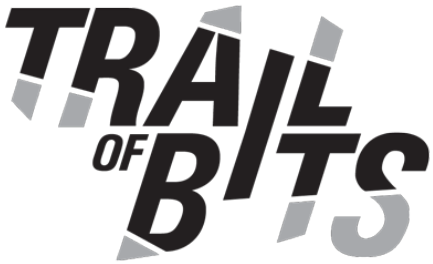

MobileCoin Secure Enclave

Security Assessment

August 21, 2020

Prepared For:
Robb Walters | *MobileCoin*
robb@mobilecoin.com

Prepared By:
Samuel Moelius | *Trail of Bits*
sam.moelius@trailofbits.com

Artur Cygan | *Trail of Bits*
artur.cygan@trailofbits.com

Executive Summary

From August 10 to April 21, 2020, MobileCoin engaged Trail of Bits to review the security of the MobileCoin secure enclave. Trail of Bits conducted this assessment over four person-weeks, with two engineers working from commit `ef7259b` of the `mobilecoin` repository.

During the first week of the assessment, we verified that we could build the codebase. We also verified that the unit tests pass and collected their code coverage using `grcov`. We ran `cargo-audit`, `cargo-upgrades`, and `cargo-clippy` over the codebase. We then began a manual review, focusing on the crates in the `consensus/enclave` and `sgx` directories. Finally, we developed a fuzzer for the enclave calls.

During the second week, we manually reviewed the implementation and use of the NOISE protocol. We also manually reviewed the transaction validation code and developed a fuzzer for it.

Our efforts resulted in nine findings ranging from medium to informational severity, including one finding of undetermined severity. One medium-severity issue concerns a dependency that is susceptible to memory exhaustion. A second medium-severity issue concerns memory exhaustion that can occur when an `EnclaveCall` is deserialized. A third medium-severity issue concerns a panic that can occur when verifying membership proofs. One low-severity finding concerns an out-of-bounds memory access that can occur when writing a `Quote` to the console. Three informational-severity findings concern updates to dependencies, the addition of unit tests, and the risk associated with relying on the Intel Attestation Service (IAS). Finally, the undetermined-severity issue concerns a panic that can arise in `Range` structures.

In addition to addressing the findings in this report, consider the following recommendations:

- Consider reorganizing the repository around the code that does and does not appear in the enclave. At present, enclave and non-enclave code are intermingled, as shown in table 1. Enclave code must be treated specially (e.g., because it must be compiled with `![no_std]`), it makes sense to separate it from other code).

Directories used in the enclave	**Directories not used in the enclave**

`account-keys`
`attest/ake`
`attest/core`
	`admin-http-gateway`
	`api`
	`attest/api`

attest/enclave-api attest/trusted common consensus/enclave/api consensus/enclave/edl consensus/enclave/impl consensus/enclave/trusted crypto/ake/enclave crypto/box crypto/digestible crypto/digestible/derive crypto/hashes crypto/keys crypto/message-cipher crypto/noise crypto/rand crypto/sig enclave-boundary sgx/alloc sgx/backtrace-edl sgx/compat sgx/debug sgx/debug-edl sgx/enclave-id sgx/libc-types sgx/panic sgx/panic-edl sgx/report-cache/api sgx/service sgx/slog sgx/slog-edl sgx/sync sgx/types transaction/core util/encodings util/from-random util/repr-bytes util/serial	attest/net attest/untrusted connection connection/test-utils consensus/api consensus/enclave consensus/enclave/measurement consensus/enclave/mock consensus/scp consensus/scp/play consensus/service crypto/message-cipher ledger/db ledger/distribution ledger/from-archive ledger/migration ledger/sync mobilecoind mobilecoind-json mobilecoind/api peers peers/test-utils sgx/compat-edl sgx/core-types sgx/core-types-sys sgx/css sgx/epid sgx/epid-sys sgx/epid-types sgx/epid-types-sys sgx/ias-types sgx/report-cache/untrusted sgx/urts-sys testnet-client transaction/core/test-utils transaction/std util/b58-payloads util/build/enclave util/build/grpc util/build/bolt-signalapp util/build/sgx util/build/script util/generate-sample-ledger util/grpc util/host-cert util/keyfile util/lmdb

	util/logger-macros util/metered-channel util/metrics util/test-helper util/uri util/url-encoding watcher watcher/api

Table 1. Directories that are used in the enclave versus those that are not

- Consider making improvements to the MobileCoin documentation.
  - The MobileCoin website does not feature a link for documentation; it is customary to provide documentation through the website for blockchain projects. Consider adding such a link.
  - The reasoning surrounding what checks should be made in the enclave versus what checks should be relegated to consensus is subtle (e.g., verifying that key images have not already been spent). This reasoning should be captured in the documentation.
  - Consider making public the documentation that was shared with us privately during the assessment.
  - The `README` files within the repository should be better linked to enable their discovery. For example, the `README`s in `attest` and `consensus` do not link to the `README`s in any of their subdirectories. Similarly, there is no `README` in `transaction/core`, even though there is one in `transaction/core/src/range_proofs`.
  - Some of the `README`s appear to require updates. For example, the `README` in `transaction/core/src/range_proofs` contains the heading `bulletproofs_utils`, though convention suggests that the heading should be `range_proofs`.
  - Information in MobileCoin Tech Talks #1 and #2 should be captured in text, if it is not already. Capturing information in text makes it both searchable and available for offline viewing.
- Many of the findings mentioned above were identified through fuzzing. Through fuzzing, we also encountered GitHub issue #287 (`ConsensusEnclave::enclave_init` panics on unparsable sealed key), though we did not write a finding for it. After we rediscovered #287, we noticed that `enclave_init` makes global state changes before panicking. In the fix for #287, consider adjusting `enclave_init` so that it makes global state changes only after an error can no longer occur. This issue is discussed further in Appendix C.
- Finally, consider having an audit done of MobileCoin's consensus code. Such an audit should include (at a minimum) enclave upgradeability and validator selection; the audit could also involve fuzzing the "untrusted" validation code.

Project Dashboard

Application Summary

Name	MobileCoin Secure Enclave
Version	ef7259b185e6723790d64ec48ae58fc05f140eb0
Type	Rust
Platform	Intel SGX

Engagement Summary

Dates	August 10–21, 2020
Method	Full knowledge
Consultants Engaged	2
Level of Effort	4 person-weeks

Vulnerability Summary

Total Medium-Severity Issues	3	■ ■ ■
Total Low-Severity Issues	2	■ ■
Total Informational-Severity Issues	3	■ ■ ■
Total Undetermined-Severity Issues	1	■
Total	9	

Category Breakdown

Access Controls	1	■
Data Validation	3	■ ■ ■
Denial of Service	1	■
Error Reporting	1	■
Patching	3	■ ■ ■
Total	9	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Strong.** Each enclave uses encryption to secure its connections to other enclaves and to protect data across enclave calls. We identified no issues.
Arithmetic	**Satisfactory.** We found an arithmetic underflow issue with regard to Range structs. The issue could be triggered through at least two distinct code paths. No other arithmetic issues were noted.
Assembly Use	**Not applicable.**
Decentralization	**Moderate.** The IAS represents a single point of failure. Nodes must have a particular enclave binary to participate in the network. Additional steps could be taken to improve the reproducibility of that binary.
Code Complexity	**Satisfactory.** There is an opportunity to organize the repository around the code that does and does not appear in the enclave. The code is sophisticated due to domain complexity. On the positive side, the more complex parts of the code are often accompanied by an explanation.
Front-Running	**Not considered.**
Key Management	**Strong.** We found no issues regarding the use of keys to secure connections to other enclaves or in the use of keys to protect data across enclave calls.
Monitoring	**Not considered.**
Specification	**Satisfactory.** The documentation could be improved in (at least) the following ways. The reasoning surrounding which checks are made in and outside of an enclave should be captured in the documentation. The documentation shared with us privately should be made public. The various README files should be better linked to enable their discovery. Information in YouTube videos should be captured in text.
Testing and Verification	**Moderate.** While the transaction validation code is reasonably well tested, the enclave calls would benefit from additional tests.

Engagement Goals

The engagement was scoped to provide a security assessment of MobileCoin secure enclave and its supporting code.

Specifically, we sought to answer the following questions:

- Are any conventional vulnerabilities (e.g., remote code execution or memory corruption) present within the codebase?
- Is it possible for a MobileCoin node to attest to an illegitimate enclave?
- Is there any way to read an enclave's state from outside of the enclave?
- Is an enclave susceptible to side-channel attacks?
- Is it possible to mint coins?
- Is it possible to double-spend coins?
- Is it possible to deanonymize a user?
- Is the NOISE protocol implemented and used correctly?
- Is any sensitive information (e.g., private keys, transaction inputs) used in operations that are not constant time?
- Are foreign function interfaces implemented and used correctly?
- Are OCALLs (e.g., logging) absent from production code?

Coverage

`attest` subdirectory. We statically analyzed the subdirectory using `cargo-audit`, `cargo-upgrades`, and `cargo-clippy`. We verified that the unit tests pass, and we reviewed them for code coverage. We manually reviewed the subdirectory, verifying its correct implementation and use of the NOISE IX protocol.

`consensus/enclave` subdirectory. We statically analyzed the subdirectory using `cargo-audit`, `cargo-upgrades`, and `cargo-clippy`. We verified that the unit tests pass, and we reviewed them for code coverage. We manually reviewed the subdirectory, verifying that sensitive information (private keys and transaction inputs) are not used in a way that could affect enclave call runtime. We fuzzed enclave calls using `cargo-afl`.

`sgx` subdirectory. We statically analyzed the subdirectory using `cargo-audit`, `cargo-upgrades`, and `cargo-clippy`. We verified that unit tests pass, and we reviewed them for code coverage. We manually reviewed the subdirectory, verifying that OCALLs would not be present in production.

`transaction` subdirectory. We manually reviewed the subdirectory and statically analyzed it using `cargo-audit`, `cargo-upgrades`, and `cargo-clippy`. We verified that unit tests pass,

and we reviewed them for code coverage. We fuzzed the transaction validation using `cargo-afl`.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short term

❑ **Upgrade `tokio-rustls` from `0.13.0` to `0.13.1`.** This will eliminate the risk of memory exhaustion attacks. TOB-MOB-001

❑ **Update dependencies to the latest version wherever possible.** Using out-of-date dependencies could mean critical bug fixes are missing. TOB-MOB-002

❑ **Add unit tests for the 17 enclave functions that are not currently exercised by unit tests.** Ideally, there will be at least one test for each "happy" (successful) path, and at least one test for each "sad" (failing) path. A comprehensive set of unit tests will help expose errors, protect against regressions, and provide a sort of documentation to users. TOB-MOB-003

❑ **Add checks of the inputs to `tx_is_well_formed` and `validate_membership_proofs`** (e.g., check that each `Range`'s `from` field is no more than its `to` field). This will eliminate the risk that a panic can arise in debug builds. TOB-MOB-004

❑ **Bound the size of the vector that `visit_seq` allocates, similar to how `serde` does** (figures 5.2 and 5.3). This will eliminate the risk of a memory exhaustion attack. TOB-MOB-005

❑ **Write a validation function for `Quote` and ensure that the function is called for any `Quote` before it is used. Alternatively, ensure that any function that operates on a `Quote` (e.g., `version`) checks its validity before accessing its internals.** (Note that this latter option may require changes to the return types of some functions, such as `version`.) Taking one of these steps will reduce the likelihood of accessing out-of-bounds memory through a `Quote`. TOB-MOB-006

❑ **Design and document the process for reproducing the enclave from the source code.** Set up at least two different CI servers that will check releases for non-reproducibility. TOB-MOB-007

❑ **Replace the uses of `expect` in figure 8.1 with `ok_or` applied to an appropriate error.** This will eliminate the risk that a panic could be exploited to carry out a denial-of-service attack. TOB-MOB-008

❑ **Document the fact that if IAS goes offline, so does MobileCoin.** This will alert users to this potential risk. TOB-MOB-009

Long term

❑ **Regularly run `cargo-audit` over the codebase.** Doing so can help reveal similar bugs. TOB-MOB-001

❑ **Regularly run `cargo-upgrades` and `cargo update --dry-run` over the codebase.** Doing so can help to ensure that the project stays up to date with its dependencies. TOB-MOB-002

❑ **Write end-to-end tests that create an enclave and call the functions inside of it, if such tests do not already exist.** Such tests will help ensure that running those functions inside an enclave does not adversely affect their behavior. TOB-MOB-003

❑ **Consider incorporating fuzzing into your continuous integration process.** Doing so could help to reveal similar bugs in the future. TOB-MOB-004, TOB-MOB-005, TOB-MOB-006, TOB-MOB-008

❑ **Investigate Nix as an alternative build system designed to maximize reproducibility.** The bootstrap size resulting from the Docker image is big, and having a build system that focuses on reproducibility would improve transparency and trust for the users. TOB-MOB-007

❑ **Consider strategies to mitigate the risks associated with IAS's being a single point of failure.** One option could be to run a backup attestation service. Another could be to try to resolve the problem using the peer-to-peer network (e.g., a subset of the nodes vouches for an enclave's authenticity in the event that IAS cannot be contacted). Finding a way to mitigate this risk could allow MobileCoin to continue to function if IAS goes offline or becomes unreachable. TOB-MOB-009

Findings Summary

#	Title	Type	Severity
1	The codebase uses a crate with a RUSTSEC advisory	Patching	Medium
2	The codebase relies on outdated dependencies	Patching	Informational
3	Insufficient enclave function tests	Error Reporting	Informational
4	The `tx_is_well_formed` and `mc_transaction_core::validation::validate` functions panic on crafted input	Data Validation	Undetermined
5	Memory exhaustion when deserializing `EnclaveCall`	Denial of Service	Medium
6	Out-of-bounds memory access in `Quote`	Data Validation	Low
7	No instructions on how to reproduce the SGX enclave build	Patching	Low
8	Panic in `derive_proof_at_index`	Data Validation	Medium
9	Intel Attestation Service (IAS) is a single point of failure	Access Controls	Informational

1. The codebase uses a crate with a RUSTSEC advisory

Severity: Medium
Type: Patching
Target: `tokio-rustls 0.13.0`

Difficulty: Undetermined
Finding ID: TOB-MOB-001

Description
MobileCoin uses a crate with a Rust Security (RUSTSEC) advisory. Specifically, the `tokio-rustls 0.13.0` crate contains a vulnerability that could lead to memory exhaustion.

RUSTSEC-2020-0019's description states the following:

> `tokio-rustls` does not call `process_new_packets` immediately after `read`, so the expected termination condition `wants_read` always returns true. As long as new incoming data arrives faster than it is processed and the reader does not return pending, data will be buffered.
>
> This may cause DoS.

In addition to the above, running `cargo-audit` over the codebase produces warnings for the following nine crates. The warnings suggest that the crates are deprecated or no longer maintained.

- `bigint 4.4.1`
- `block-cipher-trait 0.6.2`
- `mbedtls-sys-auto 2.18.1`
- `net2 0.2.34`
- `smallvec 1.4.0`
- `spin 0.5.2`
- `tempdir 0.3.7`
- `term 0.6.1`

Exploit Scenario
Eve discovers a code path leading to the vulnerable crate and uses this code path to crash nodes and corrupt memory.

Recommendations
Short term, upgrade `tokio-rustls` from `0.13.0` to `0.13.1`. This will eliminate the risk of memory exhaustion attacks.

Long term, regularly run `cargo-audit` over the codebase. Doing so can help reveal similar bugs.

References
- Fix place wrong for `process_new_packets`
- `cargo-audit`

2. The codebase relies on outdated dependencies

Severity: Informational
Type: Patching
Target: `Cargo.toml`, `Cargo.lock`

Difficulty: Undetermined
Finding ID: TOB-MOB-002

Description
Updated versions of many of MobileCoin's dependencies are available. Since silent bug fixes are common, the dependencies should be reviewed and updated wherever possible.

Dependency	Version(s) currently in use	Latest version available
aead	0.2.0	0.3.2
aes-gcm	0.3.2	0.6.0
base64	0.11.0	0.12.3
bindgen	0.51.1	0.54.1
blake2	0.8.1	0.9.0
cargo_metadata	0.9.1	0.11.1
crossbeam-channel	0.3.9	0.4.3
digest	0.8.1	0.9.0
dirs	2.0.2	3.0.1
dotenv	0.14.1	0.15.0
generic-array	0.12.3 and 0.13.2	0.14.4
hashbrown	0.6.3	0.8.2
hex	0.3.2	0.4.2
hkdf	0.8.0	0.9.0
hostname	0.1.5	0.3.1
indicatif	0.14.0	0.15.0
lru	0.1.17	0.6.0
pem	0.6.1	0.8.1
percent-encoding	1.0.1	2.1.0
proc-macro2	0.4.30	1.0.19
proptest	0.9.6	0.10.0
quote	0.6.13	1.0.7

rand	0.6.5	0.7.3
retry	0.5.1	1.0.0
rusoto_core	0.42.0	0.45.0
rusoto_s3	0.42.0	0.45.0
secrecy	0.4.1	0.7.0
semver	0.9.0	0.10.0
sentry	0.18.1	0.19.0
serial_test	0.1.0	0.4.0
serial_test_derive	0.1.0	0.4.0
sha2	0.8.2	0.9.1
sha3	0.8.2	0.9.1
signature	1.0.1	1.2.2
syn	0.15.44	1.0.38
time	0.1.43	0.2.16
zeroize	0.10.1	1.1.0

Table 2.1: Dependencies for which updates are available

In addition to the above, the build process expects SGX SDK version 2.9.1, but version 2.10 is available.

Finally, running `cargo update --dry-run` produces the warning in figure 2.1. Presumably, this warning concerns the patch in figure 2.2. If the patch is not needed, consider removing it.

```
warning: Patch `console v0.11.2
(https://github.com/mitsuhiko/console?rev=1307823#13078231)` was not used in the
crate graph.
Check that the patched package version and available features are compatible
with the dependency requirements. If the patch has a different version from
what is locked in the Cargo.lock file, run `cargo update` to use the new
version. This may also occur with an optional dependency that is not enabled.
```

Figure 2.1: Warning produced by running `cargo update --dry-run`

```
console = { git = "https://github.com/mitsuhiko/console", rev = "1307823" }
```

Figure 2.2: Cargo.toml#L115

Exploit Scenario
Eve learns of a vulnerability in an old version of a MobileCoin dependency and exploits it, knowing that MobileCoin relies on the old version.

Recommendations
Short term, update dependencies to the latest version wherever possible. Using out-of-date dependencies could mean critical bug fixes are missing.

Long term, regularly run `cargo-upgrades` and `cargo update --dry-run` over the codebase. Doing so can help to ensure that the project stays up to date with its dependencies.

References
- `cargo-upgrades`
- `cargo update`

3. Insufficient enclave function tests

Severity: Informational
Type: Error Reporting
Target: `consensus/enclave/impl/src/lib.rs`

Difficulty: Undetermined
Finding ID: TOB-MOB-003

Description
A MobileCoin enclave supports 19 functions. Only two of them are exercised by unit tests (`tx_is_well_formed` and `form_block`). Insufficient unit tests could mean that critical bugs are missed.

Test	Enclave function tested
`test_tx_is_well_formed_works`	`tx_is_well_formed`
`test_tx_is_well_formed_works_errors_on_bad_inputs`	`tx_is_well_formed`
`test_tx_is_well_form_rejects_inconsistent_root_elements`	`tx_is_well_formed`
`test_form_block_works`	`form_block`
`test_form_block_prevents_duplicate_spend`	`form_block`
`test_form_block_prevents_duplicate_output_public_key`	`form_block`
`form_block_refuses_duplicate_root_elements`	`form_block`

Table 3.1: Existing tests for enclave functions

Exploit Scenario
Eve exploits a flaw in a MobileCoin enclave function. The flaw would likely have been revealed through unit tests.

Recommendations
Short term, add unit tests for the 17 enclave functions that are not currently exercised by unit tests. Ideally, there will be at least one test for each "happy" (successful) path, and at least one test for each "sad" (failing) path. A comprehensive set of unit tests will help expose errors, protect against regressions, and provide a sort of documentation to users.

Long term, write end-to-end tests that create an enclave and call the functions inside of it, if such tests do not already exist. Such tests will help ensure that running those functions inside an enclave does not adversely affect their behavior.

4. The tx_is_well_formed and mc_transaction_core::validation::validate functions panic on crafted input

Severity: Undetermined | Difficulty: Low
Type: Data Validation | Finding ID: TOB-MOB-004
Target: `consensus/enclave/impl/src/lib.rs`, `transaction/core/src/tx.rs`, `transaction/core/src/range.rs`, `transaction/core/src/validation/validate.rs`, `transaction/core/src/membership_proofs/mod.rs`

Description
Calling `tx_is_well_formed` or `mc_transaction_core::validation::validate` with crafted data results in an "`attempt to subtract with overflow`" panic in debug mode. The panic will not occur in production, since production code will not be built in debug mode. Nonetheless, the impact of the subtraction underflow is unclear.

For the first case, the code in question appears in figures 4.1–4.4. The `tx_is_well_formed` function (figure 4.1) is called with a vector of `TxOutMembershipProof`s (figure 4.2). Each of those structures contains a `TxOutMembershipElement` (figure 4.3), and each of those structures contains a `Range` (figure 4.4). The constructor for `Range` ensures that its `from` field is no more than its `to` field. However, the values passed to `tx_is_well_formed` need not have been constructed in this way (e.g., they could have been deserialized). Thus, when `tx_is_well_formed` calls `len` on those `Range` values while inserting them into a BTree (figure 4.1), the subtraction `self.to - self.from` can underflow, resulting in a panic (in debug mode).

```
    fn tx_is_well_formed(
        &self,
        locally_encrypted_tx: LocallyEncryptedTx,
        block_index: u64,
        proofs: Vec<TxOutMembershipProof>,
    ) -> Result<(WellFormedEncryptedTx, WellFormedTxContext)> {
        // Enforce that all membership proofs provided by the untrusted system for
transaction validation
        // came from the same ledger state. This can be checked by requiring all proofs to
have the same root hash.
        let mut root_elements = BTreeSet::new();
        for proof in &proofs {
            let root_element = proof
                .elements
                .last() // The last element contains the root hash.
                .ok_or(Error::InvalidLocalMembershipProof)?;
            root_elements.insert(root_element);
        }
```

Figure 4.1: consensus/enclave/impl/src/lib.rs#L293-L308

```rust
#[derive(Clone, Deserialize, Eq, PartialEq, Serialize, Message, Digestible)]
pub struct TxOutMembershipProof {
    /// Index of the TxOut that this proof refers to.
    #[prost(uint64, tag = "1")]
    pub index: u64,

    /// Index of the last TxOut at the time the proof was created.
    #[prost(uint64, tag = "2")]
    pub highest_index: u64,

    /// All hashes needed to recompute the root hash.
    #[prost(message, repeated, tag = "3")]
    pub elements: Vec<TxOutMembershipElement>,
}
```

Figure 4.2: transaction/core/src/tx.rs#L301-L314

```rust
#[derive(Clone, Deserialize, Eq, PartialOrd, Ord, PartialEq, Serialize, Message,
Digestible)]
/// An element of a TxOut membership proof, denoting an internal hash node in a Merkle
tree.
pub struct TxOutMembershipElement {
    /// The range of leaf nodes "under" this internal hash.
    #[prost(message, required, tag = "1")]
    pub range: Range,

    #[prost(message, required, tag = "2")]
    /// The internal hash value.
    pub hash: TxOutMembershipHash,
}
```

Figure 4.3: transaction/core/src/tx.rs#L341-L351

```rust
#[derive(Clone, Copy, Deserialize, Eq, Hash, PartialEq, Serialize, Message, Digestible)]
pub struct Range {
    #[prost(uint64, tag = "1")]
    pub from: u64,
    #[prost(uint64, tag = "2")]
    pub to: u64,
}
#[allow(clippy::len_without_is_empty)]
impl Range {
    pub fn new(from: u64, to: u64) -> Result<Self, RangeError> {
        if from <= to {
            Ok(Self { from, to })
        } else {
            Err(RangeError {})
        }
    }
```

```
    /// The number of indices in this Range.
    pub fn len(&self) -> u64 {
        self.to - self.from + 1
    }
}

/// Ranges are ordered by `len`. If two ranges have equal `len`, they are ordered by
`from`.
impl Ord for Range {
    fn cmp(&self, other: &Range) -> Ordering {
        if self.len() != other.len() {
            self.len().cmp(&other.len())
        } else {
            self.from.cmp(&other.from)
        }
    }
}
```

Figure 4.4: transaction/core/src/range.rs#L13-L45

The path from `mc_transaction_core::validation::validate` is similar. This function calls `validate_membership_proofs` (figure 4.5), which calls `is_membership_proof_valid` (figure 4.6). The function `is_membership_proof_valid` tries to sort a vector of `Ranges` (figure 4.7), which causes the `Ranges` to be compared using the vulnerable code in figure 4.4.

```
pub fn validate<R: RngCore + CryptoRng>(
    tx: &Tx,
    current_block_index: u64,
    root_proofs: &[TxOutMembershipProof],
    csprng: &mut R,
) -> TransactionValidationResult<()> {
    ...
    validate_membership_proofs(&tx.prefix, &root_proofs)?;
```

Figure 4.5: transaction/core/src/validation/validate.rs#L27-L45

```
fn validate_membership_proofs(
    tx_prefix: &TxPrefix,
    root_proofs: &[TxOutMembershipProof],
) -> TransactionValidationResult<()> {
    ...
                        match is_membership_proof_valid(
                            tx_out_with_proofs.tx_out,
                            tx_out_with_proofs.membership_proof,
                            element.hash.as_ref(),
                        ) {
```

Figure 4.6: transaction/core/src/validation/validate.rs#L236-L303

```
pub fn is_membership_proof_valid(
    tx_out: &TxOut,
    proof: &TxOutMembershipProof,
    known_root_hash: &[u8; 32],
) -> Result<bool, Error> {
    ...
    let mut ranges: Vec<&Range> = range_to_hash.keys().collect();
    ranges.sort();
```

Figure 4.7: transaction/core/src/membership_proofs/mod.rs#L63-L94

Exploit Scenario
Eve sends a crafted transaction to a MobileCoin validator, causing the validator to behave incorrectly.

Recommendations
Short term, add checks of the inputs to `tx_is_well_formed` and `validate_membership_proofs` (e.g., check that each `Range`'s `from` field is no more than its `to` field). This will eliminate the risk that a panic can arise in debug builds.

Long term, consider incorporating fuzzing into your continuous integration process. Doing so could help to reveal similar bugs in the future.

References
- Add `finalizer` attribute hook to validate a deserialized structure
- Support a `#[serde(validate = "some_function")]` attribute on fields

(The implementation of either of these features in `serde` could help address the case in which `tx_is_well_formed`'s arguments are deserialized.)

5. Memory exhaustion when deserializing EnclaveCall

Severity: Medium
Type: Denial of Service
Target: `attest/core/src/traits.rs`

Difficulty: Low
Finding ID: TOB-MOB-005

Description
Deserializing an `EnclaveCall` can result in memory exhaustion.

The vulnerable code appears in figure 5.1, which is part of the `impl_sgx_wrapper_reqs` macro. When deserializing a sequence, the `visit_seq` function allocates a vector of size `seq.size_hint()`, which can be arbitrarily large. This results in memory exhaustion.

```
                    #[inline]
                    fn visit_seq<A: $crate::traits::SeqAccess<'de>>(
                        self,
                        mut seq: A,
                    ) -> core::result::Result<Self::Value, A::Error>
                    where
                        A::Error: $crate::traits::DeserializeError
                    {
                        use $crate::traits::TryFrom;
                        use $crate::traits::DeserializeError;

                        let mut bytes =
$crate::traits::Vec::<u8>::with_capacity(seq.size_hint().unwrap_or(1024usize));
```

Figure 5.1: attest/core/src/traits.rs#L116-L128

For comparison, serde's code for deserializing a sequence appears in figures 5.2 and 5.3. Note that the size of the vector is bounded to 4096.

```
            fn visit_seq<A>(self, mut seq: A) -> Result<Self::Value, A::Error>
            where
                A: SeqAccess<'de>,
            {
                let mut values = Vec::with_capacity(size_hint::cautious(seq.size_hint()));
```

Figure 5.2: serde/src/de/impls.rs#L863-L867

```
    #[inline]
    pub fn cautious(hint: Option<usize>) -> usize {
        cmp::min(hint.unwrap_or(0), 4096)
    }
```

Figure 5.3: serde/src/private/de.rs#L201-L204

Exploit Scenario

Eve sends a crafted transaction to a MobileCoin validator, causing the validator to exhaust memory.

Recommendations
Short term, bound the size of the vector that `visit_seq` allocates, similar to how `serde` does (figures 5.2 and 5.3). This will eliminate the risk of a memory exhaustion attack.

Long term, consider incorporating fuzzing into your continuous integration process. Doing so could help to reveal similar bugs in the future.

6. Out-of-bounds memory access in Quote

Severity: Low
Type: Data Validation
Target: `attest/core/src/quote.rs`

Difficulty: Low
Finding ID: TOB-MOB-006

Description
A `Quote` is a wrapper around a vector of bytes (figure 6.1). The size of the vector could be smaller than expected by certain functions that operate on the vector, resulting in an out-of-bounds memory access.

We observed out of an out-of-bounds access in `version` (figure 6.2). However, it looks as though other functions in `attest/core/src/quote.rs` may be similarly vulnerable.

```
#[derive(Clone, Deserialize, Eq, Hash, Ord, PartialEq, PartialOrd, Serialize)]
pub struct Quote(Vec<u8>);
```

Figure 6.1: attest/core/src/quote.rs#L141-L142

```
    pub fn version(&self) -> u16 {
        u16::from_le_bytes(
            (&self.0[QUOTE_VERSION_START..QUOTE_VERSION_END])
                .try_into()
                .unwrap(),
        )
    }
```

Figure 6.2: attest/core/src/quote.rs#L169-L175

We managed to trigger this vulnerability only by writing a `Quote` to the console. Therefore, this finding is of low severity. We may increase the finding's severity if additional vulnerable code paths are found.

Exploit Scenario
Eve discovers a code path leading to a vulnerable function in `attest/core/src/quote.rs`. Eve uses the code path to crash MobileCoin nodes.

Recommendations
Short term, write a validation function for `Quote` and ensure that the function is called for any `Quote` before it is used. Alternatively, ensure that any function that operates on a `Quote` (e.g., `version`) checks its validity before accessing its internals. (Note that this latter option may require changes to the return types of some functions, such as `version`.)

Taking one of these steps will reduce the likelihood of accessing out-of-bounds memory through a `Quote`.

Long term, consider incorporating fuzzing into your continuous integration process. Doing so could help to reveal similar bugs in the future.

7. No instructions on how to reproduce the SGX enclave build

Severity: Informational
Difficulty: Low
Type: Patching
Finding ID: TOB-MOB-007
Target: `libconsensus-enclave.so`

Description
It is important for users to verify that the measured enclave was built from the source code, as the released enclaves are a black box that cannot be inspected by anyone besides the signer. While we confirmed that the SGX enclave build is reproducible, there are no official instructions on how to perform the build. This lowers the chances that third-parties will build and verify the code, eroding trust in the service.

The SGX enclave is built from the open source code published on GitHub. The build dependencies are encapsulated in a Docker container that is used to build the SGX and non-SGX code. The Docker image build is a heavy process that takes time and results in a slightly different image each time it is built, despite being built from the same Dockerfile (Dockerfile builds are non-reproducible). To help with build times and ensure reproducibility, a Dockerfile-version file is used. The file tracks versions of the Dockerfile and has to be updated manually by a developer each time the Dockerfile contents change. Thanks to this mechanism, the Dockerfile builds can be cached and serve as a bootstrap for further SGX enclave builds.

Exploit Scenario
A MobileCoin user wants to check whether a node is running the same code as published. The user cannot find the process for reproducing the build and resigns from using the service, assuming it is not possible.

Recommendations
Short term, design and document the process for reproducing the enclave from the source code. Set up at least two different CI servers that will check releases for non-reproducibility.

Long term, investigate Nix as an alternative build system designed to maximize reproducibility. The bootstrap size resulting from the Docker image is big, and having a build system that focuses on reproducibility would improve transparency and trust for the users.

References
- Nix and NixOS

8. Panic in derive_proof_at_index

Severity: Medium
Type: Data Validation
Target: `transaction/core/src/membership_proofs/mod.rs`

Difficulty: Low
Finding ID: TOB-MOB-008

Description
The two `expect` statements in figure 8.1 are triggerable in the normal execution of `mc_transaction_core::validation::validate`. When triggered, they cause the program to panic. Such behavior could be exploited to carry out a denial-of-service attack.

```
    let mut derived_elements: HashMap<Range, [u8; 32]> = HashMap::default();
    ...
           // Left child.
           let left_child_hash = {
                let left_child_range = Range::new(element.range.from, mid)?;
                *derived_elements
                    .get(&left_child_range)
                    .expect("Child range should already exist.")
            };

            // Right child.
            let right_child_hash = {
                let right_child_range = Range::new(mid + 1, element.range.to)?;
                *derived_elements
                    .get(&right_child_range)
                    .expect("Child range should already exist.")
            };
```

Figure 8.1: transaction/core/src/membership_proofs/mod.rs#L184-L222

Exploit Scenario
Eve sends a crafted transaction to a MobileCoin validator, causing the validator to crash.

Recommendations
Short term, replace the uses of `expect` in figure 8.1 with `ok_or` applied to an appropriate error. This will eliminate the risk that a panic could be exploited to carry out a denial-of-service attack.

Long term, consider incorporating fuzzing into your continuous integration process. Doing so could help to reveal similar bugs in the future.

9. Intel Attestation Service (IAS) is a single point of failure

Severity: Informational
Type: Access Controls
Target: Various in `attest`

Difficulty: High
Finding ID: TOB-MOB-009

Description
MobileCoin achieves enhanced privacy by verifying ring signatures within an Intel SGX enclave. However, such enhanced privacy comes at the cost of relying on the Intel Attestation Service (IAS). If a MobileCoin node is unable to contact IAS, the node cannot verify enclave quotes sent to it. In this sense, IAS represents a single point of failure.

```
cfg_if! {
    if #[cfg(feature = "ias-dev")] {
        const IAS_BASEURI: &str =
"https://api.trustedservices.intel.com/sgx/dev/attestation/v3";
    } else {
        const IAS_BASEURI: &str =
"https://api.trustedservices.intel.com/sgx/attestation/v3";
    }
}
```

Figure 8.1: attest/net/src/ias.rs#L21-L27

Note: We tried to determine whether IAS has ever gone offline in the past. We found no record of such an event.

Exploit Scenario
The US and Wonderland engage in a trade war. As Intel is a US company, traffic to Intel servers is blocked by Wonderland. Alice, who runs MobileCoin nodes in Wonderland, is unable to verify enclave quotes sent to her nodes.

Recommendations
Short term, document the fact that if IAS goes offline, so does MobileCoin. This will alert users to this potential risk.

Long term, consider strategies to mitigate the risks associated with IAS's being a single point of failure. One option could be to run a backup attestation service. Another could be to try to resolve the problem using the peer-to-peer network (e.g., a subset of the nodes vouches for an enclave's authenticity in the event that IAS cannot be contacted). Finding a way to mitigate this risk could allow MobileCoin to continue to function if IAS goes offline or becomes unreachable.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing a system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking, or the order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices or Defense in Depth.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is relatively small or is not a risk the customer has indicated is important.
Medium	Individual users' information is at risk; exploitation could pose reputational, legal, or moderate financial risks to the client.

High	The issue could affect numerous users and have serious reputational, legal, or financial implications for the client.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is commonly exploited; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of a complex system.
High	An attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components
Arithmetic	Related to the proper use of mathematical operations and semantics
Assembly Use	Related to the use of inline assembly
Centralization	Related to the existence of a single point of failure
Upgradeability	Related to contract upgradeability
Function Composition	Related to separation of the logic into functions with clear purposes
Front-Running	Related to resilience against front-running
Key Management	Related to the existence of proper procedures for key generation, distribution, and access
Monitoring	Related to the use of events and monitoring procedures
Specification	Related to the expected codebase documentation
Testing and Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.)

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed, and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.

Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Non-Security-Related Findings

This appendix contains findings that do not have immediate or obvious security implications.

- **The build script does not check for each file's presence independently.** In the following code in [sgx/core-types-sys/build.rs](), it is unclear which `sgx.h` and `sgx_eid.h` could not be found:

```
if !sgx_h_found || !sgx_eid_h_found {
    panic!("Could not find both sgx.h and sgx_eid.h in our include paths");
}
```

  Consider having the script check for each file's presence independently.

- **Code duplication exists in `ecall_dispatcher`.** In the following code in [consensus/enclave/trusted/src/lib.rs](), the method invocation and error check ".or(Err(sgx_status_t::SGX_ERROR_UNEXPECTED))?" appears multiple times:

```
    let outdata = match call_details {
        // Utility methods
        EnclaveCall::EnclaveInit(peer_self_id, client_self_id, sealed_key) => {
            serialize(&ENCLAVE.enclave_init(&peer_self_id, &client_self_id,
&sealed_key))
                .or(Err(sgx_status_t::SGX_ERROR_UNEXPECTED))?
        }
        ...
        EnclaveCall::FormBlock(parent_block, encrypted_txs_with_proofs) => {
            serialize(&ENCLAVE.form_block(&parent_block,
&encrypted_txs_with_proofs))
                .or(Err(sgx_status_t::SGX_ERROR_UNEXPECTED))?
        }
    };
```

  Consider having it appear just once at the end:

```
    let outdata = match call_details {
        // Utility methods
        EnclaveCall::EnclaveInit(peer_self_id, client_self_id, sealed_key) => {
            serialize(&ENCLAVE.enclave_init(&peer_self_id, &client_self_id,
&sealed_key))
```

```
            }
            ...
            EnclaveCall::FormBlock(parent_block, encrypted_txs_with_proofs) => {
                serialize(&ENCLAVE.form_block(&parent_block,
&encrypted_txs_with_proofs))
            }
        }
        .or(Err(sgx_status_t::SGX_ERROR_UNEXPECTED))?;
```

- **The code that checks `root_elements` seems overly complex.** The following code in consensus/enclave/impl/src/lib.rs verifies that all of the `root_elements` are the same:

```
            // Enforce that all membership proofs provided by the untrusted system
for transaction validation
            // came from the same ledger state. This can be checked by requiring all
proofs to have the same root hash.
            let mut root_elements = BTreeSet::new();
            for proof in &proofs {
                let root_element = proof
                    .elements
                    .last() // The last element contains the root hash.
                    .ok_or(Error::InvalidLocalMembershipProof)?;
                root_elements.insert(root_element);
            }
            if root_elements.len() != 1 {
                return Err(Error::InvalidLocalMembershipProof);
            }
```

  Given how simple the problem is that the code solves, the use of a BTree seems rather heavy-handed. Consider adopting a simpler solution.

- **`enclave_init` makes global state changes before panicking.** The first thing that `enclave_init` does is call `AkeEnclaveState::init`:

```
fn enclave_init(
     &self,
     peer_self_id: &ResponderId,
     client_self_id: &ResponderId,
     sealed_key: &Option<SealedBlockSigningKey>,
```

```
    ) -> Result<(SealedBlockSigningKey, Vec<String>)> {
        self.ake
            .init(peer_self_id.clone(), client_self_id.clone())?;
```

`AkeEnclaveState::init` sets two fields protected by `Mutexes`:

```
    pub fn init(&self, peer_self_id: ResponderId, client_self_id: ResponderId)
-> Result<()> {
        let mut peer_lock = self.peer_self_id.lock()?;
        let mut client_lock = self.client_self_id.lock()?;
        if peer_lock.is_none() && client_lock.is_none() {
            *peer_lock = Some(peer_self_id);
            *client_lock = Some(client_self_id);
            Ok(())
        } else {
            Err(Error::AlreadyInit)
        }
    }
```

`enclave_init` can (erroneously) panic following the call to `AkeEnclaveState::init` (see #287, ConsensusEnclave::enclave_init panics on unparsable sealed key). Naively turning those panics into error-returns could allow the two fields to remain set, leading to subsequent problems. Consider having `enclave_init` make global state changes only after an error can no longer occur.

- **There is a typo in a comment in `crypto/noise/src/handshake_state.rs`.** In the following comment, “ee” should be “se.”

```
/// Do an identity-binding DH (that is, an "ee" or "es" operation).
```

- **There is a typo in a local variable name in `consensus/enclave/impl/src/lib.rs`.** The variable name is misspelled:

```
let mut corrputed_locally_encrypted_tx = locally_encrypted_tx.clone();
```

- **Validation functions are not imported consistently.** In the following code in `transaction/core/src/validation/validate.rs`, `validate_ring_elements_are_sorted` is imported separately from all other validation functions:

```
use crate::{
    constants::{MINIMUM_FEE, RING_SIZE},
```

```
    tx::{Tx, TxOutMembershipHash, TxOutMembershipProof},
    validation::{
        error::TransactionValidationError,
        validate::{
            validate_inputs_are_sorted, validate_key_images_are_unique,
            validate_membership_proofs, validate_number_of_inputs,
validate_number_of_outputs,
            validate_outputs_public_keys_are_unique,
validate_ring_elements_are_unique,
            validate_ring_sizes, validate_signature, validate_tombstone,
            validate_transaction_fee, MAX_TOMBSTONE_BLOCKS,
        },
    },
};

use crate::validation::validate::validate_ring_elements_are_sorted;
```

Consider moving the import of `validate_ring_elements_are_sorted` above with the other validation functions.

- **The `encrypt_with_ad` and `decrypt_with_ad` documentation is not accurate in [`crypto/noise/src/cipher_state.rs`](#).** The documentation states that the nonce will reset after 2**56 bytes are encrypted. The constant `72_057_594_037_927_940` is actually 2**56 + 4. Before the comparison, the new message length is added first to `self.sent_bytes`, which makes the comment inaccurate when `msg_len` is bigger than 4. Also, the same constant appears in two places; factor it out as `const MAX_SENT_BYTES: u64 = 2**56+4` to avoid repetition.