Hyper-V Injection Module

Microsoft gives an introduction to writing drivers for Windows operating systems in [74]. The relevant details are summarized here. Drivers are mostly associated with device communication. If the operating system wants to interact with a device, it requests an operation of the driver, which in turn knows how to communicate with the hardware device in order to perform the request. But there is also the concept of “software drivers”, which do not manage any hardware but still need access to kernel resources. They are also referred to as “kernel services”. The hypercall injection driver is exactly that. It needs to be a driver to run code with Ring 0-privilege to be able to invoke hypercalls to Hyper-V.

There are different frameworks available to use for driver development. In 1993, Windows NT was released with a new driver model, which today is called the legacy NT model. It is still usable in modern Windows operating systems; however, it can only be used to implement software drivers. A driver has to have a `DriverEntry` function, which is executed when the driver is loaded and should set up everything the driver needs. This setup includes registering handler functions for the different kinds of I/O Request Packet (IRP) that the driver can processes. IRPs are data structures used to transport information between the operating system and a driver or between drivers. Amongst others, there are the IRPs IRP_MJ_CREATE, IRP_MJ_READ, IRP_MJ_WRITE, IRP_MJ_CLOSE, and IRP_MJ_DEVICE_CONTROL. The operating system translates I/O requests from user space into IRPs and delivers them to the IRP handler function of the driver.

Together with Windows 98, the Windows Driver Model (WDM) framework was released. It introduced the concept of Plug and Play (PnP) and power management to drivers. PnP means the operating system loads the appropriate drivers for each device automatically, on startup as well as when hot-plugging devices. Drivers have to support the case that multiple devices can be present that require the same driver. Power management functionality enables drivers to react to changes of the power state of the system and possibly react by changing the state of the managed devices.

Writing a WDM driver with PnP functionality requires 1500-4000 lines of boilerplate code, according to Ionescu [75]. This was addressed with the Windows Driver Foundation (WDF) framework. It is based on WDM but tries abstracting away the boilerplate implementations as much as possible. For any WDF driver, PnP works out of the box. Also, it eases dealing with IRPs by providing I/O queues. WDF contains the User-Mode Driver Framework (UMDF), which is used to implement drivers that can run in user space, and Kernel-Mode Driver Framework (KMDF) for drivers that have to run in the kernel.

WDF/KMDF was chosen to implement the injection driver for the fact that there is a sample driver made available by Microsoft[1] that is a software driver implemented with KMDF to demonstrate different ways that user space applications and drivers can communicate. At first, the application that comes with the driver loads the driver into the kernel. The `DriverEntry` function of the driver is executed. The function tells the driver execution framework that it is not a PnP driver, and thus will not provide the otherwise mandatory `AddDevice` function. As a non-PnP driver, it has to register a `DriverUnload` function. With PnP drivers, devices accessible by user space are created automatically for all plugged-in hardware devices. To still provide a device that can be used by applications to issue I/O requests to, a control device is created, callback functions for IRPs are set, and a symbolic link is created, which makes the device accessible to applications. When the driver is loaded, the application interacts with it by issuing a `CreateFile` request to the symbolic link of the driver’s control device, proceeded by several ioctl, read, and write calls to the file.

[1] `https://github.com/Microsoft/Windows-driver-samples/tree/master/general/ioctl/kmdf`

This sample code has been adapted to implement the hypercall injection driver, whose behavior was described in pseudocode in Listing 4.7. In the `DriverEntry` function, code has been added to prepare for the execution of hypercalls. Access to the hypercall page is achieved by using the `__readmsr` function to retrieve the content of MSR 0x40000001, whose bits 63:12 contain the physical page number of the hypercall page. The remaining 12 bytes are used for flags. By setting them to 0, the physical address of the hypercall page is determined. With `MmMapIoSpace`, a new virtual address mapping for this physical address can be created. Both the input and output pages are allocated using `MmAllocateContiguousMemory`. Because 4096 bytes (the page size) is requested, this function allocates the memory page-aligned, as demanded by Hyper-V. The hypercall calling convention also demands the physical addresses of the pages. These are determined using `MmGetPhysicalAddress`.

Then, unchanged, the user space app creates file for the control device, but only performs a single write operation to it. In the buffer that is supposed to hold what should be written, it places the file path of the binary campaign file that should be executed, and a set of flags describing which values should be written to the log file. The path and expected values are given to the application via command-line arguments. The `FileEvtIoWrite` function in the driver is the one registered to process write requests. Given the file path in the buffer, this function is supposed to execute the campaign and output a log file. For the log file path, ".out" is appended to the path of the campaign.

Apart from the flags that describe whether execution times, raw timestamps, hypercall result values, or the output page should be logged, there is another flag, which specifies how file access should be handled. In one scenario, the driver sequentially reads campaign entries and input pages as needed. Thus, the driver uses very little memory, and it does not matter how large campaign files or log files are. However, performing file access costs time, which introduces small artificial delays to the campaign. So, there is also a variant, which reserves in-kernel memory buffers for the campaign and the log file, reads the campaign complete into this memory, performs it using only memory accesses, and in the end, writes the log buffer out to file in one operation. This strategy cannot be used when the buffers require more memory than the driver can allocate.

In both cases, a handle to campaign and log files are opened in the beginning using `ZwCreateFile`. The flags bits passed by the application are written to the beginning of the log file. This way, the report generators know which values are included in the log file and can interpret them correctly. So, even the user executing the campaign did specify to log no values, there is a log file, containing only the flags.

There is something that has not been mentioned when explaining the campaign listener that generates the binary campaign: The binary is not solely composed of hypercall entries, input pages, and delay entries. At the beginning, the generator places a header, which states how many bytes it takes to store the campaign, and how many hypercalls and delays it consists of. Each value is encoded with 4 bytes. In the case of the file-based approach, the driver reads the 12 header bytes and discards them. But if the memory-based execution is chosen, two memory buffers are allocated using `ExAllocatePool`, with their size depending on the values of the header. The buffer for the campaign is as big as the specified campaign size. The log buffer's required size is calculated based on the requested log values, the size of the lag values, and the number of hypercalls and delays in the campaign. Listing 5.10 presents the calculation.

Listing 5.10: Calculation of Log Buffer Size (in byte)

```
log_buffer_size = ((flags_received->exectime != 0) * 8 +
                   (flags_received->timestamps != 0) * 16 +
                   (flags_received->result != 0) * 8 +
                   (flags_received->output != 0) * 4096)
                     * info.num_calls +
                 ((flags_received->exectime != 0) * 8 +
                 (flags_received->timestamps != 0) * 16)
                     * info.num_waits;
```

Then, the campaign is read completely and stored in its dedicated buffer.

The file-based approach notices the campaign end when it receives a `STATUS_END_OF_FILE` when trying to read a new campaign entry. However, there is no way to know when the end of the memory buffer is reached - at least until the whole operating system crashes due to access to unmapped virtual memory. Thus, beforehand, the driver calculates the first memory address after the buffer and stores it in `campaign_buffer_end`.

First, the execution strategy for the memory-based technique will be explained. Then, the differences when using the file variant are laid out. The memory approach keeps track of two pointers. At the start, they are pointing to the begin of the campaign and log buffer, respectively. The campaign pointer is cast to a pointer to a `struct campaign_entry` to have access to the individual components and advanced to point to the first memory address after the entry. The struct resembles the campaign entries described in Figure 4.5, and its definition is displayed in Listing 5.11.

Listing 5.11: Injection Driver `struct campaign_entry`

```
#pragma pack(1)
struct campaign_entry {
        UINT8 type;
        union {
                struct {
                        UINT32 waiting_time;
                        UINT16 reserved;
                } waiting_info;
                struct {
                        UINT16 callcode;
                        UINT16 count;
                        UINT16 input_page_size;
                } hypercall_info;
        } info;
};
```

It is necessary to specify the `pragma pack` because otherwise, the compiler would insert padding after the `type` field to align the rest of the struct. This is beneficial for performance, but then the struct would not match up with the campaign entries in memory. The first byte can be used to differentiate between hypercall and delay. Then, the `info` union provides access to the other 6 bytes, depending on the type. In case of a delay entry, the driver reads the `waiting_info.waiting_time` field, and passes the value to the function `KeStallExecutionProcessor` to sleep for this number of microseconds. Right before and after, the system time is taken using `KeQuerySystemTimePrecise`. If execution times should be logged, the difference of the timestamps is calculated (stored as an 8-byte value), the result written to the log pointer, and the pointer is increased by 8 bytes. If the raw timestamps should be logged, the first is written to the log pointer (also 8 bytes in size), the pointer is advanced, the second timestamp is written, and the pointer advanced again.

When instead a hypercall entry is encountered, more work has to be done. The input page size is retrieved from the entry. This number of bytes is copied from the campaign buffer to the input page (which was reserved in `DriverEntry`). Accordingly, the campaign buffer is increased by this number. With the input page set up, the registers have to be set up according to Table 2.4: RCX holds the call code, RDX the physical address of the input page, and R8 the physical address of the output page. Conveniently, the calling convention for functions passes the first three arguments in these registers. Furthermore, the calling convention demands the return value to be placed in RAX, where the hypervisor places the result value. Thus, it possible to perform a hypercall by casting the virtual address of the hypercall campaign (obtained in `DriverEntry`) to a function pointer of a function with the correct signature and call it with the call code and the physical addresses of the input and output pages. The return value will be the hypercall result value. Listing 5.12 shows the C code necessary to perform this.

Listing 5.12: Injection Driver Hypercall Invocation

```
result =
    ((UINT64(*)(UINT64, UINT64, UINT64))
    hypercall_page_virtual)(call_code,
                            input_page_physical,
                            output_page_physical);
```

Again, timestamps are taken before and after the call and are logged exactly as for delays. If the result value should be logged, it is written to the log pointer, which is subsequently advanced by 8 bytes. Similarly, if the output page has to be stored in the log, the whole 4096 bytes are copied from the output page to the log buffer. Of course, then, the pointer is increased by 4096. The hypercall invocation and subsequent logging are executed as many times as instructed by the repetition count of the campaign entry.

After the processing of a campaign entry, the driver continues by reading the next campaign entry from the buffer. However, before accessing memory, it checks whether the campaign pointer has reached the `campaign_buffer_end`. If it has, the driver writes the log buffer to file and completes the write request, returning control back to the application, which unloads the driver and terminates.

The course of action for the file-based implementation is similar. The only difference is that every time values have been read and written to a buffer, and the pointer has been advanced, now it is read directly from the campaign file using `ZwReadFile`, and written directly to the log file using `ZwWriteFile`. It stops when reading from the campaign yields `STATUS_END_OF_FILE`. Because the log file is already written, the driver only has to close the file handles before finishing.