VirtIO and s3k capabilities

Fredrik Gölman

February 26, 2024

1 Vulnerabilities

There are two primary vulnerabilities in play here, both originating from the interplay (or lack thereof) between VirtIO and capabilities of the microkernel s3k. The first vulnerability is the possibility to read and write to memory regions outside of the assigned capabilities. The implementation and design choices made for this part of the problem are not discussed in great detail as the work of the group was split up into two parts for the two vulnerabilities, which is further explained in Section 5. The second vulnerability is the possibility that some other application revokes a capability involving the memory region the device driver uses for its operations.

Several different scenarios can occur from these vulnerabilities including denial-of-service, code injection, and other read and write operations that are not meant to be permitted. There are significant benefits with Direct Memory Access (DMA) as it allows fast transfers of data from Input/Output (I/O) devices without burdening the Central Processing Unit (CPU) so that it can be free to perform other tasks. On the downside, this also introduces significant security issues as the Direct Memory Access Controller (DMAC) essentially can have unrestricted access to the physical memory. Device drivers must thus be rather bug-free (which is not always the case) and uncompromised. In the case of s3k and the VirtIO driver, the driver does not respect the s3k capabilities assigned to other processes. This allows both manipulation of the VirtIO driver data structures and revocation of capabilities including the memory regions of the driver. For example, the writing and execution of program code that is not intended, which was seen in the code injection exercise where we loaded a binary overwriting the content of ”app1” and then executed the other application. Other examples include data leakage reading data from the driver buffers which are not meant to be available as well as denial-of-service situations where revocation occurs for memory regions of the driver while there are enqueued but unfinished operations yet to be processed. File systems are inherently sensitive as they are an interface to user data (and sometimes system data given sufficient privileges).

2 Design Choices

From a conceptual perspective, one could imagine running I/O device drivers in kernel space. However, that quickly becomes troublesome given the large number of different devices in different categories, such as network devices, USB devices, and storage devices, that would need to be supported. It would make the Trusted Code Base (TCB) considerably more difficult to formally verify to ensure it functions as intended. In this project the device drivers are run unprivileged in user space by default, so there is no such choice to be made. The situation still presents challenges as previously mentioned.

Given device drivers often being somewhat sizable and have a high complexity level they may often include bugs. A good approach to enhance security may in many cases be to attempt to formally verify device drivers using DMA for which a framework is provided in [1]. In this section, the focus is not on the driver itself, but on a monitor controlling interaction with the device driver during runtime.

In both of the aforementioned vulnerabilities, the monitor takes control of handling requests of the process. In the one performing operations in memory regions outside of the application's capabilities, a suitable solution may be to move the device driver data structures into the monitor and perform checks when an application wants to enqueue an operation. This is not further discussed here as the aim is on the second vulnerability involving the revocation of capabilities that involve the memory regions of the driver.

Four cases of revocation are being considered:

1. Once the capability is a parent of the driver capability
2. Once the capability is the driver capability
3. Once the capability is a child of the driver capability
4. Once the capability is independent of the driver capability

The second case is equivalent to saying the monitor's capability as the monitor owns the memory region of the driver. By sharing a pointer to the VirtIO device driver data structures the monitor can observe the current status of the driver's rings.

Another design choice is to modify the kernel to disallow any process but the monitor to revoke capabilities. Other applications must thus message the monitor a revocation request. The monitor may immediately grant a revocation of a capability of type 4 without further checks as it is independent of the driver memory region. The other cases are all handled similarly to one another. The monitor checks the VirtIO rings (or rather, the descriptors) to see if there are any entries with addresses in the capability's memory region. If there is not, the monitor takes control and revokes the capability. Otherwise, the capability is marked for future removal once it is no longer in conflict with the operations of the driver. One can imagine this being handled differently such as immediately shutting down the driver and revoking the capability. Iterating through the driver ring and removing entries that conflict with the capability under revocation could be another slightly more graceful alternative. However, both of these options would lead to reliability concerns, and similar to property 2, "Complete Mediation", of the security properties of the Wimpy Kernel [2], the monitor ensures that all "wimp applications'" accesses to shared I/O resources must be mediated.

3 Implementation

The default configuration of VirtIO in this implementation defines a set and two rings. The DMA descriptor set holds $n = 8$ descriptors which defines the operations to be performed. The two rings, the available ring, and the used ring, also hold n entries. The available ring is for the driver to write entries of which descriptors it wants the device to handle. The used ring is for the device to write entries of which descriptors it has finished processing so that they can be recycled. In both rings the entries point to the head of the chain of the DMA descriptors. For both read and write operations (which are the operations being performed in the current implementation), each operation requires three descriptors. The DMA descriptor set and the two rings are allocated non-overlapping in an array-like fashion. The separation between the available ring and the used ring, with the driver writing to the available ring but not to the used ring, and the device writing to the used ring but not the available ring is important from a reliability perspective due to potential concurrency issues.

The current VirtIO implementation recurringly calls the *virtio_disk_rw* function until the entire read/write operation has been performed, where the DMA descriptors are also chained and enqueued, and synchronize with the device in each call, so the risk of running out of descriptors is seemingly not a problem.

In the implementation, there are three applications, app0, app1, and the monitor. App0 constitutes the driver (and also performs the file operations to make sure the queue is populated for testing purposes). App1 constitutes the application wanting to revoke a memory (or PMP) capability, which both attempts to revoke a capability via the system call and once it fails messages the monitor. The monitor is on the server side of the inter-process communication (IPC) and upon receiving a request to revoke a capability first checks what relation the memory region has to the driver memory region according to the previously four defined cases (e.g. parent, child, independent). If the capability is independent, the monitor revokes the capability immediately. If the VirtIO queue is empty, it is also known that there are no conflicts. Otherwise, if there are descriptor addresses in the memory region of the capability being revoked, it is added to a "quarantine", which is a queue-like structure that is periodically iterated through, and capabilities that are no longer in conflict are revoked. The quarantine primarily holds information about what process wants to revoke what capability.

Additionally, the kernel has been modified to only allow the monitor process to revoke capabilities (by PID).

Given some issues with the built-in IPC calls, some of the client side looping for the messaging process occurs on a shared variable rather than the status of the error of the messages process.

4 Countermeasures

The primary countermeasure consists of preventing applications other than the monitor from revoking capabilities. Other countermeasures consist of letting the monitor own the driver's memory region and having access to the data structures used by the driver and the device in initiating operations and marking them as finished. This allows the monitor to continuously observe which memory capabilities may revoked. As applications may not revoke capabilities themselves, and the monitor has full access to the driver data structures, the monitor controls the revocation process.

5 Own Contributions

There were four members in our group (and still are), but two worked mainly on campus and two more remotely, so eventually we split the two issues between us, with the revocation of driver memory capabilities becoming my and Erik's issue. We subsequently had some scheduling conflicts during this exam period, so he decided to postpone his attempt to a later date. So everything in this report and the implementation (of the second vulnerability) are my contributions (with the exemption of some boilerplate code, and perhaps some work that remained from the early stages of the course when we sat all together), and also why the code is being pushed in one go in our git. This report is a bit shorter than five pages, partially due to this section being more or less nonexistent, but considering we split the task into two parts, I suspect Elias and Linus have written a similar report for the first vulnerability. Most of the implementation details have already been discussed, so discussing my "own contributions" would more or less only be a replication of previous sections.

References

[1] Jonas Haglund and Roberto Guanciale. Formally verified isolation of dma. In *Formal Methods in Computer-Aided Design*, 2022.

[2] Zongwei Zhou, Miao Yu, and Virgil D Gligor. Dancing with giants: Wimpy kernels for on-demand isolated i/o. In *2014 IEEE symposium on security and privacy*, pages 308–323. IEEE, 2014.