Configurable system call tracer in QEMU emulator

A.V. Ivanov <alexey.ivanov@ispras.ru>
P.M. Dovgaluk <pavel.dovgaluk@ispras.ru>
V.A. Makarov <vladimir.makarov@ispras.ru>
Yaroslav-the-Wise Novgorod State University,
41, Great St. Petersburg st., Velikiiy Novgorod, 173003, Russia

Abstract. Sometimes programmers face the task of analyzing the work of a compiled program. To do this, there are many different tools for debugging and tracing written programs. One of these tools is the analysis of the application through system calls. With a detailed study of the mechanism of system calls, you can find a lot of nuances that you have to deal with when developing a program analyzer using system calls. This paper discusses the implementation of a tracer that allows you to analyze programs based on system calls. In addition, the paper describes the problems that I had to face in its design and development. Now there are a lot of different operating systems and for each operating system must be developed its own approach to implementing the debugger. The same problem arises with the architecture of the processor, under which the operating system is running. For each architecture, the analyzer must change its behavior and adjust to it. As a solution to this problem, the paper proposes to describe the operating system model, which we analyze. The model description is a configuration file that can be changed depending on the needs of the operating systems. When a system call is detected the plugin collects the information downloaded from the configuration file. In a configuration file, arguments are expressions, so we need to implement a parser that needs to recognize input expressions and calculate their values. After calculating the values of all expressions, the tracer formalizes the collected data and outputs it to the log file.

Keywords: QEMU; configurable system calls; debugging; plugin; system calls; tracing.

DOI: 10.15514/ISPRAS-2018-30(3)-7

For citation: Ivanov A.V., Dovgaluk P.M., Makarov V.A. Configurable system call tracer in QEMU emulator. Trudy ISP RAN/Proc. ISP RAS, Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 3, 2018, pp. 93-98. DOI: 10.15514/ISPRAS-2018-30(3)-7

1. Introduction

Sometimes programmers face the task of analyzing the work of a compiled program to find its flaws, defects, and even search for malicious code in it. To analyze the work of such applications, we have to study their binary code or try to decompile the code, which is a laborious task. In order to simplify the analysis of applications, we can use the system calls of this application. System calls provide an essential interface between a program and the operating system. It is possible to track which system calls the application makes, and draw conclusions about the behavior of the program. This method allows us to debug the application without delving into the level of instructions and architecture features, thereby reducing the time required to find the problem.

Debugging applications using system tracing can be done inside the operating system, but still a number of problems arise:

- strong dependence of the debugger on the operating system;
- impossibility to run several debuggers at the same time;
- inaccessibility to the privileged execution;
- necessity to secure the operating system when analyzing programs that have harmful effects.

To solve these problems, we can use the virtual machine tools. In this way, we can debug applications in a wide range of different operating systems running under different processor architectures.

2. Approach and uniqueness

To date, several debuggers allow us to trace an application using system calls. All these debuggers have a drawback - they do not provide enough portability of the debugger within different operating systems and processor architectures. We offer a new approach to implementing the debugger through system calls, by loading all the information necessary for tracing from the configuration file. The configuration files will allow us to easily configure and change the parameters needed for debugging, and to simplify the addition of support for new operating systems and architectures without recompiling the program and learning the debugger code.

It was decided to implement the debugger under the virtual machine QEMU [1], using the plugin mechanism. QEMU is an open source virtual machine that emulates the hardware of various platforms. This virtual machine supports the emulation of a large number of processor architectures such as x86, PowerPC, ARM, MIPS, SPARC, m68k. In addition, this simulator supports the launch of a large number of different operating systems.

Now, there is a plugin mechanism for QEMU implemented by ISP RAS [2], which allows us to connect developed plugins to a virtual machine during its both startup and operation. The implementation of the plugin mechanism enables each additional translation of the instruction to be substituted by an additional code for execution, when this instruction is called. This mechanism is suitable for debugging through system calls, so it was decided to use it.

In addition, various mechanisms of the system call play an important role. The classical way of implementation is the use of interrupts. With the help of interrupts, control is transferred to the kernel of the operating system, with the application having to enter the number of the system call and the necessary arguments into the corresponding registers of the processor.

For many RISC processors, this method is the only one; however, the CISC architecture has additional methods. The two mechanisms developed by AMD and Intel are independent of each other, but, in fact, perform the same functions. These are SYSCALL / SYSRET or SYSENTER / SYSEXIT statements. They allow us to transfer control to the operating system without interrupts.

Each operating system supports values returned from the system call, which are passed as reference types when the system call handler is called. During the execution of the system call, the service procedure records the required values if necessary by the available links, after which the system call is exited.

One of the main tasks that we had to face was the task of supporting the plugin of different operating systems and processor architectures. The solution to this problem was the interface with the configuration file. The configuration file makes the debugger more flexible and customizable. With its help, we can disconnect a certain mechanism of system calls from the trace or disable unnecessary system calls. In addition, such a mechanism makes it easier to add support for new operating systems and processor architectures.

To implement the interface with the configuration file, it was necessary to study a wide range of different operating systems and processor architectures. After gathering the necessary information, we can determine the information necessary for debugging: what type of system call is supported by SYSCALL / SYSRET or SYSENTER / SYSEXIT and their opcodes; location of system call arguments; a list of system calls, with the name of each system call, its code, and the list of arguments. Thus, by developing an interface for debugger and configuration file interfacing, we can add support for operating systems without going into the debugger code.

When implementing the debugger interaction interface with the configuration file, it became necessary to recognize the various expressions read from the file. For this task, we used the generator of the bison parser and developed the corresponding grammar [3].

3. Background and related work

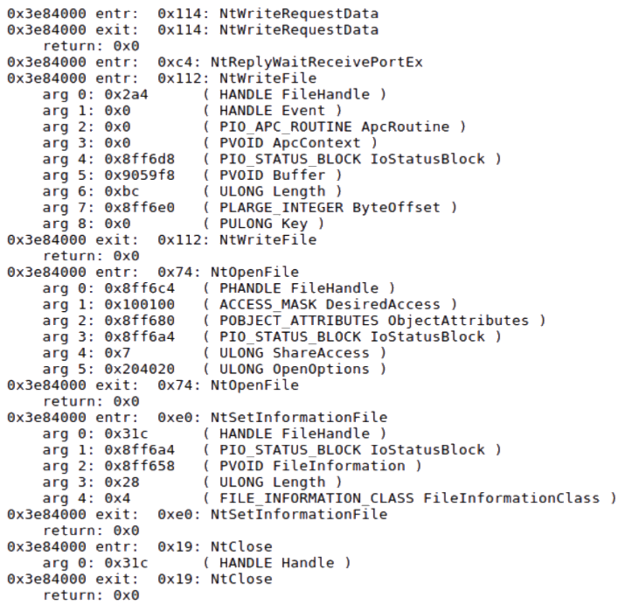
Now, there are several debuggers to solve existing problems. Nitro [4] allows us to trace system calls, but it works only under Intel x86 architecture. Another debugger – Panda [5], can also trace system calls, supporting such operating systems as Linux, Windows 7, Windows XP and two architectures of the i386 processor and ARM. The description of all system calls is found in the code of this debugger, because of which this approach makes it difficult to add support for new operating systems and processor architectures, and worsens the flexibility in configuring the plugin, since the system debugger settings mechanism is not provided.

4. Conclusion and discussion

Based on the results of the work done, the plugin was developed in the QEMU virtual machine, with which we can trace and debug an application using system calls. As input to the plugin, the configuration file corresponding to the operating system running in the QEMU virtual machine and corresponding to the selected processor architecture is used.

The structure of the configuration file consists of 4 parts. The first part provides information about the operating system, its name and bit capacity. The second part is responsible for the supported mechanisms of system calls. The next part contains the location of the system call arguments. The last part includes a list of all available system calls and service information about the arguments of the system calls.

Because of the plugin's work, a log file containing all the system calls that the plugin has intercepted is created. Each system call displays detailed information: the name and value of each system call argument, the number of the thread of execution from which this system call was made and the value that returned the system call after execution. Fig. 1 presents a small fragment of the output file that was created by the implementation of the plugin launched in the windows XP operating system and the i386 processor architecture.



```
0x3e84000 entr:  0x114: NtWriteRequestData
0x3e84000 exit:  0x114: NtWriteRequestData
    return: 0x0
0x3e84000 entr:  0xc4: NtReplyWaitReceivePortEx
0x3e84000 entr:  0x112: NtWriteFile
    arg 0: 0x2a4       ( HANDLE FileHandle )
    arg 1: 0x0         ( HANDLE Event )
    arg 2: 0x0         ( PIO_APC_ROUTINE ApcRoutine )
    arg 3: 0x0         ( PVOID ApcContext )
    arg 4: 0x8ff6d8    ( PIO_STATUS_BLOCK IoStatusBlock )
    arg 5: 0x9059f8    ( PVOID Buffer )
    arg 6: 0xbc        ( ULONG Length )
    arg 7: 0x8ff6e0    ( PLARGE_INTEGER ByteOffset )
    arg 8: 0x0         ( PULONG Key )
0x3e84000 exit:  0x112: NtWriteFile
    return: 0x0
0x3e84000 entr:  0x74: NtOpenFile
    arg 0: 0x8ff6c4    ( PHANDLE FileHandle )
    arg 1: 0x100100    ( ACCESS_MASK DesiredAccess )
    arg 2: 0x8ff680    ( POBJECT_ATTRIBUTES ObjectAttributes )
    arg 3: 0x8ff6a4    ( PIO_STATUS_BLOCK IoStatusBlock )
    arg 4: 0x7         ( ULONG ShareAccess )
    arg 5: 0x204020    ( ULONG OpenOptions )
0x3e84000 exit:  0x74: NtOpenFile
    return: 0x0
0x3e84000 entr:  0xe0: NtSetInformationFile
    arg 0: 0x31c       ( HANDLE FileHandle )
    arg 1: 0x8ff6a4    ( PIO_STATUS_BLOCK IoStatusBlock )
    arg 2: 0x8ff658    ( PVOID FileInformation )
    arg 3: 0x28        ( ULONG Length )
    arg 4: 0x4         ( FILE_INFORMATION_CLASS FileInformationClass )
0x3e84000 exit:  0xe0: NtSetInformationFile
    return: 0x0
0x3e84000 entr:  0x19: NtClose
    arg 0: 0x31c       ( HANDLE Handle )
0x3e84000 exit:  0x19: NtClose
    return: 0x0
```

Fig. 1. Part of the output file of the plugin

Upon the information gathered in the log file, we can analyze the operation of the debugged application running inside the virtual machine. The operating system load

time when using the developed plugin is increased 20% slowdown compared to the time of the operating system loading without this plugin.

Acknowledgments

The work was supported by the Russian Foundation of Basic Research (research grant 18-07-00900 A)

References

[1]. F. Bellard. QEMU, a fast and portable dynamic translator. In Proceedings of the Annual Conference on USENIX Annual Technical Conference, 2005.
[2]. Vasiliev I.A., Fursova N.I., Dovgaluk P.M., Klimushenkova M.A., Makarov V.A. Modules for instrumenting the executable code in QEMU. Problemy informacionnoj bezopasnosti. Komp'juternye sistemy [Journal of Information Security Problems. Computer Systems], no. 4, 2015, pp. 195-203 (in Russian).
[3]. GNU Bison [HTML] (https://www.gnu.org/software/bison/)
[4]. Nitro [HTML] (http://nitro.pfoh.net/index.html)
[5]. Panda. Plugin: syscalls2. [HTML] (https://github.com/panda-re/panda/blob/master/panda/plugins/syscalls2/USAGE.md)

Конфигурируемый трассировщик системных вызовов в эмуляторе QEMU

А.В. Иванов <alexey.ivanov@ispras.ru>
П.М. Довгалюк <pavel.dovgaluk@ispras.ru>
В.А. Макаров <vladimir.makarov@ispras.ru>
Новгородский государственный институт имени Ярослава Мудрого,
173003, Россия, г. Великий Новгород, ул. Б. Санкт-Петербургская, д. 41

Аннотация. Разработчики программ часто сталкиваются с проблемой анализа работы различных приложений. Для этого существует большое множество различных средств отладки, отслеживания, трассировки написанных программ. Одним из таких средств является анализ работы приложения через системные вызовы. При детальном изучении механизма системных вызовов, можно обнаружить большое количество нюансов, с которыми приходится столкнуться при разработке анализатора программ с использованием системных вызовов. В статье рассматривается реализация трассировщика, который позволяет анализировать программы на основе системных вызовов, и проблемы, с которыми пришлось столкнуться при его проектировании и разработке. На данный момент существует большое количество различных операционных систем и для каждой операционной системы должен быть разработан свой подход в реализации отладчика. Такая же проблема возникает и с архитектурой процессора, под которой запущена операционная система. Для каждой архитектуры, анализатор должен менять своё поведение и подстраиваться под неё. В качестве решения данной проблемы, в статье предлагается описать модель операционной системы, которую мы анализируем. Описание модели представляет собой конфигурационный файл, который может быть изменён в зависимости от потребностей операционных систем. При обнаружении системного вызова, в его обработчик передаются аргументы и вся сопутствующая информация, загруженная из конфигурационного файла. Изначально, в конфигурационном файле, все аргументы представляют собой выражения, поэтому возникает необходимость также реализовать синтаксический анализатор, которому необходимо распознать входные выражения и посчитать их значения. После просчёта значений всех выражений, трассировщик формализует собранные данные и выводит их в лог файл.

Ключевые слова: QEMU; конфигурируемые системные вызовы; настраиваемые системные вызовы; отладка; отладчик; плагин; системные вызовы; трассировка; трассировщик.

DOI: 10.15514/ISPRAS-2018-30(3)-7

Для цитирования: Иванов А.В., Довгалюк П.М., Макаров В.А. Конфигурируемый трассировщик системных вызовов в эмуляторе QEMU. Труды ИСП РАН, том 30, вып. 3, 2018 г., стр. 93-98 (на английском языке). DOI: 10.15514/ISPRAS-2018-30(3)-7

Список литературы

[1]. F. Bellard. Qemu, a fast and portable dynamic translator. In Proceedings of the Annual Conference on USENIX Annual Technical Conference, 2005.
[2]. Васильев И.А., Фурсова Н.И., Довгалюк П.М., Климушенкова М.А., Макаров В.А. Модули для инструментирования исполняемого кода в симуляторе QEMU. Проблемы информационной безопасности. Компьютерные системы, no, 4, 2015г., стр. 195-203
[3]. GNU Bison [HTML] (https://www.gnu.org/software/bison/)
[4]. Nitro. [HTML] (http://nitro.pfoh.net/index.html)
[5]. Panda. Plugin: syscalls2. [HTML] (https://github.com/panda-re/panda/blob/master/panda/plugins/syscalls2/USAGE.md)