HyperDbg Debugger

A debugger designed for analyzing, fuzzing and reversing

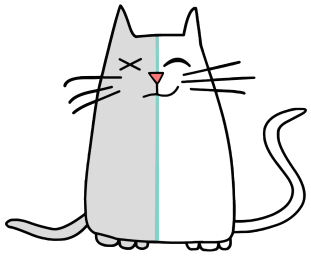
1st Edition

Kernel Debugger Design in HyperDbg

Website
`https://hyperdbg.org`

Research
`https://research.hyperdbg.org`

July 4, 2022



The kernel-mode debugger of HyperDbg is called "kHyperDbg". Unlike all the other software debuggers like *WinDbg* and *GDB*, HyperDbg is not a ring 0 (kernel) debugger. It uses ring -1 for its debugging purpose. Although using ring -1 (hypervisor) as the base of the debugger has its own benefits, many considerations are required for the implementation. In this article we present the design of HyperDbg Kernel Debugger.

---

1 Overview

Considering the fact that HyperDbg is organized to be operational in the hypervisor-level, certain key factors are to be contemplated. Moreover, as for the essential features, such as the transparency, other considerations should also be taken into account. The Figure 1 below shows an overview of the HyperDbg Kernel Debugging (kHyperDbg) mechanism, which will be thoroughly discussed.

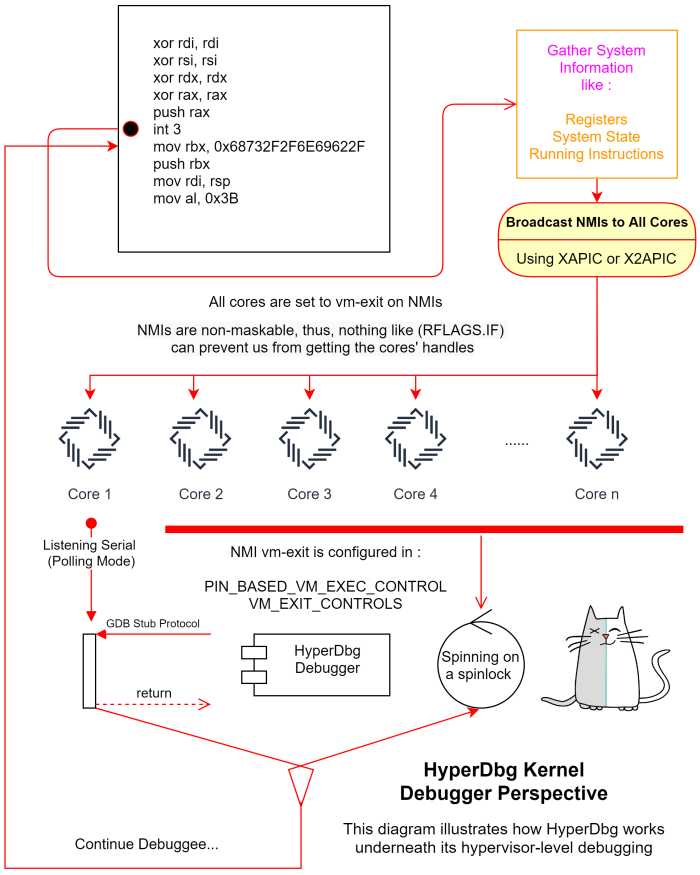


Figure 1: The Overview of Kernel Debugging in HyperDbg

2 Background:Non-Maskable Interrupts (NMIs)

HyperDbg extensively uses NMIs to perform its tasks in a multi-core system; thus, it is necessary to overview the underlying technology and different parts of NMIs in modern processors.

An (NMI) is a hardware-based interrupt that typically cannot be ignored by the system. NMI is usually triggered in order to alert for hardware errors. These errors include non-recoverable internal system chipset errors, system memory corruptions such as parity and ECC errors, and data corruptions on the system and peripheral buses. Nevertheless, for some reasons, it is possible to mask specific NMIs by employing special techniques.

Software Programs typically use debugging NMIs to diagnose, analyze, and fuzz codes. In such cases, an NMI can execute an interrupt handler transferring the control flow to a particular monitor program. In these circumstances, the developer can monitor the memory and examine the program's internal state at the instant of its interruption. This also allows the debugging or diagnosing of computers that appear hung.

On some systems, a computer software could drive an NMI through hardware, and software debugging interfaces and system reset buttons. In HyperDbg as a hypervisor-level software, the NMI triggering is extensively used for kernel debugging.

Here we investigate the NMI technology details, and this section is used as a base for future references. You can skip these parts safely and return back to the explanations whenever these terms are used in the rest of the article.

2.1 NMI Controller Bits In VMX

The controller bit functionality of the NMI behavior in Intel VT-x can be used for many purposes in VMX mode.

2.1.1 NMI-Exiting (Pin-Based VM-Execution Controls)

If the NMI-Exiting control bit is set to 0 and an NMI arrives while in VMX non-root mode, the NMI is delivered to the guest via the guest's Interrupt Descriptor Table (IDT). If the NMI-Exiting control is 1, an NMI causes a VM-exit. In other words, if this control bit is set, NMIs cause VM-exits. Otherwise, they are delivered normally using descriptor 2 of the IDT.

2.1.2 Blocking by NMI (Interruptibility state)

Delivery of a non-maskable interrupt (NMI) blocks subsequent NMIs until the upcoming IRET execution. Setting this bit indicates that blocking of NMIs is in effect, and clearing this bit does not imply that NMIs are not blocked. On the other hand, if *Virtual NMIs* control bit is set to 1, virtual-NMI blocking is enabled. This does not imply the blocking of normal NMIs.

2.1.3 NMI-window exiting (Primary Proc-Based VM Controls)

As a *window-exiting* field, NMI-window exiting makes it possible to take advantage of open window or any possible opportunity to deliver the NMI when the user tends to continue the execution (*VMRESUME*). As described by Intel,[2] if set, NMI-window exiting control executes VM-exit at the beginning of any instruction with the condition that no virtual NMI blocking. Hence, whenever the guest is ready to deliver an NMI by ensuring that NMI blocking is disabled, a VM-exit occurs if this control bit is set.

2.1.4 Virtual NMIs (Pin-Based VM-Execution Controls)

By setting up this control, NMIs are never blocked, and the *blocking by NMI* bit (bit 3) in the interruptibility-state field would indicate *virtual-NMI blocking*.

2.1.5 IRET & NMI Unblocking

The behavior of IRET with regards to NMI blocking is determined by the settings in the *NMI-exiting* and *virtual NMIs* controls:

- If the *NMI-exiting* VM-execution control is 0, IRET operates normally and unblocks NMIs. (If the *NMI-exiting* VM-execution control is 0, the *virtual NMIs* control must be 0)

- If the *NMI-exiting* VM-execution control is 1, IRET does not affect the blocking of NMIs. If, in addition, the *virtual NMIs* VM-execution control is 1, the logical processor tracks virtual-NMI blocking.

2.2 Receiving NMIs while other cores are on VMX-root

This mechanism is carried out to resolve the issue with receiving NMIs while the VMX-root mode is already enabled, e.g., when the user intends to inject NMIs to other cores and those cores are already operating in VMX-root mode. In order to tackle such a problem, the system is required to create a host IDT in VMX-root mode (Note that it should set HOST_IDTR_BASE, and there is no requirement for LIMIT since the value is fixed at 0xffff for VMX operations). Since the debugging mechanism of the Windows is utilized, the same IDT with the guest (guest and host IDT is the same) is employed. However, in future versions, this drawback is supposed to be fixed by using a specified NMI ISR handler in VMX-root mode.

2.3 Watchdog NMIs in Windows

During the implementation of HyperDbg, we realized that Windows creates NMIs probably for watchdog purposes. This is implied by the observation that even if a kernel code is in a deadlock while the interrupts are disabled, NMIs still could grant control of the processor as NMIs are non-maskable. In HyperDbg, the NMIs are used to halt all the other cores (other than the main core that triggered an event). HyperDbg does not intend to prevent Windows from its normal watchdog activities; so, particular flags are set when HyperDbg tries to broadcast NMIs. In this context, all the NMIs cause VM-exits, if the flag for NMIs is previously set. This means that the NMI is related to HyperDbg, and the debugger ignores the NMI and performs the execution to halt the core. If the NMI is caused by Windows, then HyperDbg re-injects the NMI back to the core. This way, both Windows and HyperDbg benefit from using NMIs functionality simultaneously without conflict.

3 Design primitives

- **Single-core Execution**: In order to make the debugging process precise as well as disrupt-resistant, HyperDbg is designed in such a way that all the CPU cores are halted and spinning on ring -1 (hypervisor) whenever the debugging procedure is executing. Hence, in the debugging mode, only one core (current operating core) is listening for new commands from the debugger on a polling mode serial. The underlying mechanism to realize such a feature will be elaborated in detail.

- **Pausing Scenario**: HyperDbg supports different pausing scenarios. Upon the request from the user (for instance, an interruption by pressing CTRL+C), a packet is sent to pause the debuggee. In this method, debugger process the packet on the user-mode, invoking an *IOCTL*, executing a *VMCALL* which transfers from the kernel-mode to the *VMX-root* mode and in VMX-root mode, it pauses the debuggee.

- **Operating Modes**: In HyperDbg, we operate in three different operation modes. The first and the primary mode of operation for local and remote debugging is VMI Mode. The second mode is Debugger Mode, and the third mode is Transparent Mode.

- **Event, Actions, and Conditions**: To render HyperDbg 's operations, we define three concepts of Event, Actions, and Conditions. Almost all of the HyperDbg features are developed as an event. For instance, hidden hooks are realized as events in HyperDbg. When an event is triggered, corresponding actions will be performed. Also, the event is triggered when the system executes a *SYSCALL* instruction or a *SYSRET* instruction. Generally, each event has a condition and might have zero or multiple corresponding actions. Together, any arbitrary debugging mechanism could be defined and executed in HyperDbg.

3.1 Single Core Kernel Debugging

With the employment of Intel NMI thoroughly described in the previous section, HyperDbg performs on a single core in order to provide a reliable and robust debugging experience. The details regarding the implementations and employed techniques are already studied in Section 2.

3.2 Debugger Pausing

Typically, there are two scenarios in which the kernel debugger is paused. A breakpoint is triggered either by a break request from an event or the script engine. In this context, if the user is in the kernel-mode, a *VMCALL* occurs, and a chain of events is handled accordingly. If the user is already in *VMX-root* mode, other cores should be somehow notified to prevent a system-level crash.

Operating in VMX-root mode is similar to HIGH_IRQL. In VMX-root mode, all the interrupts are masked because of RFLAGS.IF bit.

3.3 Different operating modes

Here we discuss the operation modes provided by HyperDbg.

3.3.1 VMI Mode

For a typical debugging experience, VMI Mode or Virtual Machine Introspection Mode of HyperDbg should be used. In both local debugging and remote debugging, VMI Mode features are already enabled. In VMI mode, the user can use all of the HyperDbg features, except for breaking the debugger and step instructions in kernel-mode. Nevertheless, user-mode breaks to the debugger and step instruction are functional without any limitations. Based on HyperDbg actions, users can use scripts and custom codes in both user-mode and kernel-mode in this mode of debugging. This mode can be used in both local debugging and remote debugging.

3.3.2 Debugger Mode

If HyperDbg is to be used in order to connect to the kernel and halt the system to step-in and step-over through the kernel instructions, then the Debugger Mode of operation could be taken into account. Obviously, Debugger Mode can not be used for local debugging. Here, debugging connectivity should be carried out with a serial cable or virtual serial device. Note that it is recommended to use VMI Mode when breaking and halting the system for stepping and instrumenting instructions are not required. This is due to the TCP connectivity in VMI mode, which is substantially faster than a serial device.

3.3.3 Transparent Mode

Transparent Mode is a concept provided in HyperDbg for hidden debugging. By enabling this mode, HyperDbg tries to make itself transparent from anti-debugging and anti-hypervisor methods. Thus, HyperDbg conceals itself against the hypervisor's presence on microarchitectural timing attacks and other software methods. Note that we do not claim to guarantee 100% transparency, but the proposed methodology for transparency in HyperDbg makes it substantially more challenging for the anti-debugging methods to detect the presence of the debugger. To enable this mode, *'!measure'* and *'!hide'* commands sould be used by the user. The Transparent Mode can be activated in both VMI Mode and Debugger Mode.

3.4 Events, Actions, Conditions

3.4.1 Events

The term "event" is used to describe the fundamental functionality in HyperDbg. Each time HyperDbg is executed, a special event is set. For instance, an event is set when a system call is intercepted via HyperDbg. Users should set up an event to be triggered in the case of a "syscall" execution. As another example, hidden

hooks are defined by an event on a particular function. Whenever these events are triggered, HyperDbg performs specific actions, arbitrarily configured by the user for debugging purposes.

In the first release of HyperDbg, the following events in Table 3.4.1 are supported.

!epthook	Classic EPT hook
!epthook2	EPT hook with detours hooking
!syscall	Hook execution of system-calls
!sysret	Hook execution of sysret instruction
!monitor	Monitors any access (Read/Write) to a region of memory
!cpuid	System-wide CPUID instruction execution detection
!msrread	System-wide RDMSR instruction execution detection
!msrwrite	System-wide WRMSR instruction execution detection
!tsc	System-wide RDTSC/RDTSCP instructions execution detection
!pmc	System-wide RDPMC instruction (performance counter) execution detection
!exception	Monitors and hooks first 32 entries of Interrupt Descriptor Table (IDT)
!interrupt	Monitors and hooks external-interrupts 33 to 256 entries of Interrupt Descriptor Table (IDT)
!dr	Detects any reads or write into hardware debug registers
!ioin	Monitors and ability to modify I/O ports and IN instruction
!ioout	Monitors and ability to modify I/O ports and OUT instruction
!vmcall	System-wide VMCALL instruction (hypercalls) execution detection

3.4.2 Actions

HyperDbg provides three types of actions, namely *Break*, *Script*, and *Custom Codes*. *Break* is the exact conventional feature used in classic debuggers, where all of the cores are paused, and no instruction is executed without the debugger's prior permission (note that this feature is not available in local debugging). *Script* is another type of action that helps to view the parameters, registers, and memory content without breaking into the debugger. It could be used when the user intends to analyze the target. This action could create logs or run codes in the kernel space and also modify registers or change memory. HyperDbg 's proprietary *Script* mode provides much faster debugging actions compared to the exiting debuggers[4] by performing the logging mechanism in the kernel. However, displaying the messages is handled in the user-mode. *Custom Codes* gives the ability to run custom assembly codes whenever a special event is triggered; this option is also very fast and powerful as it lets the user customize the HyperDbg according to the user's requirements.

3.4.3 Conditions

By design, each event is triggered only in the case of two scenarios. The first scenario is by setting a particular condition for the event (conditionally). So each time the event is triggered, and the condition is met, the specified action is performed.

In the second situation, the event is triggered where no condition is set (unconditionally). Hence, it is triggered anyhow with no condition check.

4 Communicating and Task Appliance

As an essential part of HyperDbg, communicating with the debugger in a safe manner is carefully considered in the design. This section will describe how the communication with HyperDbg works. Applying tasks as another important user interface part is also discussed.

4.1 Sending data over serial

It is impossible to use regular Windows API (i.e., APIs for receiving and sending data over the network) in a debugger. It is due to the fact that data should be directly received from the device. Also, interrupts

are disabled in VMX-root mode forcing the system to access data from the remote device in the polling mode. Moreover, working in HIGH_IRQL-like environments requires extra implementation for data transfer as Windows uses different device stacks in different IRQL levels to transfer data from the network. We leave this to the future release of HyperDbg.

In the initial version of HyperDbg, serial devices are considered to transfer data due to the simplicity in design and usage and their polling mode support.

There are four primary serial ports that HyperDbg is able to connect. Following the connection initialization between the device and serial port, the required COM name should be provided to HyperDbg in order to establish the connection to the target device.

HyperDbg handles all connections from debuggee to debugger only on the VMX-root to prevent/avoid deadlocks.

However, the connection from the debugger to the debuggee is both in VMX-root mode and VMX non-root mode. Whenever the debugger tries to halt debuggee, we use the interrupt mode of the serial device. The debugger sends a pause (CTRL + C) signal to the debuggee. After that, the debuggee receives the packet and passes it to the user-mode. From user-mode, an IOCTL is invoked, and the debuggee is paused in VMX-root mode. Now, we are waiting for the commands (packets) from the debugger in the polling mode.

Using interrupt-mode in pausing the debugger has made it possible to avoid unnecessary checks in polling mode when the debuggee is running.

4.2 Broadcasting tasks to all cores upon debugger continuation

In order to notify all the other cores, HyperDbg employs Non-Maskable Interrupts (NMI)[1] to inform all the cores using *XAPIC* or *X2APIC*. Cores are configured to cause *VM-exit* in the case of NMIs (e.g., PIN-Based VM-Exec controls are set to 1). Consequently, in this mechanism, all the cores in *VMX-root* mode will be spinning and waiting for a new command from the debugger.

Whenever an event is triggered, we check for the actions. If the action is a *custom code* or is a *script*, then HyperDbg executes the action without notifying other processors. If the action is a *break* action, then HyperDbg sends NMIs to all the other cores to halt them in the VMX-root mode.

5 Step-in & step-over

Step-in and step-over are two essential parts of each debugger, and HyperDbg is not an exception; however, the design of the stepping mechanism is different as we are operating in *VMX-root* mode and our stepping mechanism is operating system independent. In this section, we present stepping mechanism in HyperDbg Debugger.

5.1 Stepping Mechanism in HyperDbg

The stepping process is an essential feature in commodity debuggers, often implemented very straightforwardly. It is implemented by a trap flag in the CPU, which permits the operation of a processor in single-step mode. On Intel processors, the trap flag is carried out as *RFLAGS* which generates an exception making the system execute a single instruction and then stop. Hence, following a trap flag in the kernel, the user in the debugger can read/modify the content of the registers as well as the memory map of the system.

However, the conventional kernel stepping method can not handle a guaranteed stepping procedure due to the possible interrupts in the system. In the following section, we outline this problem in detail.

5.2 Monitor Trap Flag (MTF)

Monitor Trap Flag or MTF is a feature provided by Intel that works exactly like the trap flag in *RFLAGS* which is additionally invisible to the guest. By setting this flag on CPU_BASED_VM_EXEC_CONTROL, following a VMRESUME, the processor is forced to execute a single instruction and then performs a VM-exit.

5.3 MTFs Disadvantages

Setting a monitor trap flag does not guarantee that the next instruction is the targeted one. In this case, if the upcoming instruction is a sudden interrupt from the CPU, the next targeted instruction in the debugging program would not be executed since the interrupt handler instructions are executed first. One way to address this issue is to set a *VM-exit* on exceptions (Exception Bitmap) and external-interrupts. However, this resolution is not optimal as it might cause system inconsistency by blocking interrupts. HyperDbg presents the *i command* resolving this issue by an instrumentation stepping process.

5.4 Step-in *t* Command

In the conventional debuggers such as *WinDbg*, *GDB*, etc., the **step-in** command is implemented by using trap flag. In this scenario, the occurrence of any interrupts (for instance, a DIRQL interrupt) makes the processor executes the instructions handled by the interrupt handler and even might (and will) switch to the other threads. This leads to disrupting the single-step procedure in the debugger. A possible scenario is where the line-by-line stepping is drastically altered by an interrupt.

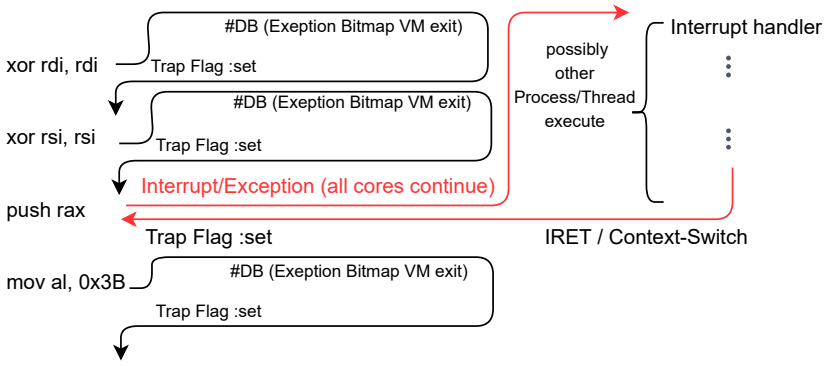


Figure 2: The *t command* Stepping mechanism in HyperDbg

In Figure 2, every instruction is stepped by executing a Debug Breakpoint (#DB) exception which causes a VM-exit. Also, it is noteworthy that all other cores and processes might be executing their routines during this stepping mode. This simple stepping mechanism as in commodity debuggers is also implemented in HyperDbg through *t* command as shown in Figure 2. Nevertheless, one can manually disable interrupts by unsetting Interrupt Flag in *RFLAGS* to ignore all the interrupts triggering toward the system. The implementation of this technique is known to be trivial using a virtualized-assisted debugger. Hence, it is possible to mask all external interrupts on the system. However, this raises a severe issue with intercepting/preventing interrupts where the debugger could easily break the OS semantics. For example, in *Windows*, queuing a self-DPC interrupt, where the *IRQL* is not adjusted, could easily damage the OS normal procedure and consequently a BSOD.

Owing to this, in addition to *t* command, HyperDbg presents *i* command to provide a guaranteed stepping mechanism in debugging routine which will be discussed later on.

The step-in *t* command is used for regular step-in process and it is implemented like other debuggers through *RFLAGS*'s trap flag. By setting exception bitmap to intercept #DB, HyperDbg is notified whenever the guest finished executing the upcoming instruction. Note that the MTF is not used for this purpose. The reason for using trap flag instead of MTF is that the context might be changed in debugging process by using the MTF. For example, it is very probable that the user-mode can be switched to the kernel-mode (because of clock-interrupt). The main reason is that the MTF handler takes so long that the kernel determines the quantum has finished when it returns back (VMETNRY).

Additionally, using the trap flag, even if the target thread execution is switched to a new thread, is assured that Windows restores the *RFLAGS* with the trap flag enabled on the next context-switch. In such a way, when the debug exception happens, the correct target thread is guaranteed to be selected.

Note that, like other conventional debuggers, HyperDbg continues the execution in all the cores (not just one core) after setting the trap flag. In the case of the generation of #DB exception, all the cores are halted again via NMIs. Consequently, at the next step, all the processes and threads are permitted to continue, but the target thread would execute only one instruction, and all the cores are halted again.

5.5 Step-over *p* Command

Step-over mechanism in HyperDbg is very similar to regular step-in. The difference is for the call instruction. In this incident, the debugger sends the length of the call instruction to the debuggee, and instead of setting the trap flag, it sets a Hardware Debug Register to the instruction after the call. Therefore, when the call is finished, the Hardware Debug Register is triggered, and the debugger is notified regarding the next instruction.

It is also necessary to consider that other threads/cores might trigger the Hardware Debug Register since all the threads/cores are continued through stepping, and different interrupts are possible to disturb the targeted instruction stepping. The solution to this problem is to check for the thread/process id to ensure the validity of the target thread supposed to trigger the hardware debug register. In this case, HyperDbg ignores the generated #DB from the debug register and re-sets the debug register for the next chance to get the correct execution context in the target thread. Figure 3 shows the overview of the *p command* stepping mechanism in HyperDbg.

In Figure 3, by inspecting a *call* instruction, a Hardware Breakpoint (HW BP) is triggered for the next instruction.

5.5.1 Hardware Debug Registers in *p* Command

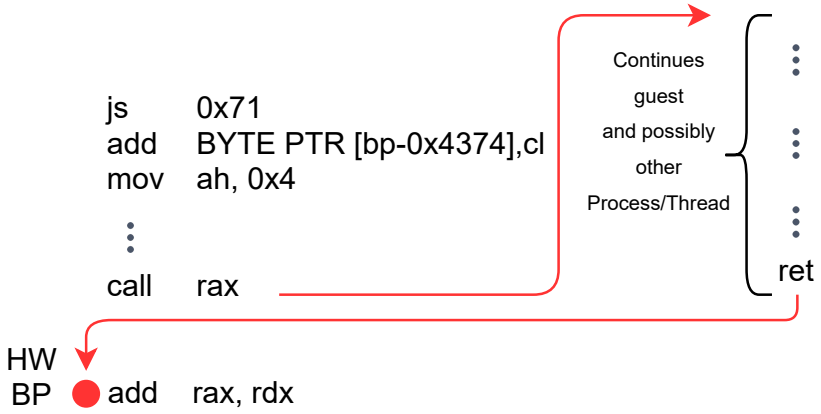


Figure 3: The *p command* Stepping over mechanism in HyperDbg

In order to determine the exact jump over the call instructions, HyperDbg sets a breakpoint via Hardware Debug Register on the instruction following the call command. Hence, the debugger is continued over the call instruction and is broken when it returns back to the main next instruction in the target program. It is also essential to set debug register for upcoming instruction in all the cores' Hardware Debug Registers. This is due to the fact that the currently executing thread might be context switched by the OS and be executed in another processor or other cores for the next time-slot. Considering this approach, it is assured that when the thread is returned from the calling function, the #DB is triggered, notifying HyperDbg about this event.

5.6 Instrumentation Step-in *i* Command

Considering the difficulty caused by the scenario in Figure 2, HyperDbg employs MTF to overcome the disruption by inevitable interrupts. To the best of our knowledge HyperDbg is the first debugger, addressing

this issue by presenting a guaranteed stepping method. As explained, MTF is a feature that works similar to *RFLAGS* but is transparent to the guest. The following listing illustrates the set/unset of MTF in an execution sequence.

```
/* Set the monitor trap flag */
void HvSetMonitorTrapFlag(BOOLEAN Set)
{
 unsigned long CpuBasedVmExecControls = 0;
 // Read the previous flag
 __vmx_vmread(CPU_BASED_VM_EXEC_CONTROL, &CpuBasedVmExecControls);
 if (Set) {
 CpuBasedVmExecControls |= CPU_BASED_MONITOR_TRAP_FLAG;
 }
 else {
 CpuBasedVmExecControls &= ~CPU_BASED_MONITOR_TRAP_FLAG;
 }
 // Set the new value
 __vmx_vmwrite(CPU_BASED_VM_EXEC_CONTROL, CpuBasedVmExecControls);
}
```

Listing 1: MTF Set/Unset in an example execution sequence

By executing each instruction, it is ensured that the specific line of code is passed to the CPU. In order to get the execution after executing an instruction, a *VM-exit* is triggered by setting an MTF which guarantees that only one succeeding instruction will be executed in the guest. In order to do so, HyperDbg continues at only one core and disables interrupts in the same core (ignoring external-interrupts by setting external-interrupts exiting bit in VMCS) to offer a fine-grained stepping. Figure 4 depicts the general procedure in

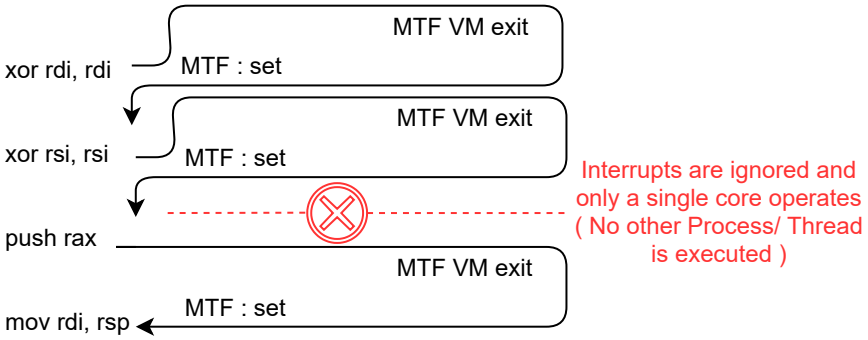


Figure 4: The *i command* Instrumentation Stepping Approach in HyperDbg

the *i command* step-in. Note that all the interrupts are ignored during this mode of stepping.

Using this method allows the user to run instructions from user-mode to kernel-mode and from kernel-mode to user-mode, which was not possible through previously available debuggers. For instance, whenever the user-mode executes a *SYSCALL* instruction, HyperDbg flow lets the debugger follow the instructions directly into the kernel, executing the next instruction from the kernel-mode SYSCALL handler. Similarly, if a page-fault occurs in the middle of a user-mode application, the debugger is moved into the kernel-mode page-fault handler. On the other hand, kernel-mode to user-mode migration is also handled through HyperDbg *i* stepping. (e.g., executing a *SYSRET* or an *IRET* returns the debugger to user-mode from kernel-mode.)

6 Processor/Execution Mode Switch

In this section, we describe the architecture relating to switches between processes, processors, and different modes of execution in HyperDbg.

6.1 Detecting execution mode changes (kernel-mode to user-mode)

The kernel-mode to user-mode (or the opposite) real-time switch detection could be very useful in debugging procedures. Detecting changes to the operating mode is performed via the same mechanism used in the *i* command in HyperDbg. The correct way is to check the CS register, fetch GDT, and check the Long Mode flag. However, since the CS for *wow64* and native code is set to a constant value across all windows versions, the CS register check is sufficient for the determination.

6.2 Switching to new processor

HyperDbg uses a straightforward mechanism to switch among cores. Each core has its own spinlock to wait on VMX-root mode. By unlocking the spinlock related to the new core and setting the spinlock of the current core, it is possible to enter a waiting state. Consequently, HyperDbg calls the command handler from the new core, and the new core is responsible for getting commands. Note that HyperDbg is designed to have a single core to get commands simultaneously.

7 Memory Access in VMX-root mode

One of the challenging parts of designing HyperDbg is safe memory access due to its architectural features. A safe memory access at the hypervisor-level raises numerous considerations which could not be implemented by simple *Mov* instructions. For example, if the debugging user accesses a user-mode memory from the VMX-root mode (even if the VMX-root mode is more privileged in the design of Intel), it most likely ends up with a system halt or an exception. Another possible issue might arise when the accessed page is paged-out. This is due to the fact that paging in is not possible in the VMX-root, which leads to a system halt if an already paged-out entity is accessed.

Addressing these considerations accompanied by other restrictions in the kernel level, targeted by HyperDbg, complicates the memory access architecture. This section presents our approaches for a secure and efficient memory access in HyperDbg.

7.1 Discovering page-table entries

The conventional method to detect whether a page is valid or not is to find the corresponding page table to the virtual address. If a valid page-table entry for the target address is found, then the address is valid. It is also necessary to double-check the page table entry's **present** bit to ensure that the target page is not paged-out and is safe to access.

The method described above is perfectly implementable and functional. However, this approach is not the best option if it's used in the user-mode, which requires invoking an IOCTL to check for the addresses each time, traversing throughout all the page tables, which makes the process time-consuming. To overcome this issue, we propose the use of *Intel's Transnational Synchronization Extension (TSX)* in the HyperDbg to minimize the overhead.

7.2 Using TSX for Page Table Detection

Intel Transactions Synchronization eXtensions (TSX) provides two x86 instruction set extensions, namely *Hardware Lock Elision (HLE)* and *Restricted Transactional Memory(RTM)*.

TSX as an essential extension used in many adversarial side-channel attacks[365] , gives a powerful and precise timing mechanism to exploit microarchitectural flows. TSX could be employed to handle exceptions at the user-level in the case of transaction failure. It means that if the transaction is failed, then no exception will occur, which means that the execution will not be passed to the kernel and remains in the user-mode.

By applying the same method in the VMX-root mode, HyperDbg exploits the TSX to check the validity of the memory address. Via a TSX section, the address is only valid if the transaction is completed without error. However, in the case where the transaction is aborted, the debugger detects the address is invalid. This method is substantially faster as the address validity check in the user-mode (not kernel-mode or VMX-root mode) could be carried out by a couple of instructions.

It is also worthy to note that not all the processors support Intel TSX; thus, HyperDbg initially checks for the processor support of TSX. If the processor supports TSX, then address validity in the user-mode is performed by TSX; otherwise, the first method (check page-table entries) is used.

The following listing shows how TSX transactions could be used to detect the validity of the address.

```
XBEGIN  $+xxx
; Using Intel TSX in order to suppress any
; page-fault in VMX-root mode


MOV RAX, Dword PTR:[RCX]
; Access the target memory address,
; if the address is invalid, then
; transaction fails; otherwise,
; the transaction is successful

XEND    ; End of TSX

; Transaction was successful
MOV RAX, 1
JMP Return

; Transaction failed
MOV RAX, 0

Return :
RETN    ; Return the result
```

Listing 2: Using Intel TSX to detect address validity

7.3 Injecting #PF to the debuggee

Contingent upon the address is not present; the debugger injects a page-fault to the debuggee in order to request VMX non-root to bring the page back from the hard disk to the RAM. Injecting page-faults consists of configuring the *cr2* register to the target virtual address, which needs to be available after handling the page-fault.

Such a procedure is unsuitable for some debugging scenarios since continuing debuggee might lead to losing the context if the targeted event cannot be triggered another time. Moreover, if the debuggee is in *DISPATCH_LEVEL IRQL*, page-faults would not work as paging is not realizable at $IRQL = DISPATCH\_LEVEL$.

Nevertheless, injecting Page Fault exceptions is possible in some useful scenarios. For instance, the debugger might check whether the SYSCALL or SYSRET instruction is located at *GUEST_RIP* by executing HyperDbg's *!syscall* or *!sysret* commands. In this scenario, the IRQL level at PASSIVE_LEVEL is assured if the guest tends to execute SYSCALL or SYSRET. Furthermore, if RIP is not incremented in this case, the guest requires to re-execute the instruction caused #UD. Hence, HyperDbg can safely inject a #PF and continue the debuggee. Then the debuggee executes the page-fault handler and re-generates the previous event.

7.4 VMX-root mode Compatible Message Tracing

Sending a message from VMX-root mode to VMX non-root mode is undoubtedly one of the most complex parts of the hypervisor design. This is mainly due to many limitations in accessing non-paged buffers. More importantly, most NT functions are not ANY IRQL compatible as they might access the buffers that

Figure 5: Caption

reside in paged pool memory. HyperDbg utilizes its custom VMX-root mode compatible message tracing mechanism to send the commands and messages from VMX-root mode to the user-mode application or the debugger in a safe manner.

7.5 Reading and writing memory

As mentioned earlier, due to the unsafety with regards to direct memory access, HyperDbg is designed not to access the memory directly. Instead, by reserving a Page Table Entry (PTE), whenever a memory access is required from VMX-root mode, HyperDbg finds the physical address of the target page and maps it to the reserved address (which is in the kernel-mode) using its Page Table Entry (PTE). By doing so, the memory is safe to be accessed. We have utilized the same approach for memory writes as well. Given a memory address to write, HyperDbg initially ensures that the address is valid. Similar to the read access, PTE eliminates the need to check whether the target address is writable or read-only since the target page is accessed through the reserved virtual address, and HyperDbg guarantees to set its writable bit in the PTE.

7.6 Pre-allocated pools

Despite the fact that most of the HyperDbg's routines are operating in VMX-root mode, it is not generally possible to allocate memory in VMX-root mode. Nevertheless, allocating memory in the VMX-root mode is unavoidable. For this purpose, we propose to use pre-allocated pools. Pre-allocated pools are allocated when the debuggee is operating in VMX non-root. We choose the driver's *IOCTL* handler, which is a safe place in the VMX non-root mode and operates in *PASSIVE_LEVEL*. In the memory manager routines, HyperDbg checks whether it has to allocate any pools. Here, new pre-allocated pools are either required to be replaced with previously allocated pools or be de-allocated for the pools that are not in use anymore.

As an example, HyperDbg maps all the pages in 2 MB granularity in EPT tables. Supposing the user intends to set a hook for a particular page, a 4 KB granularity is used. Here, to convert 2 MB size to 4 KB granularity, we employ an extra 4 KB page table to set new entries in VMX-root mode. This is one of the cases in that pre-allocated pools are taken into account. Pre-allocated pools are extensively used in many other mechanisms in HyperDbg.

7.7 Unsafe behavior

The terms "**safe**" and "**unsafe**" is used extensively with regard to the usage of HyperDbg. By "**safe**", we mean something that works all the time and will not cause a system crash or system halt. The reason for these considerations is the hindrance to manage codes in VMX-root mode. As HyperDbg gives the ability to run custom assembly codes in all execution modes, users should avoid doing "**unsafe**" behavior leading to system instability. In the VMX-root mode, interrupts are masked (disabled) and paging is disabled. So, transferring buffer from VMX-root mode to VMX non-root mode requires extra effort, and the user should be cautious to avoid executing APIs to ensure the system's safety. Yet HyperDbg provides a safe way to access the non-paged pool in the user, kernel, and VMX-root mode. Also, it sends the buffer to the user mode in a safe manner.

7.8 VMX-root mode compatible script-engine

HyperDbg uses a custom-designed script-engine. The script-engine is designed to work on VMX-root mode. It uses a MASM-Style language, combined with C keywords (if, else, for, etc.).

We designed everything from scratch, like basic operating system spinlock, memory checks, and even functions like **printf** and **strlen** as it's not possible to access memory directly in the VMX-root mode. Other script engine solutions are not working in the case of VMX-root mode.

In the script-engine, LL(1) and LALR(1) parsers are used to reach the best possible performance. The grammar of the script-engine can be customized.

In the script-engine, all the accesses to the virtual memory are checked to be safe, and accesses to registers are applied to the dumped registers and those registers that are located on each core's VMCS.

8 Challenges and Further Considerations

In this section, we discuss important challenges and other miscellaneous considerations in HyperDbg.

8.1 Getting debugging events: #BPs and #DBs

HyperDbg uses the exception bitmap of VMCS to get notified of breakpoints (#BP) and Debug Breakpoints (#DB) in order to halt the other cores. HyperDbg is the first debugger capable of being notified about the debugging event, which means that HyperDbg is notified, even earlier than the operating system. Consequently, we design the system not to notify user-mode application or kernel-mode (OS) entities regarding the debugging events. So, all the breakpoints events are handled by HyperDbg and no other debuggers will be notified.

8.2 Spinning on spinlocks

Spinning the cores in HyperDbg is considered as a primary technique in its functionalities. We study the challenges in this context. Suppose a function requires a spinlock (e.g., it is merely a buffer which is to be accessed) in a single-core processor. The function raises the IRQL to *DISPATCH_LEVEL*. Here, the Windows Scheduler can not interrupt the function until it releases the spinlock and lowers the IRQL to *PASSIVE_LEVEL* or *APC_LEVEL*. If during the execution of the function, a VM-exit occurs, the operation mode is moved into the VMX-root. (It can be interpreted that VM-exit happens similar to a HIGH_IRQL interrupt.)

Now, what if the user requires to access the buffer in the VMX-root mode? Two scenarios are possible:

- The first scenario is to wait on a spinlock that was previously acquired by a thread in the VMX non-root mode. In such a scenario, a deadlock occurs and spins forever.
- Alternatively, it is also possible to enter the function without looking at the lock (while another thread enters the function simultaneously). So it results in a corrupted buffer and invalid data. The other limitation is in Windows. In Windows, cores must not wait on a spinlock when IRQL is higher than *DISPATCH_LEVEL*. This lies in the fact that Windows raises the IRQL to 2 (*DISPATCH_LEVEL*) when a spinlock is acquired. In this case, Windows performs the workload, releases the spinlock, and lowers IRQL back afterward.

Looking at corresponding windows functions such as *KeAcquireSpinLock* and *KeReleaseSpinLock*, the IRQL arguments are given as input. Windows, in its procedure, saves current IRQL to the parameter supplied by the user in *KeAcquireSpinLock*. Then it raises the IRQL to *DISPATCH_LEVEL*. After the function is finished with the shared data, it calls *KeReleaseSpinLock* and passes the old IRQL parameter to the function. Finally, it unsets the bit and restores the old IRQL (lowering the IRQL).

Unfortunately, Windows spinlocks employ IRQLs, which do not make sense when VMX-root mode is in action. This makes it very complicated to use such functions in this mode. Hence, in order to implement spinlock for HyperDbg functionalities such as multi-core message tracing, we design a custom VMX-root compatible spinlock.

8.3 Continuation a single core

One of the exclusive features of HyperDbg is to keep execution (continuation) on one core while other cores are in a halt-state. We used this mechanism in our instrumentation step-in command to guarantee that no other cores (threads) get the chance to be executed. This mechanism's fundamental basis is ensuring that the target core is not interrupted during the debugging.

There are two approaches in which HyperDbg prevents target cores from being interrupted (e.g., by clock-interrupt or keyboard interrupts).

- First, we can unset the guest's *RFLAGS.IF* bit, so the interrupts are masked.
- Second, we can set the *PIN Based External-Interrupt Exiting bit* so all of the external interrupts cause VM-exits; thus, any interrupts could be simply ignored in the VM-exit handler.

The first method is faster and avoids unnecessary VM-exits. However, for several considerations described in the following, the second method is preferred in HyperDbg.

Notwithstanding the approach used in the first methodology, it is much safer not to change the guest's registers. As an example, if a page-fault happens or in the case of a *SYSCALL* or invalid operation such as division-by-zero, the execution is directed to the kernel, and guest's *RFLAGS* are saved by the processor. So, extra tasks are needed to locate the user-mode RFLAGS (search in the stack for exceptions and in *R11 Register* for *SYSCALLs*) because the *RFLAGS* that was previously saved in user-mode is with IF bit disabled. If this specific task is ignored, *RFLAGS* are restored without the check for IF bit, every time the guest continues and performs a context switch. In this case, the core becomes uninterruptible by unsetting this bit from the hypervisor as the OS cannot get the execution again (e.g., using clock interrupt). Consequently, after a delayed bug check, Windows realized the target core behaves abnormally and returns an error. Moreover, changing guest's *RFLAGS* is also incompatible with instructions like *CLI* and *STI*. More importantly, considering the side effects, the guest is able to detect the tampering of HyperDbg by using *PUSHF* function and check for *IF* bit in RFLAGS.

All of the investigated issues with regards to *RFLAGS* changing, along with the fact that using PIN-Based External-Interrupt Exiting bit is entirely transparent from the kernel-mode and user-mode, lead us to employ the second method in HyperDbg.

8.4 HyperDbg & Meltdown Mitigation

Meltdown vulnerability[6] and its corresponding attacks affect hypervisors drastically. In HyperDbg, the system kernel process's *cr3* is used as *HOST_CR3*. Experimentally, if other process's *cr3* is used as *HOST_CR3*, some of the functions would not be mapped correctly, causing system-halt and crash the entire system. After Meltdown's mitigation (a.k.a. KPTI or KAISER), user-mode *cr3* is differed from the kernel-mode *cr3* by the usage of a shadow-like memory layout. Consequently, it is no longer possible to use *GUEST_CR3* to access guest virtual memory. In this context, if we change the current *cr3* register (which is loaded with *HOST_CR3*) to the *GUEST_CR3* (which is the target process's *cr3*), system would crash. This is attributed to the fact that *GUEST_CR3* does not contain the HyperDbg 's routines which are mapped into the kernel-side of the debuggee if the VM-exit is caused by a user-mode instruction.

To solve these problems, we utilize Windows's *_EPROCESS* structure to find the target process's kernel *cr3*. In Windows, there is no function to export either user-mode or kernel-mode *cr3*. However, kernel-mode *cr3* is known to be at the static location from the top of *_EPROCESS* in all of the recent versions of Windows, which is exploited here.

In order to read or access the debuggee's memory, HyperDbg changes the current *cr3* to the target process's kernel *cr3*. Then it accesses those memory locations and finally restores the *HOST_CR3* again.

We successfully tested the approach for both KPTI-enabled and disabled machines.

8.5 Future Improvements

8.5.1 Events in the VMX-root mode

In the current versions of HyperDbg, some of the commands (especially events) cannot be applied immediately in Debugger Mode. Storing events in the memory consist of multiple memory allocations with dynamic sizes. In HyperDbg we use pre-allocated memory to address this problem. However, for dynamic sizes, we prefer not to allocate a huge page with only a portion used for the event. As a result, applying events directly from VMX-root mode is impossible. Also, other functions are used to sanitize the parameters for events that cannot be used in VMX-root mode as those functions are not *ANY IRQL* compatible. This means that commands go through the routines from user-mode to kernel-mode and then VMX-root mode. Thus, the

debuggee is continued for some time, and then the debuggee is halted again. It is clear that the user would lose the current context (registers and memory), and the target process (which is under debugging) will get the chance to continue its normal execution. It is important to note that whenever the user creates an event, HyperDbg continues the debuggee. In this incident, all the other running events (active events) are ignored until the current event is successfully applied. Hence, HyperDbg might ignore some of the events during this process.

This issue should be considered by the user when the events are triggered. We plan to improve the structural design of the events in HyperDbg for the upcoming updates to overcome these difficulties.

8.5.2 HyperDbg over Linux OS

HyperDbg's nature is OS-independent as it is implemented on top of the hypervisor and very near to the hardware system. Almost all of the features proposed by HyperDbg are armed with hardware technologies. So, the similar implantation presented here for Microsoft Windows can also be utilized for Linux. However, minor detailed modifications are to be contemplated. As the future works, we plan to implement a compatible HyperDbg for Linux systems.

References

[1] Control over nmis. `https://docs.hyperdbg.com/design/features/vmm-module/control-over-nmis`. Accessed: 2021-01-08.

[2] Intel Corporporation. Intel 64 and ia-32 architectures software developer manuals, 2018.

[3] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems*, pages 161–176. Springer, 2017.

[4] Mohammad Sina Karvandi, MohammadHosein Gholamrezaei, Saleh Khalaj Monfared, Soroush Medi, Behrooz Abbassi, Ali Amini, Reza Mortazavi, Saeid Gorgin, Dara Rahmati, and Michael Schwarz. Hyperdbg: Reinventing hardware-assisted debugging. 2022.

[5] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[6] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.