VirtualMouse Design and Specifications

V1.0

Prepared by Edwin Heerschap

May 7, 2020

Contents

Preface

VirtualMouse is a linux kernel driver aimed at providing programmatic mouse functionality. It is encompassed by the VirtualHideout project by SneakyHideout; which aims to create virtual peripherals. This document describes the software design and specifications for VirtualMouse v1.0. This document is not developer documentation for parts of the project.

Software Design Overview

Abstraction

A seperation of concerns approach is taken for the design of VirtualMouse. This materializes in a layering pattern for the system. Four primary layers are present in the highest abstraction of the system; kernel driver, interface library, userspace implementation, other library. Interactions among them are represented in figure 1.

Kernel Driver

VirtualMouse kernel driver is the core of the virtual mouse system. It is a character device driver for the linux kernel. Systems interpreting mouse protocols such as X11 or other kernel drivers will read input from here. The interface library requests actions from the kernel driver resulting in mouse events.

Interface Library

The interface library is a native shared object file with methods to interact with the kernel driver. Maintaining an interface library removes userspace dependence on the kernel driver implementation. Secondly, it allows language independent access to kernel driver functionality.

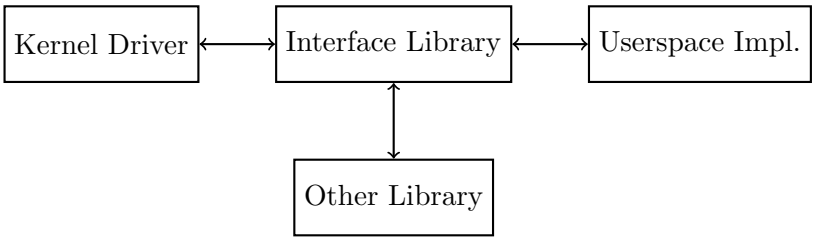
Userspace Implementation

Userspace implementation is userspace software employing the interface library to create mouse events.

Other Library

Positional information of virtual mice is not stored in the kernel driver. It is the responsibility of software such as X11 to maintain positional information. Other unknown information as of current may also be unavailable to the kernel driver. Other libraries will get this information.

Figure 1: Interaction between abstractions



Kernel Driver

The design of the kernel driver will attempt to comply with the S.O.L.I.D design principles[1] as closely as realistically possible (as *C* is not directly object oriented).

In this description the semantics for a structure having a method is a function exists which accepts the structure as one of its parameters. For example, *vmDevice*'s ioctl() method accepts a (*struct vmDevice**) as one of its parameters.

Specifications

The kernel driver is required to support multiple virtual mice. They may also be running concurrently. Each mice may have their own:

- Protocol
- Name
- Access control
- Input/Output Control (IOCTL) access

Furthermore it is required that mice can be added and removed during runtime. They may also be specificed as input parameters. To be more verbose, the kernel driver must be capable of:

- Mice
  - Select the protocol
  - Select the name
  - Select type of access control

[1] https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

  - Perform mice actions

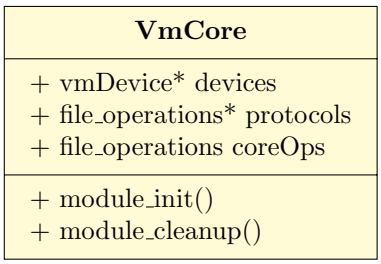
- Add new mice during runtime
- Remove mice during runtime
- Add mice as startup parameters

Protocol and syntax on how to achieve each of these will be specified further into the initial development.

VM Core

To better describe later sections the VM Core is defined. VM Core is the collection which contains the methods and variables required for bootstrapping vmDevices and passing on file_operations. This is the *module_init*, *module_cleanup* and the base *file_operations*. At some stage in the Virtual-Hideout project it may be appropriate to move the core to its own driver. The VM Core appears as seen in figure 2

Figure 2: VM Core (simplified)

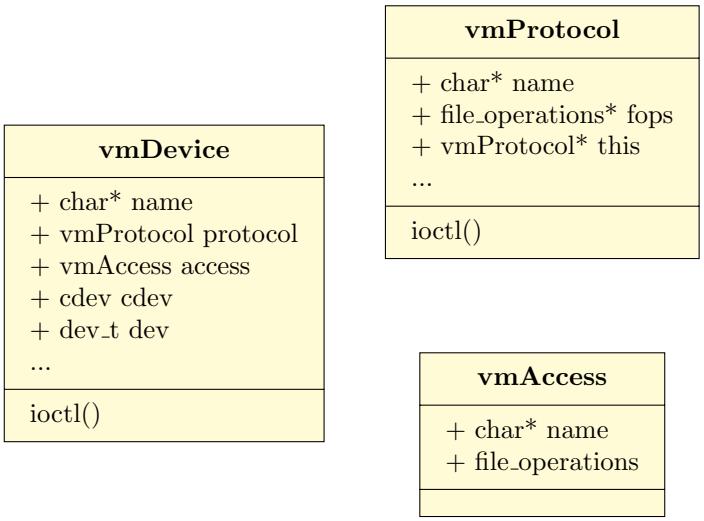


vmDevice structure

Figure 3 is a UML representation of the general software structure of *vmDevice*. *vmDevice* is an individual virtual mouse. *vmDevice*'s structure is used to store its state and point to behavior. The interaction with systems which handle mouse protocols is defined in the *vmProtocol* attribute. The ioctl() method mechanics is described in the IOCTL Chain section.

vmProtocol represents a mouse protocol; the bus protocol for example. System interactions following protocol are defined in *fops*. Non-ioctl device interactions are routed to the *fops* methods unless blocked by the fops chain; by a lock for example. The ioctl() mechanics are described in IOCTL Chain section. The fops ioctl function should only be used for protocol specific

Figure 3: vmDevice structure & dependencies



IOCTL commands. As of writing this, it is uncertain if such protocols exist, but they may do.

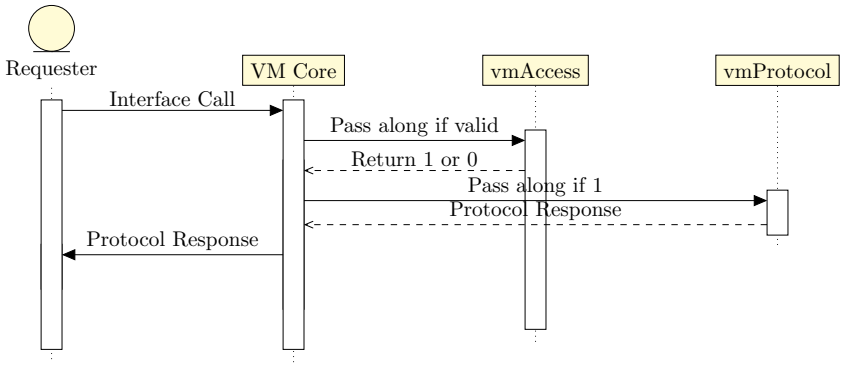
vmAccess represents the access control system used by the device. It is used to allow/deny access to protocol file_operations or IOCTL actions.

file_operations chain

A chain of responsibility approach[2] is taken to handle requests. The head of the chain is the VM Core *file_operations*; does core checks and bootstrapping. Second is the *vmAccess file_operations* which is set by the user and determines if the request should be passed along to the protocol. Lastly is the *vmProtocol file_operations* which is executes the protocol/functionality. This design allows the VM Core control on which actions are allowed to occur. It allows users to define their own access rules. See figure 5 for a sequence diagram.
Regarding figure 5, the requests makes a call to the VM call via the interface lib (arrow named interface call). The VM Core can then reject the call if required. Typically, the request will be passed along to the vmAccess. vmAccess can then return 0 for denial and 1 for allowed. If it is 1, the request is passed along to get the protocol response which is then returned to the VM Core and then to the requester.

[2]https://refactoring.guru/design-patterns/chain-of-responsibility

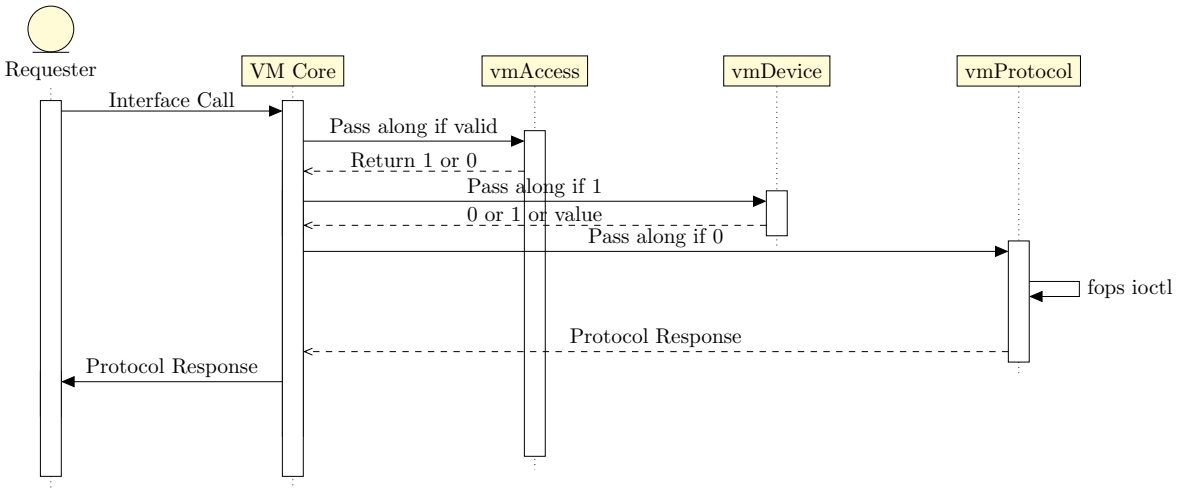
Figure 4: file_operations chain



IOCTL Chain

IOCTL chain is very similar to the file_operations chain. This chain handles IOCTL commands. IOCTL commands are passed along a chain for implementation simplicity. The difference between the two chains are 1) the chain passes the vmDevice ioctl() between vmAccess and vmProtocol 2) the vmProtocol uses the struct defined ioctl function before the fops ioctl[3].

Figure 5: ioctl chain



[3]See section vmDevice Structure

vmAccess Creation Managment

vmAccess structures are intended to be modular (not dependent on *vmProtocol* or *vmDevice*). Managing the creation for the different *vmAccess* definitions requires a generic implementation that limits flexibility restrictions as much as possible.