Final Report - DMA Monitor - Group 4

Marcus Dypbukt Källman

January 11, 2023

1 Countered Vulnerabilities

Direct Memory Access (DMA) is a process that allows a device, such as a hard drive or network card, to directly access system memory to read and write data. This allows devices to transfer data to and from system memory without going through the CPU, which can improve system performance. However, DMA also poses security risks because it allows devices to bypass security measures that are in place to protect the system and allow devices to access memory that they would normally not have access to.

Our project has focused on the vulnerabilities posed by a compromised driver of a network card. Specifically, we have worked with the virtio_net driver.

When instructing the network card to do a memory access, the virtio_net driver first creates a DMA descriptor, which is a data structure that contains information about the memory region that the network card should access. The DMA descriptor among other things includes the start address of the memory region and the length of the region in bytes. The driver then passes the DMA descriptor to the network card which can then use the information in the DMA descriptor to access the specified memory region and read or write the data at it. The network card does not have any ability to check if a received address is valid or should be prohibited and will thus blindly read and write to the address. If the virtio_net driver is compromised, it can therefore be used to instruct the network card to read and write to any address.

By compromising the driver and sending well-crafted packets to the network card we can therefore overwrite parts of the memory with the packet data. This could allow us to potentially overwrite security keys, password or crash the system among other things.

Similarly, a compromised driver can be used to leak data. The compromised driver can instruct the Network Interface Controller (NIC) to read from any address and the network card will then read from that address and send it out on the network. An adversary who has managed to compromise the driver and is able to capture the traffic coming from the NIC can therefore leak data from memory. Since the

NIC collects the packet headers from the address as well, simply giving it a random address in memory will result in "invalid" packets being sent out. However, an adversary could first use the overwrite vulnerability discussed earlier, to overwrite parts of memory with valid headers which could result in leaked packets that can be sent directly to an adversary.

2 Design Choices

In order to address the vulnerabilities outlined in the previous section, we propose implementing a monitor system. The primary role of this monitor will be to evaluate and verify the validity of addresses that are provided to the NIC by the driver and ensure that they are safe to read from or write to.

To accomplish this, the monitor will function as follows: when the driver wants to instruct the NIC to read or write to a particular address, instead of sending it directly to the NIC it will instead forward the DMA descriptor to our monitor. The monitor will then examine the information contained within the descriptor, including the address and size of the requested operation, and determine whether or not the request should be allowed. If the memory access is deemed valid, the monitor will then add the DMA descriptor to the NICs queue, allowing the operation to proceed as intended. If, however, the request is found to be invalid, the monitor can either drop the request or trigger a panic to alert the system of the possibility of a compromised driver.

This way the monitor can prevent the driver from giving the NIC invalid addresses but since the driver is assumed to be compromised, we also need to prevent the driver from simply bypassing the monitor and queuing the DMA descriptor directly as before. This can be done by revoking the driver's access to the PCI bus, making it unable to communicate directly with the NIC. This does however prevent all communication between the driver and NIC and will require some functions which currently reside in the driver to be moved to the monitor.

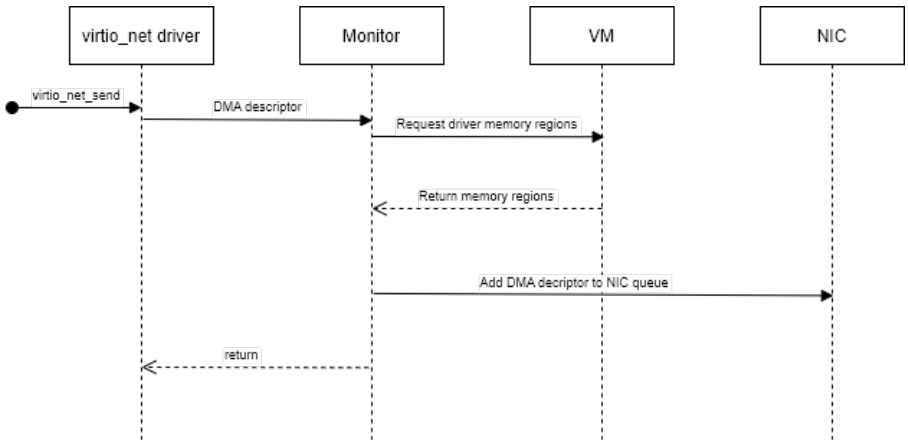

Figure 1: Operation sequence for a successful request sent through the monitor

3 Implementation Details

3.1 Monitor

The monitor is designed to run as a server process within Minix, and it will automatically start up when the operating system boots. Once the monitor server has initialized, it will continuously listen for IPC calls from other processes. When the monitor receives an IPC call, it will examine the message type and depending on the type of message received, the monitor will process the request and generate a response, which it will then send back to the calling process. This process of listening for and responding to IPC calls will continue indefinitely as long as the monitor is running.

The general function of the monitor is to receive DMA descriptors from the driver, evaluate whether the DMA descriptors contain a valid memory access and if so, pass it along to the NIC. To communicate with the monitor, we have created a wrapper library which handles creating the IPC calls and sending them to the monitor. The library has functions for each of the functions that has been moved from the driver to the monitor. The IPC calls use an added message type for sending the data required by the monitor, currently, the payload does not exceed the 56-byte limit so we don't have to use grants, but the message type could easily be changed to instead send grants. The same message type is used for all monitor functions, but unique types could be added for each.

3.2 Address Validation

When a process wants to send out a packet, the virtio_net driver will first copy the packet data into its own memory and then instruct the NIC to read from there. Similarly, when a packet is received the driver will give the NIC an address in its own memory to write the packet to before copying it to the destination processes memory. This means that the addresses that should be allowed to be passed along to the NIC should all reside inside the driver's memory.

The monitor can then determine whether a received DMA descriptor should be allowed by evaluating the address and size information contained within the DMA descriptor and checking whether the memory region defined by these values is contained within the memory allocated to the driver. This allows the monitor to validate the legitimacy of the request and ensure that the driver is not attempting to access memory that it is not authorized to use.

To do this the monitor need will need to know what memory regions are allocated to the driver, which is known by the VM server. The monitor, therefore, queries the VM server and receives back all physical memory regions allocated to the driver. The VM already had a system call VM_INFO, and using the option VMIW_REGION will return the virtual regions of a given process. We, therefore, added another option to VM_INFO, VMIW_PHYS_REGION which instead returns all the physical regions of a given process.

Once the monitor knows the physical regions allocated to the driver, it can check if the address and the size in the DMA request are fully contained inside one of these regions and if it is the request is valid.

3.3 Virtio_Net driver

The virtio_net driver will need to be altered in a few ways. Mainly, the driver will need to be altered to instead of sending the DMA descriptors to the NIC, it will send them to the monitor. The driver uses the monitor wrapper library to create and communicate the messages to the monitor. For reasons described in previous sections, we will need to disable the PCI access for the driver. This means that the driver needs to be rewritten so that function that previously communicated with the NIC directly now calls monitor functions instead.

4 Vulnerability Counters

Our solution aims to address the vulnerabilities discussed in the previous section by introducing a monitor to act as an intermediary between the driver and the NIC. This monitor will evaluate all memory addresses being sent between the two and ensure that they are valid. We have added functionality to the VM server to fetch all physical address regions allocated to the driver, and the monitor will check if the address being sent belongs to these regions. By doing so, we can ensure that the address belongs to the driver and prevent it from accessing or altering data beyond its own memory.

Previously, the driver had the ability to instruct the NIC to read from and write to any address, which allowed the two vulnerabilities discussed earlier. The monitor prevents bad addresses from being sent through the monitor, but we have also removed the driver's PCI access. This way, the driver has no other option than to communicate with the NIC through the monitor and can no longer bypass the monitor. So, while a compromised driver could still potentially disrupt the system by not sending or receiving data, it would no longer have the ability to access or alter data beyond its own memory - which was the primary goal of our project.

5 Contributions

5.1 Data leak vulnerability

Initially, I worked on determining and demonstrating how a compromised driver can be used to make the NIC leak data from memory addresses it shouldn't. To demonstrate this vulnerability, I modified the virtio_net driver to send an incorrect address to the NIC. I then used Wireshark to capture the packets sent by the NIC, which showed that they contained data from the unauthorized memory address. I then compared this leaked data to a memory dump of Minix to confirm that the data was indeed coming from the unauthorized memory address. The leak is documented in the readme on GitHub.

5.2 Monitor Server

I then worked on setting up the monitor server and the communication with the driver. I began by creating a barebones server which I based on the existing Data Store server. I then added the wrapper library for the driver to communicate with the monitor and set up the required permissions for both driver and monitor. I then extended the monitor with the functions I thought would need to be in the monitor and created the IPC message type.

5.3 Address validation

I then worked on how the monitor should validate addresses. Originally, the plan was to only verify that the addresses were not contained in kernel memory, to avoid leaking or overwriting kernel memory. This would however have still left the vulnerabilities open for all non-kernel memory. However, since the driver will, for both sending and receiving, copy the packets into its own memory first, I instead decided on validating that the received memory addresses belonged to driver allocated memory. I then had to find how the monitor could request the memory regions from the VM service. When the driver makes IPC calls to the monitor, the monitor will have the driver processes endpoint which the VM will need to identify the memory regions.

The VM server didn't have any existing call for requesting physical memory regions but did have one, VM_INFO which had an option for requesting virtual memory regions VMIW_REGION and I felt it suitable to add it as an option there as well. Examining the function which handled the VMIW_REGION option showed that it also had access to the physical addresses. So, I added the option VMIW_PHYS_REGION to the VM_INFO call and rewrote the existing function to handle both options. Which avoided adding unnecessary code and breaking existing code. Then I added a function to the monitor that requests the physical memory regions for the calling processes endpoint and checks that the memory the driver wants the NIC to access is contained in these regions. One concern though, was that the VM_INFO call is one of the calls included in the VM BASIC configuration setting. Meaning the VMIW_PHYS_REGION option is available to all services with that setting. This should be an issue but could be remedied by creating a new VM call.