Rust SPDM in the Kernel

Alistair Francis <alistair.francis@wdc.com>

01

What is SPDM?

SPDM: Secure Protocol and Data Models

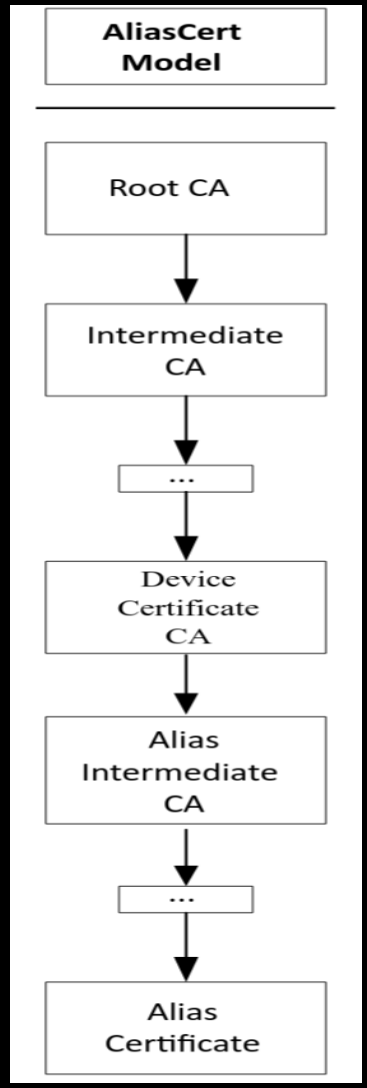
- From the kernels perspective SPDM is used to authenticate and attest devices.
  - In this threat model a device is considered untrusted until it can be verified by the kernel and userspace using SPDM. As such SPDM data is untrusted data that can be mallicious.
  - The SPDM specification is also complex, with the 1.2.1 spec being almost 200 pages and the 1.3.0 spec being almost 250 pages long.
  - As such we have the kernel parsing untrusted responses from a complex specification, which sounds like a possible exploit vector. This is the type of place where Rust excels!

SPDM: Secure Protocol and Data Models

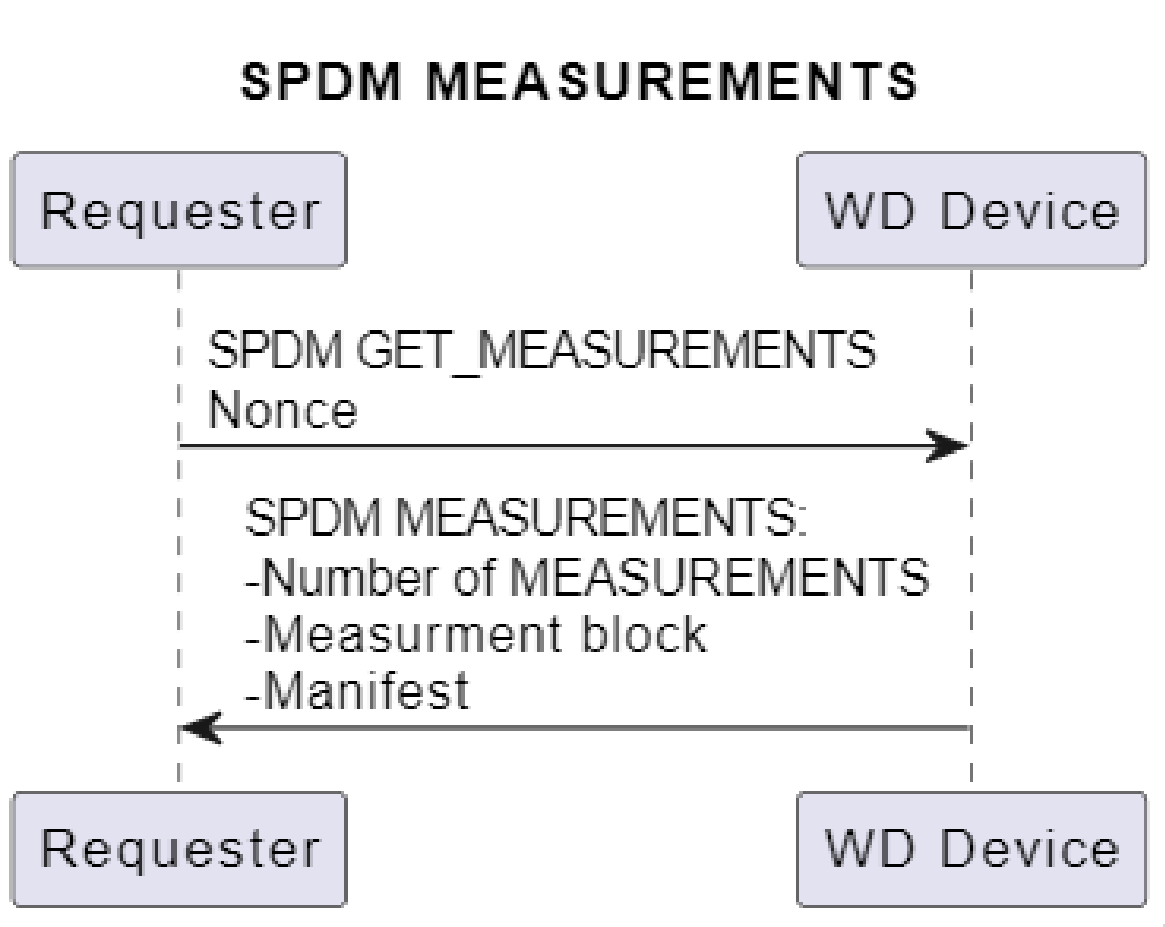
- A set of standards published by the Distributed Management Task Force (DMTF)
- A protocol designed to allow two different endpoints of a system to establish trust
  - SPDM enables the creation of a secure channel/session between two endpoints where encrypted messages can be exchanged
- SPDM follows a **requester-responder** model
- Two major features of SPDM: **Authentication** and **Attestation**
  - **Authentication**
    - Allows a host to verify the identity and integrity of an attached component and to take actions early if the device does not verify as expected
      - Each device has a Device unique ID as a certificate signed by the vendor
  - **Attestation**
    - Allows a host to verify the state of the target device
    - Achieved with multiple measurements
      - E.g. hash of immutable or mutable code, boot stages, configuration data, state variables, etc

SPDM: Authentication / Device Identity

- The identity of the device is proven by a certificate chain
- At manufacturing, the device is provisioned with a certificate chain
- A requester can issue GET_CERTIFICATE to retrieve this chain
  - The requester then verifies the certificate chain
- Optionally, the requester CHALLENGEs the responder
  - Which allows the device to prove the validity of the certificate chain (i.e that it owns the key)



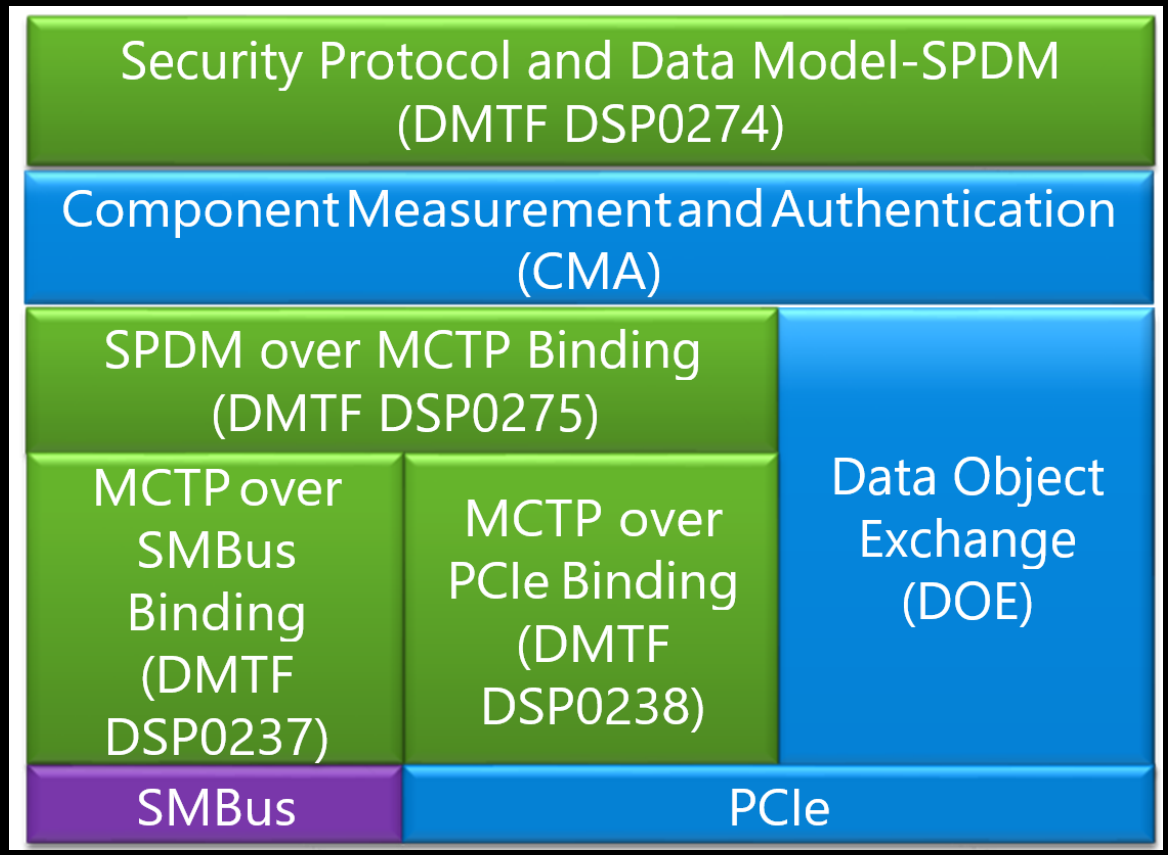
SPDM Measurements



- SPDM allows requesting measurements from a device
- This is used to read version information, running code digests, hardware and software state and configuration data
- This can then be used by a host to implement security policies
  - Useful for securing devices or maintaining large networks

SPDM: Secure Protocol and Data Models

- SPDM messages can be transported with the following transport protocols
  - SMBUS(I2C) and I3C (MCTP)
  - TCP (Networking)
  - PCIe (MCTP or DOE)
  - Storage protocols SCSI, ATA and NVMe



02

Untrusted Data API

Untrusted Data API

- Benno's Untrusted Data API v4
  - https://lore.kernel.org/all/20250814124424.516191-1-lossin@kernel.org - 14 Aug 2025

Untrusted Data API - NegotiateAlgsRsp

```rust
#[repr(C, packed)]
pub(crate) struct NegotiateAlgsRsp {
    pub(crate) version: u8,
    pub(crate) code: u8,
    pub(crate) param1: u8,
    pub(crate) param2: u8,

    pub(crate) length: u16,
    pub(crate) measurement_specification_sel: u8,
    pub(crate) other_params_sel: u8,

    pub(crate) measurement_hash_algo: u32,
    pub(crate) base_asym_sel: u32,
    pub(crate) base_hash_sel: u32,

    reserved1: [u8; 11],

    pub(crate) mel_specification_sel: u8,
    pub(crate) ext_asym_sel_count: u8,
    pub(crate) ext_hash_sel_count: u8,
    reserved2: [u8; 2],

    pub(crate) ext_asym: __IncompleteArrayField<__le32>,
    pub(crate) ext_hash: __IncompleteArrayField<__le32>,
    pub(crate) resp_alg_struct: __IncompleteArrayField<RegAlg>,
}
```

```rust
impl<'a> Validate<Untrusted<&'a [u8]>> for &'a NegotiateAlgsRsp {
    type Err = Error;

    fn validate(unvalidated: &[u8]) -> Result<Self, Self::Err> {
        if unvalidated.len() < mem::size_of::<NegotiateAlgsRsp>() {
            return Err(EINVAL);
        }

        let ptr = unvalidated.as_ptr();
        // CAST: `NegotiateAlgsRsp` only contains integers and has `repr(C)`.
        let ptr = ptr.cast::<NegotiateAlgsRsp>();
        // SAFETY: `ptr` came from a reference and the cast above is valid.
        let rsp: &NegotiateAlgsRsp = unsafe { &*ptr };

        Ok(rsp)
    }
}
```

Untrusted Data API Usage - NegotiateAlgsRsp

```rust
let response: &NegotiateAlgsRsp = Untrusted::new(response_vec.as_slice()).validate()?;

...

self.base_asym_alg = u32::from_le(response.base_asym_sel);
self.base_hash_alg = u32::from_le(response.base_hash_sel);
self.meas_hash_alg = u32::from_le(response.measurement_hash_algo);

...

// /* Responder shall select exactly 1 alg (SPDM 1.0.0 table 14) */
if self.base_asym_alg.count_ones() != 1
    || self.base_hash_alg.count_ones() != 1
    || self.meas_hash_alg.count_ones() != 1
    || response.ext_asym_sel_count != 0
    || response.ext_hash_sel_count != 0
    || response.param1 > request.param1
    || response.other_params_sel != request.other_params_support
{
    pr_err!("Malformed algorithms response\n");
    return Err(EPROTO);
}
```

Untrusted Data API - Get

```rust
#[repr(C, packed)]
pub(crate) struct GetCapabilitiesRsp {
    pub(crate) version: u8,
    pub(crate) code: u8,
    pub(crate) param1: u8,
    param2: u8,

    reserved1: u8,
    pub(crate) ctexponent: u8,
    reserved2: [u8; 2],

    pub(crate) flags: u32,

    /* End of SPDM 1.1 structure */
    pub(crate) data_transfer_size: u32,
    pub(crate) max_spdm_msg_size: u32,

    pub(crate) supported_algorithms: __IncompleteArrayFi
}
```

```rust
impl<'a> Validate<Untrusted<&'a mut KVec<u8>>> for &'a mut GetCapabilitiesRsp {
    type Err = Error;

    fn validate(unvalidated: &mut KVec<u8>) -> Result<Self, Self::Err> {
        let version = *(unvalidated.get(0).ok_or(EINVAL))?;

        let expected_length = match version {
            SPDM_VER_10 | SPDM_VER_11 => {
                core::mem::size_of::<SpdmHeader>() + 4 + core::mem::size_of::<u32>()
            }
            _ => {
                // Check to see if param1 is set
                if *(unvalidated.get(2).ok_or(EINVAL))? == 0 {
                    mem::size_of::<GetCapabilitiesRsp>()
                        - mem::size_of::<__IncompleteArrayField<__le16>>()
                } else {
                    // Not currently supported by Linux, we don't set the bit
                    // so the responder shouldn't either.
                    return Err(EINVAL);
                }
            }
        };

        // Make sure the response meets the SPDM spec version requirements
        if unvalidated.len() < expected_length {
            return Err(EINVAL);
        }

        // If the response is shorter than GetCapabilitiesRsp
        // (which is valid for older spec versions and when param1 is
        // set to 0) then we need to pad the vector to ensure
        // GetCapabilitiesRsp will be initialised.
        while unvalidated.len() < mem::size_of::<GetCapabilitiesRsp>() {
            unvalidated.push(0, GFP_KERNEL)?;
        }

        let ptr = unvalidated.as_mut_ptr();
        // CAST: `GetCapabilitiesRsp` only contains integers and has `repr(C)`.
        let ptr = ptr.cast::<GetCapabilitiesRsp>();
        // SAFETY: `ptr` came from a reference and the cast above is valid.
        let rsp: &mut GetCapabilitiesRsp = unsafe { &mut *ptr };

        Ok(rsp)
    }
}
```

Untrusted Data API - GetCapabilitiesRsp

```
if rsp_sz > response_vec.len() {
    return Err(EIO);
}

self.transcript
    .extend_from_slice(&response_vec[..rsp_sz], GFP_KERNEL)?;

let response: &mut GetCapabilitiesRsp = Untrusted::new(&mut response_vec).validate()?;

self.rsp_caps = u32::from_le(response.flags);
if (self.rsp_caps & SPDM_RSP_MIN_CAPS) != SPDM_RSP_MIN_CAPS {
    pr_err!(
        "{:#x} capabilities are supported, which don't meet required {:#x}\n",
        self.rsp_caps,
        SPDM_RSP_MIN_CAPS
    );
    self.rsp_caps = 0;
    return Err(EPROTONOSUPPORT);
}

if self.version >= SPDM_VER_12 {
    let data_transfer_size = u32::from_le(response.data_transfer_size);
```

Untrusted Data API - GetCapabilitiesRsp

```rust
if rsp_sz > response_vec.len() {
    return Err(EIO);
}

let response: &mut GetCapabilitiesRsp = Untrusted::new(&mut response_vec).validate()?;

self.transcript
    .extend_from_slice(&response_vec[..rsp_sz], GFP_KERNEL)?;
```

```
error[E0502]: cannot borrow `response_vec` as immutable because it is also borrowed as mutable
   --> lib/rspdm/state.rs:516:33
    |
513 |         let response: &mut GetCapabilitiesRsp = Untrusted::new(&mut response_vec).validate()?;
    |                                                                 ---------------- mutable borrow occurs here
...
516 |             .extend_from_slice(&response_vec[..rsp_sz], GFP_KERNEL)?;
    |                                ^^^^^^^^^^^^ immutable borrow occurs here
517 |
518 |         self.rsp_caps = u32::from_le(response.flags);
    |                                      -------------- mutable borrow later used here

error: aborting due to 1 previous error
```

03

Current Status

Current Status

- V3 patches submitted
  - o [PATCH v3 00/21] lib: Rust implementation of SPDM
    - https://lore.kernel.org/rust-for-linux/20260901010347.2614656-1-alistair.francis@wdc.com/
  - o Please review!
  - o This is based on Lukas' C implementation [https://lore.kernel.org/all/cover.1719771133.git.lukas@wunner.de/], but has been refacted during the first few RFCs


WD
TM