VirtIO and s3k capabilities

Fredrik Gölman

January 22, 2024

1 Vulnerabilities

There are two main vulnerabilities in play here with both stemming from the interplay (or lack thereof) between VirtIO and capabilities of the microkernel s3k. The first one is the possibility to write to memory ranges outside of assigned capabilities. This could, for example, allow the writing and execution of program code that was not intended, which was seen in the exercise where we loaded a binary overwriting the content of "app 1" and then executed the other application. This part of the problem is 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 the capability the file system (fs) driver uses for its operations. One could imagine data leakage if the memory of the driver is revoked and then assigned to some other process that could then read data from the fs driver buffers. There are also concerns regarding denial-of-service vulnerabilities if there are operations queued in the VirtIO driver rings (or queues, but need not be processed in order - the terms are used interchangeably going forward) that are yet to be processed by the device. File systems are inherently sensitive as they are an interface to user data (and sometimes system data given sufficient privileges). In their recent survey study of 157 Common Vulnerabilities and Exposures (CVE) cases, Cai et al. [1] found that roughly 75 percent consist of denial-of-service vulnerabilities, and 12 percent consist of data leakage. This is consistent with Sun et al. [2] findings that 61 percent of the 377 analyzed CVEs over the past 20 years consists of denial-of-service vulnerabilities.

2 Design Choices

To mitigate the aforementioned vulnerabilities there are several design choices to be made. One may move the VirtIO queues into the monitor, and that may be preferable for the first issue. For this second issue, it seems sufficient to just have access to the VirtIO queues. To handle the actual revocation attempt of driver memory one could drastically simply kill the driver, but that would leave some reliability issues as there at times would be operations queued up that would

not be performed. A more graceful approach could be to iterate through the VirtIO queue and remove any conflicting operations, but that would incur some reliability concerns. Another graceful approach that ensures reliability is to deny revocation of capabilities involving driver memory until the queued operations have been performed, and only then permit the application attempting to revoke the capability to proceed. This is the approach taken and is performed by letting the monitor hold its queue to "quarantine" capability revocation requests that are to be removed sometime in the future when they are not in conflict with the queue of the driver.

3 Implementation

This section starts with outlining a few flaws in the implementation, some corner-cutting, and some a bit more severe. The sharing of access to VirtIO's data structures is quite ugly shared with the monitor through a dummy call to the *virtio_disk_rw* function, but works. Individual file operations in VirtIO are described by three descriptors which hold information regarding disk sectors and what type of operation it is (i.e. read or write), buffer size, and a status bit for the device to write upon completing an operation. An obvious flaw in the implementation of the monitor is that it considers the addresses of the descriptors but not the size such as the buffer size of data to be read or written. It would also have been appealing to return an error to the revoking application that it could iterate on until the capability could be removed. As is, the application communicates directly via IPC with the monitor which then either revokes the capability or puts it in the "quarantine" for future removal. The quarantine is essentially just a queue that holds information about what process wants to revoke what capability for capabilities that involve driver memory ranges. The monitor checks it periodically as it iterates over its IPC communication. I believe that this core functionality of the monitor works as intended, which was the main focus, but that some of the aforementioned flaws could (and should) be improved. Time constraints, and battling with the IPC functionality of s3k, particularly after suspending and resuming an application have prevented ironing them out, but have been wanting to give it a shot nonetheless. Considering the sizes when dealing with the VirtIO queue and deciding whether to quarantine a capability revocation is not the most complex issue, but the IPC issues have me scratching my head. One can get away with using the *send* functionality for the clients and *recv* for the monitor acting as a server for the revokes that do not involve memory ranges of the VirtIO queue or ones where there are no entries in the VirtIO queue, but for the test case that depends on the VirtIO queue being populated *sendrecv* is needed to block execution to allow for the monitor to do its thing before VirtIO processes the file operation. The problem appears to arise after resuming a suspended process (appears to be some synchronization issue - suspect an ugly workaround could involve a flag in some shared memory between the monitor and process), and sending a response from the server side when using the *sendrecv* function making it spin into an infinite

loop with the error $S3K\_ERR\_NO\_RECEIVER$ on the client end. It would also be desirable to move the communication with the monitor regarding capability revocation into the kernel and return an error for the application to iterate upon as previously mentioned, but before resolving the IPC issue, it made little sense to spend time moving this around and rather focus on implementing the functionality of the monitor instead.

From a security perspective, the idea of the implementation is that the monitor is the owner of a capability and derives new capabilities from that capability that it delegates to other processes, including the driver. Thus, the child cannot revoke the parent capability, only the monitor can, and as such the process has to go through the monitor, which in turn can observe that the queued file operations are performed before allowing the capability to be revoked. From a reliability perspective, the monitor allows for existing file operations to be performed before allowing any capability revocation that involves driver memory. Improved security and reliability are two reasons for using microkernels rather than monolithic kernels, often at some performance cost.

4 Countermeasures

The countermeasures consist first of letting the monitor derive capabilities, and from them derive any other child capabilities for other processes. Any revocation attempt involving driver memory is denied as long as VirtIO is to process operations in the queue, as described in the previous two sections. The countermeasures mainly mitigate denial-of-service vulnerabilities and ensure reliability.

5 Own Contributions

There were initially four members in our group (we 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 in turn 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, but considering we split the task in two I suspect Elias and Linus have written a similar report for the first vulnerability. Most of the implementation details have already been discussed so this is more or less just reiterating over the same things as my "own contributions" does not deviate from the discussion in Section 3 regarding the implementation.

References

[1] Miao Cai, Hao Huang, and Jian Huang. Understanding security vulnerabilities in file systems. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 8–15, 2019.

[2] Jinghan Sun, Shaobo Li, Jun Xu, and Jian Huang. The security war in file systems: An empirical study from a vulnerability-centric perspective. *ACM Transactions on Storage*, 19(4):1–26, 2023.