Tracing for Hardware, Driver and Binary Reverse Engineering in Linux

Mathieu Desnoyers
École Polytechnique de Montréal
mathieu.desnoyers@polymtl.ca

Michel R. Dagenais
École Polytechnique de Montréal
michel.dagenais@polymtl.ca

Abstract

This paper introduces the new Linux Trace Toolkit Next Generation (LTTng) kernel tracer and its analysis counterpart, Linux Trace Toolkit Viewer (LTTV), a fully extensible text and graphical trace viewer. It will focus on how these tools can be used in the security field, particularly for reverse engineering.

Using a tracer to reverse engineer a software "black box" can help understanding its behavior. Such a software can be a either a driver, a library or a multithreaded application : the tracer can log every interaction between the operating system and the program. It can help eluding detection of sandboxes and debuggers due to its small performance impact compared to library wrappers and debuggers. It can collect every system call made by every program which can be later used for fuzzing.

It is not, however, limited to process examination : one could use the kernel instrumentation to reverse engineer a driver controlling a piece of hardware. This tracer should be seen as a system wide monitor for your system : It gives you the opportunity to monitor the hardware, the OS, the libraries and the programs and analyse the information with integrated plugins.

This paper will explain how you can use LTTng and LTTV for reverse engineering and how you can extend it further.

1 Introduction

Collecting information about the behavior of a computer program greatly helps understanding its inner structure. In the exercice of reverse engineering, it can save a lot of time and effort. However, the information gathering tools themselves have a non-zero impact on the studied system (Heisenberg effect), which can be used by a program that wishes to elude debuggers and tracers.

This paper presents some widely used profiling and event-based performance monitoring tools. It then focuses on the latter class of tools by presenting Linux Trace Toolkit Next Generation (LTTng), a kernel tracer in the Linux environment. It will present the particularities of this tracer relevant to the computer security field. It will present real-life debugger detectors examples and describe how LTTng can avoid being detected by such techniques.

2 Performance and Behavior Analysis

To gain a thorough understanding of a program's internals, many approaches are currently available. They mostly vary in the level of detail they give, the intrusiveness of their approach (how much impact they have on the running program), whether they can directly instrument a binary object or they require the user to recompile an instrumented version of the application. In order to fulfill their goal, some tools will use a partially statistical approach to lessen their impact while others will accept the overall system slowdown generated by a massive data output.

The profiling tools gprof, prof and tprof are based on static instrumentation of the applications (require recompiling). They count the number of entry and exit for each function and use the interrupt timer to sample the ratio of time spent in each code region. They have a relatively small impact on the running program and are good to create a list of the functions using the most CPU time within a program. Oprofile is a system-wide statistical profiler that permits profiling the Linux kernel, libraries and programs.

System wide resource usage summary tools (time, top, nmon) use data gathered by the operating system each time a scheduler call occurs. They are good to give a live overview of the system's resource usage or construct graphs with a minimal impact on the system.

System wide performance analysis can also be done by using event-based performance monitoring tools like LTT[4], SystemTAP[2], LKST and LKET. Those tools are based on instrumentation of the operating system (static or dynamic) which extracts useful information about the kernel and user space programs either to a trace (LTT, LKST, SystemTAP/LKET) or as a summary (SystemTAP).

Those tools, in addition to be useful to understand performance issues, give the ability to track a program's behavior through a trace of its execution. Some programs produce system wide traces (LTT, LKST, LKET) which are useful to understand multithreaded and multiprocesses applications. Other tools, like strace and ltrace, gives a detailed list of each interaction a program has done with, respectively, the operating system and libraries. Those tools, in order to provide a detailed list of the program's action, use various mechanisms to intercept the program's execution, making their approach more intrusive.

A second, well known class of behavior analysis tool, are the debuggers. Gdb is an example of user space application debugger, kgdb is a kernel debugger. This approach gives full access to the memory of a running program and permits hardware assisted breakpoints and watchpoints to stop execution at precise locations or when a particular memory access is made. They have, however, the highest impact on the timings of a running programs and give a detailed view of a particular moment in the program execution, which is less appropriate to follow the sequential behavior of a program.

3 LTTng/LTTV Linux Trace Toolkit

Event-based performance monitoring tools are called "tracers". The following section uses two concepts related to this category extensively : an "event" can be defined as an information record with a time stamp. Many events gathered together are called a "trace".

Linux Trace Toolkit Next Generation (LTTng) is a tracer able to extract information from the

Linux kernel, user space libraries and from programs. It is based on instrumentation of the executables, which enables the user to get information from a recompiled Linux kernel and to also get detailed information about a program by instrumenting it.

Linux Trace Toolkit Viewer (LTTV) is an integrated modular analysis tool, which offers a GUI framework and text output. Contributors can create their own view of the information by creating plugins. The analysis tool can combine information coming from multiple information sources, such as the kernel, user space programs, user space libraries, multiple kernels running in a virtualized environment and perform efficient merging of this information for the analysis plugins.

This project focuses on having a very low impact on the system studied, both in term of performance and system behavior.

4 Application in Reverse Engineering

The LTTng tracer and its analysis tool can be useful in various reverse engineering applications. To list a few, reverse engineering programs, libraries, drivers, network stacks or the whole operating system can be made easier with this tool.

Different usage scenarios are possible, depending on the type of executable that has to be analysed. The first one is the study of a program as a white box : this is the case when the source code is available. A programmer can then modify the source code and recompile to extract information at key points.

The second scenario is the study of an executable as a black box, when only the binary executable is available. It is still possible to learn about the program using the kernel instrumentation and monitoring the program intercation with the operating system by recording system calls, signals, interprocess communication and other key events. It will give useful information about how the program uses the filesystem, the network, its memory maps ; everything that passes through the operating system. It is also possible to use breakpoint based approaches, such a kprobes and SystemTAP, to instrument the binary executables at key points to call a LTTng probe to write relevant information in the trace.

This type of instrumentation is especially useful when trying to understand multi-threaded and multi-process applications. For instance, following the interaction between the X server and a graphical client often involves gathering information from various sources, which is made easier by a system-wide tracer.

Because of its small impact on the system, this kind of tracer can also be used to study applications which elude debuggers. This topic will be treated in more details in the following sections.

4.1 User Space Programs

4.1.1 Anti-debugging techniques

In Linux, both process tracers (strace, ltrace) and debuggers (gdb) use the “ptrace” mechanism extensively. It permits them to connect to a process to read and modify its memory, intercept its signals and system calls. Gdb typically uses these abilities to insert breakpoint instructions instead of the normal program code at specific locations. It also enables the debugger to use hardware watchpoints to see when a particular memory address is accessed or modified.

Silvio Cesare presented a technique[1] to detect the use of ptrace on a process based on the fact that a particular process can only be ptraced once. Therefore, a program that does a ptrace system call on itself will detect if there is a debugger or a process tracer attached on it.

Going further, I decided to demonstrate another approach to detect debuggers that doesn't imply using the ptrace system call. The idea is simple : using a precise time base (the CPU cycle counter) to detect the duration of a particular code section can detect if a ptrace process is collecting information about the program. This approach is possible due to the high impact the round-trip to the debugger process has on the system compared to the case when it is not being debugged. The different scenarios have been tested on a 3GHz Pentium 4. They are summarized in table 1.

The duration of events such as a trap (generating a SIGTRAP signal) can detect the presence of a debugger. To detect the presence of strace and ltrace, the duration of a signal, generated by a kill() system call on the process itself can be used to determine if it is being run under strace which intercepts the signals. An alternative method to discover strace is to call an unexisting system call and see how much time it takes. To be stealthier, one could call a do_gettimeofday system call, which is fast and used everywhere. It would lead to results similar to an unexisting system call without triggering alarm bells.

Clearly, these methods must be called in loops to make sure they are not too much affected by interrupts. They are also dependant on the speed of the processor and the architecture on which the binary is executed, but with proper calibration, they can be effective and hard to discover.

To verify how much the system load can modify the number of cycles per operation, we run each detection technique under low and heavy loads. It results that the number of cycles spent in each operation doubles. We can therefore consider a slowdown of 4 for the noise level (including a security margin of 2).

The impact of gdb on the trap and kill depends on the user input, while it does not impact the system call because it is not monitored. The impact of strace is a slowdown of 2.42 for the trap, 96 for the kill and 417 for the system call. Due do the imprecision of the calibration and the incoming events (interrupts) which depends on the system load, the trap technique could not determine the presence of strace due to its small impact. The ltrace slowdown is 11 for the trap, 217 for kill and 554 for the system call.

The LTTng impact on the operations is a slowdown of 1.77 for the trap, 2.40 for kill and 2.42 for the system call. Those differences do not offer a sufficient margin to determine the presence of LTTng.

4.1.2 Virus

An example executable which uses such a debugging technique is the Virus Linux.RST.B. Its advisory from Kaspersky Labs[3] states that it uses anti-debugging techniques, infects Linux binaries found in the current directory and in /bin. It also states that it adds a backdoor on the network interface.

In order to study its behavior, I created a virtual machine in Qemu and installed a Linux kernel instrumented with the LTTng tracer in it. I took a trace on the machine and launched the virus, recording all its actions on the operating system.

The result is that I could follow a trace of its execution, which would have been impossible with strace due to the debugger detecing techniques. The execution trace, in the LTTV tool,

	normal (min)	high load (max)	gdb	strace	ltrace	LTTng
trap	35108	63097	790211960	85148	405713	62152
kill	13042	28927	521129438	1257112	2832143	31417
system call	1890	3082	1920	789653	1048102	4582

Table 1: Number of cycles required for calls to the operating system (10 tests)

shows that it uses the “ptrace” method to detect debuggers 1.

Following the program execution, we can determine that it executes a temporary file ".para.tmp" which creates three other processes. It opens and lists the current directory, the /root directory, and then modifies the binary files in /bin. The advisory did not report the activity on /root, so probably a kernel tracer would have given this extra information. A secondary thread then binds a socket on the network interface and listens for incoming connexions.

4.1.3 Password snooping

Being able to extract any kind of information about the system can be very powerful when used appropriately. For instance, having the ability to dump the first bytes of each read() and write() system calls enables us to dump each character received and sent by each program.

Clearly, this includes “su” and “ssh”, which makes extraction of the password typed by the users very easy. After a trace is taken, one has to find the interesting process (su), select the interesting information by using the LTTV filter (Figure 2) and see the password typed (Figure 3).

4.2 Kernel

Kernel reverse engineering is particularly useful for binary-only drivers. They run with kernel privileges and interact directly with the kernel, wich makes them harder to trace.

The LTTng kernel tracer can be used with kprobes to add breakpoints on each exported symbol of the binary-only driver (also static symbols taken by kallsyms if debugging information is present, which is unlikely). Breakpoints can also be put at each kernel site that are called by the module : each undefined symbol referred by the kernel module is a candidate for instrumentation. The kernel stack dump is another part of LTTng that can help understanding the drivers better : at specified locations in the kernel, we can obtain the complete call stack of the program (just like a kernel OOPS). It will be recorded in the trace with the rest of the information.

LTTng can also be used for system call fuzzing : it can be used to extract the normal parameters passed to the kernel by standard programs in the first run. Then, this information can be feeded to a fuzzer that can test the system call error handlers by slightly modifying these parameters.

4.3 Hardware

Very precise information can be extracted from the kernel about the hardware connected to it. The instrumentation of interrupt handlers makes the frequency of interrupts available. Information about their maximum, minimum and average duration is also given.

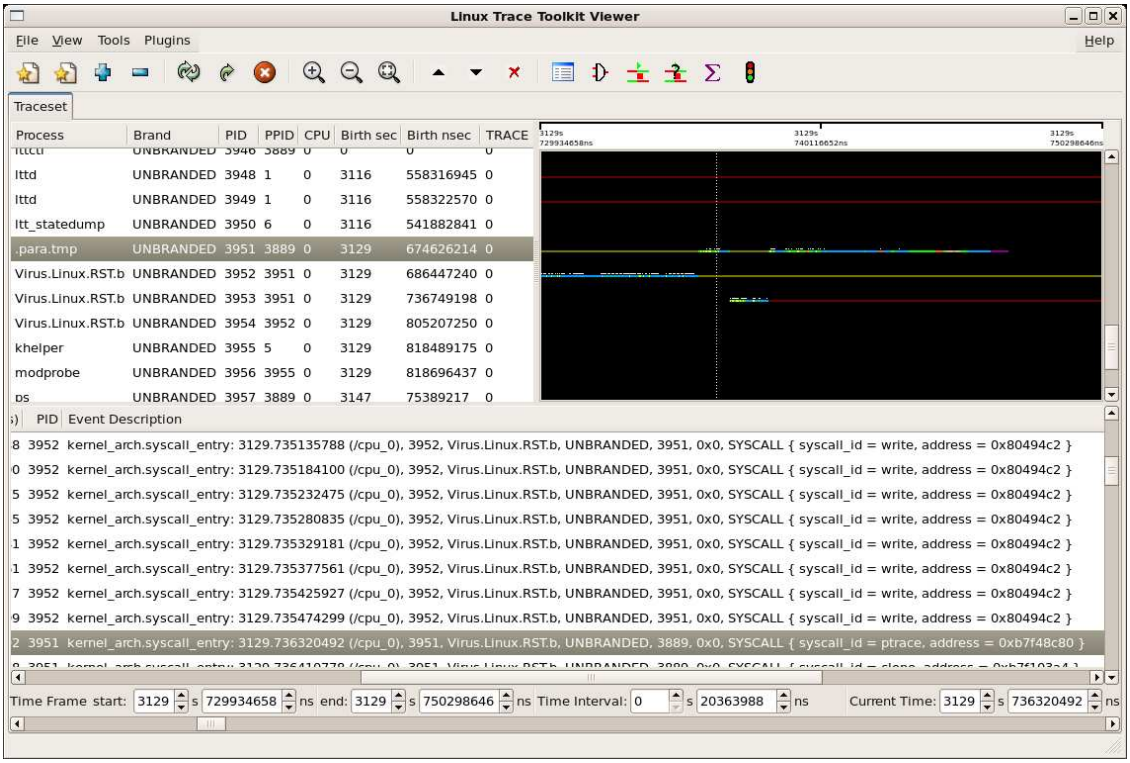

Figure 1: Virus anti-debugging

We can also think of polling the device for its state by doing a memory read of its registers either periodically (hooked on the timer interrupt) or when an IRQ from the device is received.

4.4 Conclusion

Overall, in the land of performance and behavior analysis tools, tracers offer precious information about the execution of complex programs. The LTTng tracer and LTTV, the analysis tool, provide an efficient, extensible and low impact system-wide tracing, which benefits reverse engineers for various usage scenarios, from tracing of binary-only executables to drivers.

You are welcome to try the tool and post your enhancements. The project is available under the GPL license at `http://ltt.polymtl.ca`.

References

[1] Silvio Cesare. Fooling the debugger. In *VX Heavens,* `http://www.vx.netlux.org/lib/vsc04.html`, 1999.

[2] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Brad Chen. Locating system problems using dynamic instrumentation. In *OLS (Ottawa Linux Symposium) 2005* `http://sourceware.org/systemtap/systemtap-ols.pdf`, 2005.

[3] Costin Raiu. Virus.linux.rst.b. In *Virus Encyclopedia,* `http://www.`

Figure 2: LTTV filter expression to find a password in a trace

viruslist.com/en/viruses/encyclopedia?virusid=21734, 2002.

[4] Karim Yaghmour and Michel R. Dagenais. The linux trace toolkit. May 2000.

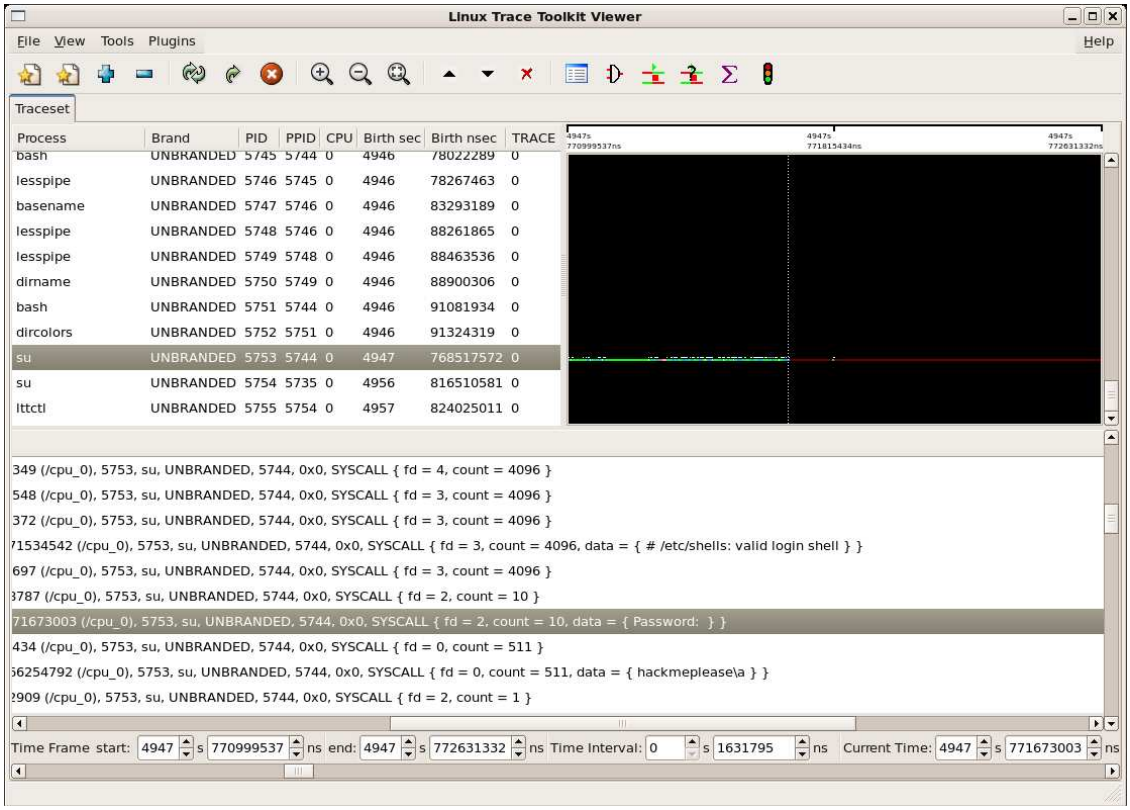

Figure 3: Expected password in a trace coming from su