Implementing a monitor to defend against DMA attacks in a secure kernel

17/01/24

Elias Lundgren

Vulnerability

Security sensitive applications must be formally verified to make them foolproof, which implicitly limits their size. In this project S3K (Simple Secure Separation Kernel), a security focused kernel built for the RISC-V architecture, is used to demonstrate and defend against DMA (Direct Memory Access) attacks.

Smaller kernels are simple by nature as they need to be formally verified. This is a process of using mathematics to prove the correctness of code and its execution. To verify and validate code using processes such as model checking it cannot be too large as it exponentially increases the complexity and thus computation required.

Instead a secure kernel should be small, but to become functionally valuable it does need to integrate with larger untrusted high level systems, like network drivers, file systems and other I/O communication. There have been some honourable efforts by OS manufacturers to try to provide trustworthy services to application developers, but they have all ultimately failed in other regards such as compatibility and updates. The integration has to be carefully considered to not compromise the kernel. Zhou. et al. (2014) describes one way to isolate wimpy (small) kernels from larger systems, by presenting a security architecture for on-demand isolated I/O channels. The paper focuses on USB as an I/O channel in practice, but the solution presented is quite universal. involves two classic computer security concepts: *outsource-and-verify* and *export-and-mediate*. The first involves verifying the correctness of device initialization and configuration, this is a lot less work and code then setting the usb hierarchy up from the kernel, while still eliminating the possibility of an attack being conducted. The second concept involves exporting required drivers and such to external (wimp) applications without OS-privileges, and instead validates the descriptors it produces to maintain isolation.

A common issue when integrating a a I/O device or file system is how a storage device's DMAC (DMA-Controller) generally has full access to device memory, and how a faulty or compromised device driver may break isolation and allow overreads or overwrites straight into any program, or worse, the kernel. A driver with control over the DMAC, thus also has control over the entire system.

It is often common for security focused kernels to either disable DMACs in their entirety and thus sacrifice some CPU performance if the functionality is required. Another solution is to rely on IOMMUs, which are IO management units that are placed between memory and DMAC to verify operations, but this creates other issues as it is slow and often not supported by hardware. There are methods to formally verify DMACs and thus be able to use them and maintain isolation (Haglund & Guanciale, 2023).

In this project we demonstrate a simple DMA attack from the driver. When the driver is tasked to read a binary from the file system we overwrite the address of where it should be placed. The VirtIO, which itself has no understanding of the capabilities defined in s3k, and cannot be halted like other processes governed by s3k, simply executes the attack. We thereafter create a defence against this taking inspiration from the previous research, particularly the export and mediate method is interesting for our solution.

Design choice

In this project we propose a solution involving a software monitor that protects the DMAC from getting illegal instructions from the driver. To implement the filesystem inside of qemu we use VirtIO, a standard for network and disk device drivers. Thus we're not really utilising a DMAC, but for demonstration purposes this works just as well. Virtio runs separately from the user defined programs, has DMA, and is provided data and communicated to by using specific pointers in memory. It continually reads blocks that need to be defined to a specification and placed in a queue at a specific address.

The queue block contains where to read from or write to in the device, as well as where to get or place the data in memory. For DMA attacks we are specifically interested in situations where the driver wants to read something to a device and place it into memory. The monitor is used to boot other processes, like the one that uses the file system, and relies on s3k's IPC features to get information about the operation.

There are two situations to consider where a DMA attack may occur, either when something is added to the queue, or when a capability is revoked from the program - thus invalidating the current queue. In the start of the project the task was to counter both of these issues, but the project was later split into two parts. This report focuses on when something is added to the queue.

Initially we let the monitor exist as a separate application that responded to the driver before it wrote to the queue with a boolean - depending on if the request was legal. But as the driver itself is compromised, we cannot trust an interaction like this, as it might as well skip the whole verification process and write to the queue either way.

The monitor has some performance implications. The biggest change is that we have one ipc call per read / write that adds some additional time, but as all the other code is more or less the same, and just happens to execute inside the monitor instead of the driver, it is barely noticeable.

Implementation Details

Instead of letting the driver add blocks to the Virtio queue by itself, we designed it to send a request to the monitor and await a response. Upon the monitor receiving a request, it pauses the process and then starts validating it. This is necessary to read the processes capabilities, and is also important from a security standpoint as it could otherwise manipulate the request after it has been validated.

The validation is simple, the request contains an address where the data will be written to when handled by VirtIO. This address is compared with all pmp and ram capabilities until one is found that contains the address within its bounds. If no capability is found, the request is illegal and is cancelled within the monitor. If this is the case, the process is never resumed as it tries to break isolation, this is similar to how s3k handles other violations.

If the process however does have a capability that allows it to modify that portion of memory, the monitor is tasked with building a request block to the queue. This block, made out of three descriptors which need to follow a specification. This portion of the code was already created in the initial file system implementation, and we only really had to change the address of the disk struct (which contains the descriptors). The disk is placed in a shared memory we set up, which is readable and writable from the monitor, and only readable from the app. This allows the driver to still keep functions like virtio_disk_status() the same and not have to switch them out to communicate via ipc. By letting the monitor take full control of the queue, the process which sends the request does not need access to the UART addresses which Virtio uses to communicate with, this makes it impossible to bypass the monitor.

For testing if the defence actually works, we set up a separate app (app1) which only contains a couple of print statements. Additionally we also create another app (app2) with another print statement, but instead of running it, we compile it and place it within the qemu filesystem. Finally, using the driver, we use the driver to read the app2 binary, and change out the address we want it to be placed at to the address of app1. Without the monitor, when we run app1 we should see the print statement from app2. But thanks to the monitor this is stopped and if we run app1 after the driver is done the same prints as before remain.

Did we counter the attack?

Yes, for an attack involving adding a read or write to the VirtIO queue, there is no longer a risk of breaking isolation. As the process that's running the driver is suspended upon request, and the capabilities are checked after the suspension, there is no way for the process to cheat the monitor by for example changing address after sending the request. The monitor then goes through the capabilities and compares the addresses and privileges. As the disk structure lies within the monitor, and the driver neither has memory access to that or the VirtIO addresses in UART memory, there is no way for it to bypass either. If the monitor accepts the operation, it will add the operation to the queue and wait until it has been processed before resuming the app.

My Contribution

Initially me and Linus spent a lot of time pair programming to be able to continually discuss and better understand how s3k works. But I did do some stuff on my own during this time too, initially I set up the dockerfile (7d80282) and makefile functions + attack in virtio_disk_rw (f77931c).

Additionally, when we started working on the monitor, I did the final part of the communication protocol after we ran into issues where the ipc s3k_ipc_sendrecv function didn’t work when suspending a process from the monitor (9d166d2). I then came up with my own solution that involved spinning on a byte in shared memory to avoid race conditions. This solution was also later refined in my own version of moving the disk.

After the presentation there was really only one task left - to move over the disk struct to the monitor to protect it properly from attacks. As this task didn’t really fit separating into chunks, and we both wanted more individual commits, me and Linus did our own solutions to this (mine in branch disk_move_test). For me, this task involved changing the attack, communication and setup of the monitor to comply with the Virtio DMAC requirements.

The attack could no longer be done the same way when the disk was moved, as we initially only sent over a pointer to the *buf* struct and a *write* boolean. To mediate this without increasing complexity an extra argument was added to the communication containing the address to where VirtIO should write to / read from.

The IPC also had to change somewhat, as we no longer had to send back a result to the app. First I tried some solutions where the only synchronisation between the app and monitor was a bit in the shared memory the app only could read, but after a while I gave up as some race condition always would manage to show up. The final design was instead a *toggle* bit in app memory (which the monitor has access to) which was switched to true upon sending a request to the monitor, and flipped back to false when the disk operation was completed. By letting the app get stuck while the toggle bit was on until the operation is complete, it avoids any

race condition. For changes in the monitor it was mostly giving it access to UART memory, and fixing the capability checking function to also take into account the RAM memory capability.

References

Z. Zhou, M. Yu and V. D. Gligor, "Dancing with Giants: Wimpy Kernels for On-Demand Isolated I/O," 2014 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 2014, pp. 308-323, doi: 10.1109/SP.2014.27 (Accessed 17 Jan 2024)

J. Haglund & R. Guanciale "Formally Verified Isolation of DMA" Formal methods in computer aided design, Vienna, 2022. https://kth.diva-portal.org/smash/get/diva2:1758722/FULLTEXT01.pdf (Accessed 17 Jan 2024)