Authenticated Northstar Container Communication

1 Scope

Containers running in the northstar runtime require an authenticated communication channel to other containers. This document discusses different implementation approaches. The example use case is an MQTT client running in a container connecting and authenticating to an MQTT broker in separate container.

2 Assumptions

1. The host operating system and the northstar runtime are trusted.
2. Containers are correctly signed and their contents are trusted.
3. The network connections between containers happen on localhost only. An attacker cannot inspect or modify such network traffic as the host is trusted (assumption 1).

3 Attack scenarios

3.1 Unauthorized connection from external network

The container network is not separated from the host network. An attacker from outside the network can therefore connect to the MQTT broker and attempt to access its data.

3.2 Compromise of container

Even though containers are signed and trusted at startup (assumption 3), a running container can be compromised during runtime. An attacker that controls a compromised container has access to all secrets of that container and can perform all actions that the container is allowed to perform. In particular, he is able to authenticate with the MQTT broker and access its data.

Additionally, the attacker can attempt to elevate his privileges. For example, he might be able to use the obtained secrets to get access to additional running containers.

4 Security measures

4.1 Global symmetric session secret

4.1.1 Mechanism

During startup, the northstar runtime generates a random symmetric secret. It is shared across all running containers for the current life-cycle of the runtime.

The secret can be used by the MQTT client as a password when authenticating to the MQTT broker running in the other container. Since the secret is shared across containers, the server is able to verify the authentication attempt of the client against his copy of the shared secret.

4.1.2 Mitigating effects

The session secret is life-cycle specific but not container specific. Since the secret is known to the runtime and containers only, an outside attacker has no knowledge of it. This security mechanism prevents unauthorized access by an outside attacker (scenario 3.1) as long as no container is compromised (scenario 3.2). If an attacker compromises a container he is able to obtain the shared secret and can impersonate any container to any other container for the duration of the life-cycle. Attack scenario 3.2 is not prevented by this measure.

4.1.3 Implementation effort

The implementation can be realized by providing a read-only `memfd` device that is mounted into every running container. The container can then read the device to obtain the shared secret.

4.2 Individual tokens

4.2.1 Mechanism

The runtime provides an API to containers to obtain and verify symmetric session tokens. The API to obtain tokens is supposed to be used by clients and requires the following information:

1. Identifier of destination container, i.e. the container a client wants to connect to.

2. Application specific string (e.g. “MQTT”)

The runtime then generates a token that is specific to the provided data and the identifier of the client’s source container. This token can then be used as a password when authenticating with the MQTT broker.

The API to verify tokens is supposed to be used by servers. An MQTT broker that receives a token as a password during an authentication attempt would then use this API to get confirmation by providing the following information:

1. Received token
2. Identifier of source container, i.e. the container the client connects from.
3. Application specific string (e.g. “MQTT”)

The runtime then returns whether the provided token was correctly signed. If it is and the application specific string matches the server’s application, the authentication succeeds.

4.2.2 Mitigating effects

The generated tokens are connection-, application- and life-cycle specific. An outside attacker (scenario 3.1) has no knowledge of these token and is unable to authenticate with the MQTT broker.

An attacker that manages to compromise a running container (scenario 3.2) is able to obtain all generated tokens and use the API to generate new ones. Existing token can be used to impersonate the compromised container to those applications for which the obtained tokens are valid. If the attacker manages to use the runtime API to generate new tokens, he is able to impersonate this container to every other container. Unlike the use of individual tokens (chapter 4.2), he is only able to impersonate the compromised container.

4.2.3 Implementation effort

Unlike measure 4.1, this approach requires a way for containers to actively communicate with the runtime to generate and verify tokens.

4.3 Asymmetric key setup

4.3.1 Mechanism

The runtime acts as a trusted key broker for all containers. It creates and stores asymmetric key pair for each container and offers the following API to all containers:

1. Sign arbitrary data with the private key of the calling container.
2. Verify signed data with the public key of another container.
3. Get public key of another container.

A container can then use a key exchange algorithm such as ECDH to derive a common secret between client and server containers. A subsequent call to a key derivation function (KDF) can turn this secret into a application specific token similar to the ones discussed in chapter 4.2.

4.3.2 Mitigating effects

It is assumed that the runtime is trusted (assumption 1) and that data transfers between containers are trusted (assumption 3). Since the private keys of the containers are already handled by the runtime, the shared secrets generated by this approach could be generated and manged by the runtime as well. This measure is therefore similar to measure 4.2 and offers similar protection.

4.3.3 Implementation effort

This approach is based on asymmetric cryptography. It requires container applications to implement ECDH and KDF algorithms and is expected to be more CPU intensive.