Using Automated Reverse Engineering for the Safe Execution of Untrusted Device Drivers

Vitaly Chipounov, George Candea, Willy Zwaenepoel
EPFL (Lausanne, Switzerland)

1. Motivation

The driver architecture employed by modern operating systems presents a safety challenge. The device driver acts as a translator between an operating system and a particular hardware device (e.g., printer, video card, scanner, digital cameras) and usually runs in kernel mode, thus having maximum privilege over the computer where they execute.

As a result, device drivers are responsible for important fractions of OS failures; e.g., in Windows 2000 systems, device drivers account for 27% of crashes reported in the field [6]. In Linux, drivers were found to have 3x to 7x more bugs than the rest of the OS [1]. Device drivers are usually written by the companies that develop the hardware, since they are the only ones with access to the proprietary details of the device's functions. Unfortunately, the company's expertise lies not in developing reliable software, but rather in building hardware. Writing a device driver requires an in-depth understanding of how the kernel works, and debugging kernel-mode code is considerably more difficult than user-mode code.

Buggy drivers do not only crash the system, but they can also compromise security. For instance, last year a zero-day vulnerability was disclosed within a third party driver that is shipped with *all* versions of Windows XP (secdrv.sys, developed by Macrovision as part of SafeDisc). The vulnerability allows a local non-privileged user to elevate his/her privileges to Local System, leading to complete system compromise [7].

Recent systems such as Xen and Nooks have improve reliability by isolating the kernel from buggy device drivers. In our approach, we depart from these proposals, by not executing the driver iself, but rather extracting the information it encodes and in effect "showing" the kernel how to interact with the device driver.

2. Proposed Solution

We consider the driver as an encapsulation of a state machine, describing how certain actions should be achieved using the hardware device (e.g., how to send or receive a data packet using a given network card). If the hardware vendor provided a formal description of this state machine, then it would be possible to verify the state machine's safety and then execute it in the kernel. In the absence of such a description, we consider the proprietary, binary driver as the next best thing to a formal specification of the state machine. We extract this state machine from the driver using a combination of I/O interception at the level of the virtual machine and test generation.

Our solution consists of two parts:

- A reverse-engineering tool that extracts from a binary driver the corresponding state machine in a form that is safe to execute in the kernel
- A mechanism for an OS kernel to use this extracted specification in order to interact with the device, obviating the need for loading and running untrusted third party code

2.1. Legal Reverse Engineering of Drivers

Extracting the state machine could in theory be done by decompiling the binary with the various available tools; this is however an approach that is technically challenging and also borders on illegality.

Instead, we observe the driver's behavior, and infer based on these observations what the underlying state machine is doing. In particular, we run the device driver inside a modified virtual machine, presenting to the driver a virtual version of the device driver it expects. Using the virtual machine monitor, we intercept and record the interactions between the driver and the (virtual) hardware; we use this trace to reconstruct the driver's state transitions. We generate specially designed requests to the driver, that enable us to explore the state machine. Once the state machine is extracted, it is replayed by a special module inside the kernel.

2.2. Obtaining Activity Traces

We use the QEMU emulator, a widely used hypervisor. QEMU has an internal table that maps I/O addresses to device handlers. In order to trace all driver/hardware interactions, we modify the appropriate table entries to redirect all I/O to our own routines, instead of the virtual device; our routines then record the call, pass it to the virtual device, and then record the response.

We record different traces generated in response to changing parameters. For example, when tracing a video card driver, we request multiple screen resolutions from the driver, in order to subsequently be able to identify differences between traces. The generation of "test patterns" is currently hard coded, but will use a feedback loop in which test patterns are generated so as to cover execution paths that previous patterns have not exercised.

2.3. Trace Analysis

The state we need to recover consists of the hardware registers (accessed via I/O ports) and the driver's own in-memory variables. A state transition occurs upon reception of an event from the device or from the operating system. Trace analysis has three important goals: to extract the semantics of hardware registers, identify the patterns in which the driver uses them, and identify the data structures used by drivers internally.

In order to identify the semantics of the status registers, we judiciously generate input patterns that trigger minor variations in driver behavior (e.g., asking the network driver to send packets of varying lengths). We then compute the differences between traces obtained for the various inputs by using the Hirschberg algorithm for longest common subsequence [3]. The statistically significant differences are then used to identify all the hardware registers involved and we then cross-correlate to the input parameters (e.g., for a network driver, if the values of a particular register differ by the same amount as the input packet lengths do, then we conclude that register holds the user-provided length plus some ε).

The utilization patterns of the hardware registers reflect the way in which the hardware device expects to be programmed. For example, polling patterns appear as a sequence of reads to the same register, until its value changes, at which point the reads stop. We supplement trace-based recognition with static analysis of the binary driver; e.g., polling takes the form of a while loop with a condition involving one or more hardware registers being read. The combination of static and dynamic analysis provides significantly higher resolution power.

Finally, to identify the driver's data structures, we intend to use a combination of static analysis of the binary driver with pattern-matching against memory access patterns, intercepted via the virtual MMU. For example, ring buffers are common among network card drivers to keep track of incomming/outgoing packets in a FIFO order; such ring buffers occupy a fixed location in memory which is accessed via a pointer that scans the buffer and wraps around. This approach is analogous to reverse-engineering the hardware register utilization patterns.

2.4. Driver State Machine Playback

We implement the playback component as a Linux kernel module; in some sense, we aim to get the benefits of Singularity [5] without needing a brand new OS. The reverse-engineered state machine is a table describing the transitions that were observed in the original driver.

We expect it is not possible to always extract the entire state machine from all drivers. In this case, we intend to run the reverse-engineered part in the kernel and the untrusted driver in user-mode using UMLinux; when an operation is encountered that cannot be handled by the (safe) kernel component, it is relayed to the user-mode (unsafe) driver. The reverse-engineering process continues over time, progressively moving portions of functionality from the isolated driver into the kernel.

3. Related Work

The idea of using different inputs to identify the semantics of registers was inspired by [4], where different combinations of operands for a given assembler instruction are used to generate the binary form. By comparing the changes, they infer the meaning of the different fields in the opcodes. The Singularity system [5] is entirely written in a type-safe garbage-collected language; the formal specification of the device driver interfaces (software-isolated processes, no sharing of data, communication through specific contract-based channels) ensures that drivers are run safely; we aim to achieve this same level of safety, without requiring that kernels be reimplemented in a different language. In [2], the authors perform automated protocol reverse-engineering; it was this paper that suggested to us the idea of automatically reverse-engineering the state machine from drivers.

4. Conclusion

With our work, we hope to offer modern operating systems the option of being able to interact with hardware devices without having to compromise the safety of the kernel. If our approach succeeds, then we hope it will motivate device manufacturers to no longer ship proprietary software to use their devices, but rather open, verifiable descriptions of how to interact with the hardware. We wish to build a system that allows OS kernels to use such descriptions in order to safely interact with devices.

References

[1] Andy Chou et al. An empirical study of operating systems errors. In *18th ACM SOSP*, 2001.

[2] W. Cui et al. Discoverer: Automatic protocol reverse engineering. In *USENIX Security Symposium*, 2007.

[3] D. S. Hirschberg. Linear space algorithm for computing maximal common subsequences. *Comm. ACM*, 1975.

[4] W. C. Hsieh et al. Reverse-engineering instruction encodings. In *USENIX Annual Technical Conference*, 2001.

[5] G. Hunt and J. Larus. Singularity: Rethinking the software stack. *Operating Systems Review*, 41(2), 2007.

[6] B. Murphy. Personal comm. Microsoft Research, 2002.

[7] Symantec. http://www.symantec.com/enterprise/security_response/weblog/2007/10.