Technical Report of Joint Security Review by Microsoft and Intel on Intel® TDX1.5

August 2024

Ben Hania, Maxime Villard, Yair Netzer
Microsoft Offensive Research & Security Engineering

Nagaraju Kodalapura, Truc Nguyen
Intel Offensive Security Research

Introduction 3
Glossary 3
Review Scope 4
Methodology 4
  Runtime Environment 4
  Source Code and Design Review 5
Findings 5
  Vulnerabilities in TD Live Migration 5
    Background 5
      Service TD Binding 6
      Instance Binding 7
      Live Migration Flow 7
      Changes in Attestation 8
    Vulnerability 1: Attestation Forgery Via Migration 8
      Remediation 9
    Vulnerability 2: Malicious Service TD Binding Post Attestation 10
      Remediation 11
    Vulnerability 3: Inconsistent Attestation Hash Post Rebinding 11
      Remediation 11
    Vulnerability 4: Missing Sanity Checks on Import 11
      Remediation 11
  Vulnerabilities in the context-switching logic 12
    Background 12
    Vulnerability 5: Incorrect Context-Switching of ProcessorTrace Registers 12
      Remediation 14
    Vulnerability 6: Unexpected SEAMCALL VMEXITs 14
      Remediation 15
Future Research 15
Acknowledgments 15
Appendix: Public Resources and References 16

Introduction

This report is the technical summary of the vulnerabilities found in the joint review by Microsoft and Intel of the security of Intel® Trust Domain Extensions (TDX) version 1.5. TDX is Intel's newest confidential computing technology, a hardware-based trusted execution environment (TEE) that facilitates the deployment of Trust Domains (TD), which are hardware-isolated virtual machines (VM) designed to protect sensitive data and applications from unauthorized access.

The review was conducted by Microsoft security researchers in close collaboration with Intel engineers, researchers and architects over four months. This collaboration included periodic technical meetings where Intel promptly addressed any questions or issues raised by the Microsoft researchers. The review culminated in an intensive two-week on-site hackathon. Throughout the review, the Intel team was highly available and responsive.

The scope of the review covered most of the code base and the specification documents of Intel TDX, with an emphasis on three new features introduced in version 1.5: TD Partitioning, TD Preserving Updates, and TD Live Migration.

29 attack vectors were scrutinized as part of the review, yielding 6 confirmed vulnerabilities and 15 recommendations for defense-in-depth improvements. Three of the vulnerabilities compromised Intel TDX's core promise of integrity and confidentiality for guest confidential VMs, either by forging attestation data of guest Trust Domains or by complete Intel TDX module takeover. Another issue impacted the integrity of legacy VMMs on Intel TDX enabled platforms related to SEAMCALL operation.

This document assumes basic knowledge of the architecture of Intel TDX. For more information see the official documentation by Intel: [Documentation for Intel® Trust Domain Extensions](#).

Glossary

Acronym	Full Name
ACM	Authenticated Code Module
BMC	Baseboard Management Controller
CSP	Cloud Service Provider
CVE	Common Vulnerabilities and Exposures
DoS	Denial of Service
ISA	Instruction Set Architecture
MCHECK	Intel firmware, verifies the security configuration of the system
MSR	Model Specific Register
NP-SEAMLDR	Non-Persistent SEAM Loader
PECI	Platform Environment Control Interface
P-SEAMLDR	Persistent SEAM Loader
SEAM	Secure Arbitration Mode
SEAMRR	SEAM Range Register
SGX	Software Guard Extensions
TD	Trust Domain
TDX	Trust Domain Extensions
TEE	Trusted Execution Environment

UUID	Universally Unique Identifier
VM	Virtual Machine
VMM	Virtual Machine Monitor

Review Scope

This review covered the Intel TDX module itself, the P-SEAMLDR and the NP-SEAMLDR. Although the security of all three components relies on MCHECK being secure, MCHECK was out of scope for the review.

Attack vectors and components that were covered in the review include:

- All code changes and new code that were introduced after the release of Intel TDX 1.0, and most of the legacy code.
- The context-switching logic implemented in the CPU, the P-SEAMLDR and the TDX module.
- The use of MSRs in the P-SEAMLDR and the TDX module.
- The signature validation logic in the NP-SEAMLDR and the P-SEAMLDR.
- The attestation mechanism on the Intel TDX side, excluding SGX.
- The BMC PECI interface as a potential side-channel attack vector.
- TD Migration – a feature that allows live migration of TD guests while retaining the TDX security promises.
- TD Partitioning – a feature that allows running 4 partitioned VMs inside one TD. A main VM called L1, which must be Intel TDX-aware, manages three additional VMs, called L2 VMs, which might not be Intel TDX-aware. This opens the possibility of running a VM with a legacy OS, that is not Intel TDX-aware, inside a TD.
- TD Preserving Update – a feature that allows updating the Intel TDX module without tearing down running TD guests. TDs are put on pause, the Intel TDX module is taken down, and the P-SEAMLDR installs the new Intel TDX module.

Methodology

Runtime Environment

The Microsoft team developed a specialized emulator called Cornelius that served as the main runtime environment during the review to perform security research and validation of certain scenarios. Cornelius allows to run the P-SEAMLDR and the TDX module as a VM on Windows, without requiring actual TDX hardware. It is useful for security testing, fuzzing, and rapid prototyping of the TDX firmware.

Cornelius supports several state-of-the-art debugging and vulnerability research features, including:

- Support for VM snapshotting, to easily implement fuzzers based on Cornelius.
- Support for sanitizers on both the P-SEAMLDR and the TDX module: Address Sanitizer (ASAN), Undefined Behavior Sanitizer (UBSAN), Coverage Sanitizer (SANCOV).
- Runtime invariant checking, to ensure that certain security properties are maintained throughout the execution of the TDX code.

Cornelius is now open-source and published under an MIT license in the following Microsoft GitHub repository: https://github.com/microsoft/Cornelius.

Source Code and Design Review

The joint research team conducted a thorough manual review of the source code of the NP-SEAMLDR, P-SEAMLDR and TDX module, and performed a design review that involved scrutinizing the relevant design documents, architectural plans and CPU specifications pertaining to all the features within the review scope.

Throughout the review, the team used automated scripts and dynamic analysis tools based on the Cornelius emulator to validate certain scenarios, reproduce identified issues, and overall build a better understanding of the TDX architecture.

Consequently, several minor issues and defense-in-depth concerns were identified, alongside more significant problems within the Live Migration feature and the context-switching logic, which are detailed in subsequent sections.

Findings

#	Title	CVSS Score	Fixed In	CVE
1	Attestation Forgery Via Migration	6.0 (Medium) - CVSS:3.1/AV:L/AC:L/PR:H/UI:N/S:U/C:H/I:H/A:N	Intel TDX 1.5.05	CVE-2023-47855
2	Malicious Service TD Binding Post Attestation	6.0 (Medium) - CVSS:3.1/AV:L/AC:L/PR:H/UI:N/S:U/C:H/I:H/A:N	Intel TDX 1.5.05	CVE-2023-47855
3	Inconsistent Attestation Hash Post Rebinding	6.0 (Medium) - CVSS:3.1/AV:L/AC:L/PR:H/UI:N/S:U/C:H/I:H/A:N	Intel TDX 1.5.05	CVE-2023-47855
4	Missing Sanity Checks on Import	7.9 (High) - CVSS:3.1/AV:L/AC:L/PR:H/UI:N/S:C/C:H/I:H/A:N	EMR PV (1.5.01.02)	CVE-2023-45745
5	Incorrect Context-Switching of ProcessorTrace Registers	7.2 (High) - CVSS:3.1/AV:L/AC:H/PR:H/UI:N/S:C/C:H/I:H/A:N	SPR PLR5 PV (1.5.01)	CVE-2024-39283
6	SEAM ISA: Unexpected SEAMCALL VMEXITs	6.5 (Medium) - CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:C/C:N/I:N/A:H	EGS UPLR1	CVE-2024-22374

Vulnerabilities in TD Live Migration

Background

A new core feature introduced in Intel TDX 1.5 is TD Live Migration. Similarly to a general purpose VM migration, TD migration allows the Cloud Service Provider (CSP) to relocate a running TD VM to a similarly-configured but different Intel TDX platform in the cloud environment. This is regularly done for the purpose of ensuring high availability, efficient resource management, and

seamless system maintenance. Live migration allows dynamic movement of VMs across physical hosts without disrupting services, enabling load balancing, proactive fault tolerance, and disaster recovery. Additionally, live migration supports scaling, compliance with SLAs, and optimization of data center resources.

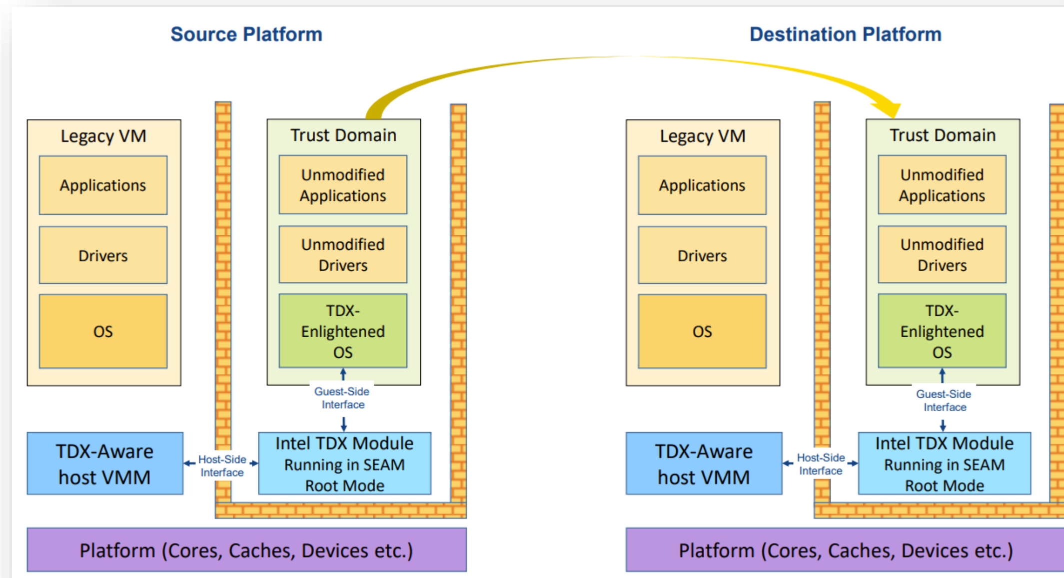

Figure 1: TD migration. Intel TDX Official Documentation.

In the case of Confidential VMs, the confidentiality and integrity of the VM must be maintained during the migration process, while allowing the host VMM to perform the migration. For this purpose, Intel introduced the Service TD, a type of TD that can be bound to a target TD and extends its TCB (Trusted Computing Base). Currently the Migration TD – or MigTD for short – is the only type of Service TD that is supported.

The main purpose of the MigTD is to negotiate a secure session and exchange migration session keys with its peer MigTD on the destination platform. The session keys are then subsequently used to protect the TD private memory pages during transport.

Service TD Binding

To enable a Service TD to access target TD metadata or perform privileged operations related to it, the host VMM must first bind it to the target TD. The binding is done using the TDH.SERVTD.BIND API function. This Intel TDX function calculates the hash of the service TD's TDINFO structure and writes it to a structure called the *binding table* of the target TD. The binding table also holds the service TD's Universally Unique Identifier (UUID) and several other attributes. Later, on each operation the Service TD will perform on the target TD, the Intel TDX module will verify that the calling TD is indeed the bound Service TD, by comparing the caller UUID and hash to the values written in the binding table. The hash is recalculated each time.

The binding operation is also reflected in the attestation report of the target TD (see Changes in Attestation below). For that reason, the host must bind, or pre-bind, the service TD before the target TD is finalized and can be executed. Pre-binding involves the host declaring in advance the hash and other attributes of the service TD that will be bound in the future, leaving only the UUID

to be filled on the actual binding. That way, the attestation report of the target TD can still reflect the Service TD even if the Service TD is not yet bound.

Instance Binding

Instance binding is a feature Intel introduced for rebinding a service TD to a target TD it was already bound to in the past, based on its UUID instead of TDINFO hash. The TDINFO hash can change when the service TD extends one or more of its RTMRs (Run-Time Measurement Register). On instance binding, the hash in the binding table is not checked, but instead modified with the new hash.

As described in Malicious Service TD Binding Post Attestation below, this feature was ultimately disabled by Intel.

Live Migration Flow

Figure 2: Simplified TD Migration Flow.

Before migration can start, the source and destination MigTDs communicate and authenticate each other using a remote attestation server. After authentication succeeds, the MigTDs generate and exchange migration transport keys to protect the TD private data (1). The MigTDs write the keys to the TD metadata using Intel TDX API (2). Note that this metadata is used and managed by the Intel TDX module and is not directly accessible to the target TD itself.

Next, the host VMM on the source platform uses a set of Export Intel TDX API functions to export the TD data and metadata. This private information is encrypted by the TDX module using the transport keys and is returned to the host VMM (3). Then, the host sends the encrypted data to the destination platform (4). On the destination platform the host VMM creates a skeleton TD and binds the destination MigTD to it. The host then Imports the data and metadata to it using a set of Import Intel TDX API functions, where the Intel TDX module decrypts and rebuilds the migrated TD (5).

This is a simplified explanation of the migration flow. For a more detailed description, see the official documentation by Intel - [Documentation for Intel® Trust Domain Extensions]

Changes in Attestation

Table 3.33: TDINFO_STRUCT Definition

Name	Offset (Bytes)	Type	Size (Bytes)	Description
ATTRIBUTES	0		8	TD's ATTRIBUTES
XFAM	8		8	TD's XFAM
MRTD	16	SHA384_HASH	48	Measurement of the initial contents of the TD
MRCONFIGID	64	SHA384_HASH	48	Software-defined ID for non-owner-defined configuration of the guest TD – e.g., run-time or OS configuration
MROWNER	112	SHA384_HASH	48	Software-defined ID for the guest TD's owner
MROWNERCONFIG	160	SHA384_HASH	48	Software-defined ID for owner-defined configuration of the guest TD – e.g., specific to the workload rather than the run-time or OS
RTMR	208	SHA384_HASH	NUM_RTMRS * 48	Array of NUM_RTMRS (4) run-time extendable measurement registers
SERVTD_HASH	400	SHA384_HASH	48	If is one or more bound or pre-bound service TDs, SERVTD_HASH is the SHA384 hash of the TDINFO_STRUCTs of those service TDs bound. Else, SERVTD_HASH is 0.
RESERVED	448	N/A	64	Must be zero

Figure 3: TDINFO Definition. Changes in version 1.5 (at time of review) marked in red. Intel TDX Official Documentation.

The TD attestation report is derived partially from the TDINFO structure. After binding a service TD to a target TD, the hash of the service TD's TDINFO is written to the binding table of the target TD. After the target TD is fully created and finalized by the host VMM, the hash of the entire binding table is written to a field in the TDINFO of the target TD called SERVTD_HASH. That way the attestation server can verify that no unexpected service TD is bound to a tenant TD.

Another addition is a new attribute bit in the TDINFO's Attributes field, called `MIGRATABLE`. This bit indicates whether the TD can be migrated or not and is reflected in the attestation report.

Vulnerability 1: Attestation Forgery Via Migration

The TDINFO structure is part of the protected metadata that is migrated along the rest of the migrated TD. This means there is a circular trust relationship between a MigTD and the attestation data. The MigTD can only be considered trustworthy if the TD attestation data is trusted. However, during migration, the protection of TD attestation data relies solely on the trustworthiness of the MigTD.

This design flaw allows the host VMM to initiate a migration flow for a malicious TD it controls and through it tamper with its TDINFO metadata, from which the attestation report is built.

Moreover, this issue is not isolated to migratable TDs only. While the Export API validates the target TD is indeed migratable, by verifying the MIGRATABLE bit in TDINFO.Attributes is 1, the Import API does not. Since this MIGRATABLE bit can also be modified during migration, it is possible to start with a malicious migratable TD, and while forging the attestation, also set MIGRATABLE to 0 before importing.

The attack scenario is as follows:

1. The host VMM creates a migratable malicious TD and binds a malicious MigTD to it.
2. The MigTD generates an encryption key and shares it with the host VMM.
3. The host VMM initiates a migration process and exports the TD's data and metadata.
4. Having the encryption\decryption key, the host VMM decrypts the TD metadata, modifies the TDINFO to match that of a legitimate victim TD, and encrypts it again.
    a. The SERVTD_HASH value is also modified to match the expected hash (a legitimate hash or 0 if none is expected).
    b. If the victim TD the attacker is trying to spoof is not migratable, the MIGRATABLE bit is also set to 0.
5. The host VMM now imports the modified and potentially non-migratable victim TD, creating a compromised TD with forged measurements.
6. The malicious TD uses the TDG.MR.REPORT API function to get a signed attestation report from the Intel TDX module and sends it to the remote attestation server.
7. The forged attestation report passes validation of the remote server, and the malicious TD receives customer secrets, breaking confidentiality and integrity.

Note that this attack does not require transferring the data to another platform. The Export and Import operations can take place on the same platform.

Another attack scenario that does **not** involve a complicit Host VMM, but instead a malicious 3rd party customer that has some foothold in the cloud network:

1. The malicious customer spins up a TD and a MigTD that is bound to it.
2. The MigTD waits for a migration to begin.
3. The MigTD shares the encryption\decryption keys with a compromised node in the cloud network, through which the encrypted TD private data passes.
4. The compromised node decrypts and modifies the private data, editing the TDINFO to match that of another customer TD, and encrypts it again.
    a. The SERVTD_HASH value is also modified to match the expected hash (a legitimate hash or 0 if none is expected).
    b. If the TD the attacker is trying to spoof is not migratable, the MIGRATABLE bit is also set to 0.
5. The compromised node sends the data to the destination platform.
6. The destination platform imports the migrated TD with the modified TDINFO.
7. The migrated TD resume execution and pass the attestation validation of the victim customer's remote server.
8. The TD that is in control of the attacker can now receive the secrets of the unsuspecting customer, breaking confidentiality and integrity.

Remediation

Intel fixed this issue by defining the SERVTD_HASH value to be immutable on migration. This means that after migration, the hash in the SERVTD_HASH value is always that of the destination

MigTD that was bound to the skeleton TD before the migration. Now if the host tries to perform this attack, the SERVTD_HASH after the migration will still reflect the malicious destination MigTD hash, causing the attestation to fail. If an attacker would create a legitimate destination MigTD, the mutual authentication between the MigTDs will fail, as the source MigTD is malicious. Further details are captured in the corresponding CVE: CVE-2023-47855.

Intel also added a validation that the TD being imported is indeed migratable (see *Vulnerability 4* below).

Vulnerability 2: Malicious Service TD Binding Post Attestation

Instance Binding relies on the UUID to be unique per TD and not modifiable by the TD or the host VMM. However, using the technique described in the previous section, the host VMM can use the migration process to modify the UUID value. By controlling a service TD's UUID, the host VMM can use instance binding to bind a malicious service TD to an already attested and running TD, as long as any legitimate service TD was already pre-bound to it.

This attack scenario is as follows:

1. A victim customer spins up a legitimate migratable TD, that is expected to be bound with a legitimate service TD.
2. The host creates the 2 TDs as expected and binds them.
3. The victim customer TD is executed and passes attestation validation via a remote server.
4. The victim customer TD receives customer secrets and begin processing confidential customer workloads.
5. The host creates a malicious and host-controlled migration TD, and using the tampering via migration technique described above, modifies its UUID value to match that of the legitimate service TD.
    a. The host creates and binds a second host-controlled migration TD to the malicious migration TD.
    b. The host exports and decrypts the migration TD metadata, using the keys provided by the second migration TD.
    c. The host modifies the UUID value to match that of the legitimate service TD.
    d. The host re-encrypts and imports the malicious migration TD.
6. The host "rebinds" the malicious migration TD to the victim customer's TD using instance-binding, replacing the legitimate service TD. The UUID value matches the one in the binding table, so the binding succeeds.
    a. The malicious migration TD can now access confidential metadata of the tenant TD.
7. The host initiates another migration process, this time with another malicious MigTD as the destination MigTD. As both source and destination MigTDs are malicious and controlled by the host, mutual authentication is not a problem.
8. The host uses the keys from the migration TDs it controls, decrypts and reads the exported private data, breaking confidentiality. The host can also break the integrity promise by modifying the content of the victim TD before encrypting and importing it back, creating a compromised TD still trusted by the victim customer.

Remediation

Intel resolved this issue by removing the Instance-binding feature. This feature will be reconsidered when it becomes necessary. Further details are captured in the corresponding CVE: CVE-2023-47855.

Vulnerability 3: Inconsistent Attestation Hash Post Rebinding

Upon rebinding with Instance Binding, the UUID of the service TD, and not the service TD hash, is checked to be equal to the one stored in the target TD's binding table. Additionally, if the UUID numbers indeed match, the service TD hash that is stored in the binding table is updated to the newly calculated hash of the service TD being bound. However, the SERVTD_HASH value in the target TD's TDINFO struct is not updated to reflect this change.

To understand the impact of this issue, suppose that a service TD extends one of its RTMRs in a way that the remote attestation server would have considered invalid and therefore would have rejected it as a possibly compromised service TD. Because the SERVTD_HASH in the TDINFO struct of the target TD is not updated, this critical change in the state of the service TD would not be reflected in the attestation report, and the target TD will keep passing attestation verification.

Remediation

As previously stated, Intel has completely disabled instance binding until further review. Further details are captured in the corresponding CVE: CVE-2023-47855

Vulnerability 4: Missing Sanity Checks on Import

Upon importing a TD guest that is being migrated, the TDX module must internalize several structures that are part of the metadata of the TD guest.

Several sanity checks were missing during the import of these structures and could cause undefined behaviors and memory corruption in the TDX module.

For instance, one of the metadata fields included in the migration process is CURR_VM, an integer index value that is supposed to be within the range [0;3]. Sanity checks were missing on import for this field, and this could cause the TDX module to perform out-of-bounds memory accesses when indexing internal structures with CURR_VM as index.

In this example, the attack scenario would be the following:

1. The host VMM creates a migratable malicious TD and binds a malicious MigTD to it.
2. The MigTD generates an encryption key and shares it with the host VMM.
3. The host VMM initiates a migration process and exports the TD's data and metadata.
4. Having the encryption\decryption key, the host VMM decrypts the TD metadata, modifies the CURR_VM value to be above 3, re-encrypts it, and re-imports the TD guest back in.
5. From now on, the TDX module will perform out-of-bounds memory accesses each time it uses CURR_VM as an index to access internal structures.

Remediation

Intel remediated the issues by adding new sanity checks in the TDX module. Further details are captured in the corresponding CVE: CVE-2023-45745.

Vulnerabilities in the context-switching logic

Background

When the TDX module is asked to execute a TD guest via the TDH.VP.ENTER command, a context-switch takes place to:

1. Save the TDX module values of the CPU registers into memory, and
2. Copy the in-memory values of the guest registers into the physical CPU registers.

This ensures that when the TD guest starts executing, it has its own values stored in the CPU's registers. Symmetrically, when a TD guest finishes executing and triggers a VMEXIT towards the TDX module, a context-switch takes place to:

1. Save the guest values of the CPU registers into memory, and
2. Copy the in-memory TDX module values of the registers back into the CPU registers.

This ensures that after a VMEXIT the TDX module is back with the register values it initially had before the VMLAUNCH.

These context-switches to enter and leave a TD guest can be performed using two approaches:

- VMCS: the VMCS of the TD guest is configured by the TDX module and tells the CPU to automatically context-switch certain registers. This approach is therefore performed by the CPU with no software intervention from the TDX module.
- XSAVE: certain types of registers cannot be context-switched via the VMCS, and for these, the TDX module performs the context-switch *manually* in software, using the XSAVE and XRSTOR instructions.

These two approaches are complementary, and combined ensure that all registers are properly saved and restored during context-switches.

The TDX module provides debugging APIs that allow the VMM to have access to the in-memory values of the registers, but only on debuggable guests.

Vulnerability 5: Incorrect Context-Switching of ProcessorTrace Registers

A critical vulnerability existed in the way the ProcessorTrace registers were being context-switched in the TDX module.

Intel ProcessorTrace is a CPU feature that provides fine-grained execution tracing. It can be configured via several privileged MSRs, two of which are important:

- IA32_RTIT_CTL: this is the control MSR which, among other things, has a TraceEn bit that dictates whether ProcessorTrace is enabled or not.
- IA32_RTIT_OUTPUT_BASE: this is an MSR that contains the memory address at which the CPU will create a log of the software execution when ProcessorTrace is enabled.

Simply said, when IA32_RTIT_CTL.TraceEn enables ProcessorTrace, the CPU creates a log buffer at the memory address pointed to by IA32_RTIT_OUTPUT_BASE and logs the software execution there, using a format documented in the Intel specifications that is not relevant to describe here.

ProcessorTrace is supported in TD guests, and therefore the two aforementioned MSRs must be context-switched between the TDX module and the TD guests. This context-switch takes place using both the VMCS and XSAVE approaches:

- VMCS: the VMCS of the TD guest is configured to restore the TD guest's IA32_RTIT_CTL value on VMLAUNCHs, and clear the MSR on VMEXITs. This guarantees that upon VMEXITs ProcessorTrace is disabled in the Intel TDX module.
- XSAVE: the IA32_RTIT_OUTPUT_BASE MSR is not handled by the VMCS, and therefore, the XSAVE approach is used by the Intel TDX module to save and restore it. However, this MSR is not individually saved and restored by XSAVE; instead, **all** of the ProcessorTrace MSRs are saved and restored by XSAVE. As an important side effect, this implies that the IA32_RTIT_CTL MSR that is context-switched by the VMCS approach is **also** context-switched by the XSAVE approach.

A critical security issue arises from the fact that the XSAVE approach covers IA32_RTIT_CTL: a malicious host VMM could create a custom debuggable TD guest, and leverage the debugging APIs to set the TD guest's in-memory IA32_RTIT_CTL.TraceEn value to 1, thereby enabling ProcessorTrace inside the TDX module itself as soon as the TDX module executes XRSTOR to copy the ProcessorTrace register values of the TD guest into the physical CPU.

By additionally initializing the TD guest's IA32_RTIT_OUTPUT_BASE value via the debugging APIs to point inside TDX memory (SEAMR), a malicious VMM could have the CPU overwrite TDX memory with a ProcessorTrace execution log, the contents of which are predictable and could also be forged by the VMM. This effectively offered the malicious VMM a write-what-where primitive to overwrite TDX memory and fully escalate privileges into the TDX module.

A possible attack scenario is as follows:

1. A malicious VMM creates a custom debuggable TD guest and enables ProcessorTrace in it.
2. The malicious VMM uses the TDH.VP.WR command to initialize the two following in-memory values of the guest registers:
    a. IA32_RTIT_OUTPUT_BASE: the malicious VMM writes the physical address of a memory page belonging to the TDX module. For simplicity let's assume that this is physical address 0xA000, and that it contains critical data such as the instruction bytes of the TDX module.
    b. IA32_RTIT_CTL: the malicious VMM sets the TraceEn bit to 1.

    It should be understood that this command does not initialize the physical CPU registers themselves, rather, it stores in memory the values of the registers that the TDX module will have to copy into the CPU when it processes a TDH.VP.ENTER command to execute a TD guest.

3. The malicious VMM uses the TDH.VP.ENTER command to start executing its custom guest. Upon processing this command, the TDX module will copy the TD guest registers from memory using the XSAVE approach, meaning that the IA32_RTIT_OUTPUT_BASE and IA32_RTIT_CTL values that the malicious VMM previously initialized in memory get copied into the physical CPU registers. Immediately at that moment, ProcessorTrace gets enabled, **while the TDX module is still executing and hasn't yet executed VMLAUNCH to run the guest**. Because the CPU is still in privileged SEAM mode, access to page

0xA000 is naturally allowed, and therefore, the CPU will proceed to log the execution and overwrite the contents of page 0xA000 with the ProcessorTrace log buffer.

4. By using additional ProcessorTrace MSRs to configure the type of logging that takes place, the VMM could control the data that the CPU writes into page 0xA000. From there, privilege escalation is achieved.

An alternative exploitation option also existed: a malicious VMM could also make IA32_RTIT_OUTPUT_BASE point to a non-TDX memory that it has access to and could simply log the execution of the TDX module and exfiltrate confidential information without overwriting TDX memory.

Remediation

Intel has remediated this issue by making sure that the TDX module doesn't allow the VMM to modify XBUFF.RTIT_CTL even for debug TD, thereby mitigating the above said attack. Further details are captured in the corresponding CVE: CVE-2024-39283.

Vulnerability 6: Unexpected SEAMCALL VMEXITs

As part of the Intel TDX architecture, a new privileged CPU instruction named SEAMCALL was implemented. This instruction can only be executed in CPL0, and has a VMEXIT reason associated with it.

A first inconsistency exists when this instruction is executed in a legacy (non-TD) VM: the VMEXIT associated with SEAMCALL triggers **before** the CPL check takes place as indicated in the pseudo-code of the instruction provided in the Intel TDX specification.

Operation

```
IF not in VMX operation or inSMM or inSEAM or ((IA32_EFER.LMA & CS.L) == 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VMexit("basic reason" = SEAMCALL,
            "VM exit from VMX root operation" (bit 29) = 0);
ELSIF CPL > 0 or IA32_SEAMRR_MASK.VALID == 0 or "events blocking by MOV-SS"
    THEN #GP(0);
```

Figure 4: pseudo-code of the SEAMCALL instruction.

As a result, it is possible to execute the SEAMCALL instruction from CPL3 in a legacy guest, and this triggers a VMEXIT towards the hypervisor without triggering a #GP(0) inside the guest.

A second, broader inconsistency related to SEAMCALL is that it is unconditionally recognized by the CPUs that support Intel TDX: it does not have an enablement control that allows to enable or disable the instruction. As a result, the SEAMCALL VMEXIT can be unconditionally triggered by legacy VMs, even if the hypervisor is not Intel TDX-aware and doesn't recognize the SEAMCALL VMEXIT reason. Each hypervisor has its own way of processing unrecognized VMEXITs, but when it comes to Hyper-V and Linux KVM, the hypervisor simply shuts down the guest: if these hypervisors do not recognize the SEAMCALL VMEXIT originating from a guest, they proceed to shut down the guest.

In configurations where a non-TDX-aware hypervisor runs on TDX hardware, these two inconsistencies in the SEAMCALL instruction lead to two security problems:

1. A usermode process running in a legacy guest can execute the SEAMCALL instruction and cause the whole guest to be shut down by the hypervisor. This effectively creates a DoS primitive, that can be critical in cloud setups where usermode containers from different customers run inside the same VM: any container executing the SEAMCALL instruction will cause the whole VM to be shut down and all the other containers of the other customers to be killed.
2. In a cloud scenario where nested virtualization is used with different customers having access to different nested guests, any guest executing the SEAMCALL instruction (even from CPL0) will cause the hypervisor to shut down the entire guest, including all the children nested guests of the different customers.

In essence, the priority inversion on the CPL check and the very unconditionality of the SEAMCALL instruction can create situations where a cloud customer is able to perform a DoS on other cloud customers if a non-TDX-aware hypervisor runs on TDX hardware.

Remediation

Intel is providing a microcode patch that inverts the priority of the CPL check on SEAMCALL, ensuring that the CPL check takes place **before** the VMEXIT. Further details are captured in the corresponding CVE: CVE-2024-22374.

To handle the nested virtualization scenario, Microsoft's hypervisor was updated to recognize the SEAMCALL VMEXIT in legacy mode and appropriately trigger VMExits to Hyper-V as soon as the issue was discovered, and all supported Windows releases are now patched.

Future Research

Possible opportunities for future research collaboration may encompass the following:

- Intel TDX Connect – The next key feature upgrade for Intel TDX product roadmap that introduces trusted device assignment to a TD, extending its TD TCB and to TEE-IO Device Interface (TDI) while protecting its data.
- Confidential AI – Investigation on how Confidential Computing features can be supported by accelerators, container technologies, and defenses against attacks inherent in the use of AI and ML.

Acknowledgments

The review team acknowledges the contributions of the following people for making the review possible through providing product training, addressing technical questions, and sharing security expertise.

We would like to extend our gratitude to the following Intel engineers: Alon Levi, Aviv Eisen, Avishai Redelman, Avraham Shalev, Boaz Tamir, Dror Caspi, Eyal Goldik, Hareesh Khattri, Ilia Rozentsvaig, Moshe Levi, Nagaraju Kodalapura, Sergey Tretiyak, Simon Johnson, Tefillah Katz, and Truc Nguyen.

Finally, we would also like to thank the following Microsoft engineers who provided technical support and guidance throughout this security assessment: Ada Ostrokol, Alexander Grest, Andrey Markovytch, Ben Hania, Carlos Vazquez, Caroline Perez-Vargas, Eli Cohen Nehemia, Jon Lange, Maxime Villard, Moti Markovitz, Netanel Ben Simon, Vladimir Abramzon, Yair Netzer.

Appendix: Public Resources and References

Intel TDX specifications and source code: Documentation for Intel® Trust Domain Extensions.

Intel TDX security assurance white papers: Intel TDX Security Research and Assurance.

Support for TDX Confidential VMs in Azure: Azure ECesv5 and ECedsv5-series confidential VMs.