Hypercall Injection Module

The hypercall injection module is supposed to take a binary campaign as an input, perform the campaign, which means to execute hypercalls as described, and wait specified delays before issuing the next. Furthermore, information about the execution should be written to a binary log file.

The choice of how to implement the kernel module is easy, because there is no choice. The problem has already been addressed during the design of the framework in Section 4.1. As mentioned there, it is possible to create a Loadable Kernel Module (LKM) for Linux or any other operating system that can perform hypercalls to Hyper-V. Yet, because many calls are only usable by the root partition, which runs a Windows OS, a Windows driver is the only way to test all of Hyper-V's available hypercalls.

Section 2.3 already went into detail about which calling conventions are supported by Hyper-V: the memory-based, default call; the register-based fast call for hypercalls with two or fewer parameters and no output value; and finally, the XMM register-based extended fast call for hypercalls with up to 112 bytes of parameters. In this thesis, we restrict the injection driver to only support the memory-based calling technique. This does not really prune any functionality regarding executable calls, because all hypercalls can be issued using this convention. However, due to the complexity of handling variably sized inputs, the driver also does not support rep calls and calls with variable header size. With these restrictions in place, the hypercall input value (see Table 2.2) reduces to only the hypercall call code, stored in the bits from 15 down to 0. So, the binary campaign has to provide the call code of each requested hypercall, which the driver has to store in the RCX register.

For the driver to comply with the memory-based calling convention, it also has to reserve two memory pages (4096 bytes in size, aligned to 4096-byte memory boundary). One of them has to be filled with parameters, which have to be supplied by the campaign. The GPA (Guest Physical Address) of this page has to be placed in the RDX register. The second page is the output page, which is written to by the hypervisor during hypercalls. So, for the output page, no information is required from the campaign. The driver only has to store the GPA of the output page in register R8 when a call is issued.

Section 2.3 also already explained that, when registers and memory are set up, control is transferred to the hypervisor either by the VMCALL or the VMMCALL instruction, depending on whether the system is running on an Intel or AMD CPU. However, it did not mention that the hypervisor takes care of abstracting the architectural differences away.

Hyper-V has the possibility to deploy overlay memory pages to the address spaces of its partitions. They are called overlay pages because the normal page with the address at that the overlay page is deployed is no longer accessible. Instead, reads and writes happen to the overlay page. Hyper-V uses such an overlay page to standardize the way hypercalls are invoked by placing the correct instruction at the beginning of the page. Thus, instead of directly executing the correct instruction, hypercall invocation code can redirect execution to the address of the overlay page, where the hypervisor placed the correct instruction. The second instruction conveniently returns execution to the caller. The contents of the hypercall page on an Intel machine are displayed in Figure 4.4.

The figure is a screenshot of the Windows Debugger (WinDbg). Each line corresponds to one instruction. The leftmost column shows the instructions' beginning memory addresses, specifically GVAs (Guest Virtual Address). Right of those are the bytes that encode the instruction, displayed in hexadecimal representation. And the rightmost column shows the instructions' assembly representations. As described, at the beginning of the hypercall page is the VMCALL, followed by a RET, which jumps back to the caller. There are also

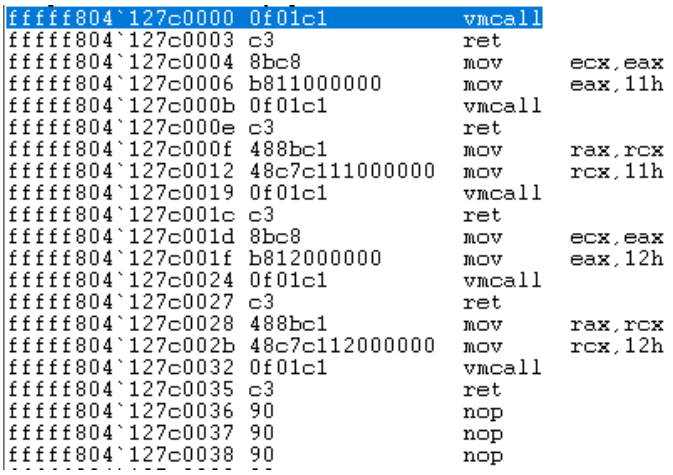

Figure 4.4.: Contents of Hyper-V's Hypercall Overlay Page (Intel)

other calling variants following, which move some registers before the VMCALL, but they are not relevant to the injection driver design.

So, at which memory address does Hyper-V deploy this overlay page? In fact, the guest operating system chooses the location and tells the hypervisor by writing it to one of the Model-Specific Registers (MSRs), which are one kind of control registers available in the x86_64 architecture. It is possible to read the MSR afterward and retrieve the specified address. However, this address is a GPA, but execution can only be redirected to GVAs. The driver could either traverse the page table to find the existing translation to this GPA or add an entry to the page table that maps an until now unused GVA to the target GPA. When a GVA is established, the driver can invoke hypercalls by executing a CALL instruction and passing the hypercall page GVA.

After performing a call, the hypervisor has placed the result value in the RAX register. Depending on the call, it might also have written to the output page. Thus, it is crucial that the driver is able to log these values. Additionally, the driver should be able to take timestamps right before and after performing a call, to measure how long the hypervisor took to process the request. The same can be done for delays to verify that they are carried out accurately.

With the necessary details discussed, it is possible to address the third research question:

RQ3: *How to inject hypercalls into Hyper-V?*

It is answered by providing the course of action of the hypercall injection Windows driver in pseudocode, as given in Listing 4.7.

Listing 4.7: Course of Action of the Hyper-V Hypercall Injection Driver

```
hypercall_page_gpa = retrieve_from_MSR()
hypercall_page_gva = get_gva_for_gpa(hypercall_page_gpa)

input_page_gva = allocate_kernel_memory_page_aligned()
input_page_gpa = lookup_gpa_for_gva(input_page_gva)

output_page_gva = allocate_kernel_memory_page_aligned()
output_page_gpa = lookup_gpa_for_gva(output_page_gva)

while (!end_of_campaign) {
    entry = campaign.next_entry()
```

```
    if (entry.is_hypercall()) {
        start_timing()
        RCX = entry.call_code
        RDX = input_page_gpa
        R8 = output_page_gpa
        execution_jump_to(hypercall_page_gpa)
        result = RAX
        stop_timing()
        log_timing()
        log_result(result)
        log_output_page()
    } else if (entry.is_delay()) {
        start_timing()
        wait(duration = entry.delay_duration)
        stop_timing()
        log_timing()
    }
}
clean_up_memory()
```

The individual steps should be comprehensible because all of them have been explained throughout this section; the pseudocode is only supposed to bring everything together and in order.

Binary Campaign Format

For the definition of the binary campaign format, the following requirements have been established:

- **Call Code**: The call code of the hypercall to be executed has to be placed in the RCX register. It is 16 bits in size.
- **Input Page**: The content of the input page describes the parameters of the hypercall. The driver has to copy it to the prepared input page, which fulfills the memory alignment demanded by Hyper-V. The total input page has a size of 4096 bytes. Because no documented hypercall actually uses the whole page and many calls only need a few bytes at the beginning of it, it is beneficial to encode only a portion the page. This makes it necessary to specify how large the encoded portion is. As the maximum size is 4096 bytes, it cannot be specified with 1 byte (highest value 255), but 2 bytes suffice (highest value 65535).
- **Repetition Count**: This has not yet been mentioned. If, e.g., in the case of stress testing, the same call is repeated many times, it is beneficial to reduce campaign entries by repeating a campaign entry multiple times. There is a trade-off between how many repetitions one entry can encode, and how large the memory overhead is when no repetitions can be used due to different call codes or parameter values. Two bytes are chosen to encode the repetition count, which allows one hypercall campaign entry to be executed 65535 times.
- **Entry Type**: As the campaign consists not only of hypercall entries but also delay entries the driver needs a way to identify the type of the entry to know how to interpret the entry's values and whether it should perform a hypercall or wait.
- **Waiting Time**: In case of a waiting entry, the driver has to know for how long it should wait before continuing with the next entry. The call code, repetition count,

and input page size occupy 6 bytes of memory. To avoid copying the waiting time and expand it to 8 bytes (necessary to access the value), only four bytes are used to store the delay duration. With a microsecond resolution, this still allows for delays of over 71 minutes.

Thus, an entry in the binary campaign file is 7 bytes large, and is, depending on its type, structured as displayed by Figure 4.5. The driver can check the first byte of an entry

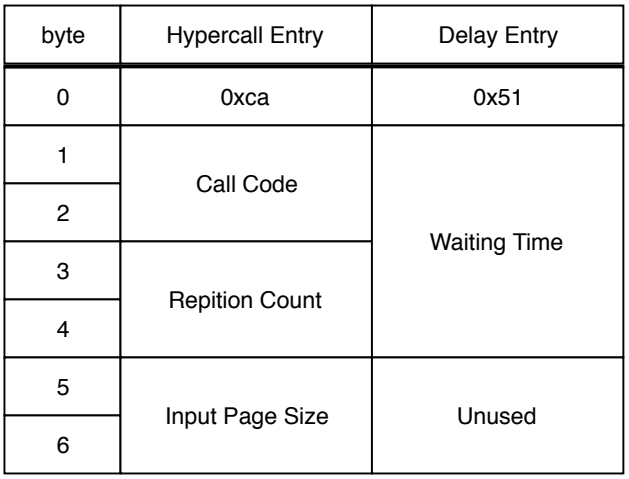

Figure 4.5.: Hypercall Campaign Entry Structure

to identify it as a hypercall or as a delay entry. According to the type, the driver can retrieve the encode information and perform the action as defined. However, in the case of a hypercall that requires parameters to be placed on the input page, the next "Input Page Size"-many bytes of the campaign file have to be copied to the hypercall input page. Consequently, the structure of a hypercall campaign file can look like the example provided

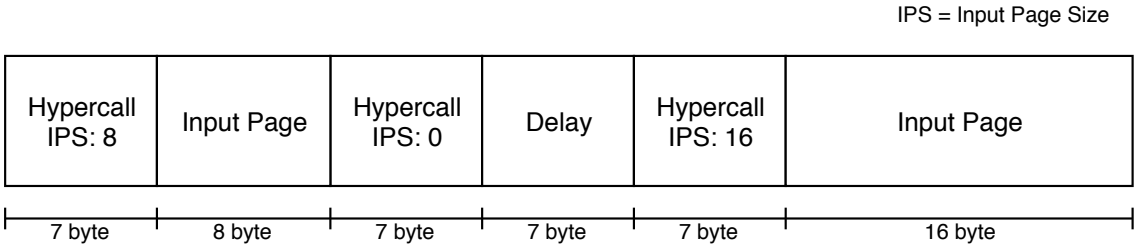

Figure 4.6.: Hypercall Campaign Structure

in Figure 4.6. At the beginning of a campaign file, there always is an entry. Therefore, the injection driver reads 7 bytes. The first byte is 0xca, so the entry describes a hypercall. The input page size, stored in bytes 5 and 6, is 8. The driver reads the next 8 bytes of the campaign file and writes them to the input page. After performing the first call, the driver reads the next 7 bytes. Again, it is a hypercall entry, but it does not require values on the input page. The same applies to the following delay, only the 7-byte entry is needed to be read before moving on to the next entry. Finally, for a hypercall with a specified input page size of 16, the driver reads 16 bytes of the campaign and stores them on the input page.

Binary Log Format

Throughout the course of this section, the following information has been identified as potentially interesting and worth logging:

- **Result Value**: The hypervisor places the result value in RAX before returning execution control to the caller. It can be used to check if hypercalls were processed successfully, or, if not, which error is indicated. Storing RAX requires eight bytes.
- **Output Page**: The output page potentially contains output values written by the hypervisor. Storing the complete page takes up 4096 bytes. Similar to the input page, no documented call actually uses the whole page, often only a few bytes are used, if at all. However, without explicit knowledge about the output parameters of the call performed, the same variably sized storing is not applicable. When running large campaigns for stress testing, logging of the output page is probably not relevant, so the driver stores the whole 4096 bytes. It should, however, be possible to enable and disable logging the page at all.
- **Timing Measurements**: An important feature is the possibility to measure the execution times of hypercalls because they can provide information about the behavior of the hypervisor. As shown in the pseudocode algorithm for the injection driver (see Listing 4.7, execution time is calculated from timestamps taken right before and after hypercall or delay execution. To quantify the time needed to execute overhead code, i.e., loading campaign entry and performing logging, it makes sense to provide the possibility to store the raw timestamps, as well. The Windows kernel system time is the number of 100ns-intervals since January 1, 1601, which requires 8 bytes to store.

There are different kinds of testing campaigns with different goals. Some might be interested in which result value is returned when varying input parameters, but do not care about execution times. Others might want to precisely specify the rate of hypercall invocation, and measure the execution times. Logging the output page in this case will cause overhead, and thus decrease the rate defined by delays in the campaign. So, logging result and output values should be disabled here. Yet another tester might want to only use campaigns to generate load but measure other metrics of the system. Then, no logging at all is the preferred solution.

Thus, the driver should support selecting if the result value should be logged, if the output page should be logged, if timing should be logged, and - if that is the case - whether the execution times suffice or the raw timestamps have to be stored.

This concludes the definition of the behavior of the injection driver, as well as its input binary campaign file format and output binary logging file format. Details about the actual implementation are provided in Section 5.3.