Kernel-Assisted Debugging of Linux Applications

Tobias Holl, Philipp Klocke, Fabian Franzen, Julian Kirsch
Technical University of Munich
Garching, Germany

ABSTRACT

On Linux, most—if not all—debuggers use *ptrace* debugging API to control their target processes. However, *ptrace* proves unsatisfactory for many malware analysis and reverse engineering tasks: So-called split-personality malware often adapts its behavior in the presence of a debugger, yet *ptrace* makes no attempt to hide from a target process. Furthermore, *ptrace* enforces a strict one-to-many relation meaning that while each tracer can trace many tracees, each tracee can only be controlled by at most one tracer. Simultaneously, the complex API and signal-based communications provide opportunities for erroneous usage.

Previous works have identified the newer *uprobes* tracing API as a candidate for building a replacement for *ptrace*, but ultimately rejected it due to lack of practical use and documentation. Building upon *uprobes*, we introduce `plutonium-dbg`, a Linux kernel module providing debugging facilities independent of the limitations of *ptrace* alongside a GDB-compatible interface. Our approach aims to mitigate some of the design flaws of *ptrace* that make it both hard to use and easy to detect by malicious software.

We show how `plutonium-dbg`'s design and implementation remove many of the most frequently named issues with *ptrace*, and that our method improves on traditional *ptrace*-based debuggers (GDB and LLDB) when evaluated on software samples that attempt to detect the presence of a debugger.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**;
• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Linux, debugging, ptrace, uprobes, GDB, debugger detection

ACM Reference Format:
Tobias Holl, Philipp Klocke, Fabian Franzen, Julian Kirsch. 2018. Kernel-Assisted Debugging of Linux Applications. In *Reversing and Offensive-oriented Trends Symposium (ROOTS '18), November 29–30, 2018, Vienna, Austria.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3289595.3289596

ROOTS '18, November 29–30, 2018, Vienna, Austria

ACM ISBN 978-1-4503-6171-2/18/11…$15.00
https://doi.org/10.1145/3289595.3289596

1 INTRODUCTION

In general, Linux debuggers rely on a single kernel API named *ptrace* to attach to and manipulate other processes. This applies not only to general-purpose debuggers such as GDB, but also to software specifically targeted at malware analysis and reverse engineering tasks. For example, both IDA Pro and Radare2 internally use *ptrace* to provide their debugging functionalities.

However, *ptrace*—first introduced in UNIX v6 (1975)[1]—was never designed to accommodate analysis of environment-aware, evasive targets such as so-called split-personality malware: Many malware samples perform some sort of environmental analysis to detect debuggers, and then alter their behavior accordingly, e.g. by performing harmless tasks instead of the intended malicious actions, or by trying to remove traces of itself from the infected system [3]. While so far these methods are not quite as common in Linux malware [5], the increasing prevalence of Linux on consumer devices may well lead to a similar "arms race" as that observed in the past decades with Windows malware. With *ptrace*, a piece of software can trivially detect if it is being debugged:

```
if (ptrace(PTRACE_TRACEME, 0, NULL, NULL) == -1) {
    /* This process is being debugged */
}
```

Other design decisions also negatively affect *ptrace*'s usefulness for malware analysis:

- When debugging, the target process or thread is made a direct child of the debugger. Some effort is made to keep presenting the "real" parent to user-space queries, but it is still possible to detect debugger presence by attempting to receive the SIGCHLD sent from a terminating child process. If the child process is being debugged, the signal will be sent to the debugger instead.
- Only one debugger can be attached to a process at any one time. If not prohibited by security settings, a malicious process can simply spawn a child process and attach to it, which stops others from debugging that process. This also affects other software where *ptrace* is used for non-malicious purposes.
- Memory transfer is limited to one machine word (8 bytes on `x86-64`) per system call[2], leading to extremely inefficient data transfer at a cost of at least two context switches per transferred machine word [2, 13].

Simultaneously, the *ptrace* API is complex to use for custom-built software whenever it cannot be abstracted by another tool like GDB [4]—so complex, in fact, that UNIX v8 (1985) opted to provide an easier interface through the `/proc` file system [2, 13]. Linux has not adopted this approach, instead adding new features to the already fragile `ptrace()` system call [1].

[1] Manual for `ptrace` on UNIX v6, `http://man.cat-v.org/unix-6th/2/ptrace`
[2] `man ptrace`, under `PTRACE_PEEKDATA` and `PTRACE_POKEDATA`

Because *ptrace* has no built-in support for breakpoints, debuggers must manually generate the appropriate instructions for the current architecture and write them to process memory. Similarly, there is no concept of threads. Like the rest of the kernel, *ptrace* treats each thread as a separate process, forcing users to manually attach to each thread of the targeted process.

This paper presents a new Linux debugging mechanism which bypasses these limitations. In particular, we make the following contributions:

- We design `plutonium-dbg`, an open source kernel-module–based debugging mechanism for Linux applications (Section 3) and show how its design circumvents some of the limitations imposed by *ptrace*.
- We describe in detail how `plutonium-dbg` uses modern kernel APIs to provide debugging functionality (Section 4).
- Finally, we show how `plutonium-dbg` impedes efforts by malicious software to detect and interfere with debuggers (Sections 5 and 6).

2 BACKGROUND

In this section, we briefly highlight the concepts needed to understand how Linux debuggers operate in detail, and some of the additional features that can assist modern Linux debugging utilities.

2.1 Debugging concepts

The following concepts are supported by all major debuggers in some form:

2.1.1 Process Relationships and Terminology. Debuggers are essentially processes that aim to manipulate other (possibly already existing) processes. However, operating systems usually strictly separate process address spaces from each other, such that no process can (unintentionally or maliciously) modify a different processes memory contents. Therefore, debuggers must be offered a way of circumventing this isolation by the operating system kernel.

In Linux, *ptrace* allows debuggers to pierce the isolation of any thread marked with the `SUID_DUMPABLE_USER` flag[3], although additional kernel security modules such as YAMA may further restrict access. Once attached, memory and registers in the foreign process may be modified using the `PTRACE_POKEDATA`/`PTRACE_POKETEXT` and `PTRACE_SETREGS`/`PTRACE_SETFPREGS`/`PTRACE_SETREGSET` operations offered by the *ptrace* system call.

Throughout this work we use the terms *debugger*, and *tracer* interchangeably to refer to the process that maintains control of some other process in order to perform debugger functionality. Similarly, we use the terms *debuggee*, *tracee*, and *target* to refer to the process whose execution is to be inspected or altered by the debugger.

2.1.2 Breakpoints. On most architectures, *software breakpoints* are created by inserting a special instruction that triggers a trap or interrupt in the CPU. On `x86-64` and `x86`, a breakpoint trap is triggered by the `int3` instruction, usually present in machine code as a single `CC` byte[4]. As opposed to software breakpoints, some architectures (such as `x86-64` and `x86`) also offer *hardware breakpoint* support. The latter have the advantage that they do not alter the program in memory but are at the same time limited in number because they rely on explicit CPU support.

The Linux kernel implements software and hardware breakpoints via signalling: If the CPU ends up executing a breakpoint instruction, the interrupt handler in the Linux kernel sends a SIGTRAP signal to the thread that hit the breakpoint. If a tracer (such as a debugger) is attached via *ptrace*, the signal is redirected to the tracer and masked from the tracee.

2.1.3 Single-step execution. In order to step over an already placed breakpoint, one strategy can be to restore the original instruction, single-step over that instruction, and then restore the breakpoint. However, this approach might introduce a race-condition on multi-threaded applications: If a second thread executes the target instruction while the breakpoint is removed in an attempt of single-stepping the first thread, the second thread might miss the breakpoint. Because of this, GDB suspends all threads of a program while executing the target instruction [18].

Single-stepping generally involves setting a bit in one of the CPU's flag or debug registers. On all Intel 8086-based architectures (including `x86-64`), bit 8 (the so-called "trap flag" `TF`) of the `FLAGS` register indicates that the processor should send a special interrupt (generally `int1`) after executing the next instruction. Once Linux receives this special interrupt, a special handler again generates a SIGTRAP signal. If a debugger is present, the signal is redirected appropriately.

2.2 Linux kernel features

At the time of writing (August 2018), the Linux kernel already abstracts away much of the hardware interaction that is required to properly support debugging: For example, implementing single-stepping on `x86-64` in user-space code alone is essentially impossible, because the implementation would either have to modify the kernel interrupt handlers (to intercept the trap directly) or the kernel's signal processing code (to redirect the SIGTRAP from the target to the debugger). Both are pieces of the operating system that cannot be accessed directly from user-space code. Instead, current debuggers use *ptrace* with the `PTRACE_SINGLESTEP` argument to inform the kernel that the tracee should execute (at most) one instruction.

However, to our knowledge, debuggers so far did not adopt a number of additional features that were introduced into the kernel over the past decade, mostly due to usability issues:

2.2.1 Kernel tracing. Over the years, the Linux kernel has itself gained a number of tracing and debugging features, mostly to identify performance issues [8]. Static and dynamic trace points enable kernel developers to register custom callbacks at specific locations within the kernel[5].

These features allow a debugger with kernel components to react to events directly in the kernel without having to rely on *ptrace* to support that specific event.

2.2.2 User-space tracing. Eventually, the *utrace* project implemented similar tracing and debugging features for user-space

[3] `man prctl`, under `PR_SET_DUMPABLE`.

[4] `CD 03` is a rarely-used alternative encoding of the same instruction.

[5] Static trace points are provided through *ftrace* and *perf_event*, while dynamic trace points were introduced with *kprobes*.

applications, with the ultimate goal of making *ptrace* just one of many *utrace* clients [12]. However, the patches were not accepted into the mainline Linux kernel [4, 20], and development on *utrace* was abandoned. Only the core tracing functionality (largely similar to *kprobes*) was retained under the name of *uprobes* and integrated into the kernel.

Unfortunately, while a tracing interface is exposed to user-space at `/sys/kernel/debug/tracing/`, the actual kernel code of *uprobes* remains sparsely documented. Combined with the fact that any significant customization requires writing a custom kernel module, *uprobes* has not seen widespread adoption for debugging-related tasks that need *interactivity* for proper operation. The *Simterpose* emulation framework has explicitly rejected using *uprobes*, at first because of its relative immaturity [10, 11] and later because it requires the use of a custom kernel module [16]. The *SystemTap* and *perf-tools* utilities both offer some form of interface to *uprobes* for tracing purposes only [9, 19].

2.2.3 Threads and processes. The Linux kernel largely treats threads exactly the same way as processes, with the only distinction that different threads of the same process share some of their properties (e.g. address space). This has given rise to some confusing terminology:

Inside the kernel, each thread or process is represented by a `struct task_struct`, which among many other properties also stores a unique ID for each task. The kernel exposes two system calls to access these values, `getpid` (returning the ID of the process to which the calling thread belongs) and `gettid` (returning the ID of the calling thread). Accordingly, we often refer to these values as the *process ID* (PID) and *thread ID* (TID) respectively.

Inside kernel code, however, the terms TID and PID are generally used interchangeably. For example, the `task_struct` member that holds the TID is actually called `pid`. Instead, the kernel perceives user-space processes as groups of threads, each with a *group leader*. The TID of the leading thread is then called the *thread group ID* (TGID), which is what user-space considers to be the ID of the process.

In order to avoid this ambiguity, we will consistently use the term TID when referring to a single thread, and TGID when referring to a process.

3 DESIGN

In order to provide a meaningful alternative to *ptrace*-based debuggers, we need to address the issues identified in Section 1. This requires us to take the following design considerations into account:

3.1 System architecture

Currently, the only API enabling us to access the kernel features we need for debugging is the `ptrace()` system call, which we explicitly avoid using in this work (cf. Section 1). Single-stepping, for example, is enabled by a call to the kernel-level function `user_enable_single_step`, which on `x86-64` is only ever called from *ptrace*-related kernel code, so we must provide an alternative interface.

This means that in order to access these functions we must either modify the kernel itself (and distribute the appropriate patches), or provide a *loadable kernel module* (LKM) that performs the necessary calls. Kernel modules can be loaded and unloaded at runtime, and significantly simplify development, maintenance and installation [6]. Not only does modifying the kernel sources themselves require end users to apply custom patches and recompile their kernel, but it also makes maintenance across kernel versions more difficult. We therefore opt for a module-based approach.

Because this work mostly presents a new debugging back-end, it also seems unwise to introduce an entirely new user interface. GDB has extensive remote-debugging support, introduced to allow clients to connect to debuggers running on other devices. However, there is no restriction that forces the other device to be running an actual GDB server—any utility that understands the publically available *GDB remote protocol* can function as a GDB server. `plutonium-dbg` can use this functionality to allow GDB clients to interface with the new kernel module.

Therefore, `plutonium-dbg` consists of two separate layers: In user-space, a small remote server translates the GDB protocol into commands for the kernel module, which then implements the actual debugging features.

Communication between the client and the server can take place through any link supported by GDB, including TCP and Unix domain sockets. Communication between the server and the kernel module (running on the same host) is implemented by means of the *ioctl* user/kernel communication interface.

3.2 Debugging functionality

Conventional debuggers use *ptrace* to control process execution through breakpoints and single-stepping, as well as to access and modify process memory and CPU registers. This provides us with a minimum baseline of features that must be present in the kernel module to make debugging use feasible.

3.2.1 Breakpoints. Breakpoints are generally created by replacing the instructions at the target address with a special breakpoint instruction (cf. Section 2). Because the different threads of a process share the same memory, they are inherently global to a process. While it would be possible to filter breakpoint "hits" by thread, such a feature can just as easily be implemented outside of the kernel by simply continuing execution of threads that should ignore this breakpoint.

A breakpoint is therefore uniquely identified by the process TGID (its target) and the address inside the address space of the process.

3.2.2 Single-stepping. In the kernel, single-stepping is enabled individually for each thread. Because behavior would be ambiguous if there were separate options to single-step the entire process or just a single thread, we explicitly require users to specify which thread they wish to single-step[6].

3.2.3 Thread suspension. Each time a debugged thread hits a breakpoint, it is automatically suspended to allow the debugger to react to the breakpoint hit (usually by prompting the user for input). However, it is also useful to allow a debugger to forcibly suspend a target thread without explicitly inserting a breakpoint. This is necessary to support interactive debuggers that suspend a target

[6] Full process single-stepping can still be emulated by enabling single-stepping for all existing threads and intercepting the creation of new threads.

process on first attach in order to allow the user to set up breakpoints. Obviously, an analogous API must be provided to allow debuggers to continue execution of a thread or process, both after handling a breakpoint and after a thread has been manually suspended. Because these operations are also sensibly defined for entire processes (they simply act on all existing threads of the target process), it should be possible for a user to specify whether they are targeting the entire process or just a single thread.

3.2.4 Memory and register access. Besides forwarding events from single-stepping and breakpoints, remote memory and register access is the core feature of *ptrace*. While it will no longer be necessary to directly manipulate process memory in order to install breakpoints (cf. Section 4.2.2), controlling the target process's memory is still very useful for debugging. We identified in Section 1 that the limited memory bandwidth of *ptrace* can sometimes be an obstacle for debugging. Our solution should therefore allow accessing memory blocks of arbitrary length as long as the target process has access to that memory. Because memory is generally shared between threads, it does not matter whether a specific thread or the whole process is addressed. On the other hand, *registers* are specific to each thread, and access can only happen through the TID of the target.

3.3 Other requirements

To address the problems described in Section 1, we take the following properties into consideration.

3.3.1 Multiple debuggers. One of the more significant limitations of *ptrace* is that only one thread can ever debug any particular target[7]. Our motivating example from Section 1 shows how this allows easy detection of attached debuggers. Of course, a common method to circumvent these checks is to remove them from the binary that we are analyzing, or to manipulate the instruction pointer to skip the checks entirely. However, it is easy to imagine more complex approaches to debugger detection that actually use *ptrace* functionality, e.g. by writing and reading arbitrary locations in memory on which subsequent program behavior could depend. These methods would then need to be emulated by the tracer to remain undetected.

Therefore, it should be possible to attach any number of debuggers to a single target thread or process, independent of the usage of *ptrace*. At that point, the notion of "attached" or "detached" becomes one of user interface logic—the need to explicitly signal that a debugger is taking control over a target no longer exists.

This has a number of effects on the remaining design. For example, breakpoints cannot be tied to a specific debugger. Traditionally, only one debugger would "own" a breakpoint, and removing the breakpoint in that debugger meant restoring the instruction to its original state. Instead, debuggers must now register to handle breakpoint hits, and the actual breakpoint instruction is only removed when there are no listening debuggers remaining. Similarly, threads that have hit a breakpoint must remain suspended until *all* debuggers that were notified of the breakpoint signal that the thread should continue execution. Furthermore, multiple debuggers must synchronize their states and must, for example, be aware that breakpoints can *disappear* because a different debugger instance removed them from memory.

3.3.2 Transparency. At the same time, there should be no direct way for the target to determine whether its thread(s) is/are being debugged. In particular, the existing *ptrace*-based methods of detection should not detect our newly proposed debugging approach, nor do we offer a trivial possibility to query the debug state of that approach. We evaluate transparency guarantees of our approach later, and discuss methods of debugger detection in Section 5.

3.3.3 Threads and processes. Because *ptrace* does not have a concept of processes, debuggers must explicitly attach to each created thread. There is explicit support to receive notifications of newly created threads through the `PTRACE_O_TRACECLONE` flag, but it is nontrivial to handle the different (and sometimes simultaneous) incoming signals that debuggers are required to handle by *ptrace*'s interface.

A usable replacement should instead work on a process level, with thread-level granularity only for those operations that require it. We discussed earlier, for example, that breakpoints can have process-wide visibility by default—if only certain threads should actually stop at that breakpoint, the debugger can instantly continue the thread instead. In our case, this type of breakpoint handling can be implemented on the GDB server's side, and does not require introducing additional complexity into the kernel module.

4 IMPLEMENTATION

There are different ways of implementing the features described in Section 3. In the following, we describe in more detail how `plutonium-dbg` uses kernel functionality to provide a meaningful alternative to *ptrace*.

4.1 User-kernel communication

The Linux kernel offers a large variety of possible communication methods between user-space programs and kernel modules. We evaluated numerous approaches (including network-based communications through either UDP or Netlink), but ultimately settled for a straightforward implementation based on file system I/O. As relying on read and write operations would require us to first design a custom serialization format for our commands, we instead rely on the `ioctl()` system call to communicate between the GDB server and the kernel module. Incidentally, this method was also used in UNIX v8 to make *ptrace* less cumbersome to use [2]. Functionality to invoke `ioctl()` is available in most higher-level programming languages including Python, in which the server component of `plutonium-dbg` is written.

4.2 Debugging functionality

Because `plutonium-dbg` is designed as a kernel module, it can only make use of those features that the kernel explicitly exports to modules. Where functions are not available to `plutonium-dbg`, we use the *kallsyms* mechanism to gain access to non-exported functions.

4.2.1 Thread suspension. In Section 2.2.3, we briefly addressed the `struct task_struct` Linux uses to represent threads internally. It

[7] Note that debugging is limited to one *thread*, not to one process at a time, so even normal multi-threaded debuggers can be affected by this issue.

also stores the thread state, which we can manipulate to hold execution of a process. We set the state to `TASK_KILLABLE`, indicating that it is blocked in the kernel and cannot otherwise be interrupted except by fatal signals [15]. Then, we force the kernel to attempt rescheduling the current thread using `schedule`. The kernel will attempt to reschedule the current thread and automatically check the state flag. If the task is not allowed to run, it is then removed from the list of active tasks. It will remain in that state until we call `wake_up_process`, where the kernel places the task back into the running state.

A suspension triggered by `plutonium-dbg` may be interrupted by the kernel. Imagine, for example, that the target thread is currently suspended inside a system call (e.g. doing I/O operations or even just calling `sleep()`). Then, once the system call finishes, the kernel will automatically wake up the target thread, regardless of whether we wanted to suspend it. In doing so, our changes to the task state will be overwritten, and suspension fails. This also means that if we are not suspending the thread that is currently running, we cannot immediately set the task state to `TASK_KILLABLE`.

Because of this, suspending another thread requires us to first redirect control of the thread to our kernel module. We do this by briefly setting the task state to `__TASK_TRACED`, a state used by *ptrace* when it is suspending a debugged thread. Then we trigger a scheduler interrupt in the target process[8] and wait for the thread to stop. Because we do not want to stay in the `__TASK_TRACED` state any longer than necessary (it is publically exposed through `/proc/<tid>/status`), we place a temporary breakpoint at the current instruction pointer and wake the thread back up. Once execution continues, it will immediately activate the breakpoint and hand control to the kernel module, where we can suspend the thread using `TASK_KILLABLE`.

4.2.2 Breakpoints. Instead of requiring debuggers to place breakpoints manually, we rely on *uprobes* (Section 2.2.2) to insert probes into the target program. While fundamentally a tracing framework, the ability to install custom probe handlers allows us to immediately suspend a target thread as soon as it executes the probed instruction.

Because *uprobes* was originally designed to allow tracing of processes sharing an executable file or library, it operates on offsets within memory mapped inodes. We therefore must first translate an address in the target's address space to an inode number (identifying the file on which the breakpoint should be placed) and an offset inside this inode. This implies that only locations within mapped files can be suitable for a breakpoint using this technique.

Breakpoint removal is another contentious issue. A *uprobes* probe cannot be removed while a thread is suspended inside its handler because cleanup routines can only run *after* the handler. If a debugger removes a breakpoint with a suspended target, removal must be deferred until after the handler has completed.

With *uprobes*, the breakpointed instruction is automatically executed *out-of-line* after the handler returns. This means that—unlike *ptrace*—the breakpoint instruction is never removed from the code [12], and that debuggers do not need to take special precautions for multithreaded targets (cf. Section 2.1.2).

4.2.3 Single-stepping. Currently, the only way to enable single-stepping for a thread inside the Linux kernel is calling *ptrace*'s `user_enable_single_step` function. For each instruction, the target thread will therefore receive a SIGTRAP signal. Because SIGTRAP is usually fatal and certainly detectable, we must intercept that signal before it is passed to the process.

Unfortunately, there is no way for us to explicitly intercept the signal before it is passed to the process (even manipulating the thread's list of blocked signals does not work, because it is reset by the kernel when the SIGTRAP is sent), and there is no official trace point hook where we could access the signal data. We must therefore fall back to using *kprobes* in order to hook arbitrary locations inside the kernel. Unfortunately, accessing the function arguments through *kprobes* would involve platform-dependent register operations[9], so we must place our hook in a location where we can access the target thread without relying on the function arguments.

We use *kprobes* to place a hook at the start of the `get_signal` function, which is executed every time a thread attempts to handle a pending signal, and immediately return if the thread in question is not being single-stepped. Next, we check if a (injected) SIGTRAP is queued for that thread. If it is, we remove it from the queue and suspend the thread just as we would on a regular breakpoint hit. This also means that other SIGTRAPs not caused by our single-stepping primitive are simply passed to the target thread's signal handler.

4.2.4 Memory and register access. Memory and register access are probably the least complex of the features identified in Section 3. The kernel provides functions to access another thread's memory (`access_process_vm`) and registers (`copy_regset_to_user` and its complement `copy_regset_from_user`). These functions are either used internally by *ptrace* or at least call the same functions, and do not suffer from the limitations imposed by *ptrace*'s system call API.

In `plutonium-dbg`'s API, memory can be accessed in arbitrary block sizes. Register access works like *ptrace*'s `PTRACE_GETREGSET`: Depending on system architecture, there can be multiple register sets; the ID of the desired register set must be specified by the user. On `x86-64`, register set 1 contains the general purpose registers, while register set 2 is made up of the floating point and vector registers.

4.3 Interaction with GDB

As explained in Section 3.1 we chose to implement GDB's remote debugging protocol[10] to allow the use of the GDB front end with `plutonium-dbg`. A full explanation of the rather complex protocol would be out of scope for this work.

In our implementation, a small Python server accepts a subset of the protocol (features not implemented by `plutonium-dbg` such as memory watchpoints are not currently supported) and translates the incoming commands into IOCTL requests for the kernel module. A brief overview of supported packets can be found in Table 1.

[8] This is necessary because the thread may currently be running on another CPU on SMP-enabled systems. A scheduler interrupt can be triggered by setting the `TIF_NEED_RESCHED` flag on the task and then using `kick_process` to force the target to enter the kernel.

[9] The *jprobes* API formerly enabled that exact feature, but was removed with Linux 4.15.

[10] `https://sourceware.org/gdb/onlinedocs/gdb/Remote-Protocol.html`

Table 1: Supported GDB protocol packets

Packet	Description
`c`	Continues a suspended thread
`g`	Requests the current register set
`H`	Sets the target thread
`m`	Reads arbitrary memory
`M`	Writes arbitrary memory
`q`	Feature queries (partially supported)
`s`	Single-steps the current thread
`T`	Queries whether a thread is alive
`z`	Removes a breakpoint
`Z`	Sets a breakpoint
`?`	Indicates reasons for thread suspension

The individual packets are explained in more detail in the GDB documentation. To connect to such a remote server from a GDB client, users can use the `target remote` command with the address and port number that the server is listening on.

4.4 Access controls

Debuggers generally have far-ranging permissions to modify other processes. This means that rigorous access controls are necessary to avoid malicious actors from abusing the debugger to escalate their privileges on the system. In *ptrace*, this is enforced through checks involving the debugger and target process credentials (in the `ptrace_may_access` function), and may be further restricted through the use of kernel security modules such as YAMA.

Access to `plutonium-dbg`'s kernel module could be restricted by setting the file permissions on the `/dev/debugging` device file, but only on a very coarsely defined basis—any user with write permissions could then attach to *any* process in the system. Instead, we duplicate the access control checks used by *ptrace* by checking for the global `CAP_SYS_PTRACE` capability and comparing the current real, effective, and saved user and group IDs (threads in the same thread group are excluded from those checks). Note that unlike *ptrace*, we do not verify the `SUID_DUMPABLE_USER` flag (c.f. Section 2.1.1), because this would again allow processes to prevent debuggers from attaching.

If the debugger and the victim are not in the same thread group (i.e. debugging is only used for introspection), we then delegate to the kernel security modules, so that access to `plutonium-dbg` is only granted if *ptrace* would also have been available.

The GDB server, of course, also needs to be secured against illegitimate access. Because TCP sockets are by their nature open for others to connect, and the GDB protocol does not implement authentication, security-minded users may prefer to use Unix domain sockets to communicate between GDB and the GDB server, so that access to the GDB server can be controlled by the socket file's permission settings.

Table 2: Scenarios for debugger detection

#	Scenario	Source
1	Uses `PTRACE_TRACEME` to detect *ptrace*-based debuggers	[14]
2	Checks that the vDSO is located above the top of the stack	
3	Checks if the debugger disabled ASLR for the target process	
4	Detects if an interactive debugger has set the `LINES` and `COLUMNS` environment variables	
5	Compares the parent process to a list of known debuggers	
6	Searches for a GDB-specific breakpoint in the dynamic loader	
7	Checks whether the heap was relocated in a PIE binary by a debugger	
8	Scans the process memory for inserted breakpoints	[21]
9	Uses a stray `0xCC` to check for the SIGTRAP signal	
10	Detects single-stepping via the `icebp` (`F1`) instruction	
11	Reads the trap flag to detect single-stepping	
12	Manually raises SIGTRAP	[23]
13	Attempts to find GDB by reading the "_" environment variable	
14	Checks for leaked debugger file descriptors	

5 EVALUATION

We tested our approach on a number of binaries designed to detect or otherwise block debugging on `x86-64`, and evaluated the performance improvement gained over *ptrace*'s `PTRACE_PEEKDATA` and `PTRACE_POKEDATA` for memory manipulation.

5.1 Detection

The `debugmenot` [14] and `pangu` [23] projects showcase (as proof-of-concept implementations) a number of approaches by which malware can attempt to detect and evade debuggers. Tung [21] outlines (but does not implement) a few others. We eliminated those examples that targeted non-Linux operating systems or exploited unrelated bugs in debuggers (e.g. in the code responsible for parsing binaries). All fourteen remaining scenarios are described in more detail in Table 2.

We then used `plutonium-dbg` to debug the remaining binaries. For comparison, the same debugging operations were also performed twice in each of GDB 7.12 and LLDB trunk (rev. 331965), once by starting the binary directly, and once by attaching to an already-running process.

Table 3 shows the results of our evaluation. Red cells indicate that the debugger was detected or otherwise failed to properly debug the target. Green cells show that debugging was successful. Rows with the `-p` flag shown contain the results obtained when the

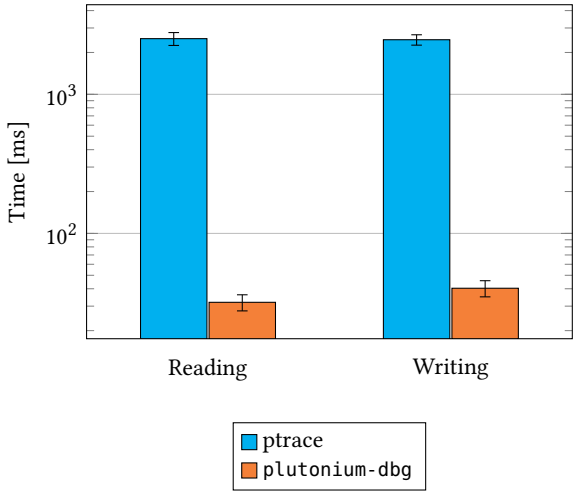
Table 3: Detection results

#	1	2	3	4	5	6	7
plutonium-dbg	■	■	■	■	■	■	■
GDB	■	■	■	■	■	■	■
GDB (`-p`)	■	■	■	■	■	■	■
LLDB	■	■	■	■	■	■	■
LLDB (`-p`)	■	■	■	■	■	■	■

#	8	9	10	11	12	13	14
plutonium-dbg	■	■	■	■	■	■	■
GDB	■	*	■	■	*	■	■
GDB (`-p`)	■	*	■	■	*	■	■
LLDB	■	*	■	■	*	■	■
LLDB (`-p`)	■	*	■	■	*	■	■

∗ unless signals are explicitly passed to the target

Figure 1: Memory Transfer Performance Reading/Writing 32MiB of Data 200 Times



debugger was attached to a process that was already running (via the `-p <pid>` command line option).

5.2 Performance

Another issue with *ptrace* is the number of context switches required to perform a simple memory transfer. We explained in Section 1 how `PTRACE_PEEKDATA` and `PTRACE_POKEDATA` are limited to transferring one machine word of data at a time, requiring two context switches for each operation.

To measure the performance improvements generated over *ptrace* by allowing reading and writing arbitrarily-sized memory sections (Section 3.2.4), we repeatedly ($n = 200$) transfer 32 MiB of data from and to the heap of a target process and measure the time taken. Obviously, transferring that amount of data is rare in debugging scenarios (short of programs that analyze the entire memory space of a target process), but was necessary here in order to gain reliable measurements independent of system load.

On average, *ptrace* takes 2.51 s ($\sigma = 0.09$s) to read and 2.47 s ($\sigma = 0.07$s) to write 32 MiB of data. In comparison, `plutonium-dbg` takes 0.03 s ($\sigma = 0.0014$s) and 0.04 s ($\sigma = 0.0018$s) respectively. Figure 1 shows these values on a logarithmic scale, with error bars representing three standard deviations (3σ) in either direction.

From this, we compute the expected speedup factors and their respective standard deviations: For reading, the mean speedup factor is 78.62 ($\sigma = 4.43$). Writing is expected to be around 61.20 times faster ($\sigma = 3.21$)[11].

6 DISCUSSION

From Table 3, we can see that of the fourteen tested methods to detect the presence of a debugger, only three succeeded, compared to up to 13 for GDB. Similarly, Figure 1 shows the performance improvements that can be realized over *ptrace*'s system call interface.

[11] Repeating the measurements with 8 MiB of data resulted in similar speedups (84.27 with $\sigma = 7.05$ for reading and 64.77 with $\sigma = 2.95$ for writing).

6.1 Evaluation results

Out of the three failed test cases, the first (8) is based on scanning the program's code segment for breakpoints or other modifications. This approach will generally detect all software breakpoints—the most common workaround is making use of the CPU's hardware breakpoint functionality. There are, however, means by which these checks can be circumvented. A possible approach can be found in Section 6.3.

The other two test cases (10, 11) in which `plutonium-dbg` was detected specifically targeted single-stepping. The `icebp` instruction (`F1`) sends the same CPU interrupt that single-stepping does. The resulting SIGTRAP is then consumed by the debugger, as if it had come from a normal step. Further analysis of the source instruction that triggered the interrupt may help alleviate this issue.

To work around the remaining detection method (11) we have to hide the real contents of the trap flag from any user-space process. Usually, the trap flag is accessible to the program and exposed to user-space. The only ways for programs to directly obtain the value of the flag without the help of the kernel are the `pushf` instruction and its 32- and 64-bit equivalents, which push parts of the RFLAGS register including the trap flag to the stack. Because single stepping only occurs at inspection points controlled from within our kernel module, it would theoretically be feasible to decide whether the target instruction is a `pushf` and modify the resulting bits pushed on the stack accordingly. Unfortunately, it may not even be possible to intercept and analyze all instructions in this manner, because some operations on the stack segment register `ss` disable interrupts temporarily [21]. This also still leaves open other paths of obtaining the processor flags with kernel assistance, including register access through *ptrace* or the `struct sigcontext` on the stack of any signal handler invoked in the program.

The performance increase gained over *ptrace* is significant, although likely not relevant in practice. For one, users that have access to system call functionality (e.g. when using the kernel module directly from C code) will prefer the even faster `process_vm_readv`

and `process_vm_writev` system calls that are already provided by the Linux kernel. Similarly, when `plutonium-dbg`'s GDB server is used, the bottleneck will not be querying the data, but encoding the raw data for the (serial) GDB remote protocol.

6.2 Other detection methods

We have not yet addressed methods specific to `plutonium-dbg` by which malicious software can detect the presence of `plutonium-dbg`. As far as we know, there is no direct way to query whether a debugger is *attached* like there is with *ptrace*. In particular, the XOL area in which *uprobes* executes probed instructions does not appear to be visible from user-space, nor does the list of installed probes. However, it is certainly possible to determine whether `plutonium-dbg` is present (and loaded) on the system at all.

The list of kernel modules is accessible to all users via `lsmod` or `/sys/module/`, and of course `plutonium-dbg` is not hidden in that list. Similarly, the IOCTL debugging interface has to be exposed to the user in order to make the kernel module usable. The presence of `/dev/debugging` and the associated entries in `/proc/devices` and `/sys/class/debugging` give away that `plutonium-dbg` is loaded[12].

While it would be possible to use methods from rootkit development to attempt to hide the debugger from user-space, we found no indication that malware alters its behavior based merely on the fact that a debugger is *installed* as opposed to *in use*. One solution could be to intercept and filter the results of module list or file system traversing system calls.

6.3 Limitations and future work

At the moment, `plutonium-dbg` can be detected through its single-stepping functionality and by programs that search their address space for breakpoints. We discussed earlier that it will require more efforts to be able to work around the former, but the latter could feasibly be addressed.

On modern Intel `x86-64` CPUs, we can set pages containing executable code as execute-only through Memory Protection Keys and other virtualization instructions (as some hypervisors do) and can reliably trigger a page fault if a program tries to read its own code segment. In the page fault handler, we could then swap permissions to read-only (or read-write) and remove all breakpoints on that page. This allows the program to only access the unmodified version of its code. When execution continues, the page fault handler is invoked again (because the page is no longer executable) and the breakpoints can be restored. Further research in this direction could allow `plutonium-dbg` to avoid being detected even when software breakpoints are in use. As long as the number of breakpoints does not surpass the number of architecturally provided breakpoints, `plutonium-dbg` could also fall back to using hardware breakpoints, although at least on `x86-64` these are visible to the user through `PTRACE_PEEKUSER`.

In this work, we limited our evaluation to `x86-64` machines, but the kernel module is not limited to any specific architecture[13]. At least in theory, *uprobes* and single-stepping (and therefore the full feature set of `plutonium-dbg`) are also both supported on 32-bit x86 as well as on the ARM64 and PowerPC architectures and on IBM mainframes (`s390`). The ARM, MIPS, and SPARC architectures also support *uprobes*, but attempting to step over individual user-mode instructions on those systems will result in an error (`arch_has_single_step` is left undefined). Further analysis of `plutonium-dbg`'s performance on other architectures than `x86-64` would be a valuable addition, especially when it comes to single-stepping–based detection methods that may differ from platform to platform. Another interesting extension to `plutonium-dbg` would be an ARM/ARM64 port of the GDB server, to enable debugging on the most common Android systems.

One of the main shortcomings of the current version of the debugger is the ability to only be able to set breakpoints on memory mapped files; this shortcoming is a direct inheritance from using the *uprobes* API. This essentially means that `plutonium-dbg` is currently unable to set breakpoints on allocations stemming from `mmap`'ed pages not backed by memory mapped files. To alleviate this issue, we provide an experimental kernel patch that extends *uprobes* to arbitrary memory areas, including those not backed by a file.

7 RELATED WORK

Shebs [17] uses a *uprobes*-based kernel module alongside a set of patches for GDB to allow placing *global* breakpoints that persist across executions of the same executable. To our knowledge, this is the first occasion that *uprobes* was used for debugging as opposed to tracing.

The *SystemTap* framework also allows the use of *uprobes* to register custom callback functions. However, there is no built-in support for actual debugging use. Registered actions are typically short and non-blocking (e.g. logging) rather than the complex suspend-wakeup cycle used by `plutonium-dbg` [19].

Even before *utrace* and *uprobes*, there were attempts to circumvent the limitations imposed by *ptrace*: The ERESI project's *Embedded ELF Debugger* (`e2dbg`) embeds their own signal handlers in a process using forced dynamic library loading through `DT_NEEDED` or `LD_PRELOAD` in order to avoid the expensive context switches that happen when a breakpoint is hit [7, 22]. Of course, registering signal handlers is a process that is much more open to detection than kernel-based approaches, especially as potentially malicious code can simply override the library's signal handlers.

8 CONCLUSION

In this paper, we showed that *uprobes* can form a reliable basis for a debugging interface that avoids some of the issues presented by *ptrace*.

Our approach to debugging will likely never replace *ptrace*. Not only is *ptrace* already widely used, but backward compatibility with legacy applications will force its persistence in the kernel more than likely [20]. To be clear, *ptrace* is—despite all of its issues—not a *bad* option for traditional debugging.

However, when it comes to malware analysis and reverse engineering, we have demonstrated that *ptrace* makes it easy for targets to interfere with the debugging process, and that other alternatives are needed to approach such software. We believe that the

[12] Theoretically, so does the *kprobe* that is registered on the `get_signal` function (Section 4.2.3)—it is listed in `/sys/kernel/debug/kprobes/list`. However, most systems require superuser permissions in order to access this file.

[13] The GDB server relies on the architecure-specific register layout, and is currently limited to `x86-64`, but could be generalized to other architectures.

hybrid kernel module / user-space client approach presented in this work—despite its shortcomings—has tremendous potential to improve debugging processes by augmenting the capabilities already present in existing debuggers.

9 AVAILABILITY

`plutonium-dbg` is freely accessible on GitHub:

https://github.com/plutonium-dbg/plutonium-dbg

REFERENCES

[1] Mojtaba Bagherzadeh, Nafiseh Kahani, Cor-Paul Bezemer, Ahmed E. Hassan, Juergen Dingel, and James R. Cordy. 2018. Analyzing a decade of Linux system calls. *Empirical Software Engineering* 23, 3 (June 2018), 1519–1551. https://doi.org/10.1007/s10664-017-9551-z

[2] Ramon Caceres. 1984. *Process Control in a Distributed Berkeley UNIX Environment.* Technical Report. University of California, Berkeley, Department of Electrical Engineering and Computer Sciences.

[3] Xu Chen, Jon Andersen, Zhuoqing Morley Mao, Michael Bailey, and Jose Nazario. 2008. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN).* IEEE, 177–186. https://doi.org/10.1109/DSN.2008.4630086

[4] Jonathan Corbet. 2010. Replacing ptrace(). LWN. Retrieved 2018-06-09 from https://lwn.net/Articles/371501/

[5] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. 2018. Understanding Linux Malware. In *2018 IEEE Symposium on Security and Privacy (SP).* 870–884. https://doi.org/10.1109/SP.2018.00054

[6] Juan-Mariano de Goyeneche and Elena Apolinario Fernández de Sousa. 1999. Loadable kernel modules. *IEEE Software* 16, 1 (Jan. 1999), 65–71. https://doi.org/10.1109/52.744571

[7] ELFsh crew [Julien Vanegue and Sebastien Soudan]. 2005. Embedded ELF Debugging: the middle head of Cerberus. *Phrack Magazine* 11, 63, Article 9 (May 2005). http://phrack.org/issues/63/9.html

[8] Mohamad Gebai and Michel R. Dagenais. 2018. Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead. *ACM Comput. Surv.* 51, 2, Article 26 (March 2018), 33 pages. https://doi.org/10.1145/3158644

[9] Brendan Gregg. 2015. Linux uprobe: User-Level Dynamic Tracing. Brendan Gregg's Blog. Retrieved 2018-06-10 from http://www.brendangregg.com/blog/2015-06-28/linux-ftrace-uprobe.html

[10] Marion Guthmuller, Lucas Nussbaum, and Martin Quinson. 2011. Émulation d'applications distribuées sur des plates-formes virtuelles simulées. In *Rencontres francophones du Parallélisme (RenPar'20).* Saint Malo, France.

[11] Marion Guthmuller, Martin Quinson, and Lucas Nussbaum. 2010. *Interception système pour la capture et le rejeu de traces.* Technical Report. Laboratoire lorrain de Recherche en Informatique et ses Applications.

[12] Jim Keniston, Ananth Mavinakayanahalli, Prasanna Panchamukhi, and Vara Prasad. 2007. Ptrace, Utrace, Uprobes: Lightweight, dynamic tracing of user apps. In *Proceedings of the 2007 Linux symposium.* 215–224.

[13] Tom J. Killian. 1984. Processes as Files. In *Proceedings of the Summer 1984 USENIX Conference.* USENIX, 203–207.

[14] Julian Kirsch. 2018. debugmenot. Retrieved 2018-06-12 from https://github.com/kirschju/debugmenot/

[15] Avinesh Kumar. 2008. *TASK_KILLABLE: New process state in Linux.* Technical Report. IBM developerWorks. Retrieved 2018-06-13 from https://www.ibm.com/developerworks/linux/library/l-task-killable/

[16] Chloé Macur. 2014. *Émulation d'applications distribuées sur des plates-formes virtuelles simulées.* Technical Report. Laboratoire lorrain de Recherche en Informatique et ses Applications.

[17] Stan Shebs. 2011. Kernel module for global breakpoints. LWN. Retrieved 2018-06-11 from https://lwn.net/Articles/469769/

[18] Nathan Sidwell, Vladimir Prus, Pedro Alves, Sandra Loosemore, and Jim Blandy. 2008. Non-stop multi-threaded debugging in GDB. In *Proceedings of the GCC Developers' Summit.* 117–127.

[19] Josh Stone. 2011. SystemTap update & overview. Linux Foundation Collaboration Summit 2011.

[20] Linus Torvalds. 2010. Linux Kernel Mailing List. Retrieved 2018-06-09 from https://lkml.org/lkml/2010/1/25/148

[21] Yu-Jye Tung. 2018. Reverse Engineering Reference Manual. Retrieved 2018-06-12 from https://github.com/yellowbyte/reverse-engineering-reference-manual/

[22] Julien Vanegue, Thomas Garnier, Julio Auto, Sebastien Roy, and Rafal Lesniak. 2007. Next generation debuggers for reverse engineering. In *4th Annual Hackers To Hackers Conference (BlackHat Europe).*

[23] Julien Voisin. 2016. pangu. Retrieved 2018-06-13 from https://github.com/jvoisin/pangu/