Analysis of Issues in the Dummy Device Driver

In this Linux module, I implemented a driver which acts as a bridge between the dummy hardware and the user applications. The driver reads half bytes of data from the dummy device through the GPIO pins each time, then assembles the data into a whole byte and stores it in a circular buffer. When the data in the circular buffer meets a particular condition, it uses a task to migrate it to an endless buffer and notifies the user applications to read the data if there is any. However, there are still some issues with the design of this module, for instance, the usage of the spin lock to protect the buffers and the implementation of the endless buffer.

To guarantee data integrity, I use two spin locks to protect the circular buffer and the endless buffer, respectively. The access to the two buffers is a traditional problem between the producers and the consumers. While each side should read or write the buffer exclusively, using the spin lock to protect the target buffer is the simplest way. However, this might influence the efficiency of the entire process, especially while the writing is in the interrupt handler and the tasklet, which have critical efficiency requirements on the code. For example, suppose the data in the endless buffer has been accumulated for a long time, and the user applications (e.g., cat) use a large buffer to read the data. In that case, the user applications will hold the lock for a long time. This causes the busy waiting of the tasklet. However, some measures can be applied to reduce the burden. One solution is to use the atomic numbers as the start and end edge of the circular buffer and update them after the data is read or written. In terms of the endless buffer, the writing process can be divided into multiple copies of small chunks of memory to reduce the waiting time of the tasklet.

In addition, using an endless buffer to store all the data read from the dummy device can be a risk to the kernel. Considering the device file can only be accessed exclusively, the only way to consume the data is read by the user application which opens the device file successfully. This can be exploited by malicious software. If the malicious software does not read data from the device after it opens the device file, the endless buffer will grow with the generation of the data from the dummy device. Finally, it will exhaust all the memory that the Linux kernel can use, which is 1GB in the 32-bit Linux system. Because all other user applications must keep waiting until the malicious application closes the device file, this design can be exploited very easily. Certainly, several strategies can be used to avoid such a drawback. For instance, limiting the size of the endless buffer, and discarding the data

after it reaches the limitation, allowing multiple processes to access the device concurrently. However, it needs the tradeoffs between the security and the data integrity.