

# MBEC, SLAT, and HyperDbg

Hypervisor-Based Kernel- and User-Mode Debugging

Björn Ruytenberg, Sina Karvandi

FOSDEM 2026 - Brussels, Belgium  
January 31, 2026



# Who We Are

Björn Ruytenberg

@0Xiphorus@infosec.exchange

- PhD Candidate @ Vrije Universiteit Amsterdam (VUSec)
- Security Researcher, HyperDbg developer
- x86-64 UEFI, hypervisor and PCI Express security
- Previous work: Intel Thunderbolt vulnerability research ([thunderspy.io](https://thunderspy.io))
- More info: [bjornweb.nl](https://bjornweb.nl)

Sina Karvandi

@rayanfam@infosec.exchange

- PhD Candidate @ Vrije Universiteit Amsterdam (VUSec)
- System Programmer, HyperDbg developer
- Windows internals, hypervisor, digital hardware design
- Blog: [rayanfam.com](https://rayanfam.com)



# HyperDbg at FOSDEM '26


- **Invisible Hypervisors: Stealthy Malware Analysis with HyperDbg**  
Security track, 13:00, UB5.132
- **MBEC, SLAT, and HyperDbg: Hypervisor-Based Kernel- and User-Mode Debugging**  
Virtualization and Cloud Infrastructure track, 18:30, H.2213 (this talk)



01

# Introduction

An introduction to HyperDbg debugger



“If debugging is the process of removing software bugs, then programming must be the process of putting them in.”

— **Edsger Dijkstra**

# Why a hypervisor-based debugger?

01

## Highly Privileged

System-wide visibility enables having access to nearly all events occurring within OS

02

## User/OS Transparency

User-mode applications and the OS run independently of the hypervisor

03

## Stealthiness

Hypervisors have relatively less artifacts, making them a better choice for malware reverse engineering and debugging

04

## New Techniques

And of course, hypervisors unlock many innovative techniques

# HyperDbg Debugger

- FOSS (GPLv3) hypervisor-assisted debugger
- Leverages hardware virtualization controls to deliver advanced debugging capabilities (e.g., EPT-based memory monitoring, system call interception, PMIO/MMIO debugging)
- Operates independently of OS-level debugging APIs, providing higher transparency than traditional debuggers
- First released for Windows (2022), actively maintained since
  - UEFI-based, OS-agnostic hypervisor agent scheduled on roadmap



Get the source code:

[github.com/HyperDbg/HyperDbg](https://github.com/HyperDbg/HyperDbg)



02

# Scenario

Real-world debugging difficulties



**Debugging Native  
Code is Difficult!**

# Why HyperDbg?

- What if you want to...
  - Find out *what* device driver writes into memory range x - y?
  - Find out *when* that device driver does it, and reverse trace the call stack up to user space?
  - Quickly write a script that triggers on the former events, without freezing up the kernel?
  - Prevent user or kernel space from modifying memory ranges, without the former noticing?
  - Reverse device drivers and user space companion apps in one go?
  - Trigger debug events when user or kernel space executes \*MSR/CPUID/RDTSC, then trace back its source?
  - ...or do anything else a hypervisor can do!

A decorative graphic on the left side of the slide consists of two overlapping squares. The bottom-left square is a dark blue, and the top-right square is a lighter blue, creating a cross-like shape.

**Hypervisor-based  
debuggers to  
rescue!**



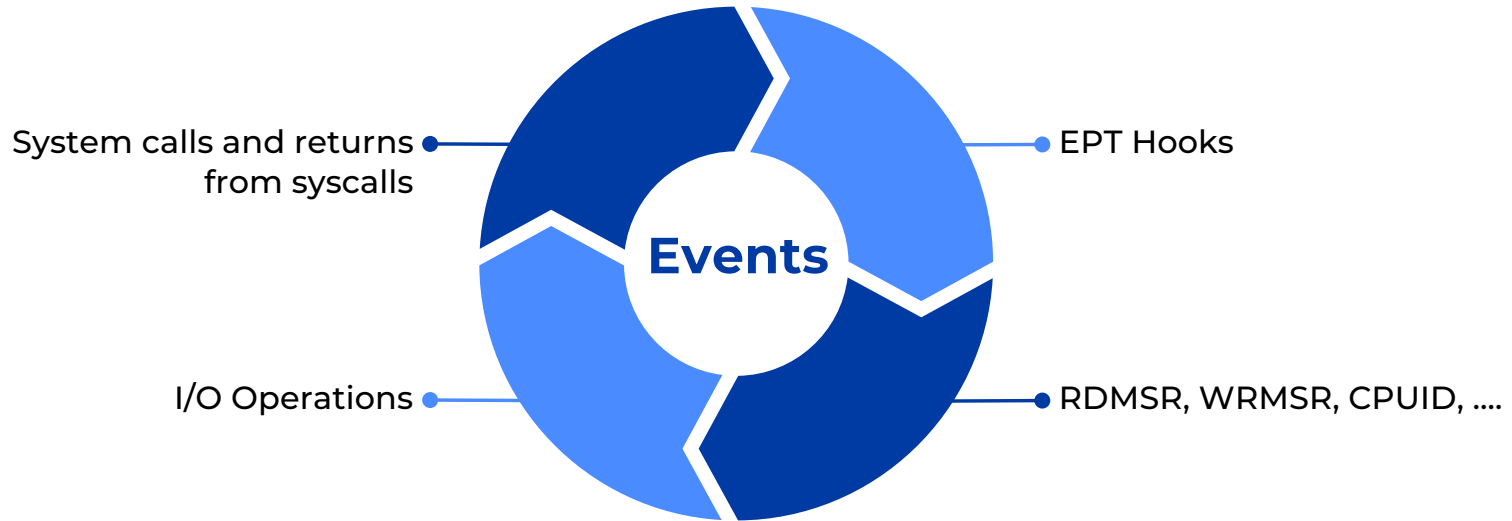
03

# Terms

New debugging terminologies in HyperDbg

# Event-driven Debugging

- Everything in HyperDbg is an **event**
- Each event could trigger one or more **actions** (*script, assembly code, break into the debugger*)



...and many more at <https://docs.hyperdbg.org/design/debugger-internals/events>

# New Debugging Terms

Event Cond.

## Event Calling Stages

- Determine when actions should be triggered
- Types: '*post*', '*pre*', and '*all*'

## Event Short Circuiting

- Ignore or modify event execution
- Bypass events (instructions or system calls)

These abstractions are based on determining whether instruction emulation is required or should be avoided when handling **VM exits**



04

# Techniques

An overview of the implementation of these techniques in HyperDbg

# Tracking Function Calls

- Uses **Monitor Trap Flag (MTF)**
- Tracks user to kernel transitions
- Useful with symbol servers

```
C:\Users\jina\Desktop\Hyper... x + v - □ ×
3: kHyperDbg> !track
ntkrnlmp!IopCreateFile (fffff802'5abc2650)
├─ ntkrnlmp!RtlpInterLockedPopEntrySList (fffff802'5a83b2b0)
├─ ntkrnlmp!ExpInterLockedPopEntrySListEnd+0xb (fffff802'5a83b2db)
├─ ntkrnlmp!PsGetCurrentSilo (fffff802'5a1ea20)
├─ ntkrnlmp!PsGetCurrentSilo+0x29 (fffff802'5a1ea49)
├─ ntkrnlmp!ObOpenObjectByNameEx (fffff802'5aace3c0)
├─ ntkrnlmp!RtlpInterLockedPopEntrySList (fffff802'5a83b2b0)
├─ ntkrnlmp!ExpInterLockedPopEntrySListEnd+0xb (fffff802'5a83b2db)
├─ ntkrnlmp!ObpCaptureObjectCreateInformation (fffff802'5aad02f0)
├─ ntkrnlmp!ObpCaptureObjectName (fffff802'5aad0580)
├─ ntkrnlmp!RtlpInterLockedPopEntrySList (fffff802'5a83b2b0)
├─ ntkrnlmp!ExpInterLockedPopEntrySListEnd+0xb (fffff802'5a83b2db)
├─ ntkrnlmp!memmove (fffff802'5a847240)
├─ ntkrnlmp!memmove+0x169 (fffff802'5a8473a9)
├─ ntkrnlmp!ObpCaptureObject+0x274 (fffff802'5aad07f4)
├─ ntkrnlmp!ObpCaptureObjectCreateInformation+0x208 (fffff802'5aad04f8)
├─ ntkrnlmp!PsReferencePrimaryTokenWithTag (fffff802'5a622eb0)
├─ ntkrnlmp!PsReferencePrimaryTokenWithTag+0x68 (fffff802'5a622f18)
├─ ntkrnlmp!SepCreateAccessStateFromSubjectContext (fffff802'5a623030)
├─ ntkrnlmp!memset (fffff802'5a847500)
├─ ntkrnlmp!memset+0x8a (fffff802'5a8475ca)
├─ ntkrnlmp!memset (fffff802'5a847500)
├─ ntkrnlmp!memset+0x8a (fffff802'5a8475ca)
├─ ntkrnlmp!MmAccessFault (fffff802'5a658450)
├─ ntkrnlmp!MmUserFault (fffff802'5a658970)
├─ ntkrnlmp!MiResolvePageTablePage (fffff802'5a6592b0)
├─ ntkrnlmp!MiFastLockLeafPageTable (fffff802'5a6594f0)
├─ ntkrnlmp!MiFastLockLeafPageTable+0x3d2 (fffff802'5a6598c2)
├─ ntkrnlmp!MiLockPageTableInternal (fffff802'5a659970)
├─ ntkrnlmp!MiLockPageTableInternal+0x256 (fffff802'5a659bc6)
├─ ntkrnlmp!MiPteInShadowRange (fffff802'5a642e00)
├─ ntkrnlmp!MiPteInShadowRange+0x21 (fffff802'5a642e21)
├─ ntkrnlmp!MiLockPageTableInternal (fffff802'5a659970)
├─ ntkrnlmp!MiLockPageTableInternal+0xf2 (fffff802'5a659a62)
├─ ntkrnlmp!MiUnLockPageTableInternal (fffff802'5a6bd260)
├─ ntkrnlmp!MiUnLockPageTableInternal+0x95 (fffff802'5a6bd2f5)
├─ ntkrnlmp!MiPteInShadowRange (fffff802'5a642e00)
├─ ntkrnlmp!MiPteInShadowRange+0x25 (fffff802'5a6d2e25)
├─ ntkrnlmp!MiIsPdeOrAboveAccessible (fffff802'5a6fc4c0)
├─ ntkrnlmp!MI_READ_PTE_LOCK_FREE (fffff802'5a642d90)
├─ ntkrnlmp!MI_READ_PTE_LOCK_FREE+0x21 (fffff802'5a642db1)
├─ ntkrnlmp!MiIsPdeOrAboveAccessible+0x21 (fffff802'5a6fc4e1)
├─ ntkrnlmp!MiInPagePageTable (fffff802'5a671f10)
├─ ntkrnlmp!memset (fffff802'5a847540)
├─ ntkrnlmp!memset+0x8a (fffff802'5a8475ca)
├─ ntkrnlmp!memset (fffff802'5a847540)
├─ ntkrnlmp!memset+0x8a (fffff802'5a8475ca)
├─ ntkrnlmp!MI_READ_PTE_LOCK_FREE (fffff802'5a642d90)
├─ ntkrnlmp!MI_READ_PTE_LOCK_FREE+0x21 (fffff802'5a642db1)
├─ ntkrnlmp!MiGetLeafVa (fffff802'5a63b470)
```

**Combining event  
calling stages with  
event short circuiting**



# Real-world Reversing Techniques

- **Bypassing privileged instructions/events**
- Changing system call assumptions
- Ignoring memory writes

```
!msrwrite 0xc000084 stage pre script {
    @eax = @eax | (1 << 9);
}
-----
!exception 0xe stage post script {
    printf("page-fault happens at: %llx", @cr2);
}
```

# Real-world Reversing Techniques

- **Bypassing privileged instructions/events**
- Changing system call assumptions
- Ignoring memory writes

*Calling stages*

```
!msrwrite 0xc000084 (stage pre) script {  
    @eax = @eax | (1 << 9);  
}  
-----  
!exception 0xe (stage post) script {  
    printf("page-fault happens at: %llx", @cr2);  
}
```

# Real-world Reversing Techniques

- Bypassing privileged instructions/events
- **Changing system call assumptions**
- Ignoring memory writes

```
!syscall stage post script {  
    if (@eax == 0x55) {  
        @ecx = @ecx & ~(1 << 5);  
    }  
}
```

```
-----  
!syscall stage pre script {  
    if (@eax == 0x55 && @ecx == 0x1) {  
        @rax = 0xc0000005;  
        event_sc(1);  
    }  
}
```

# Real-world Reversing Techniques

- Bypassing privileged instructions/events
- **Changing system call assumptions**
- Ignoring memory writes

```
!syscall stage post script {  
    if (@eax == 0x55) {  
        @ecx = @ecx & ~(1 << 5);  
    }  
}
```

```
-----  
!syscall stage pre script {  
    if (@eax == 0x55 && @ecx == 0x1) {  
        @rax = 0xc0000005;  
        event_sc(1);  
    }  
}
```

*Short-circuiting*



# Real-world Reversing Techniques

- Bypassing privileged instructions/events
- Changing system call assumptions
- **Ignoring memory writes**

```
!monitor w ff71f118210 l 4 stage all script {
  if ($event_stage == 1) {
    curr_memory = dq($context);
    eq($context, 0xdeadbeef0badcafe);
    printf("current memory ignored: %llx\n", curr_memory);
  } else {
    prev_memory = dq($context);
    printf("thread id: %x , from (RIP): %llx modified address:
%llx, previous memory: %llx", $tid, @rip, $context, rev_memory);
  }
}
```

# Real-world Reversing Techniques

- Bypassing privileged instructions/events
- Changing system call assumptions
- **Ignoring memory writes**

```
!monitor w ff71f118210 l 4 stage all script {  
  if ($sevent_stage == 1) {  
    curr_memory = dq($context);  
    eq($context, 0xdeadbeef0badcafe);  
    printf("current memory ignored: %llx\n", curr_memory);  
  } else {  
    prev_memory = dq($context);  
    printf("thread id: %x , from (RIP): %llx modified address:  
%llx, previous memory: %llx", $tid, @rip, $context, rev_memory);  
  }  
}
```

*Stage indicator*

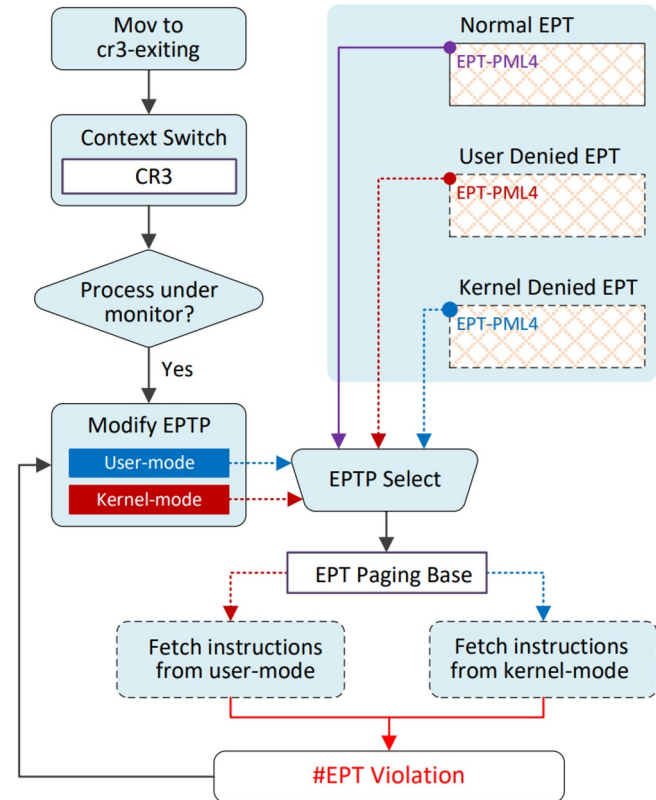
*Memory modification*

# What makes this possible?

*How does virtualization enable fine-grained user-mode debugging?*

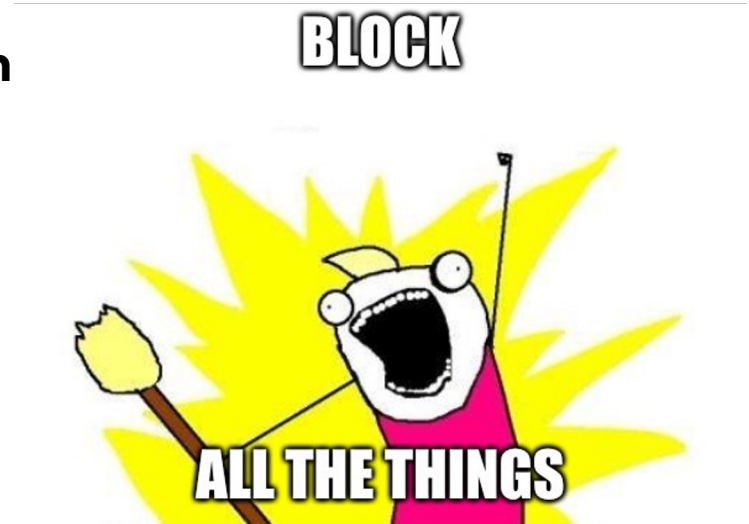
# Detecting Mode Changes

- **User mode** <-> **Kernel mode** change detection
- Uses **MBEC** and page table tricks
- Combining it with **mov-to-cr3** exiting creates a fine-grained number of **VM exits** (better performance)



# Time Freeze Debugging

- Blocks execution of selected modes
- Intercept when a **context switch** happens
- Make user mode execution impossible (thanks to **MBEC**)
- Ignoring tons of **VM exits** (**#EPT violations**)
- At some point, a new context switch happens



# Conclusion

- HyperDbg is a hypervisor-assisted debugger designed for both user-mode and kernel-mode applications
- Leverages modern hardware technologies to introduce system-wide visibility
- Offers powerful debugging features not available on traditional debuggers
- HyperDbg is FOSS (GPLv3), under active development, and available for the community to contribute to and enhance – patches welcome!

# Thanks

**Björn Ruytenberg**

 [@0Xiphorus@infosec.exchange](mailto:@0Xiphorus@infosec.exchange)

 <https://bjornweb.nl>

**Mohammad Sina Karvandi**

 [@rayanfam@infosec.exchange](mailto:@rayanfam@infosec.exchange)

 <https://rayanfam.com>



**Get the source code:**

[github.com/HyperDbg/HyperDbg](https://github.com/HyperDbg/HyperDbg)

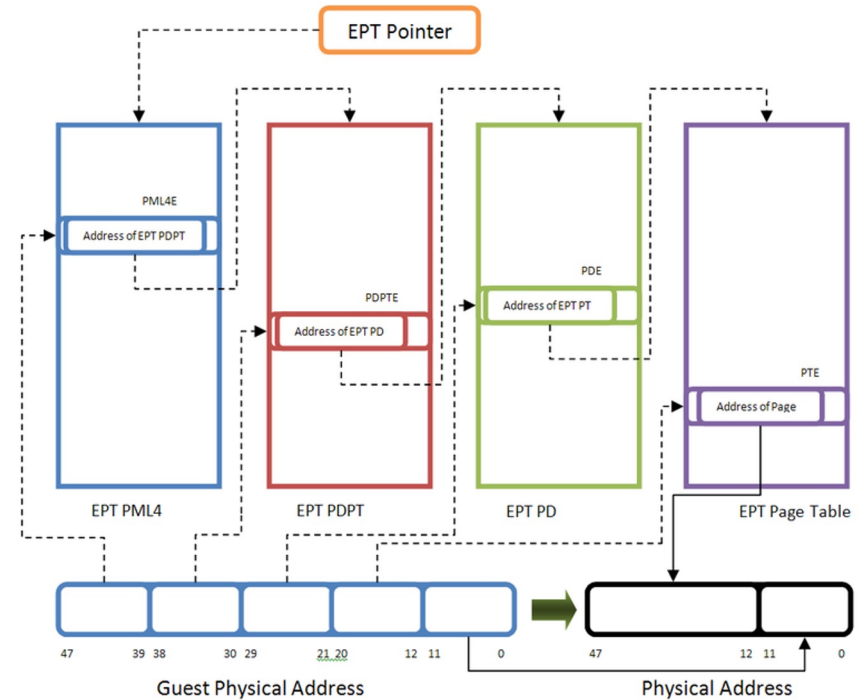
# 05

## Additional Slides



# Second Level Address Translation (SLAT)

- Hardware-assisted memory virtualization (e.g., EPT on Intel)
- Translates guest virtual memory to host physical memory
- Enables fine-grained memory monitoring and access control
- Supports execute, read, and write trapping via hardware faults



# Mode-Based Execution Control (MBEC)

- Hardware feature for execute-permission control at page granularity (MBEC support available on Intel Kaby Lake and up)
- Allows separate execute permissions for user-mode and kernel-mode
- Enables stealthy execution traps via VM exits
- Critical for transparent, artifact-free breakpoint enforcement and time freezing debugging techniques

# Freezing Modules & Applications

- Halts execution without OS awareness
- Bypasses thread-monitoring tricks
- Highly stealthy debugging

# Running Process Without Debug Flag

- Avoids DEBUG\_PROCESS flag
- Improves transparency

