call, Tracy will store the return value and restore the process to its original state. Functionally this makes no difference to the injection of a system call, but it does allow Tracy to handle the requests of other (suspended) processes instead of simply waiting for the system call of the tracee to finish.

While asynchronous injection is usually preferred, synchronous injection is supported by Tracy as well and even used in Tracy internally where asynchronous injection simply is not worth the extra work and effort.

3.5 Memory Access

Any program that uses the **ptrace** API will probably want to access the traced process' memory as well. Reasons can range from simply dumping pointer contents to monitoring, changing or even injecting new system calls with completely stand-alone data.



(a) Injection from a pre state (b) Injection from a post state

Figure 3.1: Injection for pre and post system call states

Unfortunately as with most ptrace functionality, memory access standards differ from operating system to operating system and are also affected by architectural quirks. Tracy hides these differences and provides a uniform and fast way to access memory.

3.5.1 On Linux

On Linux Tracy achieves fast memory access by employing a feature of the "/proc" filesystem that was designed to work in unison with **ptrace**.

The classical way of accessing child memory on Linux was through the *PTRACE_PEEKDATA* and *PTRACE_POKEDATA* operations. The downside of this method is its dependence on C's long type and the amount of data that can be transferred. These ptrace operations are used to respectively read and write individual processor words.

On Linux the C "long" type has the same size as the processor word size, so 32 bits on a 32-bit architecture and 64 bits on a 64-bit architecture.

This is where the trouble starts, not only is it not possible to simply read or write single bytes of memory, whenever one wants to access say 4kb of memory, depending on the architecture, this may require more than 1024 calls to ptrace. The amount of time spent performing ptrace system calls.

Alternative: Using /proc

Linux provides another way to access memory of a process. The "/proc" filesystem, used by a lot of utilities to provide information about kernel and process state can also be used to access arbitrary process memory, given certain (security related) requirements are met. Process information can usually be acquired by opening a folder named "/proc/$< pid >$" where "$< pid >$" must be substituted with the process identifier. In this folder several files, and folders, are contained which provide information on things such as, threads, open files, memory maps and memory contents.

The "/proc/$< pid >$/mem" file is the one used by Tracy to access child memory. Opening this file will present child memory as one contiguous file which can be modified using **read** and **write**..

This also means we can now write single bytes of memory without the need of first reading a processor word. We can now read 4kB of memory with a single system call or any arbitrary amount of memory provided we can allocate a large enough buffer to contain the data.

One cannot simply open the "/proc/$< pid >$/mem" file of any arbitrary process as this would cause a major security issue. The only processes capable of successfully opening this file are either processes tracing the target process, or super-user processes.

3.6 Event System

Tracy uses an event-based system to report on activities on a tracee. This is particularly fitting as the **waitpid** call blocks until an event occurs. As a result Tracy blocks until a new event occurs.

Each signal and system call (pre and post) is an event. The Tracy function *tracy_wait_event* waits for a action on (a specific or) any child and returns a new Tracy event.

Aside from events for signals and system calls, Tracy also exposes another event; an internal event which describes an internal Tracy event; this event is required for asynchronous injection and may be used for other features in the future.

3.7 Signals

Tracy obtains information about the signals directly from the **waitpid** call. The waitpid call has an argument called *status* which contains the information about the signal that was received. Retrieving the signal sent to the tracee is done using the **WSTOPSIG** macro.

To pass a signal to a tracee, Tracy passes the signal number as data argument to the ptrace *PTRACE_SYSCALL* api call. This will resume the execution of the tracee with said signal. If the data is left at zero, no signal is delivered. This makes it possible to suppress certain signals by simply not passing them along to the tracee.

3.8 Tracing Children

Tracing the child processes of a tracee very useful; it allows a tracer to truly trace everything a process (and its descendants) does. On Linux, tracing all the children of a tracee is relatively easy.

3.8.1 Linux

Linux 2.6 and onwards offer options to automatically trace all children created by **fork**, **vfork** and **clone**. Respectively: *PTRACE_O_TRACEVFORK*, *PTRACE_O_TRACEFORK*, *PTRACE_O_TRACECLONE* flags are passed to **ptrace** to enable tracing of children.

3.8.2 Safe Execution of fork, vfork

On operating systems that do not support the *PTRACE_O_TRACE** options, another solution is required to ensure that all children are traced the moment they are created. To achieve this, the **fork** (as well as vfork and clone) system call must be executed in a controlled manner.

To implement this feature, we make use of system call injection as well as the ability to be able to allocate and write to pages in the tracee. We allocate a page in the tracee and write a few lines of assembly that will safely execute **fork** to the page. Once this the assembly has been written, we deny the initial **fork** call by changing the system call to **getpid**. In the POST-getpid callback, we change the Instruction Pointer (or Program Counter) to the first instruction in the newly allocated page.

Once the tracee starts executing the instructions in this page, the tracee will once again execute a system call - **fork** in this case, and this time we allow the

call to proceed. However, after the **fork** has completed and we have stored the result of the call (the process id), both the tracee and the child execute the rest of the instructions in the page.

The only other instructions in the page are a busy while loop that calls the **sched_yield** system call. Now that both the tracee and the child of the tracee are caught in this while loop.

At this point we are still only tracing the tracee, but we can now use the "attach" mechanism of ptrace to attach to the child of our tracee. Once we are tracing the child of the tracee as well, we can restore the Instruction Pointer (or Program Counter) of each tracee to their original position (that is, just after the original **fork** call) and allow both processes continue their execution.

3.8.3 Caveats

When the feature to automatically trace newly created children in Linux is turned on, the tracer will recieve a SIGSTOP signal event for the tracee that has just been created. This signal is not passed along to the tracee. For this reason, Tracy suppresses the first SIGSTOP that is sent to a newly create tracee if the automatically trace children option is enabled.

3.9 Threaded Tracees

Threads on Linux are processes that share the same memory space and file descriptor table. If a thread in a thread group of which all the threads are being traced is suspended by ptrace; none of the other threads in the thread group are suspended. Each thread in the thread group have the same process ID but a unique thread ID; see the manual page of **gettid**. Threads (and thus, processes) on Linux are created using the Linux-specific **clone** system call[2]. The **clone** system call allows the programmer to specify if the created process shares certain attributes with its parent, such as the memory space, file descriptor tables and file system attributes.

To reliably trace opened files and sockets, programmers need to know what threads (or more general: processes) share file descriptor tables (and memory). To complicate matters, the **unshare** system call allows a process to venerate its file descriptor table as well as its file system attributes. When a process (thread) separates its file descriptor table, all the file descriptors are cloned into a new table, which is then unique to the calling process.

At the time of writing Tracy does yet implement helper functions to easily find out how processes relate to each other and what resources they share with other processes. We plan to add such functionality in the future.

3.10 Performance in Ptrace

When a tracee performs a lot of system calls, the overhead of **ptrace** becomes very noticeable. The main issue is that for every system call the kernel has to perform four context switches. The first switch is suspending the tracee and replacing the tracee with the tracer (Tracy), then Tracy issues a few (at

[2] glibc uses clone() if the user invokes fork()

least two) system calls (which require entering kernel mode), the kernel then resumes the tracee (another context switch); once the system call of the tracee is completed, the tracee is suspended again and the tracer takes over (context switch number three), again performing at least two ptrace system calls and finally the tracee is resumed again (context switch number four).

This entire process is repeated for every system call, be it the inexpensive **getpid** system call or the fairly expensive **clone** system call, they are all a lot slower for the tracee due to this overhead of suspending and resuming the tracee.

The problem is that **ptrace** does not allow the reporting of specific system calls only - the tracer has to handle every single system call event, even if the tracer will take no other action than resume the tracee on most of the system calls.

We present a (theoretical) solution to this problem in Section 5.5.

3.11 Caveats

While implementing Tracy and more specifically our "Safe Execution of fork and vfork" (Section 3.8.2) code, we ran into a few problems. The clone system call does not take the same arguments on different Linux platforms and retrieving the process id from the vfork system call proved to be another challenge.

3.11.1 clone

The Linux specific **clone** system call is a bit problematic, for several reasons. The clone system call interface differs per architecture; on x86 clone has one argument extra compared to AMD64; and Linux on x86 also *swaps* two argument registers [3].

3.11.2 vfork

We ran into a slight problem with vfork in our safe execution (Section 3.8.2) feature: vfork only returns the pid of the child after the child has executed an **exevce** call; our safe execution of fork feature relied on the fact that we could imprison both the parent and then child, first read the pid from the return value of fork (and clone). With vfork, we had to perform additional hacks. We modified the execution environment of the child to include a call to **kill** with Tracy as a target (and the signal set to *SIGUSR1* to notify Tracy the tracer) of the pid of the new child, after which we attach to the child and remove it from the safe execution environment as normal.

[3] This can be observed in Linux 3.4 *arch/x86/kernel/entry_32.S* and *arch/x86/kernel/process.c*

Chapter 4

Soxy

In order to show the power and features of Tracy, as well as finding bugs and improving the API, we have developed Soxy, a proxifier [1] on top of Tracy which works solely on existing binaries (that is, executables which do not provide such functionality natively.)

Soxy has the unique ability to tunnel all (or a subset of) the network traffic created by a process (either a child processes or a running processes) over a proxy based on the SOCKS5 [2] protocol.

The server which will be used as proxy server must be configured to accept and tunnel SOCKS5 data. There are several proxy daemons available, including `ssh`.

4.1 Python Bindings

Besides Soxy, which demonstrates an example usage of the Tracy library, we also provide Python bindings, allowing rapid development of applications which utilize Tracy. As a matter of facts, Soxy has been implemented using the Python bindings, ensuring the correctness and efficiency of these bindings.

The Python bindings provide a similar, but even easier, API for using Tracy.

For example, a pre-syscall hook on *write* which duplicates the write system call (i.e. write the same string twice) looks like the following.

```
def hook_write(child, event, args):
    if child.pre:
        child.inject(args.syscall, args)
```

4.2 Soxy Internals

4.2.1 SOCKS5

Complete documentation of the SOCKS5 algorithm can be found in RFC 1928 [3]. The global flow for TCP connections is as follows.

[1] An application which tunnels internet traffic through another server
[2] RFC 1928 - SOCKS5 Protocol
[3] RFC 1928 - SOCKS5 Protocol

- Application connects to the Proxy Server
- Optionally the application performs some sort of authentication
- A request containing the destination address is sent
- Server replies with success or failure
- Proxy connection has been established

After the proxy connection has been established, the protocol does not need any further work, because all incoming and outgoing traffic goes through the proxy server.

4.2.2 Implementation

Soxy implements the SOCKS5 protocol as follows, using hooks on the *socket* and *connect* system calls. When a socket is created, using the socket system call, Soxy determines whether the socket is TCP or UDP based. Following, when the socket attempts to connect to another machine, using the connect system call, Soxy injects a few system calls in order to connect to the proxy server, authenticate, and establish the proxy connection.

4.2.3 Asynchronous Sockets

Sockets are synchronous by default, also known as *blocking* sockets. When performing an operation on a blocking socket, the thread will hang until the operation has been finished or an error code can be returned. Applications which handle multiple sockets at once, however, tend to use asynchronous sockets, also known as *non-blocking* sockets. Non-blocking sockets return immediately when operating on them, which allows an application to handle multiple sockets at once, without hanging the application.

For Soxy this means that the *connect* system call returns immediately. However, Soxy attempts to establish a connection to the proxy server and perform authentication as well as to send over the destination address before returning from the connect system call to the application. In order not to make the underlying algorithms of Soxy too complex, Soxy makes the socket blocking until the SOCKS5 authentication has been performed (either successfully or with an error code.) After the socket has been initialized with success, Soxy makes the socket non-blocking again.

The state of a socket can be obtained and altered using the *fcntl* [4] system call, using the *F_GETFL* and *F_SETFL* commands respectively. In order to make a socket non-blocking, the *O_NONBLOCK* flag should be set. By checking the state of the socket against the O_NONBLOCK flag, Soxy determines whether a socket is blocking.

4.2.4 Differences in Architectures

There are two major differences between the x86 and AMD64 architecture. Namely the fact that x86 has one system call to perform all socket operations,

[4] fcntl - manipulate file descriptor

whereas AMD64 has different system calls for different operations. (That is, x86 has the *socketcall* [5] system call.)

For AMD64 support Soxy only intercepts three different system calls; *socket*, *connect*, and *close*. For x86 support Soxy intercepts the *socketcall* system call, upon intercepting a socketcall system call Soxy determines the type of operation (socket, connect, close, or something else) and, if it is one of the socket, connect, or close system calls it then retrieves the arguments to the function and calls the according function (as if it had been a system call intercepted on the AMD64 platform.)

4.2.5 Proxifying UDP Traffic

Although currently not implemented, Soxy aims to support UDP traffic as well. However, there are barely any SOCKS5 daemons which support tunneling UDP traffic (e.g. *sshd* does not support tunneling UDP.)

Proxifying UDP traffic offers another interesting challenge. Because one can set the address and port of an UDP packet when sending the packet itself, the SOCKS5 protocol works accordingly. When tunneling UDP traffic, the data of every regular UDP message is prefixed with the destination address and port. Soxy working on a client-parent basis requires the parent to write extra data in the client, however, not only does Soxy have to write the address and port, it also has to move the original buffer. This is because of the fact that the buffer which will be sent over UDP has got to be continous. In other words, Soxy allocates memory in the child, writes the address and port, and appends the original packet data to the data in the allocated memory.

There is no good solution to this problem, but possible techniques include.

- Allocate enough memory every time an UDP memory is sent
- Keep a list of allocated memory maps
- Perform UDP traffic synchronous

Whereas the first technique remains low on memory, it might be quite slow because every read and write operation involving UDP makes Soxy allocate memory in the child and free it afterwards (i.e. two extra system calls and a lot of extra overhead.) The second technique, however, might exhaust the application and/or system of memory, because it pre-allocates a certain amount of memory pages. And, finally, the third technique is a trade-off in speed; it does nt require much memory and performing one operation on an UDP socket is fairly cheap, but it may become slow especially when there is a lot of UDP traffic.

[5] socketcall - socket system calls on AMD86

Chapter 5

Future Work

This chapter elaborates on features that are currently lacking in Tracy to achieve the set goals by the research project as well as features that we would like to see in the future.

5.1 Secure ABI Mixing

On some platforms such as AMD64 and ARM Linux implements several ABIs. These ABIs are invoked differently and for the tracer it is not easy to tell what ABI is being used to invoke the system call. The problem here is that each ABI (sometimes) has different system call numbers, which means that **read** on AMD64 ABI is not the same as the 32 bit ABI on the AMD64 platform. Tracy currently always assumes a AMD64 ABI on 64 bit and the x86 abi on 32 bit. This is not just a functional issue (not being able to properly identify a 32 bit ABI system call) but a security problem because it allows any tracee to "fake" Tracy into thinking it is doing system call A where it is in fact doing system call B; Tracy will not call the proper hook function and (if told to do so) may deny the wrong system call or even worse, not deny the system call at all.

In a sense not being able to identify the system call ABI can be seen as a limitation of the Linux **ptrace** API. It should be possible to integrate support for identifying the system call ABI in the Linux kernel by using a system that is already in place: the *PTRACE_O_TRACESYSGOOD* extension. If *PTRACE_O_TRACESYSGOOD* is enabled, the kernal sets bit 7 of the signal being delivered to the tracer; this makes it easier to distinguish between a normal *SIGTRAP* signal and a trap caused by a system call of a tracee. The kernel could expose (and set) another bit (or more) which would indicate the ABI of the current system call.

However, a kernel change is not an immediate solution and would require all users of Tracy to install a very recent Linux kernel or even apply a patch to their current kernel.

Another solution to the problem of identifying the ABI of the current system call would be to read the current assembly instruction (that is, the assembly instruction at the instruction pointer).

AMD64 uses a different instruction to perform a system call than the 32 bit ABI; which means we can use that instruction to differentiate the two ABIs. There are a few caveats however: it is possible that the instruction in the program (memory) and the instruction in the cpu pipeline are not the same; a program could use another thread to intentionally wipe out or change the instruction that was used to invoke the system call, which would result in Tracy reading a faulty or fake instruction.

To prevent another thread in the process to change the instruction we can use the **mprotect** system call to mark the memory as read and execute only (not writable), which would make it impossible to change the contents of the memory without calling **mprotect** to mark the memory as writable. Calling **mprotect** from a tracee will however be noticed by Tracy, upon which Tracy can take appropriate actions (such as allow the memory to be written but mark it read and execute only after the write has occurred and make sure the cpu pipeline is flushed). A related technique is called W ^X [1].

One could even go as far as cache the system call instructions of each tracee so they do not have to be read again until a tracee changes its own instructions.

Even though now yet supporting identifying the ABIs is a serious security issue, it is also mainly important when someone wants to be completely sure that every single system call is traced. Most programs also do not usually modify their own instructions; nor do they create their own instructions that perform 32 bit system calls in a 64 bit process. Examples of programs that do create instructions on the fly are Just In Time (JIT) compilers.

5.2 Memory Sharing Between Tracer and Tracee

When one wants to inject a system call into a tracee; the tracee needs to be able to read the arguments and the data that arguments may point to. One way to make the data available in the tracee is by simply allocating memory in the tracee and have the arguments point to the newly allocate memory to which the data is written. This approach is sensitive to race conditions, as threads (that are not currently suspended) may be able to read (and perhaps even write) to the newly allocated memory. A solution to this problem is presented in [8] and we plan to implement such as solution and add it to the Tracy API.

5.3 Threaded Tracees

As discussed previously in Section 3.9, Tracy needs an API to describe what resources are shared amongst tracees. To Tracy, each thread is a new tracee. When managing resources of tracees, it is pertinent to know exactly what resources (file descriptors, memory) are shared. Tracy will in the near future support an API to query what resources a tracee shares with other tracees.

[1] `https://en.wikipedia.org/wiki/W%5EX`

5.4 Instant Injection Using Signals

An interesting addition to Tracy would be the ability to inject a system call even when a tracee is not performing a system call: that is, when the tracee is not stopped. We can simply stop the tracee by sending it a signal will **kill**; which will automatically suspend the tracee as **ptrace** gives us complete control over the tracee. So, before the signal is actually received by the tracee, we have control over the tracee and we can resume it later while suppressing the signal that should never be delivered as we only send the signal to suspend the tracee and gain control.

Once the tracee is suspended, we can jump to some previously crafted assembly similar to our safe fork code (Section 3.8.2), and execute a system call by simply invoking the system call instruction and modifying the arguments and system call instruction in Tracy. Once the system call has completed we can store the return code and jump back to the original code, and the finally resume the tracee.

The process just described can also be used to inject system calls in the signal hook exposed by Tracy.

While it would not be too hard to implement support for this particular feature, we have postponed the feature due to time constraints. We also did not find the feature particularly useful because Tracy is event based, which means that in most cases the tracee is already stopped.

5.5 Improving Ptrace Performance

As previously discussed in Section 3.10, the overhead of **ptrace** is very noticeable when a tracee performs a lot of small and inexpensive system calls - they suddenly become very expensive.

The problem boils down to the fact that **ptrace** currently has no mechanism which allows the tracer to only get notified on specific system calls, so that other system calls can execute without requiring intervention from the tracer.

It then follows that this performance issue can be solved by adding such an API to the Linux kernel.

We wrote a simple Linux kernel patch (against Linux 3.4) to demonstrate our claim and we also added experimental support for this feature to Tracy.

However, the patch is incomplete and lacks features such as support for multiple ABIs (Section 5.1).

The API is very basic and allows one to add (and delete) specific system calls to a list. This list can then either function as a whitelist or a blacklist, in other words, the Linux kernel either notifies the tracer of all the system calls in the list, or notifies the tracer of all system calls but the system calls in the list.

Because Tracy is aware of the all the system calls that are hooked by the program using Tracy, Tracy can make informed decisions about system calls that do not need to be received - as they would never result in a hook in the event system (Section 3.6) being executed.

Unless Tracy uses this particular system call internally, Tracy could decide to advice the kernel to not notify Tracy of in the event that this particular

system call is executed.

5.6 Threaded Tracer

As the number of tracees for a single Tracy instance increases, the likelyhood of Tracy becoming a bottleneck increases. Since Tracy is single threaded, it is very possible a lot of tracees will have to wait on Tracy to resume their execution.

We believe that it is possible to add multithreading support to Tracy where each thread can perform its own **waitpid**. Access to certain datastructures in Tracy will have to be guarded with semaphores, but we think multithreading will be both viable to program and an effective wait to increase the performance of Tracy with a lot of tracees.

5.7 BSD Support

Support for Tracy on the BSD variants is planned, but will require some additional work. We have not extensively examined **ptrace** and system calls in general on BSD variants, but a few things jumped out.

BSD uses **rfork** instead of Linux' **clone** and BSD also does not support automatically tracing processes created by a tracee. While we can emulate this effectively with Tracy's safe fork Section(3.8.2) the Tracy code still requires some work.

Apart from this functional difference, there is also a difference in how BSD variants treat the system call arguments. Arguments of a system call are pushed onto the stack (the last parameter is pushed first). [1]

Alternatively, if Linux emulation is enabled, BSD variants (at least FreeBSD) also support the Linux way of system calls, by passing the arguments in registers in the same manner as Linux.

BSD also returns errors differently than Linux (although again, it supports Linux emulation). BSD sets the "carry flag" upon failure and sets the error in the register of the return code, whereas Linux sets the register of the return code to the negative equivalent of the error number[2].

Chapter 6

Discussion

As stated in Section 5.1, Tracy is not yet able to safely trace processes, the solution presented in Section 5.1 should however eliminate all currently known ways to "escape" from Tracy. Implementation of the aforementioned solution will also require the ability to "understand" multiple ABIs as presented in Section **??**. We believe that upon completion of features "Secure ABI Mixing" (Section 5.1) and "Threaded Tracees" (Section 5.3), Tracy will be fit for creating programs which require safe tracing, such as a *jail*.

We believe that the Tracy API is relatively simple. The API provides functionality that is not easily implemented using ptrace and also provides a solid and platform agnostic base for tracing applications. As mentioned in Section 5.3 the API will be extended to support thread detection and the ability to query what resources are shared amongst specific processes.

Creating a completely cpu architecture agnostic program is possible with Tracy, but we have found that in certain cases, such as Section 4.2.4, it is simply not possible to create code that works on all architectures without architecture specific code, because the Linux system call API differs per architecture. This makes it impossible to inject or hook a system call such as **socket** without writing architecture specific code. Normal programs are not hindered by this design flaw as they mostly rely on *glibc*, which works around architecture specific quirks. We do not plan to extend the Tracy API to cover this kind of architectural differences. As a result, some programs that rely on Tracy will still need architecture specific code.

Writing operating system independent code should be viable once Tracy works on a BSD variant (see Section 5.7). However, if a programmer decides to rely on (or inject) operating system dependant system calls, the resulting program will obviously not work on all platforms. However, we believe that the tracing and system call modification and injecting part of Tracy should work on all platforms Tracy supports without architecture specific code on the users end.

The fact that Tracy programs are not completely architecture and operating system agnostic raises the question of the impact on supporting several architectures in one code base. We believe that most architecture quirks will be due to differences in the Linux system call API. However, the changes required to work around these issues should be relatively small in comparison to writing

code which utilises ptrace directly for each architecture.

The performance of a tracee is highly dependent on the amount of system calls the tracee performs. As noted previously (Section 3.10, Section 5.5) the performance of a tracee suffers when the tracee performs a lot of system calls. We have presented a possible (simplistic) solution but we see no short-term solution to the performance issues. The performance issues however should be neglectable when tracing a program that doesn't perform a lot of IO.

The viability of a cross platform tracing framework is demonstrated by Soxy, which implements a ptrace-based proxifier in three hundred lines of C code. Soxy can trace relatively simple programs such as *curl* and *irssi*, but it also succesfully traces complex programs such as *firefox* and *ssh*. We believe that even with Tracy's current limitations, it should be possible to write complex applications that utilise Tracy for system call tracing and injection.

Once we support more architectures[1], we will be able to further investigate the cross architecture promise.

[1] We currently support ARM, x86 and AMD64

Bibliography

[1] 2.1. default calling convention. http://www.int80h.org/bsdasm/#default-calling-convention.

[2] Freebsd assembly language programming. http://www.int80h.org/bsdasm/#where-errno.

[3] Gnu debugger. http://sourceware.org/gdb/.

[4] Go language. http://golang.org/.

[5] Posix - portable operating system ix. .

[6] Socks 5. http://www.ietf.org/rfc/rfc1928.txt.

[7] strace. http://sourceforge.net/projects/strace/.

[8] Guido Van 't Noordende, Ádám Balogh, Rutger Hofman, Frances M. T. Brazier, and Andrew S. Tanenbaum. A secure jailing system for confining untrusted applications. In *International Conference on Security and Cryptography (SECRYPT)*, 2007.

Appendix A

Creating and tracing a process with ptrace(2)

```
pid = fork();

/* Child */
if (pid == 0) {
    r = ptrace(PTRACE_TRACEME, 0, NULL, NULL);
    if (r) {
        fprintf(stderr, "PTRACE_TRACEME failed.\n");
        _exit(1);
    }

    raise(SIGTRAP);

    execve(...);

    /* ... */
}

if (pid == -1)
    return NULL;

waitpid(pid, &status, __WALL);
signal_id = WSTOPSIG(status);
if (signal_id != SIGTRAP) {
    return NULL;
}

/* Resume child and tell ptrace to stop at the next system call */
r = ptrace(PTRACE_SYSCALL, pid, NULL, 0);
```

Appendix B

ptrace(2) API

Every method except for *PTRACE_TRACEME* is called by the tracer.

PTRACE_TRACEME	Called by a process to make the parent of the process a tracer. This is forcibly called by the first child that is spawned by Tracy.
PTRACE_ATTACH	Attach to process.
PTRACE_DETACH	Detach from a tracee.
PTRACE_SYSCALL	Continue the execution of the tracee, stopping at the next system call or signal event.
PTRACE_PEEKDATA	Read a word (long) from a tracee.
PTRACE_POKEDATA	Write a word (long) to a tracee.
PTRACE_GETREGS	Returns the cpu registers of a tracee.
PTRACE_SETREGS	Set the cpu registers of a tracee.
PTRACE_SETOPTIONS	Set a ptrace option of a tracee.

Table B.1: Basic ptrace api

The following table provides an overview of the Linux specific ptrace options set using PTRACE_SETOPTIONS.

PTRACE_O_TRACESYSGOOD	When delivering a system call trap, bit 7 in the signal is set: SIGTRAP|0x80.
PTRACE_O_TRACEFORK	Stop the tracee at the next fork event and automatically tracee any children created. The child is started with SIGSTOP.
PTRACE_O_TRACEVFORK	Stop the tracee at the next vfork event and automatically tracee any children created. The child is started with SIGSTOP.
PTRACE_O_TRACECLONE	Stop the tracee at the next clone event and automatically tracee any children created. The child is started with SIGSTOP.
PTRACE_O_TRACEEXEC	Stop the tracee at the next execve system call and report the death of each thread.
PTRACE_O_TRACEVFORKDONE	Stop the tracee (parent) at completion of the vfork call.
PTRACE_O_TRACEEXIT	Stop the tracee before exit.

Table B.2: Linux-specific ptrace options