Linus Below Blomkvist
19-01-2024 libl@kth.se

VirtIO Queue Monitor

Explain the countered vulnerabilities:

Our project focused on ensuring isolation between kernel components by implementing a monitor to regulate access requests. This initiative was crucial when addressing vulnerabilities associated with Direct Memory Access (DMA) attacks. In such attacks, a malicious host can take advantage of buffer overruns in VirtIO devices, which historically trust their drivers, to access and manipulate the guest memory.

Our project drew inspiration from two previous research to understand these vulnerabilities better. The paper “Dancing with giants: wimpy kernels for on-demand isolated I/O” illustrates the fundamental challenge in secure computing: the smaller, security-sensitive 'wimpy' kernels struggle to safely interact with larger, less trustworthy 'giant' systems. This is a similar scenario to our challenge in ensuring that smaller, secure components within s3k (our security-focused kernel) can safely interact with larger systems like VirtIO drivers without compromising security.

The second, “Formally Isolating Direct Memory Access” underlines the importance of formally verifying DMACs to prevent unauthorized memory access. This concept is crucial in understanding the risks associated with VirtIO drivers, which traditionally operate on a trust-based model with the device (host), potentially leading to security breaches such as buffer overruns and unauthorized memory access.

In our project, we brought these concepts into an attack scenario involving three applications: app0, app1, and app2. It is set up by having app0 overwrite app1 with app2's contents. which displays a breach of isolation, related to the risks highlighted in the referenced papers. This was showcased through the code injection string “APP2: heheh we take over now," which presents the potential for misuse when the trust model between devices and drivers is exploited.

```
f_read(&Fil, buffer, 1023, &bw);

fr = f_open(&Fil, "app2.bin", FA_READ);
```

This scenario was an illustrative example of the risks posed by uncontrolled capabilities within the kernel. And the cause of this problem is the inherence trust in VirtIO.

VirtIO is a virtualization standard for network and disk drivers where just the guest's driver cooperates with the hypervisor. This enables guests to get high performance from network and disk operations.

But there is one problem: VirtIO devices and their corresponding drivers operate under the assumption that the device is trusted by the driver. Which is a problem derived from inherent trust between the device and driver. The security concerns from this revolve around the shifting trust model between VirtIO devices and drivers. Since the VirtIO drivers assumed the

device (i.e., the host) could be trusted there is a lack of necessary checks for the data that is shared with the devic. As a result, vulnerabilities like buffer overruns could be exploited by a malicious host to access guest memory.

To solve this there have been proposals to harden the VirtIO framework against malicious devices. One approach is disabling certain VirtIO modes, like indirect descriptors and packed mode. The problem with this approach however is that this would lead to significant performance loss. Our choice was to instead implement a monitor.

Explain you design choice

Our design approach involved setting up a monitor to block DMA attacks. We set up a monitor to inspect and validate each access request against the capabilities to the address. The initialisation of the entire process, including app0 and app1, was conducted from the monitor. We also derived and allocated Physical Memory Protection (PMP) capabilities from the monitor to the apps. The int main(void) function in the monitor was then responsible for starting the apps, establishing the socket, and setting up shared memory and awaiting requests for processing.

The design strategy was quite basic. Every access request had to pass through this monitor for inspection, where the monitor would control and verify the capabilities of the request.

In our design choice there were two parts we needed to take into account: first performing DMA attack into a invalid address directly or bypassing the monitor by adding an event into the queue. For that we also had to find a way to separate and isolate the construction of the queue. In the previous design we added the event into the queue after the request had passed the monitor. That was a design from earlier when we were given the code in the virtio_disk.c file.

The choice of design for the second part is where my lab partner and I took different directions. After the monitor had been implemented the queue was still being constructed in app0 in the virtio_disk.c file. Because of that the construction of the queue had to be moved to the monitor. This was done by setting up a queue build function in the main.c file in the monitor. Then we once again use the socket to this time pass a buffer pointer, int write and a disk pointer. Then we can build the queue from the monitor.

Discuss you implementation details that you think are important

Key to our project was the implementation of an IPC socket. This socket was instrumental in bridging communication between the virtio_disk.c file in app0 and the main.c file in the monitor. Each request's capabilities were transmitted to the monitor for verification and validation, thus reinforcing kernel component isolation. The direct construction of the queue within the monitor, via the queue_build function, further solidified our defense mechanism against potential attacks.

The second important implementation was the building of the queue: To enhance security, we transferred the queue construction process to the monitor. We achieved this by implementing a queue_build function in the main.c file of the monitor. By then using the socket, we passed a buffer pointer, an integer write variable, and a disk pointer to the

monitor. This allowed us to construct the queue directly within the monitor, further reinforcing the security and isolation of the queue.

Explain why software counter the selected attacks

Our solution effectively counters the identified attacks for two main reasons. Firstly, by designing the system to route all requests through the monitor, we ensured a consistent check on the correspondence between the capabilities of the request and the address. This guaranteed isolation.

Secondly, the relocation of the queue construction from app0 to the monitor closed a critical security loophole. Currently an attacker can use the virtIO Queue to access portions of memory without having the required capability. Previously, an attacker could bypass the monitor by accessing the queue construction in app0. Now, with the queue building process centralized in the monitor, only the monitor has the authority to construct and pass the queue back to app0, thereby securing the system against such attacks.

However while the infrastructure should work since it is only moving a section of a code, the program hasn't managed to run so far. Still in progress. Therefore I can only report a successful countermeasure for the first part with the monitor and not with the queue build function.

Individual contribution

My contribution involved two parts. First, I relocated the setup process from app0 to the monitor. Secondly, I proposed a method for transferring the queue from app0 to the monitor as the final part of the project.

The relocation was necessary because, in the initial phase of our code, app0 set up and started the process. This required deriving and moving capabilities from app0 to app1 to start app1. To address this, I moved the set_up_app0, set_up_app1, start_app0, and start_app1 functions to the main.c file in the monitor. I then called these functions in the int main(void) function of the monitor. This change means that we now derive and move PMP capability from the monitor to both app0 and app1. This step is crucial for maintaining isolation, a core goal of our project. Without this change, deriving capabilities from app0 to app1 would have compromised isolation from the start.

For my second task, I tackled the issue of the queue. In our earlier solution on the defense-alt-1 branch, the system was set up such that when there was a request to an address, the monitor would check if the allocated capability on the address matched the capability from the request source. If not, the monitor would deny the request. However, this solution had a flaw: the queue built in the virtio_disk.c file remained within the app0 folder. Consequently, if someone accessed the queue directly, it could bypass the monitor, breaching isolation. My solution is to move the code responsible for building the queue into the monitor. Specifically, in virtio_disk.c, lines 330-355 in the virtio_disk_rw function handle adding events to the queue. This segment, along with the disk structure defined on line 35, should be relocated to the main.c file in the monitor. We can then create a queue_build function in the monitor that takes a buffer pointer, the int write variable, and a disk pointer as inputs.

Linus Below Blomkvist
19-01-2024 libl@kth.se

In this new setup, the virtio_disk_rw function in the virtio_disk.c file will be modified to send three messages through the socket: the buffer pointer, the int write variable, and the disk pointer to the queue_build function in the monitor. This approach ensures that while the disk and buffer remain on app0, their construction occurs in the monitor using pointers. This shift of essential code from app0 to the monitor also requires granting the monitor read and write access to app0 to write the queue there.

However, this approach does violate isolation to some extent. Although the monitor is permitted to do so, granting it additional read and write access may not be the most optimal since it requires further breach of isolation. My lab partner Elias made an alternative solution, which was to move the whole disk to the monitor and place it in shared memory for app0 to read only. This solution might be more effective as it relocates the queue building process to the monitor while maintaining isolation for the disk.