Protecting against Persistently Compromised PCI Devices

Making Qubes OS and OpenXT Live Up To Their Promises

Demi Marie Obenour

Invisible Things Lab

2024-03-14

PCI Passthrough

Why use PCI passthrough?

- The only way to isolate drivers for PCI devices.
- Relies on the IOMMU to enforce isolation.
- Believed to be secure if properly implemented...

PCI Passthrough

Why use PCI passthrough?

- The only way to isolate drivers for PCI devices.
- Relies on the IOMMU to enforce isolation.
- Believed to be secure if properly implemented...
- ...at least until system reboot!

The problem

- Real devices are stateful.
- Some of that state may be persistant across reboots.
  - Option ROMs
  - Firmware
  - Intentional storage.
- Even when it is *not* persistant, it may not be cleared by reset.
- **Attackers can exploit this!**

The attack

Attack flow:

The attack

Attack flow:

1. Compromise a VM with an attached PCI device.

The attack

Attack flow:

1. Compromise a VM with an attached PCI device.
2. Compromise the device itself, or trick it into doing an operation “late” (after reset).

The attack

Attack flow:

1. Compromise a VM with an attached PCI device.
2. Compromise the device itself, or trick it into doing an operation “late” (after reset).
3. Wait for the host to reset.

The attack

Attack flow:

1. Compromise a VM with an attached PCI device.
2. Compromise the device itself, or trick it into doing an operation “late” (after reset).
3. Wait for the host to reset.
4. Exploit the system during boot!

Consequences

- sys-usb $\implies$ dom0 or sys-net $\implies$ dom0 breakout, if network or USB device is stateful (integrated or chipset ones aren’t).
- GPU passthrough isn’t secure.
- Storage passthrough is quite likely not secure.

Why does this work?

- IOMMU disabled during ExitBootServices() for compatibility.
- First-instruction DMA protection requires Boot Guard/Platform Secure Boot.
  - This is therefore a requirement, sadly.
- Some devices, such as GPUs, require option ROMs to initialize.
- Firmware drivers might be vulnerable to malicious PCI devices.
- Most importantly, the OS, and likely firmware as well, trust that devices are what they claim to be.

The cloud provider solution

Custom board designs

Cloud providers use custom board designs.

- These may use SPI interposers or even completely emulated SPI flash.
- This allows the provider to ensure that any persistant mutable state is wiped out by a device power cycle.
- On-board MCU resets the device while CPU is in reset.

The cloud provider solution

Custom board designs

Cloud providers use custom board designs.

- These may use SPI interposers or even completely emulated SPI flash.
- This allows the provider to ensure that any persistant mutable state is wiped out by a device power cycle.
- On-board MCU resets the device while CPU is in reset.

Evaluation:

+ Multi-tenant safe: device can be passed through to one client, then another.
+ Can be designed into new hardware at reasonable cost.
– Cannot be retrofitted into existing hardware.
– Custom board designs are expensive (unless one is a cloud provider).

Hardening software against malicious devices

The cheaper solution

- Accept that persistent compromise might happen.
- Design the system to limit the impact of such a compromise.
- Users should be able to safely use a system even if one or more of its devices is permanently under attacker control!

Hardening software against malicious devices

The cheaper solution

- Accept that persistent compromise might happen.
- Design the system to limit the impact of such a compromise.
- Users should be able to safely use a system even if one or more of its devices is permanently under attacker control!

Evaluation:

– Not multi-tenant safe: device cannot be passed from one user to another.

\+ **Can** be retrofitted into existing hardware.

\+ Custom board designs **not required**!

Device trust

The root cause of this problem is that **the firmware and OS do not know which devices to trust.**

- A server can afford to trust no peripherals, provided that attestation is available.
- A desktop system *must* trust at least some devices.
  - To the user, some of the devices (keyboard, mouse, microphone, speaker, and display) *are* the system!
- These devices *must not* be assigned to an untrusted entity, and therefore cannot be compromised by a malicious VM.
- Only devices assigned to a guest must be considered potentially compromised!

How do we know which devices can be trusted?

- Most devices do not have a cryptographic identity.
- A device can pretend to be any other kind of device.
- The one unspoofable information about the device is the **physical slot into which it is plugged**.
- This turns out to be enough!

From physical location to device identity

- Firmware knows which device was shipped in each slot.
  - The keyboard will always be in the same place, every time.
  - If there is an NVMe device where the NIC should be, something is wrong – do not trust that device!
- Firmware also knows which slots were shipped empty (e.g. extension slots on desktop motherboards).
- Using this information, firmware can **determine if a device should be trusted or not**.
  - Most devices shipped by the vendor are trusted (exceptions: NICs, USB controllers, removable media).
  - Third-party devices are *untrusted* by default.

Preserving device identity through boot

- Firmware already provides a very large amount of information to the OS.
- "Which devices are trusted?" and "What kind of device is in each slot?" can just be new tables.
- The OS can simply refuse to use devices marked untrusted in firmware. (Linux: assign them to pciback).
- This works with DRTM too! Firmware can sign the information during early boot (with a TPM-bound key that is unusable later) and the MLE can check this signature.

Exposing device identity to the user

- The OS doesn't know enough information to tell the user where each slot is.
- The firmware does, though!
- One can provide the user a visual representation of the system and each and every slot that it has.

Updating trust

- Devices can go from trusted to untrusted over time.
- New devices may be trusted or untrusted.
- OS informs the firmware that something has changed.
- Firmware uses updated information for new trust decisions in the future.

Device becomes untrusted

- Alice's system ships with iGPU and discrete GPU, both trusted by default.
- Alice assigns their discrete GPU to their Windows gaming VM.
- Qubes OS tells the firmware that the Nvidia GPU is now **untrusted**.
- On subsequent boots, the discrete GPU is not used by the firmware, and Xen assigns it to the quarantine domain until it is assigned to the Windows VM.

New trusted device

- Bob's system ships with 2TiB NVMe storage.
- Bob decides to add another 2TiB NVMe storage device.
- After next boot, the system pops up a message explaining the physical location of the slot and asking the user what they inserted.
- Bob answers that they inserted a trusted NVMe storage device.
- Subsequent boots will consider this device **trusted**.

New untrusted device

1. Clair needs to recover information from a system she doesn't trust.
2. She inserts a different NVMe device into a slot.
3. She says the device is **untrusted**.
4. She assigns the device to a disposable VM, gets the data off, and reboots.
5. She forgets to remove the device first, but it isn't a problem: the device is simply ignored until they unplug it.

Device remains trusted

1. User assigns a device that has been designed for safe PCI passthrough.
2. The firmware knows that it can forcibly reset this device by asserting a GPIO in a way that the device's firmware can't override.
3. Firmware **ignores** the message and continues to treat the device as trusted after reboots (but not before resetting it!).

Device replaced by another device

1. Virgo removes their failing NVMe drive from their storage slot.
2. After booting, the system **forgets about what was in the slot**.
3. When Virgo inserts a new NVMe drive, they are **re-prompted**.

Requirements

- No special hardware required!
- Can be retrofitted via firmware update.
- Requires interaction with OS.

Firmware-OS communication

Firmware ⟹ OS

- Firmware already provides many tables to the OS
- A new ACPI is the best choice for x86, as it is used even by non-UEFI firmware.
- For other platforms, Device Tree or a known location in memory can be used.
- For DRTM support, the firmware can sign the table early, with a TPM-bound key that is later unusable.

Firmware-OS communication

OS ⟹ Firmware

- OS needs to be able to update device trust status.
- On non-DRTM systems, this is possible via EFI variables.
- On DRTM systems, UEFI runtime services are unavailable for security reasons, unless a protected EFI call mechanism is available.
- Disabling DRTM is highly undesirable – it might well mean that the user can't even boot the OS without a recovery phrase.
- Possible solution: signed file on the EFI system partition (signed by TPM)

Conclusion

- Persistant compromise via PCI devices is a significant threat to Qubes OS users.
- Thankfully, it is possible to solve this problem via firmware and OS updates.
- Let's start working on them!