

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 malicious.
 - 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!

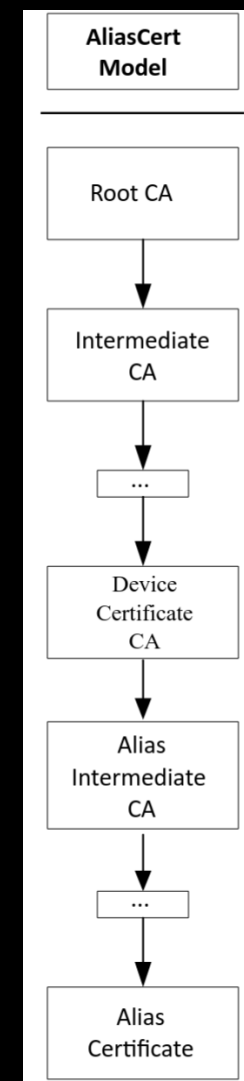
SPDM: Secure Protocol and Data Models



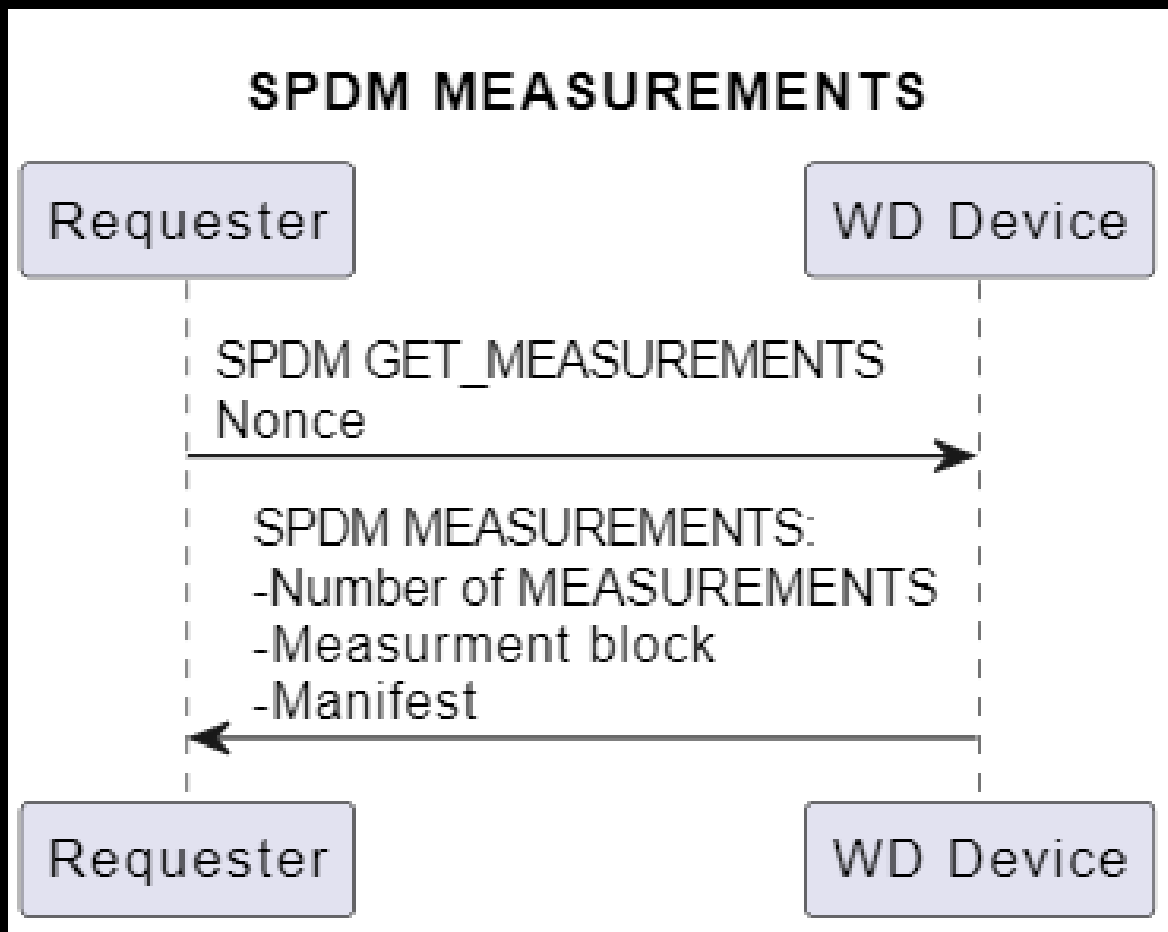
- A set of standards published by the Distributed Management Task Force (DMTF)
- A protocol designed to allow two different endpoints of a system to establish trust
 - SPDM enables the creation of a secure channel/session between two endpoints where encrypted messages can be exchanged
- SPDM follows 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

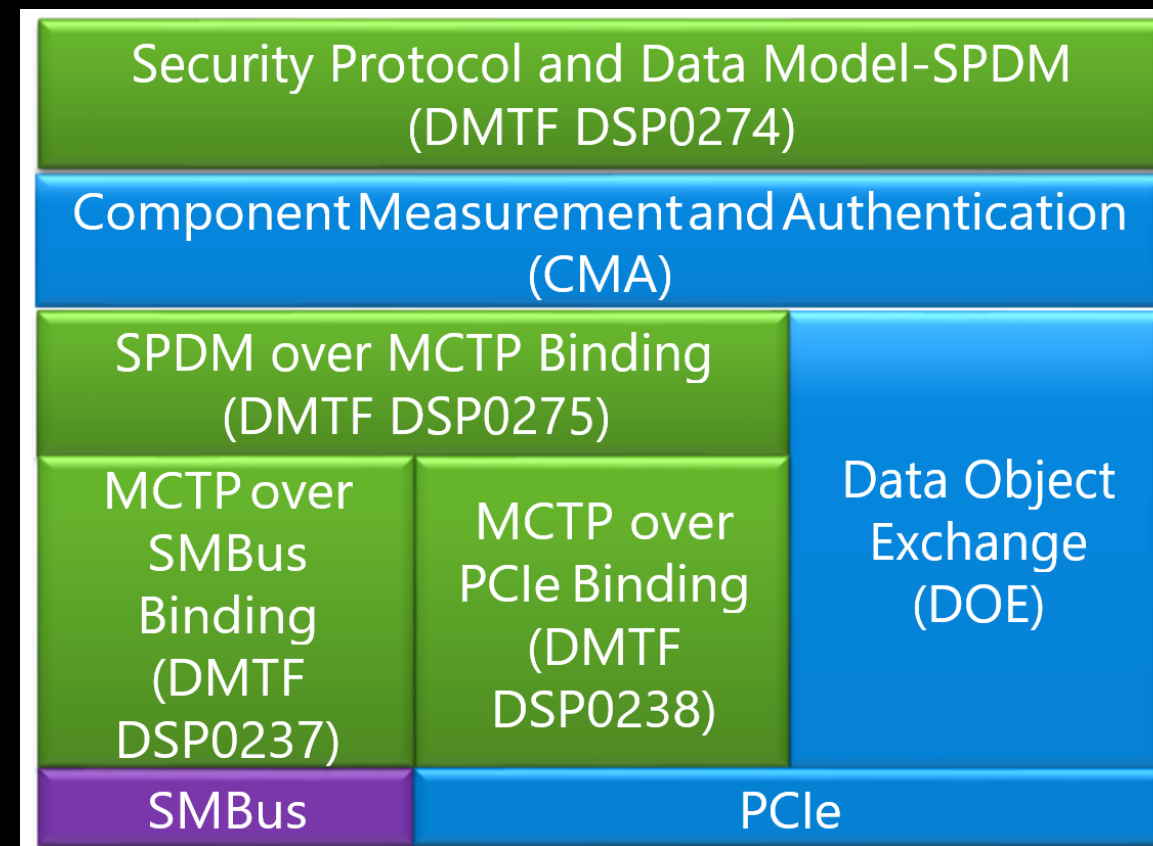


- SPDM allows requesting measurements from a device
- This is used to read version information, running code digests, hardware and software state and configuration data
- This can then be used by a host to implement security policies
- Useful for securing devices or maintaining large networks

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

Current Status

Using Validate API

```
let mut response_vec: KVec<u8> = KVec::from_elem(0u8, SPDM_GET_VERSION_LEN, GFP_KERNEL)?;

let rc =
    self.spdm_exchange(request_buf.as_mut_slice(), response_vec.as_mut_slice())? as usize;

// The transport must report a length within the buffer we provided.
if rc > response_vec.len() {
    return Err(EINVAL);
}
response_vec.truncate(rc);

let response: GetVersionRsp = Untrusted::new(response_vec.as_slice()).validate(&*self)?;
```

Validating Data - GetVersionRsp

```
fn validate(unvalidated: &[u8], context: &'c SpdmState<'c>) -> Result<Self, Self::Err> {
    let header: SpdmHeader =
        Untrusted::new(unvalidated.get(0..4).ok_or(EIO)?).validate(context)?;

    if header.code != SPDM_GET_VERSION - 0x80 {
        return Err(EINVAL);
    }

    if header.version != SPDM_MIN_VER {
        return Err(EINVAL);
    }

    let version_number_entry_count = *unvalidated.get(5).ok_or(EIO)?;

    // Entries follow header (4) + reserved (1) + count (1) = 6 bytes.
    let mut offset = mem::size_of::<SpdmHeader>() + 2;

    let mut version_number_entries = KVec::new();
    for _ in 0..version_number_entry_count {
        let entry = u16::from_le_bytes(
            unvalidated
                .get(offset..offset + mem::size_of::<u16>())
                .ok_or(EIO)?
                .try_into()
                .map_err(|_| EINVAL)?,
        );
        version_number_entries.push(entry, GFP_KERNEL)?;
        offset += mem::size_of::<u16>();
    }
}
```

```
#[expect(dead_code)]
pub(crate) struct GetVersionRsp {
    pub(crate) header: SpdmHeader,

    pub(crate) version_number_entry_count: u8,
    pub(crate) version_number_entries: KVec<u16>,

    /// Size of the response, not public
    length: usize,
}
```

Validating Data - NegotiateAlgsRsp

```
impl<'a, 'c> Validate<'c, Untrusted<&'a [u8]>> for NegotiateAlgsRsp {
    fn validate(unvalidated: &[u8], context: &'c SpdmState<'c>) -> Result<Self, Self::Err> {

        let mut ext_hash = KVec::new();
        for _ in 0..ext_hash_sel_count {
            ext_hash.push(read_le32(offset)?, GFP_KERNEL)?;
            offset += mem::size_of::<u32>();
        }

        let mut resp_alg_struct = KVec::new();
        for _ in 0..header.param1 {
            let alg_type = *unvalidated.get(offset).ok_or(EIO)?;
            let alg_count = *unvalidated.get(offset + 1).ok_or(EIO)?;
            let fixed_alg_count = (alg_count & 0xf) as usize;
            let ext_alg_count = (alg_count >> 4) as usize;
            offset += 2;

            let mut alg_supported = KVec::new();
            alg_supported.extend_from_slice(
                unvalidated
                    .get(offset..offset + fixed_alg_count)
                    .ok_or(EIO)?,
                GFP_KERNEL,
            )?;
        }
    }
}
```

CMA TSM Driver

```
static const struct pci_tsm_ops pci_cma_tsm_ops = {
    .link_ops = {
        .probe      = pci_cma_tsm_probe,
        .remove     = pci_cma_tsm_remove,
        .connect    = pci_cma_tsm_connect,
        .disconnect = pci_cma_tsm_disconnect,
        .bind       = pci_cma_tsm_bind,
        .unbind     = pci_cma_tsm_unbind,
        .guest_req  = pci_cma_tsm_guest_req,
    },
};
```

```
/* device is a TSM host and capable of 'connect' */
static inline bool is_pci_tsm_host(struct pci_dev *pdev)
{
    if (!pdev)
        return false;

    if (!pci_is_pcie(pdev))
        return false;

    if (pdev->is_virtfn)
        return false;

    /*
     * Report capable if CMA is supported, which can be supported on any PCIe
     * device.
     */
#ifdef CONFIG_PCI_DOE
    if (pdev->doe_cma)
        return true;
#endif

    return is_pci_tsm_dsm(pdev);
}
```

Other Drivers

- NVMe, ATA, SCSI, MCTP?

Current Status

- V4 patches submitted
 - o [PATCH v4 00/22] lib: Rust implementation of SPDML
 - <https://lore.kernel.org/lkml/20260928011123.450800-1-alistair.francis@wdc.com>
 - Please review!
 - o This is based on Lukas' C implementation [<https://lore.kernel.org/all/cover.1719771133.git.lukas@wunner.de/>], but has been refactored during the first few RFCs

03

Userspace

netlink

- Maximum netlink attribute size is ~64K (nlattr is 16 bits)
- We need much larger blobs, closer to 16M
- "multi-message object receive"
 - Nacked: <https://lore.kernel.org/all/20260721182046.16219426@kernel.org>

More netlink-y

Multi message netlink attributes

- Option 1: Delete the F_MULTI and replicate other attrs in each message and add an offset attr; this will make each message in the dump more standalone
- Option 2: Keep the F_MULTI but object_val has to be a multi-attr, and then we have to teach YNL to correctly append the attrs
 - <https://lore.kernel.org/all/20260318170014.6650d2bf@kernel.org>
- Ankit Agrawal seems to be working on option 1
 - <https://lore.kernel.org/all/SAWPR12MB9992679DEB9AFB41A21B116FDBB0D62@SAWPR12MB999267.namprd12.prod.outlook.com>

Zero Copy Blob Attribute

Multi message netlink attributes

- Lukas's "netlink: Introduce nla_put_blob() to add zero-copy blob attribute to skb"
- Attach the pages backing the blob as fragments to the skb. It's very simple, overcomes the skb size limitation and allows for zero-copy
 - <https://github.com/l1k/linux/commit/6e73bb999128>
 - <https://lore.kernel.org/all/apl39vOiMPP7VloW@wunner.de>

ioctl-based iface

CTLV

- <https://lore.kernel.org/all/arzr32ZDComnfmny@FV6GYCPJ69/>
- https://github.com/jpirko/linux_mlxsw/commits/wip_ctlv_pre_rfc_draft1/
- It introduces a provider-neutral, fd-based interface for evidence
- retrieval, userspace verdicts tied to exact device/evidence generations,
- measurement registers and their journal, events, and device-security
- state transitions. Large evidence is written directly to referenced
- buffers instead of being split across Netlink messages.

04

Next Steps

Next steps?

- Zero copy?
- SPDM 1.4 Features?
- PQC?
- Other Transports?



