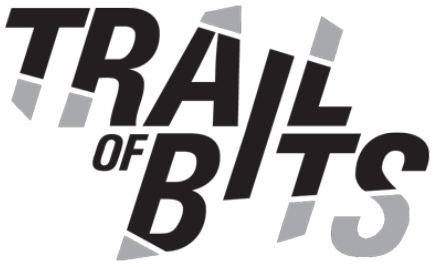

MobileCoin BFT

Security Assessment

November 2, 2020

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

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

Dominik Czarnota | *Trail of Bits*
dominik.czarnota@trailofbits.com

Will Song | *Trail of Bits*
will.song@trailofbits.com

Changelog:
October 23, 2020: Initial report delivered
November 2, 2020: Updated TOB-MOBBFT-004 and TOB-MOBBFT-005. Added TOB-MOBBFT-014.

Executive Summary

From October 13 to October 23, 2020, MobileCoin engaged Trail of Bits to review the security of its Byzantine fault tolerance (BFT) consensus implementation. Trail of Bits conducted this assessment over four person-weeks, with three engineers working from commit `11a2932e` of the `mobilecoin` repository.

During the first week, we verified that we could build the codebase. We also verified that the unit tests pass, and we ran them with thread sanitizer enabled. We ran `cargo-audit` and `cargo-clippy` over the codebase. We then began a manual review, looking for unintentional deviations from IETF's draft of the Stellar Consensus Protocol (SCP) specification. Finally, we developed fuzzers for `consensus_msg::from_scp_msg` and `consensus_msg::verify_signature`.

During the second week, we continued our manual review. We developed a fuzzer for `Node::handle_messages`. We also tested the consensus algorithm in the presence of Byzantine (i.e., non-responsive or malicious) nodes.

Our efforts resulted in 14 findings ranging from high to informational severity, with three findings of undetermined severity. The two high-severity findings concern safety violations observed in the presence of malicious nodes and private key files created with overly broad permissions.

One medium-severity finding concerns the reliance on a crate with a Rust Security (RUSTSEC) advisory. A second medium-severity finding concerns failures observed in metamesh tests with certain parameters. Two additional medium-severity findings involve potential denial-of-service vectors.

The five informational-severity findings concern the staleness of the `RustCrypto-utils` dependency, insufficient validation of `ResponderID`s, insufficient consensus message validation, and unused metrics counters. Two findings of undetermined severity concern an assertion violation and an arithmetic underflow revealed through fuzzing. The third finding of undetermined severity concerns the potential failure to report errors from `Node::handle_messages`.

In addition to the 14 findings, we provide an appendix containing non-security-related findings (Appendix C).

Our main recommendation is to write additional tests for the consensus implementation. Such tests should expand on the existing set of parameter choices and should incorporate Byzantine nodes. We implemented some of these tests and encountered numerous errors in the process. Expanding the consensus implementation tests could reveal similar errors.

Project Dashboard

Application Summary

Name	MobileCoin Byzantine Fault Tolerance (BFT)
Version	11a2932ec941e2ab8d1c852ab6414eeafe0705a2
Type	Rust
Platforms	POSIX

Engagement Summary

Dates	October 13–23, 2020
Method	Full knowledge
Consultants Engaged	3
Level of Effort	4 person-weeks

Vulnerability Summary

Total High-Severity Issues	2	■ ■
Total Medium-Severity Issues	4	■ ■ ■ ■
Total Low-Severity Issues	0	
Total Informational-Severity Issues	5	■ ■ ■ ■ ■
Total Undetermined-Severity Issues	3	■ ■ ■
Total	14	

Category Breakdown

Access Controls	1	■
Auditing and Logging	1	■
Data Validation	2	■ ■
Denial of Service	4	■ ■ ■ ■
Patching	2	■ ■
Undefined Behavior	4	■ ■ ■ ■
Total	14	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Not applicable.**
Arithmetic	**Satisfactory.** A potential arithmetic underflow issue was found regarding `Ballot` counters. No other arithmetic issues were noted.
Assembly Use	**Not applicable.**
Centralization	**Strong.** The SCP is designed for networks with open membership. We saw nothing in MobileCoin's implementation that would prevent this.
Function Composition	**Satisfactory.** The SCP (like most consensus protocols) is complex. In at least one instance, we believe this complexity may have led to a type-confusion error. However, the code is well documented and well structured, which somewhat remediates the complexity.
Front-Running	**Moderate.** Several safety violations were observed in the presence of a node that maliciously broadcasts random nominate messages.
Key Management	**Moderate.** We noted that the `keygen` binary creates files with overly broad permissions.
Monitoring	**Not considered.**
Specification	**Strong.** An IETF draft specification exists for the SCP. (An academic paper and at least one helpful blog post also exist.)
Testing and Verification	**Moderate.** The metamesh tests should be expanded to include additional parameter choices. Additional mesh tests involving Byzantine nodes should be written. There are also areas of the code that lack testing, such as tests for the `ByzantineLedgerWorker` struct. On the positive side, assertions and invariants are used extensively and appropriately.

Engagement Goals

The engagement was scoped to provide a security assessment of MobileCoin's SCP implementation.

Specifically, we sought to answer the following questions:

- Are double spends possible?
- Is it possible to create money?
- Is it possible to steal others' money?
- Could any of these risks occur without compromising a majority of the enclaves?
- Is it possible to make the blockchain halt?
- Is it possible to make the blockchain fork?

Coverage

Peer management. We verified that the unit tests pass and ran them under ThreadSanitizer. We manually reviewed the component, statically analyzed it using `cargo-audit` and `cargo-clippy`, and fuzzed `consensus_msg::from_scp_msg` and `consensus_msg::verify_signature` using `cargo-afl`.

SCP implementation. We verified that the unit tests pass and ran them under ThreadSanitizer. We manually reviewed the component, statically analyzed it using `cargo-audit` and `cargo-clippy`, and fuzzed `Node::handle_messages` using `cargo-afl`. We incorporated malicious nodes into the mesh tests and expanded the set of parameters used in the metamesh tests.

Consensus service. We verified that the unit tests pass and ran them under ThreadSanitizer. We manually reviewed the component and statically analyzed it using `cargo-audit` and `cargo-clippy`.

Ledger. We verified that the unit tests pass and ran them under ThreadSanitizer. We manually reviewed the component and statically analyzed it using `cargo-audit` and `cargo-clippy`.

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

❑ **Disable use of `sentry` until a fix becomes available.** This will eliminate the memory corruption that could arise from using this crate. TOB-MOBBFT-001

❑ **Update the `eranrund/RustCrypto-utils` library to contain the patches from the upstream.** This will help prevent bugs related to OID validation if the modified code is one day used in the MobileCoin codebase. TOB-MOBBFT-002

❑ **Change the `ResponderId::from_str` function so that it validates the input's format completely, rather than simply checking whether it contains a colon character.** This will prevent issues in code that expects the function to fully validate the format. TOB-MOBBFT-003

❑ **Determine the cause of the assertion violation and either expand the existing invariant checks or add a runtime error check, as appropriate.** This will eliminate an assertion violation that could be used to carry out denial-of-service attacks. TOB-MOBBFT-004

❑ **Determine the cause of the arithmetic underflow and either expand the existing invariant checks or add a runtime error check, as appropriate.** This will eliminate an arithmetic underflow that could lead to undefined behavior. TOB-MOBBFT-005

❑ **Determine why safety violations occur when a node broadcasts random nominate messages.** This will help honest, non-faulty nodes reach consensus. TOB-MOBBFT-006

❑ **Determine the cause of the poor performance associated with hierarchical quorum sets and small thresholds.** Since these parameters are configurable by node operators, a poor choice could adversely affect the network as a whole. TOB-MOBBFT-007

❑ **Change the `write_keyfile` function to create the private keyfiles with `0700` permissions.** This will prevent other users on the same system from accessing private keys generated with the `keygen` utility. TOB-MOBBFT-008

❑ **Implement the rejection of incorrectly ordered values in the `handle_messages` function** to ensure that messages from malfunctioning nodes are not processed needlessly. TOB-MOBBFT-009

❑ **Update or remove the unused metrics counters in MobileCoin's consensus code.** This will let users analyze the statistics of the protocol used by MobileCoin. TOB-MOBBFT-010

❑ **Implement a maximum gRPC message-length limit of a few megabytes to prevent denial-of-service attacks prompted by massive server requests.** TOB-MOBBFT-011

❑ **Consider changing the logic in the `process_consensus_msgs` function to first handle messages and then to broadcast only those that are successfully handled.** This will prevent the system from processing messages that will fail on multiple nodes anyway. TOB-MOBBFT-012

❑ **Investigate the `node`'s `handle_messages` function and the functions it calls that return a `Result` type to determine whether they can return an error.** If they cannot return an error, refactor the functions to return proper value types (e.g., `Vec<Msg<V>>` instead of `Result<Vec<Msg<V>>, String>` for the `node`'s `handle_messages` function). This will increase the readability and correctness of the code. TOB-MOBBFT-013

❑ **Three recommendations:**

- **Eliminate the phase checks** in `check_prepare_phase_invariants` and `check_commit_phase_invariants`.
- **Ensure that `check_prepare_phase_invariants` is called** along all code paths in `do_prepare_phase`.
- **Ensure that `check_commit_phase_invariants` is called** along all code paths in `do_commit_phase`.

This will eliminate assertion violations that could be used to carry out denial-of-service attacks. TOB-MOBBFT-014

Long term

❑ **Regularly run `cargo-audit` over the codebase to reveal similar bugs.** TOB-MOBBFT-001

❑ **Regularly check the forked dependencies** for updates in their upstream repositories or use the upstream repositories instead of the forks (e.g., `RustCrypto/utils` instead of `eranrund/RustCrypto-utils`). TOB-MOBBFT-002

❑ **Add tests for the `ResponderId::from_str` function to ensure that it returns an error for invalid inputs.** This will help prevent issues if the code is changed and the responder IDs are used differently. TOB-MOBBFT-003

❑ **Incorporate fuzzing into your continuous integration process** to reveal similar bugs in the future. TOB-MOBBFT-004, TOB-MOBBFT-005, TOB-MOBBFT-014

❑ **Three recommendations:**

- **Expand the set of mesh tests** to include tests with malicious nodes. This will flag bugs like the one that now seems to involve nodes broadcasting random nominate messages.
- **Adjust the code** so that a "value" is a sequence of `TxHash`es. This will help align the code with the draft SCP specification.
- **Allow nodes to nominate only one "value,"** such as a sequence of `TxHash`es. Punish nodes that do not (e.g., by removing them from quorum sets, to limit the effect of malicious nodes on the network). TOB-MOBBFT-006

❑ **Expand the set of metamesh tests to include more widely varied parameters, especially with small thresholds.** This will flag bugs like the one that now seems to involve small thresholds. TOB-MOBBFT-007

❑ **Investigate whether the handling of the users' private keys could be offloaded to a service such as Vault.** Avoid handling sensitive information to prevent its disclosure. TOB-MOBBFT-008

❑ **Ensure that all "TODO" items are monitored in an issue-tracking system and dealt with before deployment.** TOB-MOBBFT-009

❑ **Add tests for MobileCoin metrics counters so that the values are set correctly according to what happens within the SCP protocol.** This will help ensure such counters do not go unused accidentally. TOB-MOBBFT-010

❑ **Carefully consider decisions to deviate from network-related defaults.** These defaults are based on the experience of the community as a whole, so deviations must be absolutely necessary and weighed carefully. This will keep your systems configured to withstand common threats and to not attract undue attention. TOB-MOBBFT-011

❑ **As suggested in TOB-MOBBFT-006, incorporate malicious nodes into your tests.** Do this even for end-to-end tests. This could help alert you to denial-of-service vectors associated with the order in which expensive operations are performed. TOB-MOBBFT-012

❑ **When implementing functions that may return a `Result` type, ensure that all return types are covered by unit tests.** Here, if the `Result` return types are intended, add such tests for the `node`'s `handle_messages` function and the functions it calls that return a `Result` type. TOB-MOBBFT-013

Findings Summary

#	Title	Type	Severity
1	Codebase relies on a crate with a RUSTSEC advisory	Patching	Medium
2	The RustCrypto-utils dependency is behind two commits that added validation	Patching	Informational
3	Insufficient validation of responder IDs	Data Validation	Informational
4	Assertion violation in `Slot::out_msg`	Denial of Service	Undetermined
5	Arithmetic underflow in `Slot::out_msg`	Undefined Behavior	Undetermined
6	Mesh tests fail sporadically in the presence of malicious nodes	Undefined Behavior	High
7	Metamesh tests fail sporadically with certain parameters	Undefined Behavior	Medium
8	The keygen binary saves keyfiles with overly broad permissions	Access Controls	High
9	Messages with incorrectly ordered values are not rejected	Data Validation	Informational
10	Some metrics counters are never updated	Auditing and Logging	Informational
11	Potential denial of service due to excessive gRPC message-length limit	Denial of Service	Medium
12	Broadcasting and then handling resolved messages may fail	Denial of Service	Informational
13	The `node`'s `handle_messages` function always returns an “Ok” `Result` in the current codebase	Undefined Behavior	Undetermined

14	Overly restrictive checks in `Slot::check_prepare_phase_invariants`	Denial of Service	Medium

1. Codebase relies on a crate with a RUSTSEC advisory

Severity: Medium
Type: Patching
Target: `mc-common` crate

Difficulty: Undetermined
Finding ID: TOB-MOBBFT-001

Description
MobileCoin's `mc-common` crate relies on `sentry`, which provides support for logging events and errors/panics to the Sentry error logging service. The `sentry` crate is affected by Rust Security (RUSTSEC) advisory RUSTSEC-2020-0041. The advisory specifically affects the `sized-chunks` crate (an indirect dependency of `sentry`) and indicates the following:

> Chunk:
>
> - Array size is not checked when constructed with `unit()` and `pair()`.
> - Array size is not checked when constructed with `From<InlineArray<A, T>>`.
> - `Clone` and `insert_from` are not panic-safe; A panicking iterator causes memory safety issues with them.
>
> InlineArray:
>
> - Generates unaligned references for types with a large alignment requirement.

At the time of this writing, no fix appears to be available for either `sentry` or `sized-chunks`.

Note that `cargo-audit` warns that MobileCoin is also vulnerable to RUSTSEC-2020-0019, which we reported previously (TOB-MOB-001). That advisory concerns `tokio-rustls 0.13.0`, which seems to be referenced only by the `Cargo.lock` file. We recommend updating the `Cargo.lock` file to eliminate the needless warning.

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, disable use of `sentry` until a fix becomes available. This will eliminate the potential memory corruption that could arise from using this crate.

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

References
- Multiple soundness issues in Chunk and InlineArray
- Soundness issues in dependency `sized-chunks` (via `im` crate)
- `cargo-audit`

2. The RustCrypto-utils dependency is behind two commits that added validation

Severity: Informational
Type: Patching
Target: `RustCrypto-utils` crate

Difficulty: High
Finding ID: TOB-MOBBFT-002

Description
The `eranrund/RustCrypto-utils` dependency used by MobileCoin is behind two commits from its upstream version that added compile-time `const-oid` validation.

It seems that the modified code is not used directly in the MobileCoin codebase. However, if it is used in the future, the additional validation may prevent bugs.

Recommendations
Short term, update the `eranrund/RustCrypto-utils` library to contain the patches from the upstream. This will help prevent bugs related to OID validation if the modified code is one day used in the MobileCoin codebase.

Long term, regularly check the forked dependencies for updates in their upstream repositories or use the upstream repositories instead of the forks (e.g., `RustCrypto/utils` instead of `eranrund/RustCrypto-utils`).

3. Insufficient validation of responder IDs

Severity: Informational
Type: Data Validation
Target: `mobilecoin/common/src/responder_id.rs`

Difficulty: Medium
Finding ID: TOB-MOBBFT-003

Description
The `ResponderId::from_str` function validates the input string by checking whether it contains a colon character; the responder ID should consist of a host and a port delimited by a colon (figure 3.1). This check is insufficient and makes it possible to create `ResponderId` objects from invalid formats.

The severity of this issue is informational, as it does not seem to be currently exploitable. The only place in which the `ResponderId::from_str` function is used with an input is the send_consensus_msg function, which then passes the responder ID to the handle_consensus_msg function, which checks whether it is a known responder ID.

```
impl FromStr for ResponderId {
    type Err = ResponderIdParseError;

    fn from_str(src: &str) -> Result<ResponderId, Self::Err> {
        // ResponderId is expected to be host:port, so at least ensure we have a single
colon as a
        // small sanity test.
        if !src.contains(':') {
            return Err(ResponderIdParseError::InvalidFormat(src.to_string()));
        }

        Ok(Self(src.to_string()))
    }
}
```

Figure 3.1: The `ResponderId::from_str` function (mobilecoin/common/src/responder_id.rs#L38-L50)

Recommendations
Short term, change the `ResponderId::from_str` function so that it validates the input's format completely, rather than simply checking whether it contains a colon character. This will prevent issues in code that expects the function to fully validate the format.

Long term, add tests for the `ResponderId::from_str` function to ensure that it returns an error for invalid inputs. This will help prevent issues if the code is changed and the responder IDs are used differently.

4. Assertion violation in Slot::out_msg

Severity: Undetermined | Difficulty: Undetermined
Type: Denial of Service | Finding ID: TOB-MOBBFT-004
Target: `consensus/scp/src/slot.rs`, `consensus/scp/src/msg.rs`

Description
Fuzzing `Node::handle_messages` produced an assertion violation in `Slot::out_msg`, which could be used to carry out denial-of-service attacks.

We fuzzed both the `self` and `msg` parameters of `Node::handle_messages` (figure 4.1) and obtained several crashes. In several cases, the crashes were eliminated by adding invariant checks to the start of `Node::handle_messages` (figure 4.2). However, several crashes could not be ruled out. This finding describes two of them. The others are described in TOB-MOBBFT-005 and TOB-MOBBFT-014.

```
    /// Handle incoming message from the network.
    fn handle_messages(&mut self, msgs: Vec<Msg<V>>) -> Result<Vec<Msg<V>>, String> {
```

Figure 4.1: consensus/scp/src/node/node_impl.rs#L181-L182

```
    fn handle_messages_invariants(&self) -> bool {
        if !self.Q.is_valid() {
            return false;
        }

        if !self.current_slot.quorum_set.is_valid() {
            return false;
        }

        if let (Some(C), Some(H)) = (&self.current_slot.C, &self.current_slot.H) {
            if !(C.N <= H.N) {
                return false;
            }
        }

        return true;
    }
```

Figure 4.2: Invariant checks added to the beginning and end of `Node::handle_messages`

Certain inputs produced an assertion violation in `Slot::out_msg` (figure 4.3). The assertion is that the current outgoing message is valid. The message can be invalid because it contains a `PreparePayload` whose `CN` value is larger than its `HN` value, or whose `HN` value is larger than its `B.N` value (figure 4.4).

```
    fn out_msg(&mut self) -> Option<Msg<V>> {
        ...
        let topic_opt = match self.phase {
            ...
        };
```

```
        let msg_opt = topic_opt.map(|topic| {
            Msg::new(
                self.node_id.clone(),
                self.quorum_set.clone(),
                self.slot_index,
                topic,
            )
        });

        // Suppress duplicate outgoing messages.
        if let Some(msg) = msg_opt {
            assert_eq!(msg.validate(), Ok(()));
```

Figure 4.3: consensus/scp/src/slot.rs#L1231-L1393

```
    pub fn validate(&self) -> Result<(), String> {
        ...
        let validate_prepare = |payload: &PreparePayload<V>| -> Result<(), String> {
            ...
            if payload.CN > payload.HN {
                return Err(format!("CN > HN, msg: {}", self.to_display()));
            }

            if payload.HN > payload.B.N {
                return Err(format!("HN > BN, msg: {}", self.to_display()));
            }
            ...
        };

        match self.topic {
            ...
            Prepare(ref payload) => {
                validate_prepare(payload)?;
```

Figure 4.4: consensus/scp/src/msg.rs#L362-L409

We added an invariant check to the start of `Slot::handle_messages` to verify that the current outgoing message was either valid or nonexistent. However, this did not make the crash go away, which indicates that the invalid outgoing message was generated within the call to `Node::handle_messages`.

We cannot rule out the possibility that the node was in an unreachable state upon entry to `Node:handle_messages`. For this reason, this finding is of undetermined severity.

Exploit Scenario
Eve sends a crafted transaction to a MobileCoin node, causing an assertion violation followed by a crash.

Recommendations
Short term, determine the cause of the assertion violation and either expand the existing invariant checks or add a runtime error check, as appropriate. This will eliminate an assertion violation that could be used to carry out denial-of-service attacks.

Long term, incorporate fuzzing into your continuous integration process to reveal similar bugs in the future.

5. Arithmetic underflow in Slot::out_msg

Severity: Undetermined
Type: Undefined Behavior
Target: `consensus/scp/src/slot.rs`

Difficulty: Undetermined
Finding ID: TOB-MOBBFT-005

Description
Fuzzing `Node::handle_messages` produced an arithmetic underflow in `Slot::out_msg`, which could lead to undefined behavior.

This finding describes one of several crashes that could not be ruled out by adding invariant checks to the start of `Node::handle_messages`. The others are described in TOB-MOBBFT-004 and TOB-MOBBFT-014.

Certain inputs produced an arithmetic underflow in `Slot::out_msg` (figure 5.1). The underflow occurs when `self.B.N` is decremented. Thus, the underflow occurs when `self.B.N` is 0.

```
    fn out_msg(&mut self) -> Option<Msg<V>> {
        // Prepared is " the highest accepted prepared ballot not exceeding the "ballot"
field...
        // if "ballot = <n, x>" and the highest prepared ballot is "<n, y>" where "x < y",
        // then the "prepared" field in sent messages must be set to "<n-1, y>" instead of
"<n, y>""
        // See p.15 of the [IETF
draft](https://tools.ietf.org/pdf/draft-mazieres-dinrg-scp-04.pdf).

        let mut clamped_P: Option<Ballot<V>> = None;
        if let Some(P) = &self.P {
            if *P > self.B {
                if P.X > self.B.X {
                    clamped_P = Some(Ballot::new(self.B.N - 1, &P.X));
```

Figure 5.1: consensus/scp/src/slot.rs#L1231-L1241

Note that when a new slot is created, its `B.N` value is 0, so adding an invariant check that `self.B.N` is at least 0 would not be meaningful.

We cannot rule out the possibility that this crash could be eliminated with a more complicated invariant check. For this reason, this finding is of undetermined severity.

Exploit Scenario
Eve sends a crafted transaction to a MobileCoin node, causing an arithmetic underflow followed by undefined behavior.

Recommendations
Short term, determine the cause of the arithmetic underflow and either expand the existing invariant checks or add a runtime error check, as appropriate. This will eliminate an arithmetic underflow that could lead to undefined behavior.

Long term, incorporate fuzzing into your continuous integration process to reveal similar bugs in the future.

6. Mesh tests fail sporadically in the presence of malicious nodes

Severity: High
Type: Undefined Behavior
Target: `consensus/scp/src/node/node_impl.rs`

Difficulty: Low
Finding ID: TOB-MOBBFT-006

Description
The mesh tests fail sporadically when they include a node that maliciously and repeatedly broadcasts random nominate messages.

The main change to each `SCPNode` thread appears in figure 6.1. Exactly one node was designated as malicious (i.e., its `byzantine` flag was set to `true`). That node did not process consensus messages. Rather, it simply generated and broadcast nominate messages with random slots and random values.

```
        if byzantine && MALICIOUS {
            thread_local_node.reset_slot_index(rng.next_u64() % n_slots);

            thread_local_node.malign_current_slot();

            let mut values_to_propose = BTreeSet::<String>::default();
            let value = mc_util_test_helper::random_str(&mut rng, CHARACTERS_PER_VALUE);
            values_to_propose.insert(value);

            let outgoing_msg: Option<Msg<String>> = thread_local_node
                .propose_values(values_to_propose)
                .expect("propose_values() failed");

            if let Some(ref outgoing_msg) = outgoing_msg {
                (broadcast_msg_fn)(logger.clone(), outgoing_msg.clone());
                total_broadcasts += 1;
            } else {
                panic!("Should have outgoing_msg");
            }
        }

        if byzantine {
            continue;
        }
```

Figure 6.1: The main change made to each `SCPNode` thread, added just after the node's `receiver.try_recv()` handler

The presence of the malicious node often led to safety violations. An example appears in figure 6.2, in which two nodes have ledgers of different lengths. (Note that node `m2` is the malicious node in figure 6.1, as can be seen by its 11,446 broadcast messages. Compare this to 628 and 642 broadcast messages for nodes `m0` and `m1`, respectively.)

```
2020-10-21 16:53:57.411635791 UTC INFO ( testing ) externalized 1007/1000 values at node
m0, mc.test_name: test_mesh_networks::mesh_3k1, mc.module:
test_mesh_networks::mock_network, mc.src: consensus/scp/tests/mock_network/mod.rs:644
2020-10-21 16:53:57.412015225 UTC WARN ( testing ) externalized extra values at node m0,
```

```
mc.test_name: test_mesh_networks::mesh_3k1, mc.module: test_mesh_networks::mock_network,
mc.src: consensus/scp/tests/mock_network/mod.rs:656
2020-10-21 16:53:57.415272114 UTC INFO ( testing ) externalized 1007/1000 values at node
m1, mc.test_name: test_mesh_networks::mesh_3k1, mc.module:
test_mesh_networks::mock_network, mc.src: consensus/scp/tests/mock_network/mod.rs:644
2020-10-21 16:53:57.415596365 UTC WARN ( testing ) externalized extra values at node m1,
mc.test_name: test_mesh_networks::mesh_3k1, mc.module: test_mesh_networks::mock_network,
mc.src: consensus/scp/tests/mock_network/mod.rs:656
2020-10-21 16:53:57.418758569 UTC ERRO first_node_ledger.len() != other_node_ledger.len()
in run_test(), mc.test_name: test_mesh_networks::mesh_3k1, mc.module:
test_mesh_networks::mock_network, mc.src: consensus/scp/tests/mock_network/mod.rs:727
2020-10-21 16:53:57.419159952 UTC INFO thread results: m1,642,117, mc.test_name:
test_mesh_networks::mesh_3k1, mc.module: test_mesh_networks::mock_network, mc.src:
consensus/scp/tests/mock_network/mod.rs:470
2020-10-21 16:53:57.419869040 UTC INFO thread results: m0,628,117, mc.test_name:
test_mesh_networks::mesh_3k1, mc.module: test_mesh_networks::mock_network, mc.src:
consensus/scp/tests/mock_network/mod.rs:470
2020-10-21 16:53:57.429566784 UTC INFO thread results: m2,11446,0, mc.test_name:
test_mesh_networks::mesh_3k1, mc.module: test_mesh_networks::mock_network, mc.src:
consensus/scp/tests/mock_network/mod.rs:470
FAILED
```

Figure 6.2: Sample output in the presence of a malicious node

One solution to this problem might involve punishing nodes that nominate too many "values" for a slot. However, this solution cannot be immediately implemented in MobileCoin. For example, one would have to determine how many "values" are too many.

A closely related issue concerns MobileCoin's interpretation of the notion of "value" in IETF's draft of the SCP specification. A re-interpretation of this notion would bring MobileCoin more in line with the specification, and would suggest an answer to the question of how many "values" are too many.

The draft SCP specification states the following regarding "combining functions":

> ...SCP requires a..._combining function_ that reduces multiple candidate values into a single _composite_ value. (page 5)
>
> ...if one or more values are confirmed nominated, then "ballot.value" is taken as the output of the deterministic combining function applied to all confirmed nominated values. (page 15)

These passages suggest that the input to a combining function should be a collection of "values" and that the output of a combining function should be a single "value." However, MobileCoin's combining functions (figure 6.3) map sequences of "values" to sequences of "values" (with each "value" being a `TxHash`). Thus, there is a kind of "type mismatch" between the specification and the MobileCoin implementation.

```
    /// Combines the transactions that correspond to the given hashes.
     fn combine(&self, tx_hashes: &[TxHash]) -> TxManagerResult<Vec<TxHash>>;
```

Figure 6.3: consensus/service/src/tx_manager/tx_manager_trait.rs#L32-L33

One way to fix this "type mismatch" would be to interpret a "value" as a sequence `TxHash`es. Under such an interpretation, MobileCoin's combining functions would (following adjustment) map *sequences of sequences of* `TxHash`es to sequences of `TxHash`es.

Such an interpretation could also determine how many "values" are too many for a node to nominate during a slot. It may simply be "one" (i.e, a node could be allowed to nominate at most one sequence of `TxHash`es for a slot). Attempting to nominate any more than one sequence of `TxHash`es would be grounds for punishment.

Note that it is common for other systems (such as Eth2.0) to punish participants who propose multiple "values" within a round of consensus.

Exploit Scenario
Eve compromises Alice's MobileCoin node and causes it to broadcast random nominate messages. The nodes in Alice's quorum set cannot agree on the state of the ledger.

Recommendations
Short term, determine why safety violations occur when a node broadcasts random nominate messages. This will help honest, non-faulty nodes reach consensus.

Long term, take the following actions:

- Expand the set of mesh tests to include tests with malicious nodes. This will flag bugs like the one that now seems to involve nodes broadcasting random nominate messages.
- Adjust the code so that a "value" is a sequence of `TxHash`es. This will help align the code with the draft SCP specification.
- Allow nodes to nominate only one "value," such as a sequence of `TxHash`es. Punish nodes that do not (e.g., by removing them from quorum sets to limit the effect of malicious nodes on the network).

7. Metamesh tests fail sporadically with certain parameters

Severity: Medium | Difficulty: Low
Type: Undefined Behavior | Finding ID: TOB-MOBBFT-007
Target: `consensus/scp/tests/test_metamesh_network.rs`

Description
The metamesh tests fail sporadically when small thresholds are involved.

The existing metamesh tests appear in figure 7.1. Note that there is only one test in which a threshold differs from its associated number or nodes by more than one.

```
#[test_with_logger]
#[serial]
fn metamesh_3k2_3k1(logger: Logger) {
    metamesh_test_helper(3, 2, 3, 1, logger.clone());
}

#[test_with_logger]
#[serial]
fn metamesh_3k2_3k2(logger: Logger) {
    metamesh_test_helper(3, 2, 3, 2, logger.clone());
}

#[test_with_logger]
#[serial]
fn metamesh_3k2_4k3(logger: Logger) {
    metamesh_test_helper(3, 2, 4, 3, logger.clone());
}

#[test_with_logger]
#[serial]
fn metamesh_3k2_5k4(logger: Logger) {
    metamesh_test_helper(3, 2, 5, 4, logger.clone());
}
```

Figure 7.1: Existing test cases in `test_metamesh_network.rs` (consensus/scp/tests/test_metamesh_networks.rs#L34-L56)

To test the consensus implementation in the presence of Byzantine nodes (e.g., as in TOB-MOBBFT-008), we expanded the set of metamesh parameters that were tested. A node cannot expect all of its peers to agree upon a value if one of those peers is Byzantine. Therefore, we added some tests with thresholds that differ from their associated number or nodes by two (figure 7.2).

```
#[test_with_logger]
#[serial]
fn metamesh_4k2_3k1(logger: Logger) {
    metamesh_test_helper(4, 2, 3, 1, logger.clone());
}

#[test_with_logger]
#[serial]
```

```
fn metamesh_4k2_4k2(logger: Logger) {
    metamesh_test_helper(4, 2, 4, 2, logger.clone());
}

#[test_with_logger]
#[serial]
fn metamesh_4k2_5k3(logger: Logger) {
    metamesh_test_helper(4, 2, 5, 3, logger.clone());
}

#[test_with_logger]
#[serial]
fn metamesh_4k2_6k4(logger: Logger) {
    metamesh_test_helper(4, 2, 6, 4, logger.clone());
}
```

Figure 7.2: Test cases added to `test_metamesh_network.rs`

Even without a Byzantine node, we saw several such tests fail (i.e., time out without completing). When we increased the number of values to submit beyond one (figure 7.3), we often saw the tests stall halfway through.

```
    test_options.values_to_submit = 1;
```

Figure 7.3: consensus/scp/tests/test_metamesh_networks.rs#L27

Exploit Scenario
MobileCoin node operators configure their nodes to use hierarchical quorum sets with small thresholds, and the network stalls.

Recommendations
Short term, determine the cause of the poor performance associated with hierarchical quorum sets and small thresholds. Since these parameters are configurable by node operators, a poor choice could adversely affect the network as a whole.

Long term, expand the set of metamesh tests to include more widely varied parameters, especially with small thresholds. This will flag bugs like the one that now seems to involve small thresholds.

8. The keygen binary saves keyfiles with overly broad permissions

Severity: High | Difficulty: Medium
Type: Access Controls | Finding ID: TOB-MOBBFT-008
Target: `mobilecoin/util/keyfile/src/keygen.rs and lib.rs`

Description
The `keygen` binary built by the `util/keyfile` project creates private key files with `0666` permissions. Depending on the configuration of Linux umask, this may allow an attacker to read the private keyfile if he gets filesystem access to the machine on which the keys are generated.

Figure 8.1 shows a way to see the permissions that the keyfiles are created with.

```
$ strace -e file ./keygen --name keyfile
// (...)
openat(AT_FDCWD, "/home/disconnect3d/mobilecoin/target/debug/keyfile.json",
O_WRONLY|O_CREAT|O_TRUNC|O_CLOEXEC, 0666) = 3
openat(AT_FDCWD, "/home/disconnect3d/mobilecoin/target/debug/keyfile.pub",
O_WRONLY|O_CREAT|O_TRUNC|O_CLOEXEC, 0666) = 3
```

Figure 8.1: Using `strace` to see the permissions that the keyfiles are created with

Exploit Scenario
Alice generates her keys through the `keygen` tool. Eve, another user on the same system, reads the generated private keyfile due to its overly broad permissions and hijacks Alice's node or account.

Recommendations
Short term, change the `write_keyfile` function to create private keyfiles with `0700` permissions. This will prevent other users on the same system from accessing private keys generated with the `keygen` utility.

Long term, investigate whether the handling of users' private keys could be offloaded to a service such as Vault. Avoid handling sensitive information to prevent its disclosure.

9. Messages with incorrectly ordered values are not rejected

Severity: Informational
Type: Data Validation
Target: `mobilecoin/consensus/scp/src/slot.rs`

Difficulty: Medium
Finding ID: TOB-MOBBFT-009

Description
The `handle_messages` function (figure 9.1) does not reject messages with incorrectly ordered values. This may allow messages from malfunctioning nodes to be processed.

The severity of this issue is informational since MobileCoin is aware of this issue and there is a "TODO" in the code to address it.

```
/// Handle incoming messages from peers. Messages for other slots are ignored.
fn handle_messages(&mut self, msgs: &[Msg<V>]) -> Result<Option<Msg<V>>, String> {
    // (...)

    'msg_loop: for msg in msgs_for_slot {
        let is_higher = match self.M.get(&msg.sender_id) {
            Some(existing_msg) => msg.topic > existing_msg.topic,
            None => true,
        };

        if is_higher {
            // This message is higher than previous messages from the same sender.
            if msg.validate().is_ok() {
                // Reject messages with invalid values.
                // This Validation can be skipped during the Externalize phase
                // because this node no longer changes its ballot values.
                if self.phase != Phase::Externalize {
                    for value in msg.values() {
                        if self.is_valid(&value).is_err() {
                            // Ignore this msg because it contains an invalid value.
                            continue 'msg_loop;
                        }
                    }
                }

                // TODO: Reject messages with incorrectly ordered values.
```

Figure 9.1: mobilecoin/consensus/scp/src/slot.rs#L370

Recommendations
Short term, implement the rejection of incorrectly ordered values in the `handle_messages` function to ensure that messages from malfunctioning nodes are not processed needlessly.

Long term, ensure that all "TODO" items are monitored in an issue-tracking system and dealt with before deployment.

10. Some metrics counters are never updated

Severity: Informational | Difficulty: Medium
Type: Auditing and Logging | Finding ID: TOB-MOBBFT-010
Target: `mobilecoin/consensus/service/src/counters.rs`

Description
The metrics counters in figure 10.1 are never updated in the MobileCoin logic. These un-updated metrics counters may mislead users who want to see certain federated voting statistics associated with MobileCoin's SCP implementation.

```
    // Previous slot number of voted nominated values.
    pub static ref PREV_SLOT_NUM_VOTED_NOMINATED: IntGauge =
OP_COUNTERS.gauge("prev_slot_num_voted_nominated");

    // Previous slot number of accepted nominated valutes.
    pub static ref PREV_SLOT_NUM_ACCEPTED_NOMINATED: IntGauge =
OP_COUNTERS.gauge("prev_slot_num_accepted_nominated");

    // Previous slot number of confirmed nominated values.
    pub static ref PREV_SLOT_NUM_CONFIRMED_NOMINATED: IntGauge =
OP_COUNTERS.gauge("prev_slot_num_confirmed_nominated");

    // Previous slot nomination round.
    pub static ref PREV_SLOT_NOMINATION_ROUND: IntGauge =
OP_COUNTERS.gauge("prev_slot_nomination_round");

    // (...)

    // Number of consensus messages dropped due to referencing an invalid previous block id.
    pub static ref SCP_MESSAGES_DROPPED_DUE_TO_INVALID_PREV_BLOCK_ID: IntCounter =
OP_COUNTERS.counter("scp_messages_dropped_due_to_invalid_prev_block_id");

    // Number of times catchup is initiated
    pub static ref CATCHUP_INITIATED: IntCounter = OP_COUNTERS.counter("catchup_initiated");

    // Number of times attestation is initiated
    pub static ref ATTESTATION_INITIATED: IntCounter =
OP_COUNTERS.counter("attestation_initiated");
```

Figure 10.1: mobilecoin/consensus/service/src/counters.rs#L63-L88

Recommendations
Short term, update or remove the unused metrics counters in MobileCoin's consensus code. This will let users analyze the statistics of the protocol used by MobileCoin.

Long term, add tests for MobileCoin metrics counters so that the values are set correctly according to what happens within the SCP protocol. This will help ensure such counters do not go unused accidentally.

11. Potential denial of service due to excessive gRPC message-length limit

Severity: Medium
Difficulty: Medium
Type: Denial of Service
Finding ID: TOB-MOBBFT-011
Target: `mobilecoin/mobilecoind-json/src/bin/main.rs`

Description
The `mobilecoind-json` server sets a ~2 GB message-length limit for receiving and sending gRPC messages (figure 11.1). Setting such a big limit may allow an attacker to exhaust the server's memory by sending big requests to the `mobilecoind-json` server.

We also reported an issue about an incorrectly documented `max_receive_message_len` function to the library's upstream repository on grpc-rs#491.

```
    // Set up the gRPC connection to the mobilecoind client
    let env = Arc::new(grpcio::EnvBuilder::new().build());
    let ch = ChannelBuilder::new(env)
        .max_receive_message_len(std::i32::MAX)
        .max_send_message_len(std::i32::MAX)
        .connect_to_uri(&config.mobilecoind_uri, &logger);
```

Figure 11.1: mobilecoin/mobilecoind-json/src/bin/main.rs#L646-L647

Exploit Scenario
Alice sets up a MobileCoin node with a `mobilecoind-json` server. Eve sends big HTTP requests to Alice's server, exhausting its available memory and causing a denial of service.

Recommendations
Short term, implement a maximum gRPC message-length limit of a few megabytes to prevent denial-of-service attacks prompted by massive server requests.

Long term, carefully consider decisions to deviate from network-related defaults. These defaults are based on the experience of the community as a whole, so deviations must be absolutely necessary and weighed carefully. This will keep your systems configured to withstand common threats and to not attract undue attention.

12. Broadcasting and then handling resolved messages may fail

Severity: Informational | Difficulty: High
Type: Denial of Service | Finding ID: TOB-MOBBFT-012
Target: `mobilecoin/consensus/service/src/byzantine_ledger/worker.rs`

Description
The `process_consensus_msgs` function (figure 12.1) first broadcasts resolved messages and then processes them in the `handle_messages` function. This order of operations may fail, as messages that may be considered invalid could be broadcast to other nodes. A large load of such messages may result in the waste of computation resources or even cause denial-of-service scenarios.

```
    fn process_consensus_msgs(&mut self) {
        // (...)
        // Process compatible messages in batches.
        for chunk in compatible_msgs.chunks(CONSENSUS_MSG_BATCH_SIZE) {
            // (...)
            // Broadcast resolved messages.
            for (consensus_msg, from_responder_id) in &resolved {
                self.broadcaster
                    .lock()
                    .expect("mutex poisoned")
                    .broadcast_consensus_msg(consensus_msg.as_ref(), &from_responder_id);
            }

            let scp_msgs: Vec<Msg<_>> = resolved
                .into_iter()
                .map(|(consensus_msg, _)| consensus_msg.scp_msg().clone())
                .collect();

            match self.scp_node.handle_messages(scp_msgs) {
                Ok(outgoing_msgs) => {
                    for msg in outgoing_msgs {
                        let _ = self.issue_consensus_message(msg);
                    }
                }
                Err(err) => {
                    log::error!(self.logger, "Failed handling messages: {:?}", err);
                }
            }
```

Figure 12.1: mobilecoin/consensus/.../byzantine_ledger/worker.rs#L507-L528

Recommendations
Short term, consider changing the logic in the `process_consensus_msgs` function to first handle messages and then to broadcast only those that are successfully handled. This will prevent the system from processing messages that will fail on multiple nodes anyway.

Long term, as suggested in TOB-MOBBFT-006, incorporate malicious nodes into your tests. Do this even for end-to-end tests. This could help alert you to denial-of-service vectors associated with the order in which expensive operations are performed.

13. The node's handle_messages function always returns an “Ok” result in the current codebase

Severity: Undetermined
Type: Undefined Behavior
Target: `mobilecoin/consensus/service/src/byzantine_ledger/worker.rs`

Difficulty: Medium
Finding ID: TOB-MOBBFT-013

Description
The node's `handle_messages` function returns a `Result<Vec<Msg<V>>, String>` type. However, given the calls it makes, it always returns `Ok(outbound_msgs)`, which may be a bug or a code-quality issue.

The node's `handle_messages` function can return an error result only if the calls it makes through Rust's "?" operator return an error result. Those calls include the following:

- The `slot`'s `handle_messages` function (figure 13.2), which can return only `Ok(self.out_msg())` or `Ok(None)` in the current code
  - (However, it has a “TODO,” also described in TOB-MOBBFT-009, which may allow an error return to be added to that function.)
- The `externalize` function (figure 13.3), which always returns an `Ok(())` result

```
    /// Handle incoming message from the network.
    fn handle_messages(&mut self, msgs: Vec<Msg<V>>) -> Result<Vec<Msg<V>>, String> {
        // (...)
        // Handle messages for recent externalized slots. Messages for older slots are
ignored.
        for slot in self.externalized_slots.iter_mut() {
            if let Some(msgs) = slot_index_to_msgs.get(&slot.get_index()) {
                if let Some(response) = slot.handle_messages(msgs)? {
                    outbound_msgs.push(response);
                }
            }
        }

        // Handle messages for current slot.
        if let Some(msgs) = slot_index_to_msgs.get(&self.current_slot.get_index()) {
            if let Some(response) = self.current_slot.handle_messages(msgs)? {
                if let Topic::Externalize(ext_payload) = &response.topic {
                    self.externalize(&ext_payload)?;
                }
                outbound_msgs.push(response);
            }
        }

        Ok(outbound_msgs)
    }
```

Figure 13.1: The node's `handle_messages` function (mobilecoin/consensus/scp/src/node/node_impl.rs#L181-L240)

```
    /// Handle incoming messages from peers. Messages for other slots are ignored.
    fn handle_messages(&mut self, msgs: &[Msg<V>]) -> Result<Option<Msg<V>>, String> {
        // (...)

        'msg_loop: for msg in msgs_for_slot {
            let is_higher = match self.M.get(&msg.sender_id) {
                Some(existing_msg) => msg.topic > existing_msg.topic,
                None => true,
            };

            if is_higher {
                // This message is higher than previous messages from the same sender.
                if msg.validate().is_ok() {
                    // (...)
                    // TODO: Reject messages with incorrectly ordered values.
                    // (...)
                }
            }
        }

        if has_higher_messages {
            // (...)
            Ok(self.out_msg())
        } else {
            Ok(None)
        }
    }
```

Figure 13.2: The `slot`'s `handle_messages` function (mobilecoin/consensus/scp/src/slot.rs#L322-L389)

```
    // Record the values externalized by the current slot and advance the current slot.
    fn externalize(&mut self, payload: &ExternalizePayload<V>) -> Result<(), String> {
        // (...)

        // Log an error if any invalid values were externalized.
        // This is be redundant, but may be helpful during development.
        for value in &payload.C.X {
            if let Err(e) = (self.validity_fn)(value) {  // shall this return an Err?
                log::error!(
                    self.logger,
                    "Slot {} externalized invalid value: {:?}, {}",
                    slot_index,
                    value,
                    e
                );
            }
        }

        // (...)
        self.push_externalized_slot(externalized_slot);
        Ok(())
    }
```

Figure 13.3: The `externalize` function (mobilecoin/consensus/scp/src/node/node_impl.rs#L96-L128)

Recommendations

Short term, investigate the `node`'s `handle_messages` function and the functions it calls that return a `Result` type to determine whether they can return an error. If they cannot return an error, refactor the functions to return proper value types (e.g., `Vec<Msg<V>>` instead of `Result<Vec<Msg<V>>, String>` for the `node`'s `handle_messages` function). This will increase the readability and correctness of the code.

Long term, when implementing functions that may return a `Result` type, ensure that all return types are covered by unit tests. Here, if the `Result` return types are intended, add such tests for the `node`'s `handle_messages` function and the functions it calls that return a `Result` type.

14. Overly restrictive checks in Slot::check_prepare_phase_invariants

Severity: Medium
Type: Denial of Service
Target: `consensus/scp/src/slot.rs`

Difficulty: Undetermined
Finding ID: TOB-MOBBFT-014

Description
Fuzzing `Node::handle_messages` produced an assertion violation in `Slot::check_prepare_phase_invariants`, which could be used to carry out denial-of-service attacks.

This finding describes one of several crashes that could not be ruled out by adding invariant checks to the start of `Node::handle_messages`. The others are described in TOB-MOBBFT-004 and TOB-MOBBFT-005.

Certain inputs produced an assertion violation in `check_prepare_phase_invariants` (figure 14.1). The assertion is that the current phase is `NominatePrepare` or `Prepare`. The assertion fails when `do_prepare_phase` transitions to the `Commit` phase. Note that the assertion does not fail more often because `do_prepare_phase` does not call `check_prepare_phase_invariants` along all code paths (figure 14.2).

```
fn check_prepare_phase_invariants(&self) {
    assert!(
        self.phase == Phase::NominatePrepare || self.phase == Phase::Prepare,
        "self.phase: {:?}",
        self.phase
    );
```

Figure 14.1: consensus/scp/src/slot.rs#L653-L658

```
/// Prepare phase message handling.
fn do_prepare_phase(&mut self) {
    self.check_prepare_phase_invariants();
    ...
    let accepted_prepared = self.ballots_accepted_prepared();

    // Find the highest ballot accepted prepared.
    if let Some(new_P) = accepted_prepared.iter().max() {
        match &self.P {
            Some(current_P) => {
                // self.P should not decrease.
                if new_P >= current_P {
                    self.P = Some(new_P.clone());
                } else {
                    ...
                    return;
                }
            }
            ...
        }
        ...
    }
```

```
        ...
        if let Some(c) = c_opt {
            ...
            return;
        }
        ...
        // Check invariants.
        self.check_prepare_phase_invariants();
    }
```

Figure 14.2: consensus/scp/src/slot.rs#L679-L977

Also note that `check_commit_phase_invariants` and `do_commit_phase` have similar problems. Specifically, `check_commit_phase_invariants` checks that the current phase is `Commit` (figure 14.3), which will fail if `do_commit_phase` transitions to the `Externalize` phase. Furthermore, `do_commit_phase` does not call `check_commit_phase_invariants` along all code paths (figure 14.4).

```
    fn check_commit_phase_invariants(&self) {
        assert_eq!(self.phase, Phase::Commit);
```

Figure 14.3: consensus/scp/src/slot.rs#L983-L984

```
    // Commit phase message handling.
    fn do_commit_phase(&mut self) {
        self.check_commit_phase_invariants();
        ...
        if let Some((cn, hn)) = self.ballots_confirmed_committed() {
            ...
            return;
        }
        ...
        self.check_commit_phase_invariants();
    }
```

Figure 14.4: consensus/scp/src/slot.rs#L1012-L1090

Exploit Scenario
Eve sends a crafted transaction to a MobileCoin node, causing an assertion violation followed by a crash.

Recommendations
Short term, take the following actions:

- Eliminate the phase checks in `check_prepare_phase_invariants` and `check_commit_phase_invariants`.
- Ensure that `check_prepare_phase_invariants` is called along all code paths in `do_prepare_phase`.
- Ensure that `check_commit_phase_invariants` is called along all code paths in `do_commit_phase`.

This will eliminate assertion violations that could be used to carry out denial-of-service attacks.

Long term, incorporate fuzzing into your continuous integration process to reveal similar bugs in the future.

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 code features two structs called `MockLedger`.** One is declared in `mock_ledger.rs`, and the other is declared by use of the `automock` procedural macro in `ledger_trait.rs`. Since the latter `MockLedger` does not appear explicitly within the code, there is potential for confusion. Consider renaming the former one.

- **`LedgerDB::create` panics unnecessarily and hides the error that caused the panic:**

```
   pub fn create(path: PathBuf) -> Result<(), Error> {
        let env = Environment::new()
            .set_max_dbs(22)
            .set_map_size(MAX_LMDB_FILE_SIZE)
            .open(&path)
            .unwrap_or_else(|_| {
                panic!(
                    "Could not create environment for ledger_db. Check that path exists
{:?}",
                    path
                )
            });
```

  `LedgerDB::create`'s return type already allows for errors. Consider returning an error that wraps the error that caused `open` to fail.

- **Typo in `test_ballot_set_predicate_blocking_set` comment.** The comment refers to a "quorum" when it should instead refer to a "blocking set":

```
       // Look for quorum intersecting with ballot_1 and some ballot for which there is
no quorum
        let (node_ids, pred) = local_node_quorum_set.findBlockingSet(
```

- **The code features a struct and a protobuf message called `ConsensusMsg` that do not correspond to each other directly.** The protobuf message contains the serialized structure as one of its fields. Consider changing the name of one or the other to prevent developers from confusing the two.

- **The MLSAG challenge's computation code is duplicated in the sign and verify code.** Consider moving the calculation code to a helper function. The shared part can be seen below, highlighted in yellow.

`c[(i + 1) % ring_size] = {`	`recomputed_c[(i + 1) % ring_size] = {`

```
 let mut hasher = Blake2b::new();

hasher.update(&RING_MLSAG_CHALLENGE_DOMAIN_TAG);
 hasher.update(message);
 hasher.update(&key_image);
 hasher.update(L0.compress().as_bytes());
 hasher.update(R0.compress().as_bytes());
 hasher.update(L1.compress().as_bytes());
 Scalar::from_hash::<Blake2b>(hasher)
};
```

```
 let mut hasher = Blake2b::new();

hasher.update(&RING_MLSAG_CHALLENGE_DOMAIN_TAG);
 hasher.update(message);
 hasher.update(&self.key_image);
 hasher.update(L0.compress().as_bytes());
 hasher.update(R0.compress().as_bytes());
 hasher.update(L1.compress().as_bytes());
 Scalar::from_hash::<Blake2b>(hasher)
};
```

- **The `ByzantineLedgerWorker.broadcaster` lock is locked once to process all messages and once to process each message.** Consider unifying this behavior or document why it is done this way in the code.

```
self.broadcaster
    .lock()
    .map(|mut broadcast| {
        broadcast
            .broadcast_consensus_msg(...)
    })
    .map_err(|_e| "Mutex poisoned:...")?;
```

```
for (consensus_msg, from_responder_id) in &resolved {
    self.broadcaster
        .lock()
        .expect("mutex poisoned")
        .broadcast_consensus_msg(...);
}
```

- **The `Amount` type that holds picomobs does not check against the maximum number of picomobs ($2.5 * 10^{20}$).** Add a sanity check to determine whether the value passed to the `Amount::new` function does not exceed the maximum picomobs value, at least in debug builds. This will help catch calculation bugs if they occur and also prevent the creation of a transaction output (through `TxOut::new`) with an oversized amount.

- **`Slot` fields `base_round_interval` and `base_ballot_interval` are declared with public visibility, which is inconsistent with other `Slot` fields**:

```
/// This parameter sets the base interval for round timeout.
/// SCP suggests this should be one second.
pub base_round_interval: Duration,

/// This parameter sets the base interval for ballot timeout.
/// SCP suggests this should be one second.
pub base_ballot_interval: Duration,
```

  All other `Slot` fields have, at most, crate visibility. Moreover, `base_round_interval` and `base_ballot_interval` do not seem to be used outside of the crate. Consider restricting them to crate visibility at most.

- **`Msg` features a `to_display` method; consider implementing the `Display` trait instead**:

```
/// Provides a display string for the Msg.
pub fn to_display(&self) -> String {
```

For example, implementing the `Display` trait would eliminate the need to call `to_display` in the Byzantine worker:

```
        log::warn!(
            self.logger,
            "Msg refers to a different blockchain. Msg {}, previous block ID: {:?}",
            consensus_msg.scp_msg().to_display(),
            consensus_msg.prev_block_id(),
        );
```

- **The mock network tests' log messages provide little information on nodes' progress besides that of the first node.** This is largely due to how the tests are structured. An outer loop iterates over all nodes, while an inner loop waits for each node to externalize all values:

```
for node_id in node_ids.iter() {
     let mut last_log = Instant::now();
     loop {
         ...
         let num_externalized_values = simulation.get_ledger_size(&node_id);
         if num_externalized_values >= test_options.values_to_submit {
             ...
             break;
         }
         ...
     }
     ...
 }
```

Consider restructuring the tests so that the inner loop iterates over nodes and provides information on the progress of each.

- **The metamesh tests consense on only a single value:**

```
test_options.values_to_submit = 1;
```

As mentioned in TOB-MOBBFT-007, we saw some metamesh tests stall with larger choices of this parameter. Consider running the metamesh tests with larger values of this parameter regularly.

- **Fix the code documentation around CryptoNote-style onetime keys:**
    - Document the discrepancies between MobileCoin's implementation and the CryptoNote paper: that subaddresses were introduced and that the `tx_public_key` is calculated based on `D`, the public subaddress spend key, instead of `G`, the Ristretto base point.

- Add the subscript "`_i`" to "`D`" in the equation "`C_i = a * D`" so that it reads "`C_i = a * D_i`."