ZKSECURITY

Audit of dstack

Date: May 26th, 2025

Introduction

On May 26, 2025, zkSecurity was engaged to perform a security audit of Phala Network's dstack project. The audit focused on parts of two public codebases: dstack at commit `be9d0476`, and meta-dstack at commit `5b63aec3`. The audit was conducted between May 26th, and June 13th, 2025 by two consultants.

Scope

The scope for the project was twofold:

Low-level libraries and tooling. The first part of the scope involved reviewing low-level libraries and tooling, including the following components from https://github.com/Dstack-TEE/dstack:

- `ra-tls` and `ra-rpc`, implementing an augmented TLS service (which relies on an external x509 cert library which was deemed out of scope)
- `guest-agent`, a service that runs in a confidential VM (CVM) to serve a container's key derivation and attestation requests
- `dstack-util`, a CLI with subcommands like full-disk encryption (FDE).

Image-related files. The second part of the scope focused on image-related files for the dstack OS, such as Yocto BitBake recipes and base initialization scripts. This includes Yocto BitBake files (rootfs, initramfs, initramfs-files/init) from https://github.com/Dstack-TEE/meta-dstack/tree/main/meta-dstack/recipes-core/images and `basefiles` from https://github.com/Dstack-TEE/dstack

Methodologies

We approached the project in two phases. In the first phase we focused on:

- Understanding the intended attacker model and trust boundaries based on the documentation provided.
- Understanding and reviewing RA-TLS, an SGX-inspired protocol that integrates Intel SGX Remote Attestation with TLS (Integrating Intel SGX Remote Attestation with Transport Layer Security).
- Understanding internal and external CVM interfaces and the access control in place for them.
- Evaluating privilege escalation strategies and potential vulnerabilities and their impact on the intended countermeasures.

In a second phase we focused on the dstack OS images, configuration and QEMU launching of the CVM:

- Reviewing the reproducibility of builds.
- Checking if measurements reflect necessary events at the right time.
- Reviewing the differences between dev and prod images, and understanding hardening of the production image.
- Assessing the role of dm-verity in the booting process.
- Focusing on the host operator attacker model and the potential vulnerabilities operators can exploit.

Strategic Recommendations

The following recommendations are intended to strengthen the overall security posture of the system.

Documentation. Given the complexity of the system and the challenges of producing secure, minimal images, we recommend documenting both the rationale and the decisions behind the hardening of the BitBake recipes. See VMM Is Currently Trusted In OVMF Build, qemu-guest-agent Is Present In Production and more generally Lack Of Documentation On Design and Hardening Decisions In meta-dstack Layer

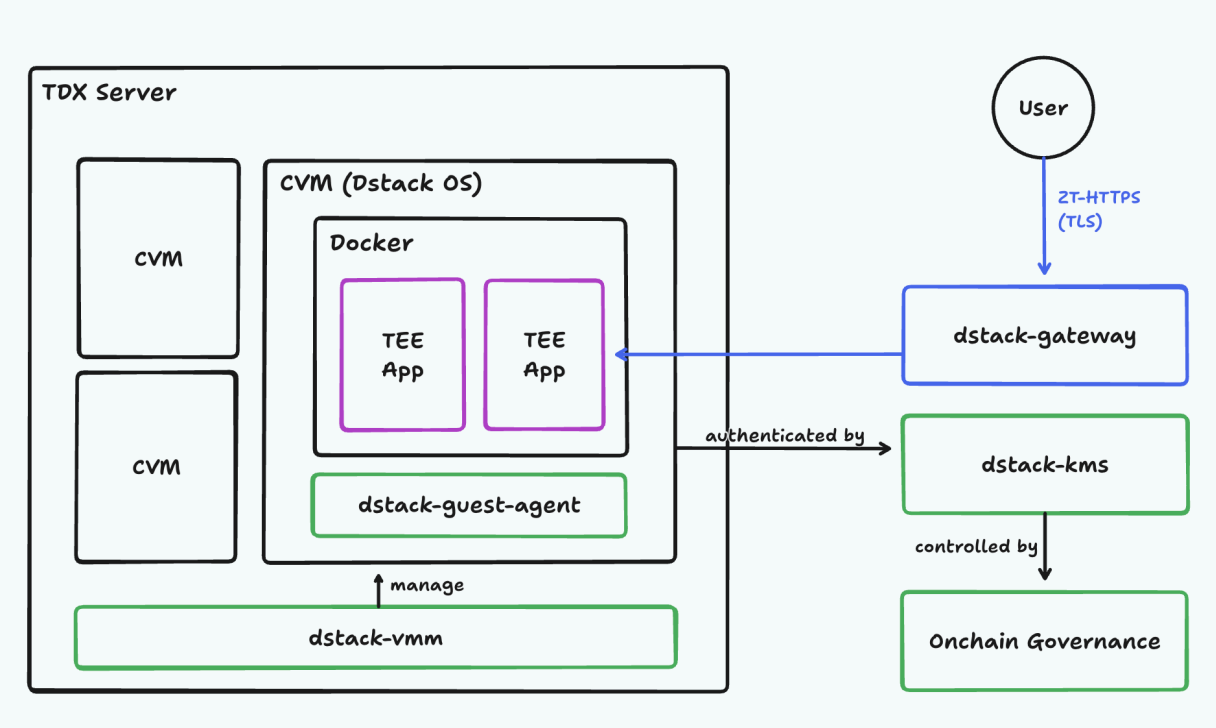
Further Audits. While this audit served as a solid entry point, several related areas remain unreviewed. We recommend conducting additional audits on the following components:

- **dstack-kms and related flows**. As Pre-Launcher Code Can Be Used To Leak Secrets On Default KMS pointed out, some KMS flows and interactions within the scope of this audit were overlooked.
- **dcap-qvl**. Phala Network rewrote the reference C++ implementation in Rust, which was outside the scope of this review. A dedicated audit would help ensure its correctness. See Incomplete TD Under Debug Checks, Underdocumented Root of Trust and Vendored Attestation Code, and Lack Of Revocation Checks In Quote Verification Library.

Overview

Architecture Overview

The architecture overview of Dstack is depicted in the following image from their official repository:



CVM (dstack-os). The Confidential Virtual Machine (CVM) serves as the primary virtualized environment within the Intel TDX (Trust Domain Extensions) framework. It runs the guest operating system known as `dstack-os`, which is also referred to as `meta-dstack`. The `meta-dstack` is constructed as a Yocto Project meta layer, designed to build reproducible guest image.

Within the CVM, the guest image includes a Docker runtime that is responsible for launching and managing the containerized workloads. These containers host the main application logic, making the CVM the execution environment for end-user services.

dstack-vmm. The `dstack-vmm` is a service operating on the bare-metal TDX host, responsible for managing the full lifecycle of the CVM. Acting as the Virtual Machine Manager (VMM), it coordinates the creation, configuration, execution, suspension, and termination of CVMs.

dstack-guest-agent. The `dstack-guest-agent` is an in-guest service that runs within the CVM. It is primarily responsible for handling sensitive operations such as container-specific key derivation and servicing remote attestation requests.

dstack-gateway. The `dstack-gateway` functions as a reverse proxy that mediates encrypted communications (via TLS connection) between the CVM and external, public-facing networks.

dstack-kms. The `dstack-kms` component operates as a Key Management Service (KMS) designed to generate, store, and manage cryptographic keys for CVMs.

Intel TDX Runtime Measurements

Intel TDX (Trust Domain Extensions) provides four runtime measurement registers: `RTMR0`, `RTMR1`, `RTMR2`, and `RTMR3`. These registers function similarly to TPM PCRs: they are append-only (i.e. they can be extended but not reset) and are designed to reflect the integrity of the runtime environment.

At runtime, both the firmware and guest software can extend these RTMRs by hashing data and appending it to the current register value. This process, called measuring, ensures that any change in the system's state results in different measurement values.

The first three registers, RTMR0–RTMR2, are used for predefined components in the boot process. RTMR3 is available for applications to use during runtime (e.g., to log and prove that specific data or events occurred). In the dstack OS, RTMR3 is used to register information about the CVM, such as the `compose-hash` (a hash of the `app_compose.json` which the container stack). This is done before keys for persistent data and TLS communication are generated or received from the KMS. The idea is roughly to guarantee through RTMR0-RTMR2 that the expected dstack OS is running. RTMR3, in contrast, captures measurements related to the Docker application stack running on top of that OS.

The expected values of the registers can be computed on a machine without TDX support using the `dstack-mr` tool, a client-side reimplementation of the measurement logic. It can replay the measurement process and help verify expected hashes.

In addition to these registers, the `MRTD` (Measured Root for TDX) is initialized by the TDX module itself and contains measurements of the early firmware loaded by the hypervisor (e.g., OVMF). It's immutable after launch and is critical for root trust. This is sometimes referred to as `TDMR` in earlier Intel documents.

The QEMU command below shows the components involved in booting a TDX guest:

```
/usr/bin/qemu-system-x86_64 \
    ...
    -bios ovmf.fd \
    -kernel bzImage \
    -initrd initramfs.cpio.gz \
    -drive file=rootfs.img.verity,format=raw,readonly=on \
    ...
    -append "console=ttyS0 ... dstack.rootfs_hash=... dstack.rootfs_size=..."
```

Here we can see how each parameter contributes to computing the registers:

- OVMF (`-bios`): Measured by the firmware and contributes to **RTMR0**.
- Kernel and Initramfs (`-kernel`, `-initrd`): Measured into **RTMR1**.
- Kernel command-line (`-append`): Includes critical fields such as the hash of the root filesystem and is measured into **RTMR2**.
- Application Events: Applications can call into the guest agent (via RA RPC) to emit custom events that extend **RTMR3**.

The fundamental applications events recorded by `dstack-util/src/system_setup.rs` are:

```
extend_rtmr3("system-preparing", &[])?;
extend_rtmr3("app-id", &instance_info.app_id)?;
extend_rtmr3("compose-hash", &compose_hash)?;
extend_rtmr3("instance-id", &instance_id)?;
extend_rtmr3("boot-mr-done", &[])?;
```

These are added in order to mark distinct phases of runtime initialization before requesting/generating keys.

Together, this strategy aims at ensuring that:

- Only a trusted kernel with trusted OVMF is used.
- The kernel parameters are not modified by a malicious host.
- Only a trusted `rootfs` image is mounted.
- Runtime integrity can be remotely verified.
- RA-TLS certificates can bind to an attested software state (see the RA-TLS section for more details)

In sum, proper measurement is the foundation of trust in TDX-based confidential computing environments.

RA-TLS

A key feature of the dstack framework is the ability of an application running inside a CVM to prove it is running a trusted dstack OS image and a particular Docker application. In order to do this using the TDX platform, it is crucial to obtain a fresh *quote*, which is a hardware-signed structure produced by the TDX module. This quote contains the current values of the measurement registers (MRTD and RTMR0–RTMR3), as well as a 64-byte `report_data` payload supplied by the caller. Because the CPU cryptographically signs those register values, any client holding Intel's public key can verify that (a) the measurements truly came from an untampered TDX guest and (b) the included `report_data` is exactly what the guest intended to prove.

To build a secure channel with an application running in the CVM, dstack uses a TLS certificate that carries remote-attestation evidence. The idea—borrowed from SGX RA-TLS is to bind the TLS public key to a given attestation by requesting a quote whose `report_data` is the application's public key (or a hash of it). Dstack's implementation lives in `ra-tls/src/cert.rs` inside the `generate_ra_cert` function:

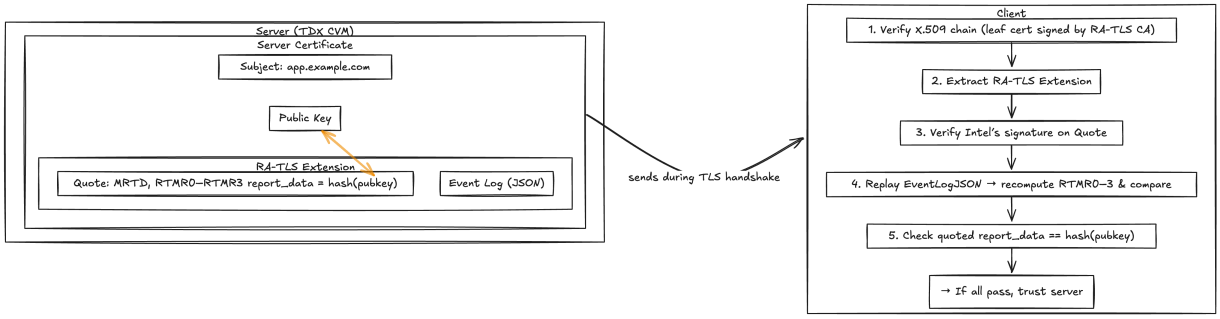
```
let report_data = QuoteContentType::RaTlsCert.to_report_data(&pubkey);
let (_, quote) = get_quote(&report_data, None).context("Failed to get quote")?;
let event_logs = read_event_logs().context("Failed to read event logs")?;
let event_log = serde_json::to_vec(&event_logs).context("Failed to serialize event
logs")?;
let req = CertRequest::builder()
    .subject("RA-TLS TEMP Cert")
    .quote(&quote)
    .event_log(&event_log)
    .key(&key)
    .build();
```

First, the code computes `report_data` by embedding the freshly generated TLS public key. Then it calls `get_quote(report_data)` to receive a signed quote over MRTD and RTMR0–3 plus that exact `report_data`. It also reads the runtime event log (JSON-serializable) so a verifier can recompute RTMR0–3. Finally, it builds a certificate request that includes the new private key, the raw quote blob, and the serialized event log. The CA signs this request, producing a leaf certificate that contains a custom extension with both the quote and the event log.

Concretely the extensions used for the TLS certificate are:

```rust
/// OID for the SGX/TDX quote extension.
pub const PHALA_RATLS_QUOTE: &[u64] = &[1, 3, 6, 1, 4, 1, 62397, 1, 1];
/// OID for the TDX event log extension.
pub const PHALA_RATLS_EVENT_LOG: &[u64] = &[1, 3, 6, 1, 4, 1, 62397, 1, 2];
/// OID for the TDX app ID extension.
pub const PHALA_RATLS_APP_ID: &[u64] = &[1, 3, 6, 1, 4, 1, 62397, 1, 3];
/// OID for Special Certificate Usage.
pub const PHALA_RATLS_CERT_USAGE: &[u64] = &[1, 3, 6, 1, 4, 1, 62397, 1, 4];
```

The resulting certificate can be presented by the application during the TLS handshake. A remote client then performs these checks in sequence: verify the X.509 signature chain (ensuring the leaf cert was issued by the trusted RA-TLS CA), extract the embedded quote and event log, verify Intel's signature on the quote (confirming the quoted measurements came from a genuine TDX CVM), replay the JSON event log to recompute RTMR0–3 and compare those values to what the quote claims, and finally check that the quoted `report_data` exactly matches the certificate's public key. If and only if all of these checks succeed can the client be confident that "this TLS endpoint is running on an untampered Dstack OS image inside a genuine TDX guest, and its TLS key is freshly generated inside that enclave."



Binding the TLS public key into `report_data` is vital. Without that step, an attacker could replay a valid quote (which signs some arbitrary `report_data`) inside a new certificate using a different keypair, tricking clients into believing the new key was attested. Moreover, by integrating attestation into standard TLS, one can build a secure channel to the TDX CVM without developing an ad-hoc protocol.

RA-RPC

RA RPC is the internal gRPC-style interface exposed by the Dstack guest agent inside a CVM. It extends traditional remote procedure calls with attestation information, by securing the prpc channel with RA TLS. The guest agent uses RA RPC to allow containerized applications to perform attestation related operations without needing direct access to TDX hardware. For instance through `GetTlsKey, DeriveKey, GetQuote` and `EmitEvent` applications can request new cryptographic key material whose generation is bound to specific TDX measurements, obtain fresh TDX quotes (including the event log needed to replay RTMR extensions), and extend the RTMR3 register with application-defined events. All of these requests and responses are marshalled via prpc and carry the necessary payloads (e.g., TLS public keys or application data) so that the dockerized applications can verify the authenticity and integrity of the TDX guest state.

With the information obtained from the guest agent via RA RPC, apps within the CVM can establish secure channels (via RA-TLS certificates) and prove to remote clients that they are running on an unmodified, correctly measured Dstack OS image. In sum, RA RPC provides a programmatic API for leveraging TDX attestation and key derivation directly from application code running in standard Docker containers, via the guest agent.

Deployment

The deployment of a new dstack instance is performed by the VMM server. Since the service acts as a bridge between the CVM and the application developer, the security assumption is that this service is untrusted. The application developer can interacts with the VMM server from the VMM client either via browser or CLI.

The application developer needs the following configuration to deploy the application:

- **App compose**: stored as `app-compose.json`
- **Instance information**: stored as `.instance-info`
- **System configuration**: stored as `.sys-config.json`
- **Environment variables**: stored as `.encrypted-env` in encrypted form
- **Application-specific configuration**: stored as `.user-config`

These configurations will be temporarily stored in a shared folder on the host server. When the CVM starts, it will mount this folder and load the configurations into the CVM.

Once the instance is created, the user can manage its lifecycle, including starting, shutting down, terminating, or updating its configuration.

Application compose

This is the main configuration of the application that consists of the following data structure in JSON format:

```
manifest_version: integer               # Version (currently default to "2")
name: string                            # Name of the instance
runner: string                          # Name of the runner (currently default to
"docker-compose")
docker_compose_file: string             # YAML string representing docker-compose config
docker_config: object                   # Additional docker settings (currently empty)
kms_enabled: boolean                    # Enable/disable KMS
gateway_enabled: boolean                # Enable/disable gateway
public_logs: boolean                    # Whether logs are publicly visible
public_sysinfo: boolean                 # Whether system info is public
public_tcbinfo: boolean                 # Whether TCB info is public
local_key_provider_enabled: boolean     # Use a local key provider
allowed_envs: array of string           # List of allowed environment variable names
no_instance_id: boolean                 # Disable instance ID generation
secure_time: boolean                    # Whether secure time is enabled
prelaunch_script: string                # Prelaunch bash script that runs before
starting containers
```

By default, the SHA256 digest of this compose content will become the application ID (`app_id`). Note that after created, any update to the application compose will not change the `app_id`.

Instance information

This is the metadata information that describe the application instance information with the following data:

- `app_id`: The application ID, which by default is determined by the SHA256 digest of the `app-compose.json` (truncated to the first 20 bytes)
- `instance_id`: The instance ID, which by default is determined by the SHA256 digest of the `instance_id_seed || app_id` (truncated to the first 20 bytes). This value is empty if the `no_instance_id` in the `app-compose.json` is `true`.

- `instance_id_seed`: The random seed that determine the instance ID.
- `bootstraped`: The boolean value whether the instance has been initialized or not.

Note that these values are generated runtime inside the CVM when system setup is running (see system setup - stage0).

System configuration

This is system configuration that determines the external config of the VM with the following data structure in JSON format:

```
kms_urls: array of string       # List of URL of the KMS services
gateway_urls: array of string   # List of URL of the gateway services
pccs_url: string                # URL of the PCCS service
docker_registry: string         # URL of the docker registry
host_api_url: string            # VSOCK URL of host API
vm_config: string               # JSON string of the VM configuration containing
os_image_hash, cpu_count, and memory_size
```

All values here except the `cpu_count` and `memory_size` of the `vm_config` are defined by the VMM server and cannot be defined from the client.

Environment variables

Dstack employs encrypted environment variables to facilitate the app developer to load secret configurable values into the CVM. Since, these variables need to be stored temporary in the host server before being loaded into the CVM, the content needs to be encrypted so that the confidentiality is not broken by the host server.

The workflow of the encryption is as follows:

- The App developer specify all the envs needed in the `app-compose.json` via the VMM client (Web UI or CLI)
- Before being sent to the VMM server:
    - The VMM client fetches the encryption public key of the App by making RPC call to the KMS by specifying the `app_id`
    - The KMS responses with the public key along with the ECDSA k256 signature
    - The VMM client can verify the signature to verify that the signer is trusted and the resulting encryption public key is legitimate and trusted
    - With the encryption public key, the VMM client does the following:
    - Converts the given environment variables to JSON bytes
    - Generates an ephemeral X25519 key pair
    - Computes a shared secret using this ephemeral private key and the encryption public key
    - Uses the shared key directly as the 32-byte key for AES-GCM
    - Encrypts the JSON string with AES-GCM using a randomly generated IV
    - The final resulting encrypted value is: `ephemeral public key || IV || ciphertext`
- When the app developer deploys the App, the client will send all the required configuration along with this encrypted value to the VMM server and stored as `.encrypted-env` in the shared host directory

- The CVM can later retrieves the envs by does the following:
    - The CVM requests the app keys directly from the KMS, giving the `env_crypt_key` that is equivalent to the private key of the encryption public key
    - The CVM derives the shared secret using the ephemeral public key that is attached in the encrypted value via X25519 key exchange
    - The CVM performs AES-GCM decryption of the ciphertext using the derived shared secret, resulting the JSON bytes back
    - The CVM parses the JSON and will only store the variable keys that are listed in the `allowed_envs` in the `app-compose.json`. It also performs basic check using regex to verify the validity of the value.
    - The final result is transformed into env-formatted file and stored in the `/dstack/.host-shared/.decrypted-env`, which then later will be loaded using `app-compose.service` to become system-wide environment variables.

Application-specific configuration

This is an optional application-specific configuration that can be accessed by the application inside the docker container. It will be stored in the `/dstack/.user-config`.

CVM Runtime Workflow

When the CVM instance is launched, it follows a deterministic sequence of operations to initialize, verify, and prepare the environment for secure execution of the user application.

Booting

The VMM server initiates the CVM using QEMU with a set of predefined parameters described in the TDX Runtime Measurements. The system hardware and boot environment are initialized using the firmware specified in the `ovmf.fd` file. Subsequently, the kernel image (`bzImage`) is loaded by OVMF.

Upon kernel execution, it extracts and runs the initial userland defined in `initramfs.cpio.gz`. This initramfs is responsible for mounting the root filesystem (`rootfs.img.verity`) and verifying its integrity using **dm-verity**, based on the rootfs hash provided in the filename. The kernel command-line arguments passed via `-append` are interpreted and executed by the `/init` script.

System Setup

Once the operating system has fully booted, `systemd` triggers the `dstack-prepare` service to set up the system prior to application startup. This setup is executed by the `dstack-util setup` tool, which runs in two main phases: `stage0` and `stage1`.

Stage 0

This phase primarily focuses on preparing the CVM's file system and performing application-info measurements.

1. Copy and read shared files

Configuration files generated during the deployment phase are mounted into the CVM from the shared host folder as read-only, then copied into the `/dstack` directory.

All files, except `.encrypted-env` and `.user-config` are deserialized from JSON and parsed into internal structures:

- `app-compose.json` → `app_compose`
- `.instance-info` → `instance_info`
- `.sys-config.json` → `sys_config`

2. Measure app info

- The SHA256 hash of `app_compose` is computed as `compose_hash`. If the `mr_config_id` is defined (non-zero) in the initial attestation quote, this hash must match it. A mismatch results in setup failure.
- Empty values in `instance_info` are generated using mechanisms defined in instance information.
- Measurements are sequentially extended into **RTMR3** with the following event sequence:
  - `system-preparing` → zero value
  - `app-id` → `instance_info.app_id`
  - `compose-hash` → `compose_hash`
  - `instance-id` → `instance_info.instance_id`
  - `boot-mr-done` → zero value

3. Request app keys

Application-specific keys are requested from a KMS or a local Intel SGX key provisioner (if KMS is not enabled). The retrieved keys include:

- **disk_crypt_key**: Used for full disk encryption
- **env_crypt_key**: X25519 private key for decrypting environment variables
- **k256_key**: ECDSA private key for digital signatures
- **k256_signature**: ECDSA signature signed by the root k256 key
- **gateway_app_id**: Application ID used by the gateway reverse proxy
- **ca_cert**: TLS CA certificate for secure HTTPS communication
- **key_provider**: Details about the key provider config (KMS or local)

These values are saved in `/dstack/.appkeys.json`.

4. Mount data disk

To protect persistent storage from host access, dstack uses **dm-crypt LUKS2** full disk encryption. The `disk_crypt_key` is used to encrypt and decrypt the data volume.

If this is the first initialization of the CVM, the disk is formatted using:

```
echo -n $disk_crypt_key | cryptsetup luksFormat --type luks2 --cipher aes-xts-plain64 --
pbkdf pbkdf2 -d- /dev/vdb dstack_data_disk
```

Then, the encrypted disk will be opened and mounted using **zfs** file system.

After successful completion, the setup logs a final `system-ready` event (with zero value) into RTMR3, indicating that the system is fully initialized.

Stage 1

1. Unseal encrypted envs

The encrypted environment variables are decrypted using the env_crypt_key, as described in environment variables, and stored in `/dstack/.host-shared/.decrypted-env`.

2. Setup guest agent

The setup writes the configuration file (stored in `/dstack/agent.json`) of the **dstack-guest-agent** service with the following data based on the user configuration in `app_compose` and `sys_config`:

```
{
    "default": {
        "core": {
            "app_name": app_compose.name,
            "public_logs": app_compose.public_logs,
            "public_sysinfo": app_compose.public_sysinfo,
            "pccs_url": sys_config.pccs_url,
            "data_disks": mount_point,
        }
    }
}
```

3. Setup dstack gateway

In order to communicate with the network outside the CVM securely, it will set up a secure connection using WireGuard. In this process, it will set up the key and certificate of the WireGuard and then try to register the CVM to the defined gateway URLs in the configuration. Based on the gateway response, it constructs the WireGuard config and iptables firewall rules.

4. Setup docker registry

If supplied, the setup will try to use a custom Docker registry URL specified in the `app_compose` config (along with Docker credentials such as username and token). If present, it will update the Docker configuration by updating the JSON config in `/etc/docker/daemon.json`. Later, Docker will use the specified credentials and registry mirror when pulling images.

Application compose

After everything is set up, the CVM will start `app-compose.service`, which mainly tries to start the Docker service with the main application.

First, it reads all config from the `.sys-config.json` file and sets `PCCS_URL` as an environment variable (if present). Then, it will execute the pre-launch script that is defined in the `app-compose.json` with the `source` command. After the pre-launch script finishes, it removes all orphan containers and restarts the Docker service, then launches the Docker container using the compose file specified in the `app-compose.json` in detached mode.

Guest agent service

After the main application is launched via Docker container, the **dstack-guest-agent** service is started using the configuration from `agent.json`. Then, the end user can start interacting with the application.

Meta-dstack

Overview of the build

The meta-dstack repository uses Yocto to build a UEFI firmware image, a kernel image, an initram filesystem, and root filesystems (one for development and one for production environments):

The main entry point is `reprobuild.sh` which launches a Docker container and use it to run the repo's `build.sh` script. The Docker container makes it easier to create a reproducible environment.

The command ran is `build.sh guest ./bb-build` with environment variable `DSTACK_TAR_RELEASE=1`. It then produces artifacts that it moves to the `/dist` folder on the host.

The build uses a default config file (unless a `build-config.sh` is created):

```
# DNS domain of kms rpc and dstack-gateway rpc
# *.1022.kvin.wang resolves to 10.0.2.2 which is the IP of the host system
# from CVMs point of view
KMS_DOMAIN=kms.1022.kvin.wang
GATEWAY_DOMAIN=gateway.1022.kvin.wang

# CIDs allocated to VMs start from this number of type unsigned int32
VMM_CID_POOL_START=$CID_POOL_START
# CID pool size
VMM_CID_POOL_SIZE=1000

VMM_RPC_LISTEN_PORT=$BASE_PORT
# Whether port mapping from host to CVM is allowed
VMM_PORT_MAPPING_ENABLED=true
# Host API configuration, type of uint32
VMM_VSOCK_LISTEN_PORT=$BASE_PORT

KMS_RPC_LISTEN_PORT=$(($BASE_PORT + 1))
GATEWAY_RPC_LISTEN_PORT=$(($BASE_PORT + 2))

GATEWAY_WG_INTERFACE=dgw-$USER
GATEWAY_WG_LISTEN_PORT=$(($BASE_PORT + 3))
GATEWAY_WG_IP=10.$SUBNET_INDEX.3.1
GATEWAY_SERVE_PORT=$(($BASE_PORT + 4))
GATEWAY_CERT=
GATEWAY_KEY=

BIND_PUBLIC_IP=0.0.0.0

GATEWAY_PUBLIC_DOMAIN=app.kvin.wang

# for certbot
CERTBOT_ENABLED=false
CF_API_TOKEN=
CF_ZONE_ID=
ACME_URL=https://acme-staging-v02.api.letsencrypt.org/directory
```

The `build.sh` scripts eventually runs the repo's `Makefile` in this way:

```
if [ -z "$BBPATH" ]; then
    source $SCRIPT_DIR/dev-setup $1
fi
make -C $META_DIR dist DIST_DIR=$IMAGES_DIR BB_BUILD_DIR=${BBPATH}
```

It first check whether `BBPATH` is set or not. If not set, it will source a script called `./dev-setup` that will initialize Bitbake environment and sets `BBPATH`.

Later, `BBPATH` is passed as `BB_BUILD_DIR` to the `make` command along with the `META_DIR` and `IMAGES_DIR` as repo path and path to an `images` subfolder, respectively.

`./dev-setup` simply adds all the layers to the build folder (`./bb-build/conf/bblayers.conf` in this case) by calling `bitbake-layers add-layer` :

```
LAYERS="$THIS_DIR/meta-confidential-compute \
    $THIS_DIR/meta-openembedded/meta-oe \
    $THIS_DIR/meta-openembedded/meta-python \
    $THIS_DIR/meta-openembedded/meta-networking \
    $THIS_DIR/meta-openembedded/meta-filesystems \
    $THIS_DIR/meta-virtualization \
    $THIS_DIR/meta-rust-bin \
    $THIS_DIR/meta-security \
    $THIS_DIR/meta-dstack"

# needed to initialize bitbake binaries
source $OE_INIT $BUILD_DIR

bitbake-layers add-layer $LAYERS
```

(Note that here the default `bb-build/conf/local.conf` is used.)

The following python scripts are added to `PATH`. Those scripts do not end up in the BitBake image, but are useful to quickly deploy and interact with recently created images.

```
scripts/bin
├── dstack -> dstack.py
├── dstack.py
├── host_api.py
└── lsproc.py
```

The `Makefile` is relatively small, calling `bitbake` to build five images:

```
export BB_BUILD_DIR
export DIST_DIR

DIST_NAMES ?= dstack dstack-dev
ROOTFS_IMAGE_NAMES = $(addsuffix -rootfs,${DIST_NAMES})

all: dist

dist: images
    $(foreach dist_name,${DIST_NAMES},./mkimage.sh --dist-name $(dist_name);)

images:
    bitbake virtual/kernel dstack-initramfs dstack-ovmf $(ROOTFS_IMAGE_NAMES)
```

As a result, five images are built:

- `dstack-ovmf` : The UEFI firmware used to boot the kernel.
- `virtual/kernel` : The linux kernel tweaked to support TDX.

- `dstack-initramfs`: The first (and temporary) filesystem that the kernel will mount in volatile memory.
- Two rootfs images, `dstack-rootfs` for production and `dstack-dev-rootfs` for development

The production `dstack-rootfs` image is a minimal root filesystem built for production. It strips out shells, SSH servers, getty services, and most debug or development tools, and only contains the bare packages needed to run Dstack guests. The development `dstack-dev-rootfs` image inherits everything in the prod image plus additional convenience packages (e.g. SSH/getty support, shells, editors, networking and debugging utilities) and so developers can quickly SSH in, edit files, and troubleshoot without rebuilding the image.

Layers

The following is a list of Yocto layers included in the build environment, as shown by the `bitbake-layers show-layers` command. The priority column indicates the precedence of the layer when multiple layers provide the same metadata.

LAYER	PATH	PRIORITY
core	meta-dstack/poky/meta	5
yocto	meta-dstack/poky/meta-poky	5
yoctobsp	meta-dstack/poky/meta-yocto-bsp	5
confidential-compute	meta-dstack/meta-confidential-compute	20
openembedded-layer	meta-dstack/meta-openembedded/meta-oe	5
meta-python	meta-dstack/meta-openembedded/meta-python	5
networking-layer	meta-dstack/meta-openembedded/meta-networking	5
filesystems-layer	meta-dstack/meta-openembedded/meta-filesystems	5
virtualization-layer	meta-dstack/meta-virtualization	8
rust-bin-layer	meta-dstack/meta-rust-bin	7
security	meta-dstack/meta-security	8
dstack	meta-dstack/meta-dstack	20

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME	RISK
#00	meta-dstack/ovmf	VMM is Currently Trusted in OVMF Build	High
#01	meta-dstack recipes	Terminal Binaries Present in Production Dstack Image	Medium
#02	dstack-util system setup	Host Can Pass Symbolic Links To Shared Folder With Guest	Medium
#03	dstack-util	Env Injection via Unauthenticated Shared Files	Medium
#04	app-compose service	Pre-Launcher Code Can Be Used To Leak Secrets on Default KMS	Medium
#05	meta-dstack	qemu-guest-agent is Present in Production	Medium
#06	dcap-qvl	Incomplete TD Under Debug Checks	Medium
#07	app-compose service	Unchecked Container Image Digest	Medium
#08	guest-agent	Unrestricted Exposure of stdout/stderr From CVM Docker Containers	Low
#09	dstack-util	Incomplete Measurement of CVM Configuration Files	Low
#0a	*	Underdocumented Root of Trust and Vendored Attestation Code	Low
#0b	dcap-qvl	Lack of Revocation Checks in Quote Verification Library	Low
#0c	meta-dstack	Lack of Documentation on Design and Hardening Decisions in meta-dstack Layer	Informational

ID	COMPONENT	NAME	RISK
#0d	*	Insufficient Guidance for Secure Production Deployment of CVMs	**Informational**

#00 - VMM is Currently Trusted in OVMF Build

Severity: High **Location:** meta-dstack/ovmf

Description. *Trust Domain Virtual Firmware (TDVF)* is a minimal UEFI project by Intel documented in Intel® TDX Virtual Firmware Design Guide. There are two official implementations of a TDVF:

- the main one integrated in *Open Virtual Machine Firmware (OVMF)*. OVMF is the virtual firmware of the EDK II firmware framework (https://github.com/tianocore/edk2)
- a minimal and more recent one called *td-shim* (https://github.com/confidential-containers/td-shim)

dstack makes use of OVMF as initial firmware to boot in the Intel TD. A hash of that firmware is the first measurement produced by TDX in the MRTD measurement register. TDX integration comes in two configurations in OVMF according to the documentation:

> *There are 2 configurations for TDVF.*
> ***Config-A:***
>
> - *Merge the basic TDVF feature to existing **OvmfPkgX64.dsc**. (Align with existing SEV)*
> - *Threat model: VMM is **NOT** out of TCB. (We don't make things worse)*
> - *The OvmfPkgX64.dsc includes SEV/TDX/normal OVMF basic boot capability. [TRUNCATED...]*
>
> ***Config-B:***
>
> - *Add a standalone **IntelTdxX64.dsc** to a TDX specific directory (**OvmfPkg/IntelTdx**) for a full feature TDVF. (Align with existing SEV)*
> - *Threat model: VMM is out of TCB. (We need necessary change to prevent attack from VMM) [TRUNCATED...]*

As one can see, configuration A trusts the VMM whereas configuration B is a full implementation of the TDVF.

Recommendation. Move to the `IntelTdxX64.dsc` platform description file, following the build instructions provided in the README.

Client response. The move to the configuration B was implemented in https://github.com/Dstack-TEE/meta-dstack/pull/7/commits/c9b5b92098d9063ce4e0f6186f03b89b754d0f5f.

#01 - Terminal Binaries Present in Production Dstack Image

Severity: Medium **Location:** meta-dstack recipes

Description. The `meta-dstack` repository contains the Yocto recipes to produce both a development and a production bootable image of the dstack OS. One recipe defines a base image `meta-dstack/recipes-core/images/dstack-rootfs-base.inc` that is inherited by both the production and the development images. The production image is hardened by including the `nologin` configuration:

```
include dstack-rootfs-base.inc
IMAGE_FEATURES += "nologin"
```

This `nologin` configuration triggers a call to a function that disables various terminal related binaries and services to harden it against a malicious host:

```
disable_getty_services() {
    for srv in getty getty-pre; do
        rm -f ${IMAGE_ROOTFS}/etc/systemd/system/${srv}.target
        rm -f ${IMAGE_ROOTFS}/usr/lib/systemd/system/${srv}.target
    done
    for srv in autovt container-getty console-getty getty-generator serial-getty getty;
do
        rm -f ${IMAGE_ROOTFS}/etc/systemd/system/${srv}.service
        rm -f ${IMAGE_ROOTFS}/etc/systemd/system/${srv}@.service
        rm -f ${IMAGE_ROOTFS}/usr/lib/systemd/system/${srv}.service
        rm -f ${IMAGE_ROOTFS}/usr/lib/systemd/system/${srv}@.service
    done
}
```

However not all binaries are captured by this function, in particular the `agetty` binary and the `systemd-getty-generator` are not explicitly removed. The `agetty` binary is the getty implementation that spawns login prompts on serial or virtual consoles, while `systemd-getty-generator` is the systemd component that dynamically generates `getty@.service` units for any detected console devices. We have confirmed their presence in a production CVM by means of a pre-launcher script:

```
#!/bin/sh

echo "===== Serial Getty Testing ====="

echo -n "[*] agetty binary: "
if [ -x /sbin/agetty ]; then
  echo "[!] /sbin/agetty present"
elif [ -x /usr/sbin/agetty ]; then
  echo "[!] /usr/sbin/agetty present"
else
  echo "[✓] no agetty binary found"
fi

echo -n "[*] systemd-getty-generator: "
if [ -x /usr/lib/systemd/system-generators/systemd-getty-generator ]; then
  echo "[!] present"
else
  echo "[✓] not present"
fi

echo -n "[*] getty@ttyS0.service running: "
if systemctl is-active --quiet getty@ttyS0; then
  echo "[!] running"
else
  echo "[✓] not running"
fi

echo "======================================"
```

and confirmed:

```
===== Serial Getty Testing =====
[*] agetty binary:
    [!] /sbin/agetty present
    [!] /usr/sbin/agetty present
[*] systemd-getty-generator: [!] present
[*] getty@ttyS0.service running: [✓] not running
======================================
```

Impact. Leaving `agetty` and `systemd-getty-generator` in a production CVM image increases the risk of exploitation by a malicious host if the `getty` service is ever started. This would potentially enable a host or privileged attacker with QMP access to spawn login prompts over the serial console (e.g., `/dev/ttyS0`) without altering TDX measurements. This would allow an attacker to gain root shell post-boot, bypassing attestation. By default the service does not start and no direct QMP-only trigger is known, but an attacker could exploit other vulnerabilities in a crafted chain to launch the console login.

Recommendation. Extend the `disable_getty_services()` function in `dstack-rootfs-base.inc` to also remove the `agetty` binary (`/sbin/agetty`, `/usr/sbin/agetty`) and the generator executable (`/usr/lib/systemd/system-generators/systemd-getty-generator`). Verify in the production build that no `agetty` or getty-generator files remain, and that `getty@ttyS0.service` cannot be started.

Client response. Client has acknowledged the issue and further hardened the images in PR https://github.com/Dstack-TEE/meta-dstack/pull/7.

#02 - Host Can Pass Symbolic Links To Shared Folder With Guest

Severity: Medium **Location:** dstack-util system setup

Description. While setting up a CVM guest after booting, `dstack-util/src/system_setup.rs` uses `fs_err::copy` to copy the files on the shared folder from the host to the guest. This is executed on the guest side. The copy operation happens immediately after the 9p `host-shared` mount.

```
let copy = |src: &str, max_size: u64, ignore_missing: bool| -> Result<()> {
    let src_path = host_shared_dir.join(src);
    let dst_path = host_shared_copy_dir.join(src);
    if !src_path.exists() {
        if ignore_missing {
            return Ok(());
        }
        bail!("Source file {src} does not exist");
    }
    let src_size = src_path.metadata()?.len();
    if src_size > max_size {
        bail!("Source file {src} is too large, max size is {max_size} bytes");
    }
    fs_err::copy(src_path, dst_path)?;
    Ok(())
};
```

Here `fs_err` is wrapper around `std::fs::copy`, and so it inherits the standard-library behavior of dereferencing any symbolic link and copying the link's target bytes. This lets a malicious host replace, say, `app-compose.json` with a symlink to `/proc/kcore` or `/proc/self/mem`, causing the guest to copy live kernel memory into its staging directory.

Impact. At this point of the system setup the guest has not yet received keys from the KMS or decrypted persistent files. Moreover the target file is copied to a directory inside the guest so exfiltration to the host is not trivial. Such an attack will also likely cause a boot error since it will overwrite some expected well formed file (usually a JSON file). However this vulnerability increases the likelihood that some contents of the confidential file target leak through `stderr` or public logs to the host. At this phase in booting `/proc/kcore` and `/proc/self/mem` contain runtime secrets such as TDX injected system randomness and ASLR randomness, so those files hold non-deterministic secrets. Even if the copy files (files are large), some bytes may leak into error logs.

Recommendation. The code should defensively assume the host maybe malicious and detect and reject a symbolic link. Just checking the link before copying it may be still misused by the host by a race condition so the check should be robust against a concurrent attack.

Client response. Client has acknowledged the issue and issued a fix in commit `95814f4`.

#03 - Env Injection via Unauthenticated Shared Files

Severity: Medium **Location:** dstack-util

Description. During the deployment process, the VMM stores several temporary, user-configurable files as inputs to the Confidential VM (CVM):

- `.encrypted-env`
- `.instance-info`
- `.sys-config.json`
- `.user-config`
- `app-compose.json`

However, since there is no cryptographic integrity protection or authentication mechanism in place, any entity with access to the host server (including the cloud provider) can read and modify these files prior to CVM boot. Note that the shared files other than `.encrypted-env` are public and thus can be detected by the TDX measurement (see also the related finding).

Although `.encrypted-env` is in the encrypted form (as explained in here), the attacker can still modify this value as long as they know the encryption public key of the application, which is trivially possible to be obtained by using the VMM instance to fetch it from the KMS.

Impact. The impact of this issue depends on how the application interprets and uses the values from `.encrypted-env`. In many scenarios, altering environment variables can subtly or significantly affect application behavior, potentially undermining integrity of the CVM.

Note that a malicious host can't just load arbitrary environment variables (ie. system wide variables such as `LD_PRELOAD`, `PATH`, etc..) because the `app-compose.json` file enforces the environment variables that can be set, and the boot sequence will check that the decrypted envs only contain the keys that are listed in the `allowed_envs` field. For example, this `app-compose.json` only allows a malicious host to alter the `API_KEY` environment variable:

```
{
  "manifest_version": 2,
  "name": "kvin-nb",
  "runner": "docker-compose",
  "docker_compose_file": "services:\n  jupyter:\n    image: quay.io/jupyter/base-
notebook\n    user: root\n    environment:\n      - GRANT_SUDO=yes\n    ports:\n      -
\"8888:8888\"\n    volumes:\n      - /:/host/\n      -
/var/run/tappd.sock:/var/run/tappd.sock\n      -
/var/run/dstack.sock:/var/run/dstack.sock\n    logging:\n      driver: journald\n
options:\n        tag: jupyter-notebook\n",
  "docker_config": {},
  "kms_enabled": true,
  "tproxy_enabled": true,
  "public_logs": true,
  "public_sysinfo": true,
  "public_tcbinfo": false,
  "local_key_provider_enabled": false,
  "allowed_envs": ["API_KEY"],
  "no_instance_id": false
}
```

Recommendation. There are two possible approaches to mitigate this:

1. General approach: Add an authentication layer to the encrypted environment variables, such as signing the ciphertext with digital signatures
2. Application-specific approach: Delegate verification to the application itself, requiring the application developer to manually validate the integrity of the decrypted values within the CVM.

Client response. The client is aware of the issue and has expanded the documentation, including a security guide in commit `d007d0c`. The client also plans to introduce `LAUNCH_TOKEN` mechanism and built-in authentication mechanism to protect the integrity of the env content.

#04 - Pre-Launcher Code Can Be Used To Leak Secrets on Default KMS

Severity: Medium **Location:** app-compose service

Description. When using the testing setup described in the project's README, a KMS is automatically launched on the host which is by default not configured with an upgradeability policy. Therefore, once a CVM is deployed once by the host with the KMS setup, it instantiates keys dependent on the initial `instance_id`, which is persistent on the host for upgradability. The idea is that future changes to the `app-compose.json` receive the same keys from the KMS upon booting if the upgrade is whitelisted. However the default KMS in this setup will always return the original keys regardless of the `compose_hash` that fingerprints the update. The app-compose JSON allows users to define a pre-launch script. This entire pre-launch script is already folded into `compose_hash` (and therefore into `RTMR-3`), but the demo KMS never checks that hash during attestation, so any change to the script slips through unchanged key-release logic.

Impact. Given that the upgrade is not checked against a policy, a malicious host can upload a malicious pre-launcher script which will be executed by the app-compose service `basefiles/app-compose.sh` before starting the docker service inside the CVM:

```
if [ $(jq 'has("pre_launch_script")' app-compose.json) == true ]; then
    echo "Running pre-launch script"
    dstack-util notify-host -e "boot.progress" -d "pre-launch" || true
    source <(jq -r '.pre_launch_script' app-compose.json)
fi
```

The injected script executes inside the TD as UID 0 with the full Linux capability set, so it can read any file and open outbound sockets. Because the KMS policy never validates RTMR-3, the TD's quote still verifies and the KMS releases the long-term keys—despite the script change. The script can then leak persistent secrets via `stderr`, `dstack-utils notify-host` or a network exfiltration channel for example.

Recommendation. The default setup should require a policy or define a clear guideline for the `compose_hash` to prevent users that follow it (and do not set up a smart contract to authorize specific upgrades) to be vulnerable to this attack scenario. In general given that upgrades carry the risk of a misconfiguration on the KMS side, it is advised to consider defining a white-list or DSL for the specific operations the pre-launcher can do, as well as limiting the privileges in which this script is executed.

Client response. Client has acknowledged that extending the documentation for this scenario is meaningful and have done so in commit `ea1a64f`.

#05 - qemu-guest-agent is Present in Production

Severity: Medium **Location:** meta-dstack

Description. The meta-dstack yocto layer contains two recipes to build the rootfs used by the dstack OS, a recipe for development and one for production. Both inherit the same base `dstack-rootfs-base.inc` which lists the packages to be installed on both filesystems:

```
inherit core-image

IMAGE_BASENAME = "${PN}"

IMAGE_INSTALL = "\
    ${VIRTUAL-RUNTIME_base-utils} \
    ${ROOTFS_BOOTSTRAP_INSTALL} \
    base-files \
    base-passwd \
    systemd \
    netbase \
    iptables \
    docker-moby \
    docker-compose \
    tdx-guest-ko \
    dstack-guest \
    wireguard-tools \
    cryptsetup \
    curl \
    jq \
    chrony \
    chronyc \
    qemu-guest-agent \
    dstack-zfs \
    kernel-module-tun \
"
```

Notice `qemu-guest-agent`, a package of qemu's guest agent (https://wiki.qemu.org/Features/GuestAgent) which allows a host to issue JSON commands on a guest over a virtio-serial channel.

The package is defined in poky (`poky/meta/recipes-devtools/qemu/qemu.inc`):

```
# TRUNCATED...
PACKAGES =+ "${PN}-guest-agent"
SUMMARY:${PN}-guest-agent = "QEMU guest agent"
FILES:${PN}-guest-agent += " \
    ${bindir}/qemu-ga \
    ${sysconfdir}/udev/rules.d/60-qemu-guest-agent.rules \
    ${sysconfdir}/init.d/qemu-guest-agent \
    ${systemd_unitdir}/system/qemu-guest-agent.service \
"

INITSCRIPT_PACKAGES = "${PN}-guest-agent"
INITSCRIPT_NAME:${PN}-guest-agent = "qemu-guest-agent"
INITSCRIPT_PARAMS:${PN}-guest-agent = "defaults"

SYSTEMD_PACKAGES = "${PN}-guest-agent"
SYSTEMD_SERVICE:${PN}-guest-agent = "qemu-guest-agent.service"
```

It relies on the following udev rule (`poky/meta/recipes-devtools/qemu/qemu/qemu-guest-agent.udev`):

```
SUBSYSTEM=="virtio-ports", ATTR{name}=="org.qemu.guest_agent.0", \
  TAG+="systemd", ENV{SYSTEMD_WANTS}="qemu-guest-agent.service"
```

This rule asks systemd to start the qemu-guest-agent service once it detects a new qemu (virtio-ports) device with a specific attribute. If the host attaches such a device it allows them to read/write arbitrary files, run commands, and perform a number of other operations from within the guest.

Impact. A malicious host that manages to start the qemu-guest-agent service can fully compromise the CVM by interacting with the guest agent using the QMP protocol. Plugging such a device and starting the guest agent is easy to do by modifying the qemu launch argument, for instance by adding:

```
-chardev socket,id=qga0,path=/tmp/qga.sock,server,nowait \
-device virtio-serial-pci,id=virtio-serial0 \
-device virtserialport,chardev=qga0,name=org.qemu.guest_agent.0
```

however this impacts the expected runtime measurements. Currently, thanks to the q35 QEMU configuration there are no available PCIe root-ports to which you can hot-plug a virtio-serial port, so any runtime `device_add virtio-serial-pci` will fail with:

```
{"error": {"class": "GenericError", "desc": "Bus 'pcie.0' does not support hotplugging"}}
```

Note that this barrier is brittle: if a `pcie-root-port` is ever added to the base configuration, hot-plugging a virtio-serial channel succeeds and the udev rule will fire. As with the `agetty` finding, any other vulnerability that allows starting the guest-agent service will likewise bypass TDX measurements and let the host fully compromise the CVM.

Recommendation. Only list qemu-guest-agent as a package on the **dev** rootfs recipe (`dstack-rootfs-dev.inc`).

Client response. Client has removed the qemu-guest-agent from the base image on PR https://github.com/Dstack-TEE/meta-dstack/pull/7.

#06 - Incomplete TD Under Debug Checks

Severity: Medium **Location:** dcap-qvl

Description. In the dcap-qvl crate in charge of validating TDX attestations, the following checks are performed:

```
fn validate_tcb(report: &Report) -> Result<()> {
    fn validate_td10(report: &TDReport10) -> Result<()> {
        let is_debug = report.td_attributes[0] & 0x01 != 0;
        if is_debug {
            bail!("Debug mode is not allowed");
        }
        Ok(())
    }
    fn validate_td15(report: &TDReport15) -> Result<()> {
        if report.mr_service_td != [0u8; 48] {
            bail!("Invalid mr service td");
        }
        validate_td10(&report.base)
    }
    fn validate_sgx(report: &EnclaveReport) -> Result<()> {
        let is_debug = report.attributes[0] & 0x02 != 0;
        if is_debug {
            bail!("Debug mode is not allowed");
        }
        Ok(())
    }
    match &report {
        Report::TD15(report) => validate_td15(report),
        Report::TD10(report) => validate_td10(report),
        Report::SgxEnclave(report) => validate_sgx(report),
    }
}
```

According to the TDX DCAP Quoting Library specification, section 2.3.2:

> *Verify that all TD Under Debug flags (i.e., the TDATTIBUTES.TUD field in the TD Quote Body) are set to zero. If any flag is non-zero, the TD should not be trusted and thus should not be provisioned with production secrets.*

The test above is only checking that a single bit is zero, instead of checking the whole byte. Thus, the TD should still not be considered trusted according to the specification.

That being said, the other bits are currently reserved for further used (via a microcode update), not making this currently exploitable.

Recommendation. Check that the full byte of `report.td_attributes[0]` is zero.

Client response. The issue was fixed in https://github.com/Phala-Network/dcap-qvl/commit/53130199282e84b6d094e37b3c370bbbee1d9152 by checking the whole byte. Instead of doing:

```
let is_debug = report.td_attributes[0] & 0x01 != 0;
```

It now does:

```
let is_debug = report.td_attributes[0] != 0;
```

#07 - Unchecked Container Image Digest

Severity: Medium **Location:** app-compose service

Description. The application deployment process references container images in the Docker Compose file without enforcing immutable digests (e.g., `repo/image:latest`). As a result, these image references can change over time. The RTMR3 measurement currently includes only the digest of the compose file and not the digests of the individual container images. Consequently, changes to the actual image content may go undetected.

Impact. An attacker or any entity with access to the image registry can overwrite the image behind a mutable tag (e.g., `latest`) without altering the hash of the compose file. On subsequent boots, the CVM will fetch and run the new image, yet still produce a valid attestation, since the `compose-hash` remains unchanged. This undermines the trust model, as the application behavior may differ despite the attestation appearing valid.

Moreover, the risk is amplified in the case of private container registry and can be chained with the env injection attack by changing the docker credentials in the environment variables silently.

Recommendation. The most ideal mitigation is to include image digests directly in the measurement hash. However, a practical and immediate defense is to reject any container in the docker compose file that is not pinned to an immutable digest (eg., `repo/image@sha256...`). This ensures all images are explicitly specified and verifiable before being pulled.

Client Response. Client is aware of the issue and agreed to mention it in the documentation and official examples, making digest pinning a mandatory requirement for production deployments.

#08 - Unrestricted Exposure of stdout/stderr From CVM Docker Containers

Severity: Low **Location:** guest-agent

Description. By default, the docker logs of a CVM (stdout and stderr of the docker containers) are exposed to the host. While convenient for debugging and logging purposes, this behavior undermines the confidentiality guarantees typically expected from a trusted execution environment. Enclaves are designed to process sensitive data securely and isolate it from a potentially malicious host. Any unfiltered output may inadvertently disclose secrets or facilitate side-channel attacks due to logging the wrong value or displaying too much in an error message.

Recommendation. Disable this feature by default and warn users of the potential risks. Consider disabling this feature entirely to force developers to expose filtered data through a more conscious API.

Client response. The client acknowledged the issue and planned on disabling docker logs by default in production images.

#09 - Incomplete Measurement of CVM Configuration Files

Severity: Low **Location:** dstack-util

Description. During the deployment process, the VMM stores several temporary, user-configurable files as inputs to the Confidential VM (CVM):

- `.encrypted-env`
- `.instance-info`
- `.sys-config.json`
- `.user-config`
- `app-compose.json`

Except for `.encrypted-env`, all of these files should be treated as public inputs to the CVM and included in the RTMR3 measurement to ensure the integrity and correctness of the configuration. However, the current implementation only includes changes to `.instance-info` and `app-compose.json` in the measurement.

Impact. Modifications to `.sys-config.json` and `.user-config` are not reflected in RTMR3 and therefore cannot be detected through remote attestation. This undermines the assurance that the CVM is running with the expected configuration and may open potential vectors for misconfiguration or tampering that go unnoticed.

Recommendation. Ensure that all relevant configuration files are extended into the RTMR3 measurement to maintain the integrity of the CVM deployment.

Client Response. The client has provided additional documentation clarifying why measuring other files is unnecessary in https://github.com/Dstack-TEE/dstack/pull/216/files.

#0a - Underdocumented Root of Trust and Vendored Attestation Code

Severity: Low **Location:** *

Description. In the threat model of dstack, application users as well as application developers (who might not be the hosts) have the option to audit the code (for correctness and backdoors) and ensure that what is deployed is exactly what they're looking at. Currently, this is made harder for a number of reasons, including the lack of documentation around hardcoded constants or vendored code. We offer a few example in this finding.

1. Pinned Intel Root Certificate Not Properly Documented

To verify TDX remote attestations, the dcap-qvl crate is used with a hardcoded root Intel certificate. More specifically, only the subject, subject public key info, and name constraints fields of the certificate are hardcoded (in `src/constants.rs`):

```
pub static DCAP_SERVER_ROOTS: &[webpki::types::TrustAnchor<'static>; 1] =
    &[webpki::types::TrustAnchor {
        subject: webpki::types::Der::from_slice(&[
            49, 26, 48, 24, 06, 03, 85, 04, 03, 12, 17, 73, 110, 116, 101, 108, 32, 83,
71, 88, 32,
            82, 111, 111, 116, 32, 67, 65, 49, 26, 48, 24, 06, 03, 85, 04, 10, 12, 17,
73, 110,
            116, 101, 108, 32, 67, 111, 114, 112, 111, 114, 97, 116, 105, 111, 110, 49,
20, 48, 18,
            06, 03, 85, 04, 07, 12, 11, 83, 97, 110, 116, 97, 32, 67, 108, 97, 114, 97,
49, 11, 48,
            09, 06, 03, 85, 04, 08, 12, 02, 67, 65, 49, 11, 48, 09, 06, 03, 85, 04, 06,
19, 02, 85,
            83,
        ]),
        subject_public_key_info: webpki::types::Der::from_slice(&[
            48, 19, 06, 07, 42, 134, 72, 206, 61, 02, 01, 06, 08, 42, 134, 72, 206, 61,
03, 01, 07,
            03, 66, 00, 04, 11, 169, 196, 192, 192, 200, 97, 147, 163, 254, 35, 214, 176,
44, 218,
            16, 168, 187, 212, 232, 142, 72, 180, 69, 133, 97, 163, 110, 112, 85, 37,
245, 103,
            145, 142, 46, 220, 136, 228, 13, 134, 11, 208, 204, 78, 226, 106, 172, 201,
136, 229,
            05, 169, 83, 85, 140, 69, 63, 107, 09, 04, 174, 115, 148,
        ]),
        name_constraints: None,
    }];
```

As of now, it is unclear to users of dstack how they should verify that this is indeed the Intel root certificate. While users are supposed to use a Provisioning Certificate Caching Service (PCCS), these systems are dynamic and can be compromised or misused. For this reason root CAs are long-term certificates that can and should be pinned by user applications.

We recommend documenting where these values were obtain. Currently, the root CA can be downloaded in pem or der format from the documentation (https://api.portal.trustedservices.intel.com/content/documentation.html#pcs):

Furthermore, we recommend uploading the Intel root certificates to the repository, and asserting in CI that the bytes contained in `src/constants.rs` correctly match the relevant portion of the certificate.

2. Unexplained Vendoring of Intel Attestation Code

The `tdx-attest` (and its inner dependency `tdx-attest-sys`) seems to be taken from https://github.com/intel/SGXDataCenterAttestationPrimitives/blob/main/QuoteGeneration/quote_wrapper/tdx-attest-rs. dstack should document where these vendored files can be found so that one can verify that they match the source.

Client response. The client added comments (https://github.com/Phala-Network/dcap-qvl/commit/07b0605edc7bad924340855accf03085dbc0ebec) and a README file (https://github.com/Dstack-TEE/dstack/commit/85407eab66d7f34db198b198b7a93da9afe2cd04) documenting where the relevant files were taken from to address both of these issues.

#0b - Lack of Revocation Checks in Quote Verification Library

Severity: Low **Location:** dcap-qvl

Description. The dcap-qvl crate is in charge of verifying remote attestations.

We have found that dcap-qvl does not handle certificate revocation checks which the reference library (https://github.com/intel/SGX-TDX-DCAP-QuoteVerificationLibrary) handles.

The documentation Intel® Trust Domain Extensions Data Center Attestation Primitives (Intel® TDX DCAP): Quote Generation Library and Quote Verification Library says that this is one of the baseline tests:

> *To verify a TD Quote, the QVL needs verification collateral, which **at least** includes the root Intel CA certificate of the Intel® CA that signed the PCK Cert and the reference values, **the Certificate Revocation Lists (CRLs)**, and reference values for components in the platform TCB.*
> *TRUNCATED...*
> *With this verification collateral, the QVL performs at least the following checks on the TD Quote:*
>
> - *Check the PCK Cert (signature chain).*
> - ***Check if the PCK Cert is on the CRL.***
> - *Check the verification collaterals' cert signature chain, including PCK Cert Chain, TCB info chain and QE identity chain*
> - ***Check if verification collaterals are on the CRL.***
>
> *TRUNCATED...*

One can manually check, using Intel's API that these CRLs already have a number of revoked certificates:

```
$ wget -qO- "https://api.trustedservices.intel.com/sgx/certification/v4/pckcrl?
ca=platform" | openssl crl -noout -text
Certificate Revocation List (CRL):
        Version 2 (0x1)
    Signature Algorithm: ecdsa-with-SHA256
        Issuer: /CN=Intel SGX PCK Platform CA/O=Intel Corporation/L=Santa
Clara/ST=CA/C=US
        Last Update: Jun 12 22:47:58 2025 GMT
        Next Update: Jul 12 22:47:58 2025 GMT
        CRL extensions:
            X509v3 CRL Number:
                1
            X509v3 Authority Key Identifier:
                keyid:95:6F:5D:CD:BD:1B:E1:E9:40:49:C9:D4:F4:33:CE:01:57:0B:DE:54

Revoked Certificates:
    Serial Number: 6FC34E5023E728923435D61AA4B83C618166AD35
        Revocation Date: Jun 12 22:47:58 2025 GMT
        CRL entry extensions:
            X509v3 CRL Reason Code:
                Key Compromise
    Serial Number: EFAE6E9715FCA13B87E333E8261ED6D990A926AD
        Revocation Date: Jun 12 22:47:58 2025 GMT
        CRL entry extensions:
            X509v3 CRL Reason Code:
                Key Compromise
    TRUNCATED...
```

Recommendation. Use the reference implementation or implement the certificate revocation checks in dcap-qvl. If the later, additionally consider a more thorough security review of dcap-qvl.

Client response. The revocation checks were added in https://github.com/Phala-Network/dcap-qvl/pull/8.

#0c - Lack of Documentation on Design and Hardening Decisions in meta-dstack Layer

Severity: Informational **Location:** meta-dstack

Description. The meta-dstack layer aims at creating a minimaly safe image to boot its CVMs. Its design aims at reducing as much attack surface as possible while providing TDX-aware functionality. For this reason, it is important to understand the trade-offs and the rationale behind the different architectural decisions. We give a few examples in this finding.

1. Yocto development vs stable

The choice of the Yocto linux kernel development recipes (`linux-yocto-dev`) over the stable recipes (`linux-yocto`) should be discussed as there are significant security trade-offs in choosing one over the other (one might argue that living on the edge is dangerous, but some others would argue that not getting the latest patches might lead to a vulnerable image). Note that from discussions with developers it appears that `linux-yocto-dev` was chosen for pragmatic reasons, due to compatibility issues with Intel TDX.

2. TDVF vs td-shim

td-shim (https://github.com/confidential-containers/td-shim) is a more recent addition from Intel that attempts at minimizing the TDVF with a Rust implementation that directly boots the kernel. Currently dstack uses Intel's TDVF implementation integrated in OVMF. While both are legitimate choices, documenting the choice of one over the other allows contributors to debate or even ask for changes. Note that from discussions with the develoeprs it appears that as of now td-shim could not boot the dstack system and was deemed too new to be used.

3. Custom driver vs built-in driver

meta-dstack make uses of the confidential-compute layer (https://github.com/Dstack-TEE/meta-confidential-compute) which correctly configures the firmware and the kernel to support TDX. But the final layer meta-dstack later on sets `CONFIG_TDX_GUEST_DRIVER=n` to use a custom TDX guest driver. From discussions with the developers this decision was made to add the capacibilities of extending RTMR.

4. Correct seeding of randomness

Randomness is used in a number of places in the CVM, from generating certificates and keypairs to providing randomness to docker containers. It is important that randomness does not come directly from the host as the host is untrusted. The official recommendation is to use Intel's RDRAND instruction to provide randomness through a hardware RNG. An application can directly use RDRAND, but the kernel should also make sure that the special files `/dev/random` and `/dev/urandom` are correctly seeded using RDRAND as well as they are the main sources of randomness for many applications.

The kernel is correctly passed the following command line arguments (which are made public as they as kernel command line arguments are measured as part of RTMR2):

```
random.trust_cpu=y
random.trust_bootloader=n
```

Which allows the kernel to trust RNG instructions from the CPU (like RDRAND) and to discard randomness from the bootloader. This is not enough though and the kernel needs to additional make sure that this enough RDRAND entropy is used to seed the system's randomness. This is pointed out in Intel® Trust Domain Extension Linux Guest Kernel Security Specification:

> *The Linux RNG uses timing from interrupts as the default entropy source; this can be a problem for the TDX guest because timing of the interrupts is controlled by the untrusted host/VMM. However, on x86 platforms there is another entropy source that is outside of host/VMM control: RDRAND/RDSEED instructions. The commit x86/coco: Require seeding RNG with RDRAND on CoCo systems ensures that a TDX guest cannot boot unless 256 bits of RDRAND output is mixed into the entropy pool early during the boot process.*

Recommendation. In general, we would recommend that every decision (e.g. every package installed in rootfs) is documented in a design document.

Client response. The client acknowledged the issue and added a design and hardening decision document in https://github.com/Dstack-TEE/dstack/blob/bbb1d0ac4c56e852148ae56f60a96ee781988d7a/docs/design-and-hardening-decisions.md.

#0d - Insufficient Guidance for Secure Production Deployment of CVMs

Severity: Informational **Location:** *

Description. Phala Network's production checklist provides some guidance for operators seeking to deploy Confidential Virtual Machines (CVMs) securely. On the other hand it does not mention how users can assess the security of an open source CVM code (assuming that dstack is secure if correctly used). What follows are a few examples.

Exposed logs. Users should be concerned about docker logs being exposed (see Unrestricted Exposure Of stdout/stderr From CVM Docker Containers)

Strange environment variables. Users should be concerned to see strange environment variables (e.g. `LD_PRELOAD`, `PATH`, `BASH_ENV`, `DOCKER_*`, `COMPOSE_*`) in the allowed environment variables of an `app-compose.json`. This could allow a malicious app developer to alter the correct behavior of scripts ran during the boot sequence. For example, the app-compose systemd unit configuration file `dstack/basefiles/app-compose.service` injects all of these environment variables in the environment before running `/bin/app-compose.sh`:

```
[Unit]
Description=App Compose Service
Wants=docker.service
After=docker.service tboot.service dstack-guest-agent.service

[Service]
Type=oneshot
RemainAfterExit=true
EnvironmentFile=-/dstack/.host-shared/.decrypted-env
WorkingDirectory=/dstack
ExecStart=/bin/app-compose.sh
ExecStop=/bin/docker compose stop
StandardOutput=journal+console
StandardError=journal+console

[Install]
WantedBy=multi-user.target
```

Recommendation. We recommend writing down a security document on how users should approach the security of CVMs, and add a section on intentional backdoors in the threat model section of the whitepaper.

Client response. The client added more documentation in https://github.com/Dstack-TEE/dstack/pull/215.