Detecting Hypervisor-assisted Hooking

</> momo5502.com/blog/

I recently started to experiment with hypervisors and their use for bypassing anti-cheat or anti-tampering systems.

This post will describe the concept of hypervisor-assisted hooking and a few simple approaches to detect such hooks.

What is a hypervisor?

In short: A hypervisor allows to run virtual machines with hardware acceleration.

The concept of hypervisors in general is a huge topic, but for this post, all that depth doesn't really matter.

The way virtualization is done using a hypervisor is by abstracting certain critical components. For example privileged instructions or memory.

This allows uncritical instructions to run on real hardware, while privileged operations can be intercepted and virtualized.

How is memory virtualized?

In modern hypervisor technologies, for example Intel VT-X or AMD-V, memory is virtualized by adding "Second Level Address Translation" (SLAT).

When resolving a virtual memory address to physical space in ram, the translation is not only done once by the kernel, but a second time by the hypervisor. That way, memory of virtual machines can be controlled by the host.

Intel calls this functionality "Extended Page Tables" (EPT), for AMD this is called "Nested Page Tables" (NPT).

This technology, however, is not limited to running virtual machines, because the CPU does not distinguish between host OS and virtualized OS. The hypervisor decides that. Therefore all kind of memory can be virtualized, including memory of the host OS.

How does hypervisor-assisted hooking work?

The idea of hypervisor-assisted hooking is to redirect code execution from a virtual page to a different physical page than reads or writes to the same virtual page.

I'm going to focus on Intel's EPT technology. A description of how this works for AMD can be found here. However, due to limitations, the technique is by far not as powerful on AMD.

I'm not going to describe how address translation works on x86 in detail. If you want a good explanation, I can recommend reading this article.

However, in short, address translation is done using tables that describe where virtual memory pages reside in physical memory.

Those pages can have certain permissions. They can be executable, readable or writable. Or any combination of those three (on AMD a page can never be executable without also being readable, this is the limitation mentioned above).

If a page is readable, but not executable, any kind of execution will trigger an exception. However, with EPT, this address translation happens a second time in the hypervisor. The permissions also need to be valid there. Meaning if a page is readable and executable in the first level of translation, but not in the second level, the one done using EPT, a "VM exit" will be triggered upon execution.

This will exit the VM and pass execution to a handler that was previously registered by the hypervisor. This handler can then decide how to proceed with this error.

To abuse this behaviour to install stealth hooks, we first need to clone the physical page. That way, there will be an original page and a fake page, that will contain all our modifications.

By marking the entry in the page table as non-executable, we will be able to intercept executions to that page in the VM exit handler of our hypervisor.

The handler can then replace the reference to the physical page with our fake page and retrigger the execution.

However, once the CPU tries to read (or write) that memory, it would read our fake page. To prevent that, permissions of the page are marked as execute-only. That way, reads and writes will trigger a VM-exit again, which allows swapping in the original page. By marking this one as no-execute, the hypervisor can continuously swap our fake page in and out, to hide it from any kind of memory-based integrity check.

Is it possible to detect such hooks?

Usually, hooks, or any kind of memory manipulations, are detected by reading memory and comparing it to what it is supposed to look like, for example by hashing it.

With an EPT hook, those reads will always go to the original unmodified page and never reveal any manipulation. Therefore, using conventional integrity checks will not work.

Of course, trying to detect the driver or hypervisor in general is a viable way. However, not every hypervisor is necessarily evil and installs EPT hooks.

Microsoft's hypervisor (Hyper-V) is usually active on modern Windows systems, to for example assert kernel integrity (HyperGuard). So detecting EPT hooks here would result in a false positive.

Additionally, the hypervisor can hide itself from the list of loaded driver using EPT hooks, making it invisible.

Instead, I tried detecting EPT hooks using side channels, so for example by trying to observe the effects of a VM exit. Essentially, I came up with 3 different methods:

Write Check

The first method is pretty weak, but works with many open-source hypervisors. It essentially abuses the fact that most hypervisors don't reflect memory writes in their fake page.

This means if a write happens to the virtual page, it will be performed on the original physical page. But the fake physical page won't contain the change.

To detect an EPT hook by writing to a page, we first need to find a spot on that page that is safe to write to.

When compiling code, modern compilers usually align code of a function to 16 bytes. Meaning, in between two functions, there are often quite a few unused bytes that serve no purpose except aligning the next function.

Those bytes are usually nops (0x90) or software breakpoints (int 3, 0xCC).

Microsoft's compiler emits software breakpoints:

```
00000001800170B0 4C 24 30 E8 48 08 00 00 85 C0 78 D8 83 7C 24 30
00000001800170C0 00 74 CF E9 F6 C3 01 00 CC CC CC CC CC CC CC CC
00000001800170D0 48 89 5C 24 08 57 48 83 EC 20 8B FA 48 8B D9 BA
```

Therefore, searching for 2 or more consecutive CC instructions often reveals unused space. Of course, those CCs could be an operand, for example in `mov rax, 0xCCCCCC...`. So for a robust implementation, it might be useful to use a disassembler. But for the sake of simplicity, two consecutive CCs reveal unused space.

Executing such a CC instruction will result in an exception. This exception can be caught using exception handlers or using Microsoft's *__try/__finally* constructs.

We can now overwrite this instruction and replace it with, for example, a return instruction (0xC3). Without EPT hooks, executing the new instruction will not yield any exception. If an exception still occurs, it is very likely that an EPT hook is installed with a hypervisor that does not reflect writes to the fake page.

Of course this can easily be mitigated by checking for writes to the original page before swapping in the fake page.

However, this check can be extended. Writing to memory on one core should yield the same result on a different one (on Intel). As hypervisors are installed per core, writes not only to be reflected to the fake page of one core, but also to pages maintained by different cores.

Therefore, spawning a few threads and executing the written 0xC3 instruction can also reveal EPT hooks.

An example implementation of this check can be found here.

This check as a whole relies on bad or lazy implementation of EPT hooking. Therefore, I would not consider this a very good check. However, if it ever detects a hook, it is unlikely to be a false positive, which makes it very safe. As long as one makes sure the C3 is reverted back to a CC after performing the check 😛

Timing Check

This mechanism abuses the fact that VM exits and page swaps take time.

The idea is to create an execution pattern that triggers VM exits due to EPT violations and measure the execution.

First, we need to find something we can easily execute. Searching for a return instruction (0xC3) in the target page will allow us to execute this instruction as function and measure the time.

To measure the time, we will use the RDTSC instruction which will return the current CPU ticks.

Once this is ready, the check will measure timing of two patterns:

1. Repeatedly reading the same instruction without executing it
2. Alternating executions and reads of the same instruction

Reding the instruction without executing it should be quite fast, even with EPT hooks installed, because there is no need to page-in any page over and over again, as it was already paged in during the first read. Doing a warmup can even optimize the result.

If an EPT hook is present on the page, alternating between executions and reads to the page will cause a VM exit and thus a page swap every time. This takes time.

If no EPT hook is present, this should take about twice as long as just reading the memory (assuming that reading a single return instruction takes as long as executing it, which is roughly the case).

Therefore, due to the induced VM exits of EPT hooks, the timing should diverge quite a lot. More than a factor of 10, even 100.

An example implementation of this check can be found here.

A limitation is that RDTSC can be virtualized by the hypervisor. A carefully crafted hypervisor could therefore measure the time spent during VM exits and subtract that from the value RDTSC would initially return.

Additionally, due to time being involved, false positives can occur. Context switches can happen and many other disturbances can cause inaccurate results.

Thread Check

This check improves upon the previous mechanism. Again, timing of reads vs. alternating executions/reads is measured and evaluated.

However, instead of relying on RDTSC as a clock, time is measured by synchronizing two threads and letting them race. For example by incrementing a counter on one thread while another thread performs the task to be measured.

To reduce the probability of context switches, it is possible to bind the two threads to different physical cores using SetThreadAffinityMask. Additionally SetThreadPriority can be used to further reduce the probability of context switches.

An example implementation of this check can be found here.

Even if the probability of false postives is quite high here, probability of false negatives is low. Meaning if no EPT hook was detected, it is unlikely there is one installed on the checked page. Turns out probability of false negatives is equally likely. If the counting thread gets context switched, but not the other thread, a false negative can occur.

However, averaging out the times of multiple checks can yield very reliable results.

Additionally, it is definitely not trivial to mitigate this check for a hypervisor. Therefore, I consider this to be the best of the three approaches.

Summary

The mechanisms described here show the possibility to detect EPT hooks from usermode, without the need of elevated privileges.

One thing to remember is, they only scan a single page. Checking the entire executable address space of a process can get very costly, but might be desirable in a real world application.

Still, this shows that ways to detect EPT hooks exist. In the end, however, it remains a cat and mouse game.

A quick google search did not really reveal existing research on the topic of detecting EPT hooks. But, as I rather wanted to spend my time experimenting, I'm sure others have found different, if not better detections.

My hypervisor experiments with EPT hooking can be found here.
My implementation of the checks can be found here.