SEV-SNP Platform Attestation Using VirTEE/SEV

Publication	58217
Revision	1.2
Issue Date	July, 2023

Date	Version	Changes
Mar, 2023	1.0	Initial public release
Apr, 2023	1.1	Incorporated community-generated API changes
Jul, 2023	1.2	Updated the code to match library SEV-SNP update.

Audience

This document describes using the SEV-SNP update in the VirTEE/SEV crate to interface with 3rd Gen and newer AMD EPYC™ processors. You must have admin OS access and familiarity with OS-specific configuration, monitoring, and troubleshooting tools.

Author

Larry Dewey, Diego Gonzalez Villalobos

Table of Contents

This page intentionally left blank.

Chapter

1

Introduction

The VirTEE/sev* crate offers a Rust-friendly, simple-to-use API for interfacing with the AMD Secure Processor included within AMD EPYC processors. The library has support for interacting with legacy SEV firmware, but the VirTEE community has recently added support to interact with the new SEV-SNP firmware that can be found in 3rd Gen and newer AMD EPYCs. This User Guide includes potential solutions for performing general platform attestation for trusted execution environments (TEE) secured by AMD SEV-SNP for both:

- **Platform owners:** Those who manage the system software where virtual machines or containers will be deployed, such as a host or Cloud Service Provider (CSP). This is the system software, including the hypervisor, where a confidential virtual-machine (VM) or container will be deployed. See "Platform Owner" on page 3.
- **Guest owners:** Those seeking to deploy the workloads. See "Guest Owner" on page 7.

This User Guide explores various capabilities available to both platform owners and guest owners provided in the SEV-SNP update in the VirTEE/sev crate, provides examples of various use cases, and explains why you may prefer various options.

This page intentionally left blank.

Chapter 2 Platform Owner

The Platform owner is the system software, including the hypervisor, where a confidential virtual-machine (VM) or container will be deployed.

2.1 System Requirements

- **Hardware:** Your system must be powered by 3rd Gen or later AMD EPYC processors.
- **Firmware:** You should always use the latest SEV firmware supported by your BIOS to have the latest features and security protection. The VirTEE/sev SEV-SNP features are compatible with firmware (version 1.54.01).
- **Kernel:** SEV-SNP development is ongoing. AMD recommends using the latest host patches* for the Linux kernel until this support is upstream.
- **Software:** If the guest owner and/or platform owner needs to:
  - Validate unique portions of the Identity Block (structure definition) provided by a guest to be included within an attestation report.
  - Validate the `kernel`, `initrd`, or `cmdline` parameters (only OVMF / EDK II)

  Then use:

  - **OVMF/EDK II:** SNP latest*
  - **QEMU:** SNP latest*

  If neither of the above are required, then AMD recommends using upstream OVMF/EDK II and QEMU version 7.1.

2.2 API Capabilities

Platform owners may:

- Request the AMD Secure Processor status. See "Request the Status of the AMD Secure Processor" on page 4.
- Load new extended configurations. See "Load New Extended Configurations" on page 4.
- Request existing extended configurations. See "Request Existing Extended Configurations" on page 6.

2.2.1 Request the Status of the AMD Secure Processor

AMD recommends checking the status of the AMD Secure Processor before beginning the process of configuring a host platform for cached certificates. To do this:

1. Include the VirTEE/sev crate into your Rust project.

```
// Import library
use sev::firmware::host::*;
```

2. Connect to the firmware and request the `snp` status of the AMD Secure Processor.

```
// Open a connection to the firmware.
let mut firmware: Firmware = Firmware::open().unwrap();

// Request the current snp status of the AMD Secure Processor.
let snp_status: SnpPlatformStatus = firmware.snp_platform_status().unwrap();
```

2.2.2 Load New Extended Configurations

Storing policies and certificate-chains in hypervisor memory allows platform owners to enhance ease-of-use for guest owners by eliminating the need to manually request a certificate-chain from the AMD Key Distribution Server (KDS). This also reduces the extent to which CSP operations depend on the AMD KDS. Further, this mitigates manually requesting certificate-chains per VM/container when attempting to scaling to large deployments.

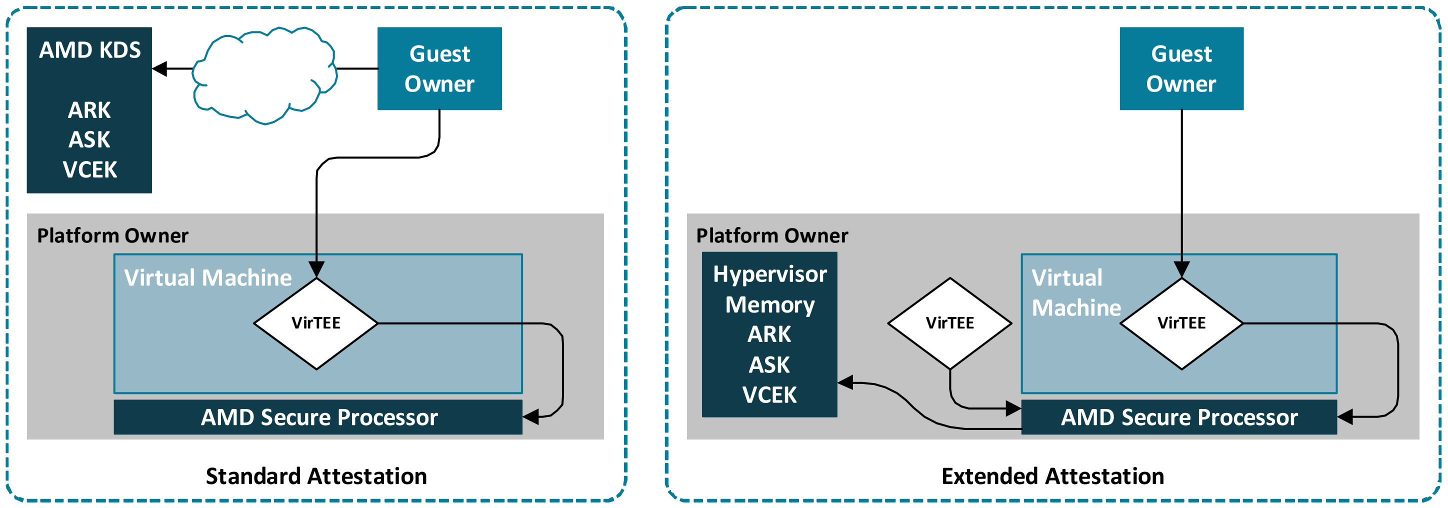

Figure 2-1: Standard (left) and extended (right) attestation flows

To load new extended configurations:

1. Include the VirTEE/sev crate into your Rust project.

```
 // Import library
use sev::firmware::host::*;
```

2. Read the bytes of the certificates which will be stored in hypervisor memory. You can do this using `include_bytes!()` as shown below, or by some other means.

```
// Read certificate bytes.
pub const ARK: &[u8] = include_bytes!("ark.pem");
pub const ASK: &[u8] = include_bytes!("ask.pem");
pub const VCEK: &[u8] = include_bytes!("vcek.pem");
```

3. Create a configuration for when guests request an extended report using one of the following three options:

   - **Certificates Only:** Allows platform owners to store a certificate-chain in hypervisor memory without modifying the contents of the reported TCB.

     ```
     // Generate a vector of certificates to store in hypervisor memory.
     let certificates: Vec<CertTableEntry> = vec![
         CertTableEntry::new(CertType::ARK, ARK.to_vec()),
         CertTableEntry::new(CertType::ASK, ASK.to_vec()),
         CertTableEntry::new(CertType::VCEK, VCEK.to_vec()),
     ];

     // Call the `new_certs_only` constructor to generate the extended configuration.
     let ext_config: ExtConfig = ExtConfig::new_certs_only(
         certificates
     );
     ```

   - **Configuration Only:** Useful for setting or updating the contents of the reported TCB without storing a certificate-chain.

     ```
     // Specify the desired configuration
     let configuration: Config = Config::new(
         TcbVersion::new(3, 0, 10, 169),
         0,
     );

     // Call the `new_config_only` constructor to generate the extended configuration.
     let ext_config: ExtConfig = ExtConfig::new_config_only(
         configuration
     );
     ```

   - **Configuration and Certificates:** Useful for setting or updating the contents of the reported TCB while also storing the certificate-chain.

     ```
     // Specify the desired configuration
     let configuration: Config = Config::new(
         TcbVersion::new(3, 0, 10, 169),
         0,
     );

     // Generate a vector of certificates to store in hypervisor memory.
     let certificates: Vec<CertTableEntry> = vec![
         CertTableEntry::new(CertType::ARK, ARK.to_vec()),
         CertTableEntry::new(CertType::ASK, ASK.to_vec()),
         CertTableEntry::new(CertType::VCEK, VCEK.to_vec()),
     ];

     // Call the `new` constructor to generate the extended configuration.
     let ext_config: ExtConfig = ExtConfig::new(
         configuration,
         certificates
     );
     ```

4. Connect to the firmware and forward the extended request to the AMD Secure Processor:

```
// Open a connection to the firmware.
let mut fw: Firmware = Firmware::open().unwrap();

// Forward the certificates to the AMD Secure Processor to be loaded.
if let Err(error) = fw.snp_reset_config(&ext_config) {

    // Handle an error if one is encountered.
    ...
}
```

2.2.3 Request Existing Extended Configurations

To request existing extended configurations:

1. Include the VirTEE/sev crate into your Rust project.

```
// Import library
use sev::firmware::host::*;
```

2. Connect to the firmware and request for the current configuration:

```
// Open a connection to the firmware.
let mut fw: Firmware = Firmware::open().unwrap();

// Request the current configuration.
let current_configuration: ExtConfig = fw.snp_get_ext_config().unwrap();
```

Chapter 3 Guest Owner

The Guest owner is a tenant of a virtualization provider. They may have one or more guest confidential virtual machines (VM) or containers that may be deployed in a Platform Owner's environment.

3.1 System Requirements

Guest-level support has been completed and is upstream. Use Linux Kernel 5.19 or newer to access the guest driver.

3.2 API Capabilities

Guest owners may:

- Request a standard attestation report. See "Requesting and Attesting a Standard Attestation Report" on page 7.
- Request an extended attestation report. See "Requesting and Attesting an Extended Attestation Report" on page 12.
- Request a unique key cryptographically derived from a hardware-owned secret. See "Requesting a Derived Key" on page 13.

This User Guide includes possible steps for platform that are beyond the scope of this library.

3.2.1 Requesting and Attesting a Standard Attestation Report

To request and attest a standard attestation report:

1. Import the necessary pieces from the crate.

```
// Import the modules
use sev::firmware::guest::* ;
```

2. Create and supply 64 bytes of unique data to include in the attestation report.

```
// This could be a unique message, a public key, etc.
let unique_data: [u8; 64] = [
    65, 77, 68, 32, 105, 115, 32, 101, 120, 116, 114, 101, 109, 101, 108, 121, 32, 97, 
119,
    101, 115, 111, 109, 101, 33, 32, 87, 101, 32, 109, 97, 107, 101, 32, 116, 104, 101, 
32,
    98, 101, 115, 116, 32, 67, 80, 85, 115, 33, 32, 65, 77, 68, 32, 82, 111, 99, 107, 
115,
    33, 33, 33, 33, 33, 33,
];
```

3. Connect to the firmware and request a SEV-SNP attestation report.

```
// Open a connection to the firmware.
let mut fw: Firmware = Firmware::open()?;

// Request a standard attestation report.
let attestation_report: AttestationReport = fw.get_report(None, Some(unique_data), 
None);
```

4. Validate the Root of Trust. This is one of the most significant steps in the attestation process. The openssl* crate provides all of the tools needed to verify the signature chain. The VirTEE/sev library includes structures and functions that simplify certificate handling. The current AMD root of trust is:

   - The AMD Root Key (ARK) is self-signed.
   - The ARK signed the AMD Signing Key (ASK).
   - The ASK signed the Versioned Chip Endorsement Key (VCEK).

   For example:

   a. Import the necessary certificate pieces for certificate handling.

   ```
   use sev::{
     certs::snp::{ca, Certificate, Chain},
     firmware::host::CertType,
   };
   ```

   b. Import the necessary pieces from the `openssl` crate.

   ```
       ecdsa::EcdsaSig,
       pkey::{PKey, Public},
       sha::Sha384,
       x509::X509,
   ```

   c. Pull the certificate-chain from the AMD Key Distribution Server (KDS), as described in the VCEK Specification. All fields are expected to be a minimum of two characters in length, as well as zero-padded (ex. 8 => 08). You will find that the `hwid` matches the `chip_id` on the attestation report.

   ```
   const KDS_CERT_SITE: &str = "https://kdsintf.amd.com";
   const KDS_VCEK: &str = "/vcek/v1";
   const KDS_CERT_CHAIN: &str = "cert_chain";

   /// Requests the certificate-chain (AMD ASK + AMD ARK)
   /// These may be used to verify the downloaded VCEK is authentic.

   pub fn request_cert_chain (sev_prod_name: &str) -> (ask, ark) {

       // Should make -> https://kdsintf.amd.com/vcek/v1/{SEV_PROD_NAME}/cert_chain
   ```

```
    let url: String = format!("{KDS_CERT_SITE}{KDS_VCEK}/{sev_prod_name}/
{KDS_CERT_CHAIN}");

    println!("Requesting AMD certificate-chain from: {url}");

    let rsp: Response = get(&url).unwrap();

let body: Vec<u8> = rsp.bytes().unwrap().to_vec();

let chain: Vec<x509> = X509::stack_from_pem(&body).unwrap();

// Create a ca chain with ark and ask

let ca_chain: ca::Chain = ca::Chain::from_pem(&chain[1].to_pem, &chain[0].to_pem);

ca_chain
};

/// Requests the VCEK for the specified chip and TCP
pub fn request_vcek(chip_id: [u8; 64], reported_tcb: TcbVersion) -> X509 {
    let hw_id: String = hexify(&chip_id);
    let url: String = format!(
    "{KDS_CERT_SITE}{KDS_VCEK}/{SEV_PROD_NAME}/\
    {hw_id}?blSPL={:02}&teeSPL={:02}&snpSPL={:02}&ucodeSPL={:02}",
    reported_tcb.boot_loader,
    reported_tcb.microcode
    );

    println!("Requesting VCEK from: {url}\n");

    let rsp_bytes = get(&url).unwrap().bytes().unwrap().to_vec();

      Certificate::from_der(&rsp_bytes)
};
```

d. Verify the Root of Trust

```
let ca_chain: ca::Chain = request_cert_chain("milan");

// chip_id and reported_tcb should be pulled from the host machine,
// or an attestation report. let vcek: Certificate = request_vcek(
chip_id,
reported_tcb
);

// Create a full-chain with the certificates:
Let cert_chain = Chain{ca: ca_chain, vcek: vcek};

//Now you can simply verify the whole chain in one command.
cert_chain.verify().unwrap();

//Or you can verify each certificate individually
let ark = cert_chain.ca.ark;
let ask = cert_chain.ca.ask;

if (&ark,&ark).verify().unwrap() {
  println!("The AMD ARK was self-signed...");
  if (&ark,&ask).verify().unwrap() {
  iprintln!("The AMD ASK was signed by the AMD ARK...");
  f (&ask,&vcek).verify().unwrap() {
  println!("The VCEK was signed by the AMD ASK...");
} else {
  eprintln!("The VCEK was not signed by the AMD ASK!");
  }
} else {
  eprintln!("The AMD ASK was not signed by the AMD ARK!");
  }
} else {
  eprintln!("The AMD ARK is not self-signed!");
}
```

5. Verify the guest Trusted Compute Base by verifying the following fields in an attestation report and a VCEK:

    - Bootloader
    - TEE
    - SNP
    - Microcode
    - Chip ID

    Neither the `openssl` nor the VirTEE/sev crates support validating X509v3 Extensions (at time of writing). One possible solution is to use the x509_parser* crate in conjunction with the asn1_rs* crate (for OIDs). The following examples will be built off these definitions:

```
/
*****************************************************************************************
 *                                RELEVANT X509v3 EXTENSION OIDS
*****************************************************************************************
/
use asn1_rs::{oid, Oid};
use x509_parser::{
    self,
    certificate::X509Certificate,
    pem::{parse_x509_pem, Pem},
    prelude::X509Extension,
};

enum SnpOid {
    BootLoader,
    Tee,
    Snp,
    Ucode,
    HwId,
}

impl SnpOid {
    fn oid(&self) -> Oid {
        match self {
            SnpOid::BootLoader => oid!(1.3.6.1.4.1.3704.1.3.1),
            SnpOid::Tee => oid!(1.3.6.1.4.1.3704.1.3.2),
            SnpOid::Snp => oid!(1.3.6.1.4.1.3704.1.3.3),
            SnpOid::Ucode => oid!(1.3.6.1.4.1.3704.1.3.8),
            SnpOid::HwId => oid!(1.3.6.1.4.1.3704.1.4),
        }
    }
}

impl std::fmt::Display for SnpOid {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{}", self.oid().to_id_string())
    }
}
```

```
/
******************************************************************************************
 *                                       HELPER FUNCTIONS

******************************************************************************************
/


fn check_cert_ext_byte(ext: &X509Extension, val: u8) -> bool {
    if ext.value[0] != 0x2 {
        panic!("Invalid type encountered!");
    }

    if ext.value[1] != 0x1 && ext.value[1] != 0x2 {
        panic!("Invalid octet length encountered");
    }

    if let Some(byte_value) = ext.value.last() {
        *byte_value == val
    } else {
        false
    }
}

fn check_cert_ext_bytes(ext: &X509Extension, val: &[u8]) -> bool {
    ext.value == val
}


/
******************************************************************************************
 *                                  EXAMPLE ATTESTATION FUNCTION:

******************************************************************************************
/

fn validate_cert_metadata(
    cert: &X509Certificate,
    report: &AttestationReport,
) -> bool {
    let extensions: HashMap<Oid, &X509Extension> = cert.extensions_map().unwrap();

    if let Some(cert_bl) = extensions.get(&SnpOid::BootLoader.oid()) {
        if !check_cert_ext_byte(cert_bl, report.reported_tcb.boot_loader) {
            eprintln!("Report TCB Boot Loader and Certificate Boot Loader mismatch
encountered.");
            return false;
        }
        println!("Reported TCB Boot Loader from certificate matches the attestation
report.");
    }

    if let Some(cert_tee) = extensions.get(&SnpOid::Tee.oid()) {
        if !check_cert_ext_byte(cert_tee, report.reported_tcb.tee) {
            eprintln!("Report TCB TEE and Certificate TEE mismatch encountered.");
            return false;
        }
        println!("Reported TCB TEE from certificate matches the attestation report.");
    }

    if let Some(cert_snp) = extensions.get(&SnpOid::Snp.oid()) {
        if !check_cert_ext_byte(cert_snp, report.reported_tcb.snp) {
            eprintln!("Report TCB SNP and Certificate SNP mismatch encountered.");
            return false;
        }
        println!("Reported TCB SNP from certificate matches the attestation report.");
    }

    if let Some(cert_ucode) = extensions.get(&SnpOid::Ucode.oid()) {
        if !check_cert_ext_byte(cert_ucode, report.reported_tcb.microcode) {
            eprintln!("Report TCB Microcode and Certificate Microcode mismatch
encountered.");
            return false;
        }
        println!("Reported TCB Microcode from certificate matches the attestation
report.");
    }

    if let Some(cert_hwid) = extensions.get(&SnpOid::HwId.oid()) {
        if !check_cert_ext_bytes(cert_hwid, &report.chip_id) {
            eprintln!("Report TCB Microcode and Certificate Microcode mismatch
encountered.");
            return false;
        }
        println!("Chip ID from certificate matches the attestation report.");
    }

    true
}
```

6. Verify that the signature contained in the attestation report truly came from the VCEK.

```
let ar_signature: EcdsaSig = EcdsaSig::try_from(&report.signature).unwrap();
let signed_bytes: &[u8] = &bincode::serialize(&report).unwrap()[0x0..0x2A0];
let amd_vcek_pubkey: EcKey<Public> = vcek.public_key().unwrap().ec_key().unwrap();
let mut hasher: Sha384 = Sha384::new()
hasher.update(signed_bytes);
let base_message_digest: [u8; 48] = hasher.finish();

if ar_signature.verify(base_message_digest.as_ref(), vcek_pubkey.as_ref()).unwrap() {
    println!("VCEK signed the Attestation Report!");
} else {
    eprintln!("VCEK did NOT sign the Attestation Report!");
}
// Or you can use a complete certificate chain to verify the attestation report
(&report, &certificate_chain).verify().unwrap()
```

3.2.2 Requesting and Attesting an Extended Attestation Report

To request and attest an extended attestation report:

1. Create and supply 64 bytes of unique data to include in the attestation report.

```
// This could be a unique message, a public key, etc.
let unique_data: [u8; 64] = [
    65, 77, 68, 32, 105, 115, 32, 101, 120, 116, 114, 101, 109, 101, 108, 121, 32, 97,
119,
    101, 115, 111, 109, 101, 33, 32, 87, 101, 32, 109, 97, 107, 101, 32, 116, 104, 101,
32,
    98, 101, 115, 116, 32, 67, 80, 85, 115, 33, 32, 65, 77, 68, 32, 82, 111, 99, 107,
115,
    33, 33, 33, 33, 33, 33,
];
```

2. Connect to the firmware and request the extended report.

```
let mut fw: Firmware = Firmware::open().unwrap();

let (extended_report, certificates): (AttestationReport, Vec<CertTableEntry>) =
fw.get_ext_report(None, Some(unique_data), 0)
```

3. Parse the ARK, ASK, and VCEK obtained from the AMD Secure Processor using the VirTEE/SEV library.

```
// Assumes all certificates are in PEM format (for simplicity).

let certs: Chain = Chain::from_cert_table_pem(certificates).unwrap();
```

4. Proceed with standard attestation report Root of Trust verification, skipping the HTTP requests to the AMD Key Distribution Server. See "Requesting and Attesting a Standard Attestation Report" on page 7.

3.2.3 Requesting a Derived Key

There are many use cases when a guest owner may want to generate a unique encryption key that was derived from the hardware Root of Trust. The guest can request that the key derivation be made dependent on several TCB-related parameters that allow the guest to re-derive the key only when the same parameter(s) are provided. To do this:

1. Construct a `DerivedKey` as per the specification:

```
let request: DerivedKey = DerivedKey::new(false, GuestFieldSelect(1), 0, 0, 0);
```

2. Connect to the firmware and request a derived key:

```
let mut fw: Firmware = Firmware::open().unwrap();

let derived_key: [u8; 32]= fw.get_derived_key(None, request).unwrap();
```

This page intentionally left blank.