COMP3301 2023 Assignment 2 - PCI device driver

- Due: Week 9
- $Revision: 493 $

1 OpenBSD `vkey(4)` Cryptographic Token Driver

The task in this assignment is to develop an OpenBSD kernel driver to support a virtual cryptographic token device connected over the PCI bus.

The purpose of this assignment is to demonstrate your ability to read and comprehend technical specifications and apply low level C programming skills to an operating system kernel environment.

You will be provided with a specification for the token device, the emulated device itself on the COMP3301 virtual machines, and userland code to test your driver functionality.

This document should be read in conjunction with the *COMP3301 A2 Device Interface Specification*, which contains the in-depth technical details of the hardware interface your driver must use.

2 Background

In many scenarios involving deployment of Virtual Machines (VMs), it is desirable for the VMs to be able to prove their identity to each other and to other systems.

For example, one VM may be running a database server, while another VM is running a client which connects to it. Having the client automatically be able to prove its identity to the database server VM based only on their VM configuration in the hypervisor enables much simpler deployment and management.

One way this can be achieved is by the hypervisor providing cryptographic proof of identity for VMs through an emulated cryptographic token device.

In this assignment, we will be providing a heavily simplified example of such a virtual token device, and you will be implementing a kernel driver for using it in an OpenBSD guest.

2.1 Cryptographic signatures

Proof of identity is most commonly secured using a cryptographic signature. A "signature" can be made from any cryptographic construct that allows the signer to generate a proof that is difficult to replicate, but easy to verify. The classic example of such a construct is a "public key" algorithm such as RSA or (EC)DSA.

Public-key cryptosystems are built around "key pairs", consisting of a public key and a private key. The private key is a large randomly generated number kept secret by the signer, while the public key is a related number which is openly shared with anybody.

The signer can use the private key to sign some data, producing a signature value. Anybody in possession of the public key can easily verify the signature's validity against the data, but cannot use the public key to generate a fake signature in any reasonable time frame.

Signatures of this type can be used for many applications, but one of the most important is in authentication protocols. For example, if our system is aware that a particular user has possession of a cryptographic key, we can challenge them to sign a random string of data we provide to prove that they are who they claim they are.

There are many fine details involved in using these cryptosystems for engineering, which greatly impact their security and performance. Many of these details are deeply technical and require understanding of the mathematics behind the cryptosystem in question in addition to the CPU and OS environment in use, so cryptosystem engineering is treated as a specialty field in its own right.

As a result, when reaching for a cryptographic solution to a problem, as engineers we should be even more careful than usual to re-use off-the-shelf solutions built by specialists wherever possible. Designing and building new cryptosystems without deep specialist knowledge is perilous and ill-advised.

2.2 `ssh-agent(1)`

One example of a software system that depends heavily on cryptography is the Secure Shell protocol (SSH). It uses cryptography to verify the identities of both SSH servers and the users connecting to them, and also to protect the SSH traffic in transit from interception or modification.

Part of the SSH design includes a "key agent" called `ssh-agent`. This is a specialised process which can be run solely as a container for the private keys used as part of signature authentication.

The idea is that the `ssh-agent` has the private key data, and will answer requests from other parts of the SSH system (e.g. the command line `ssh` client) to compute signatures with that private key. It will not, however, divulge the private key to any other process.

This reduces the "attack surface" of the code which has direct access to private key data, lowering the risk that it can be compromised and stolen by an attacker. The `ssh-agent` contains far less code, doing less complicated tasks, than the full SSH client would, or any of the other clients which use the agent.

Even though other processes can still make the SSH agent sign any data they like on their behalf, they can't easily make a copy of the private key. This means that e.g. if the owner of the agent realises it is being attacked and shuts it down, the attacker's access to the private key is cut off. It also enables the `ssh-agent` to enforce policy on its clients by inspecting the data to be signed (e.g. by only signing authentication challenges from particular remote hosts).

Clients of the `ssh-agent` communicate with the agent over UNIX domain sockets (UDS). They use the same `socket(1)` and `connect(1)` system calls that a TCP socket client would use, but give a different socket family and type of address to connect to. Addresses in UDS do not have a port number, and rather than an IP address or hostname, they contain a file system path (e.g. `/var/run/ssh-agent.sock`). Once open, they function almost identically to a TCP socket, supporting all the normal socket-related system calls.

Over the UDS, clients send command messages, and the agent replies with reply messages back. Clients can only send one message at a time on a given UDS connection, and the server must reply before they send another. These messages have a fixed format with an identifying message number, outlined in the Device Interface Specification.

Since this interface is simple and straight-forward, and allows clients to request a signature over some data using a private key held elsewhere, we can re-use the design for our proof of identity application.

The "agent" in our design will run in the hypervisor, with the private key data outside the guest memory and inaccessible to it. The "client" will run inside the guest, and can request the hypervisor sign authentication challenges on its behalf so it can prove its identity to others.

2.3 The PCI bus

The most commonly used and reliable means by which hypervisors expose virtual devices to guests is via the PCI bus.

PCI (the Peripheral Component Interconnect) originated on the Intel x86 PC platform, but has also become used on other CPU architectures. It specifies physical connectors and electrical signalling of add-in cards and components, as well as the logical data link traffic layer on top of it, and most aspects of the host software interface used by operating systems.

VM hypervisors provide an emulated PCI bus because it is the most common form of extensible hardware interface on the real computers the hypervisor is emulating.

PCI focuses primarily on peripherals which can be operated via memory-mapped registers, and most modern PCI devices also have DMA capabilities. It allows for device auto-discovery by the OS and system firmware, as well as configuration of the location and size of memory-mapped register regions, and provides some basic control to e.g. enable and disable DMA for each device.

The interface between the guest and hypervisor for our proof-of-identity design will come in the form of an emulated PCI device.

In this course, the tutorial content and practicals 4 and 6 provide a lot of information and practical experience with implementing drivers for PCI devices on OpenBSD. You will be using this experience and content for this assignment.

2.4 Descriptor rings

The device driver interface to memory-mapped devices has undergone several stages of evolution over the life of the PCI bus, as demands for performance and capacity for memory bandwidth have both increased.

Many early devices focussed entirely on memory-mapped I/O, where drivers would read and write all traffic via the device BAR. This was also prudent in an era where DMA support was sometimes patchy and unreliable across different hardware.

In the modern era, DMA use is ubiquitious and devices focus on ways to improve performance by making the best out of this capability.

One of the most common ways modern devices make use of DMA is via descriptor rings. These are regions of memory describing commands sent to the device, or responses from it, which are operated as a circular buffer. Host device drivers write new commands into the circular buffer, and the device reads them and carries them out. This enables the host to produce a lot of commands in a short period, and have the device carry them out in parallel without having to wait for previous commands to finish.

Typically descriptor rings come in pairs, with one ring containing commands and the other containing "completions", so that the device driver can be made aware when commands have been executed. The completion ring references the command ring in some way (e.g. by specifying what index in the ring was completed, or copying a value from the command across so that the driver can use it to correlate them).

The PCI device in this assignment uses an interface based around descriptor rings.

3 Specifications

You will implement a driver called `vkey(4)` that will attach to the PCI device specified in the Device Specification document, and provide access to that token device via a character device special file under `/dev`.

You may assume that userland tools which use your driver will directly open the device and use the `ioctl` interface specified below. You do not need to implement any additional userland components to go with it.

3.1 Device functionality

All the basic features of the device must be implemented by your driver. These include:

- Allocating and configuring descriptor rings
- Sending `ssh-agent` commands, including long commands spread across multiple buffers
- Receiving responses to those commands, including long responses
- Detecting hardware errors in the `FLAGS` register

You may, if you wish, also implement resetting the device following a fatal error. However, there will be no additional marks awarded for doing so.

Your device driver must be able to handle commands and responses with data payloads up to 16384 bytes in total length.

You may choose the strategy your driver uses for checking the `FLAGS` register yourself. The device specification has some suggestions, but you should consider the trade-offs of each in terms of performance and timeliness of detection.

Upon detecting an error in the `FLAGS` register, your driver should print a brief description of the error to the system console and `dmesg` (e.g. using the kernel `printf(9)` function).

If a command is in progress at the time an error occurs, then your driver must also return an error to userland as described below (e.g. a `VKEYIOC_CMD` in progress must not "hang", it must immediately return `EIO` to userland).

3.1.1 Concurrency

For full marks (see the criteria sheet for further details) your device must support concurrent use of the device special file interface by multiple userland processes and/or threads at once. This includes both multiple single-threaded processes, and multiple threads within the same process, all of which may be using the same device major/minor node at once.

As well as concurrent use of the device special file interface, your driver must also be able to have multiple commands in-flight being processed by the device at once, and it must support these completing in a different order to the order in which they were submitted.

You may place an upper bound on the level of concurrency possible, but it must be at least 16 concurrent operations.

3.2 Device special file interface

The `vkey(4)` device special file should use major number 101 on the amd64 architecture for a character device special file.

The device minor number maps 1:1 with the `vkey(4)` unit number.

The naming scheme for the special files in `/dev` must be of the form `/dev/vkeyX`, where `X` is the unit number as a numeric string.

Examples:

Device	Major	Minor	(Unit)
`/dev/vkey0`	101	0	0
`/dev/vkey1`	101	1	1
`/dev/vkey2`	101	2	2

Your driver may, if you wish, return additional `errno(2)` values as well as those specified below, in situations that are not covered by the `errno` values and descriptions listed.

3.2.1 open() entry point

The `open()` handler should ensure that the minor number has an associated kernel driver instance with the same unit number attached.

Then it should set up any resources needed for the device to begin operation. This is a good place to allocate any expensive resources that should only be consumed if the device is actively being used.

Access control is provided by permissions on the filesystem (your driver code does not need to perform extra permissions checks).

The open() handler should return the following errors for the reasons listed. On success it should return 0.

ENXIO no hardware driver is attached with the relevant unit number
ENOMEM the kernel was unable to allocate necessary resources

3.2.2 close()

Can be used to release any resources that were allocated by `open()`. It is permissible for some driver resources allocated at `open()` to not be released at `close()`, if these will be re-used by a subsequent `open()` or are still in use by hardware at the time.

If an I/O operation is currently in-flight (i.e. an `ioctl` is currently being processed in another thread), then `close()` must block until that call has completed.

`close()` must not return an error. If any I/O operations during `close()` fail, or the device returns an error, then a message should be written to the system console and `dmesg`.

3.2.3 ioctl()

3.2.3.1 VKEYIOC_GET_INFO

Returns the device version information, in a `struct vkey_info_arg` (defined in `<dev/vkeyvar.h>`):

```
struct vkey_info_arg {
        uint32_t        vkey_major;
        uint32_t        vkey_minor;
};
```

ENXIO no hardware driver is attached with the relevant unit number

3.2.3.2 VKEYIOC_CMD

Performs a command-reply cycle. Takes a `struct vkey_cmd_arg` (defined in `<dev/vkeyvar.h>`) as input and updates it on output.

```
struct vkey_cmd_arg {
        /* input */
        uint            vkey_flags;
        uint8_t         vkey_cmd;
        struct iovec    vkey_in[4];

        /* output */
        uint8_t         vkey_reply;
        size_t          vkey_rlen;
```

```
        /* input + output */
        struct iovec    vkey_out[4];
};
```

The `struct iovec` members of `struct vkey_cmd_arg` function are the same `struct iovec` defined in `sys/uio.h` for `vread(1)`, `vwrite(1)` etc.

The command number must be written into `vkey_cmd_arg` by userland, and the reply message number will be written into `vkey_reply` by the driver.

All vectors with non-zero `iov_len` within `vkey_in` will be used as payload data for the command. All vectors with non-zero `iov_len` within `vkey_out` will be used for output data in order, and the total length of data written will be placed in `vkey_rlen`.

The `vkey_flags` field may be contain the flag `VKEY_FLAG_TRUNC_OK` to specify that truncated responses will be accepted by userland. If this flag is set, and insufficient output buffer space is provided for the size of the reply payload, no error will be returned, and the reply will be truncated to the buffers provided. If this flag is not set, an error is returned instead (see below).

The `VKEYIOC_CMD` ioctl blocks until the command is completely finished and reply data (if any) is available.

It must block in an interruptible fashion, such that userland processes can receive signals such as `SIGINT` (e.g. resulting from `Ctrl+C`) and immediately exit the ioctl system call.

It is permissible for an interrupted command to still continue executing on the device and have its output discarded by the driver.

ENXIO no hardware driver is attached with the relevant unit number
EIO a hardware error prevented handling this command
EFBIG the provided output buffers were not big enough for the reply data

3.3 Testing

3.3.1 Userland tools

In the base patch for this assignment, we provide a `vkeyd` daemon that accepts UDS connections speaking the `ssh-agent` protocol and communicates with your device driver.

You will need to install and start the daemon in order to test with it, for which instructions are provided below in the section *Provided code*.

The daemon is designed to not open any of the vkey device special files until the first request is made for it to perform an operation. This avoids the case where, if the daemon is set to run on startup, it immediately panics the machine due to a bug in your driver code.

To communicate with the daemon, you can use the `ssh-add(1)` tool (built into OpenBSD), as well as the provided `vkeyadm` command, which is capable of sending arbitrary commands, including those with long data fields.

3.3.2 Control interface

Your VM control interface will be extended to include control commands related to the A2 device, including the ability to make certain commands take extra time to complete, complete out of order, and introduce error conditions.

The commands for controlling this will be documented in the updated *COMP3301 VM Control Interface* document.

3.3.3 Other testing

You can also write your own additional tools for testing your device driver. You may find this the easiest way to trigger or examine certain kinds of bugs in your implementation.

This is also very useful in the early stages of the project when your driver may not yet support enough features to allow `vkeyd` to function.

3.4 Code Style

Your code is to be written according to OpenBSD's style guide, as per the `style(9)` man page.

An automatic tool for checking for style violations is available at https://stluc.manta.uqcloud.net/comp3301/public/2023/cstyle.pl. This tool will be used for calculating your style marks for this assignment.

3.5 Compilation

Your code for this assignment is to be built on an `amd64` OpenBSD 7.3 system identical to your course-provided VM.

We will only be compiling and installing the kernel for this assignment; any modifications you make to userland testing tools or `vkeyd` for your own testing will be ignored when we mark your work.

Your kernel code must compile as a result of `make obj; make config; make` in `sys/arch/amd64/compile/GENERIC.MP` relative to your repository root.

As a result of marking only your kernel, you may not modify the userland interface from what is shown above or the structures defined in `vkeyvar.h`.

3.6 Provided code

A patch will be provided that includes source for the `vkeyd` userland program, the `vkeyadm` command, and the `sys/dev/vkeyvar.h` header file containing the ioctl's that the `vkey(4)` driver must implement.

The provided code which forms the basis for this assignment can be downloaded as a single patch file at:

https://stluc.manta.uqcloud.net/comp3301/public/2023/a2-base.patch

You should create a new `a2` branch in your repository based on the `openbsd-7.3` tag using `git checkout`, and then apply this base patch using the `git am` command:

```
$ git checkout -b a2 openbsd-7.3
$ ftp https://stluc.manta.uqcloud.net/comp3301/public/2023/a2-base.patch
$ git am < a2-base.patch
$ git push origin a2
```

The patch contains new headers, so you will need to run `make includes` once you have applied the patch:

```
$ cd /usr/src/include
$ doas make includes
```

To build the provided userland components, you will need to change into their relevant directories and compile and install them:

```
$ cd /usr/src/usr.sbin/vkeyd
$ make obj
$ make
$ doas make install
```

```
$ cd /usr/src/usr.sbin/vkeyadm
$ make obj
$ make
$ doas make install
```

3.7 Recommendations

You should refer to the course practical exercises (especially pracs 4, 5 and 6) for the basics of creating a PCI device driver and attaching to a device.

The following are examples of some APIs you will need to use in this assignment:

- The `bus_space(9)` family of functions
- The `bus_dma(9)` family of functions
- `pci_intr_map_msix(9)` and `pci_intr_establish(9)`

4 Submission

Submission must be made electronically by committing to your Git repository on `source.eait.uq.edu.au`. In order to mark your assignment the markers will check out the `a2` branch from your repository. Code checked in to any other branch in your repository will not be marked.

As per the `source.eait.uq.edu.au` usage guidelines, you should only commit source code to your repository, not binary artefacts, cscope databases, or patch files.

Remember to use `git status` before committing to examine the list of files being added.

Your `a2` branch should consist of:

- The openbsd-7.3 base commit
- The a2 base patch commit
- Commit(s) implementing the `vkey(4)` driver

We recommend making regular commits and pushing them as you progress through your implementation. As this assignment involves a kernel driver, it is quite likely you will cause a kernel panic at some point, which may lead to filesystem corruption. Best practise is to commit and push before every reboot of your VM, just in case.

4.1 Marking

Your submission will be marked by course tutors and staff, during an in-person demo with you, at your weekly lab session.

You must attend your lab session in-person, otherwise your submission will not be marked. Online attendance (e.g. via Zoom) is not permitted.

5 Testing

We will be testing your code's functionality by compiling and running it in a virtual machine set up in the same way we detail in the Practical exercises.

Tests will be run against the same virtual agent device available in your personal virtual machine for the course.