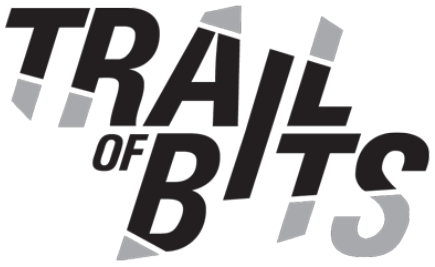

DFINITY Consensus

Security Assessment

February 15, 2022

Prepared For:
Robin Künzler | *DFINITY*
robin.kunzler@dfinity.org

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

Fredrik Dahlgren | *Trail of Bits*
fredrik.dahlgren@trailofbits.com

Changelog:

November 30, 2021:	Initial report draft
February 15, 2022:	Added Appendix D: Fix Log

Executive Summary

Project Dashboard

Code Maturity Evaluation

Engagement Goals

Coverage

Recommendations Summary
- Short term
- Long term

Findings Summary
- 1. Maliciously crafted catchup package shares could cause memory resource exhaustion
- 2. The consensus protocol uses vulnerable dependencies
- 3. Inconsistent handling of duplicate shares
- 4. Misbehaving nodes are not reported or punished by the consensus layer
- 5. Invalid notarizations cause the validator to skip block validation

A. Vulnerability Classifications

B. Code Maturity Classifications

C. Code Quality Recommendations

D. Fix Log
- Detailed Fix Log

Executive Summary

From November 11 to November 26, 2021, DFINITY engaged Trail of Bits to review the security of the DFINITY consensus mechanism. The review covered block validation, signature generation and verification, XNet, disaster recovery, update mechanisms, node resumption, and distributed key generation. Trail of Bits conducted this assessment over two person-weeks, with one engineer working from commit `b5b3728` of the `dfinity-lab/core/ic` repository.

This review resulted in five issues, two of which are of high severity. The remaining issues are of low, informational, and undetermined severity. One of the high-severity issues (TOB-DCA-001) involves missing input validation, which could lead to memory exhaustion. The issue could be leveraged by a malicious node operator to launch denial-of-service attacks against other nodes on the same subnet. The second high-severity issue (TOB-DCA-006) is related to block proposal validation and could be abused by a malicious node to stop other nodes from validating and notarizing select block proposals, allowing the malicious node to gain control over which new blocks are finalized by the subnet. Both issues have been addressed by the DFINITY team.

One issue (TOB-DCA-005) is of undetermined severity. This issue is related to missing mechanisms for reporting or punishing Byzantine nodes. It is unclear how this issue affects the consensus mechanism, but we are concerned that without clear economic disincentives to acting maliciously, more nodes may choose to become Byzantine. Missing reporting mechanisms may also make it more difficult to investigate incidents resulting from malicious behavior. Currently, this issue is mitigated by removing malicious nodes from the network following a Network Nervous System (NNS) proposal. The DFINITY community has also adopted a long-term proposal to strengthen the system against malicious behavior.

Overall, the DFINITY consensus protocol is well designed and easy to review. The protocol is based on a random beacon, which is used to rank potential block makers in each round. The random beacon is generated using Boneh–Lynn–Shacham (BLS) threshold signatures over BLS12-381, which is also used to notarize and finalize block proposals. Since the BLS scheme has unique signatures (which are verifiable by all participants in a threshold setting), Byzantine nodes have much less leeway to act maliciously; overall, the choice to use the BLS signature scheme sidesteps a number of issues common to other threshold signature schemes.

The protocol's implementation is based on a simple round-robin scheduling algorithm, in which a number of components take turns validating incoming artifacts, aggregating signature shares, proposing new blocks, notarizing validated blocks, and finalizing notarized blocks. This type of loosely coupled implementation makes it easier to ensure that the protocol's security properties are upheld by the implementation. The codebase has

good test coverage and implements tests in which a number of nodes can be initialized to act maliciously in a limited fashion.

Going forward, we recommend that the team extend the test harness to test for more types of malicious behavior, such as message spam (e.g., nodes spamming the network with duplicate or invalid messages). We also recommend that the team implement property tests (e.g., using a framework like `quickcheck`) to ensure that messages with randomized fields are handled correctly by the artifact validation mechanism.

To improve auditing and logging throughout the network, we recommend that the DFINITY team start building out an auditing system in which nodes can report potentially malicious behavior to the other nodes in the network. The DFINITY team could then leverage this system if it chooses to implement a slashing mechanism for Byzantine behavior.

Update: After the completion of the assessment, Trail of Bits reviewed the fixes implemented for the issues presented in this report. Detailed information on the results of the fix review can be found in Appendix D.

Project Dashboard

Application Summary

Name	DFINITY consensus
Version	Commit b5b3728
Type	Rust
Platform	Native/Linux

Engagement Summary

Dates	November 11–26, 2021
Method	Full knowledge
Consultants Engaged	1
Level of Effort	2 person-weeks

Vulnerability Summary

Total High-Severity Issues	2	■ ■
Total Medium-Severity Issues	0	
Total Low-Severity Issues	1	■
Total Informational-Severity Issues	1	■
Total Undetermined-Severity Issues	1	■
Total	5	

Category Breakdown

Auditing and Logging	1	■
Data Validation	3	■ ■ ■
Patching	1	■
Total	5	

Code Maturity Evaluation

Category Name	Description
Centralization	**Strong.** The protocol achieves the standard safety and liveness guarantees in the presence of at most *f* Byzantine nodes for networks with at least *3f + 1* nodes. The use of distinct subnets also allows the Internet Computer to scale in order to handle increases in incoming traffic and in the number of deployed canisters.
Upgradeability	**Strong.** The Internet Computer allows the replica (the guest OS and node client software) to be upgraded through a voting system implemented by the NNS. This provides a reliable upgrade mechanism and helps the network avoid forks as long as a majority of the nodes upgrade to the latest version.
Function Composition	**Strong.** Overall, the codebase is well structured and easy to navigate. The implementation consists of a small number of loosely coupled components with clear areas of responsibility, which makes the code easy to review and maintain. The implementation also uses properties of the Rust type system to provide correctness guarantees at compile time. This makes the security properties of the codebase easier to reason about, as the compiler rules out a large number of issues.
Auditing and Logging	**Moderate.** The implementation logs invalid messages locally, but there is no functionality to report nodes as malicious or to punish nodes for malicious behavior. (For more detail, see TOB-DCA-005.)
Specification	**Strong.** The DFINITY team provided ample documentation covering the high-level details of the consensus algorithm, as well as internal documentation covering the actual implementation. However, we recommend adding documentation to the codebase describing the security properties upheld by each function.
Testing and Verification	**Satisfactory.** The consensus implementation has an extensive test suite, which also tests how some types of malicious behavior could affect a running system. However, the test suite is missing tests for additional types of malicious behavior, such as message spam and messages sent with invalid (randomized) fields. This type of testing could have uncovered some of the issues in this report (such as TOB-DCA-001 and TOB-DCA-006).

Engagement Goals

The engagement was scoped to provide a security assessment of the DFINITY consensus protocol implementation.

Specifically, we sought to answer the following questions:

- Are BLS threshold and multisignature shares aggregated correctly?
- Could a node send multiple shares to compromise signature aggregation?
- Does the system reject signature shares from nodes that are not part of the relevant committee?
- Is the signed content properly validated against the current state of the node?
- Are distributed key generation (DKG) summaries and dealings properly validated?
- Could a node send multiple DKG dealings in a single round?
- Could a node send invalid messages that would remain in the unvalidated pool?
- Is it possible to cause other nodes to fail to validate valid block proposals?
- How does the system handle blocks from nodes that claim to run a different replica version?
- How do nodes handle malicious behavior?

Coverage

Consensus. First, we reviewed the provided documentation on the DFINITY consensus protocol. During this phase, we tried to identify any implicit assumptions made by the protocol, potential edge cases, and under-specified components of the protocol. Shifting our focus to the implementation, we ran a number of static analysis tools, such as Clippy, Dylint, and `cargo-audit`, to identify unidiomatic Rust code and common vulnerability patterns. This analysis resulted in one issue related to known vulnerabilities in a number of the implementation's dependencies (TOB-DCA-003). We then began a manual review of each of the following components of the consensus algorithm:

- Validator
- Share Aggregator
- Block Maker
- Notary
- Finalizer
- Catchup Package Maker
- Random Tape Maker
- Random Beacon Maker
- Purger
- DKG

Our review covered each component, but we focused mainly on message validation (to ensure that nodes do not accept invalid messages), signature share aggregation (to ensure that signature shares are correctly validated and aggregated), and signature verification (to ensure that the implementation does not accept invalid signatures).

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

❑ **Validate the height of all incoming artifacts, ensuring that it is within a given, predetermined interval from the last random beacon, finalized block, or notarized block, when any of these are available.** TOB-DCA-001

❑ **Update the project dependencies to their newest versions.** Monitor the referenced GitHub thread regarding the `chrono` crate segfault issue. TOB-DCA-003

❑ **Add comments to each function detailing the preconditions required by each function and any postconditions that are guaranteed to hold by each function.** TOB-DCA-004

❑ **Improve the validation of incoming messages and revise the validation mechanism so that it returns a different error type for messages that are malicious.** Ensure that the system broadcasts any malicious behavior across the network so that it becomes known to all nodes. TOB-DCA-005

❑ **Remove the `continue` statement highlighted in figure 6.1 to allow block validation to continue when an invalid notarization is found in the unvalidated pool.** TOB-DCA-006

Long term

❑ **Consider purging unvalidated messages older than a given bound.** TOB-DCA-001

❑ **Run `cargo-audit` as part of the CI/CD pipeline and ensure that the team is alerted to any vulnerable dependencies that are detected.** TOB-DCA-003

❑ **Refactor the message validation mechanism to ensure that duplicate messages do not enter the validated pool.** TOB-DCA-004

❑ **Consider implementing a slashing mechanism to disincentivize nodes from acting maliciously.** TOB-DCA-005

❑ **Review any control flow choices that are based on untrusted inputs from the network.** TOB-DCA-006

Findings Summary

#	Title	Type	Severity
1	Maliciously crafted catchup package shares could cause memory resource exhaustion	Data Validation	High
2	The consensus protocol uses vulnerable dependencies	Patching	Low
3	Inconsistent handling of duplicate shares	Data Validation	Informational
4	Misbehaving nodes are not reported or punished by the consensus layer	Auditing and Logging	Undetermined
5	Invalid notarizations cause the validator to skip block validation	Data Validation	High

1. Maliciously crafted catchup package shares could cause memory resource exhaustion

Severity: High
Type: Data Validation
Target: `dfinity-labs/rs/consensus`

Difficulty: Medium
Finding ID: TOB-DCA-001
Status: Fixed

Description
The purger deletes old artifacts from the consensus pool. By sending two malicious catchup package (CUP) shares, it is possible to cause the purger to ignore all artifacts in the unvalidated pool, exhausting the node's memory resources.

Suppose a malicious node creates two CUP shares, `S` and `S'`, with random beacons of height `0u64` and `u64::MAX` (`0xffff_ffff_ffff_ffffu64`), respectively. (The values of the other `CatchupContentT` fields are not important as long as all hashes are correct.) Note that `S.height() = 0` and `S'.height() = u64::MAX`, since the share height is the height of the corresponding random beacon.

The malicious node broadcasts the two CUP shares to all other nodes, which add the shares to the unvalidated pool. The following are the only ways for the two messages to be removed from the unvalidated pool:

1. They are moved to the validated pool by the validator.
2. They are marked as invalid by the validator.
3. They are purged by the purger.

When the validator runs, the `Validator::validate_catch_up_package_shares` method will ignore `S` since `S.height()` is less than the height of the last CUP; the method will call `Validator::validate_catch_up_share_content` with malicious share `S'`.

```
    fn validate_catch_up_package_shares(&self, pool_reader: &PoolReader<'_>) -> ChangeSet {
        let catch_up_height = pool_reader.get_catch_up_height();
        let max_height = match pool_reader
            .pool()
            .unvalidated()
            .catch_up_package_share()
            .max_height()
        {
            Some(height) => height,
            None => return ChangeSet::new(),
        };
        let range = HeightRange::new(catch_up_height.increment(), max_height);

        let shares = pool_reader
            .pool()
            .unvalidated()
            .catch_up_package_share()
```

```
            .get_by_height_range(range);

        shares
            .filter_map(|share| {
                debug_assert!(share.height() > catch_up_height);
                if !share.check_integrity() {
                    // ... <redacted>
                }
                match self.validate_catch_up_share_content(pool_reader, &share.content) {
                    Ok(block) => {
                        // ... <redacted>
                    }
                    Err(ValidationError::Permanent(err)) => {
                        // ... <redacted>
                    }
                    Err(ValidationError::Transient(err)) => {
                        // ... <redacted>
                        None
                    }
                }
            })
            .collect()
    }
```

Figure 1.1: S will not be validated since the height of S is less than the CUP height plus one. S' will be passed to `Validator::validate_catch_up_share_content`. (consensus/src/consensus/validator.rs)

The `Validator::validate_catch_up_share_content` method will attempt to obtain a finalized block for the malicious block's height, `S'.height()` = `u64::MAX`, and will return a `TransientError::FinalizedBlockNotFound` error when the block is not found. This will leave both S and S' in the unvalidated pool.

```
    fn validate_catch_up_share_content(
        &self,
        pool_reader: &PoolReader<'_>,
        share_content: &CatchUpShareContent,
    ) -> Result<Block, ValidatorError> {
        let height = share_content.height();
        let block = pool_reader
            .get_finalized_block(height)
            .ok_or(TransientError::FinalizedBlockNotFound(height))?;

        // ... <redacted>
    }
```

Figure 1.2: `Validator::validate_catch_up_share_content` will return a transient error when called with S'. (consensus/src/consensus/validator.rs)

When the purger runs and attempts to purge the unvalidated pool in the `Purger::purge_unvalidated_pool_by_expected_batch_height` method, the highlighted

condition in figure 1.3 will be true, which means that the function will return early with an empty change set.

```
    fn purge_unvalidated_pool_by_expected_batch_height(
        &self,
        pool_reader: &PoolReader<'_>,
        changeset: &mut ChangeSet,
    ) {
        let finalized_height = pool_reader.get_finalized_height();
        let expected_batch_height = self.message_routing.expected_batch_height();
        let mut prev_expected_batch_height = self.prev_expected_batch_height.borrow_mut();
        if *prev_expected_batch_height < expected_batch_height
            && expected_batch_height <= finalized_height.increment()
        {
            let catch_up_height = pool_reader.get_catch_up_height();
            let unvalidated_pool = pool_reader.pool().unvalidated();
            fn below_range_max(h: Height, range: &Option<HeightRange>) -> bool {
                range.as_ref().map(|r| h < r.max).unwrap_or(false)
            }
            fn above_range_min(h: Height, range: &Option<HeightRange>) -> bool {
                range.as_ref().map(|r| h > r.min).unwrap_or(false)
            }
            // ... <redacted>

            // Skip purging if we have unprocessed but needed CatchUpPackageShare
            let unvalidated_catch_up_share_range =
                unvalidated_pool.catch_up_package_share().height_range();
            if below_range_max(catch_up_height, &unvalidated_catch_up_share_range)
                && above_range_min(expected_batch_height, &unvalidated_catch_up_share_range)
            {
                return;
            }

            // ... <redacted>
        }
    }
```

Figure 1.3: The purger will exit early with an empty change set since the highlighted condition is true. (consensus/src/consensus/purger.rs)

Because the purger exits early with an empty change set, nodes will no longer be able to purge the unvalidated pool. Malicious nodes are free to continue sending messages of height `0`, which will remain forever in the unvalidated pool (since message validation considers only messages of heights greater than some lower bound). This will increase the memory pressure on all nodes, eventually leading to memory resource exhaustion, which could take down nodes in the subnet.

Exploit Scenario
A malicious node uses the vulnerability to launch a denial-of-service attack, affecting the consensus mechanism and stopping honest nodes from being able to propose new blocks.

Recommendations
Short term, validate the height of all incoming artifacts, ensuring that it is within a given, predetermined interval from the last random beacon, finalized block, or notarized block, when any of these are available.

Long term, consider purging unvalidated messages older than a given bound.

2. The consensus protocol uses vulnerable dependencies

Severity: Low
Type: Patching
Target: `dfinity-labs/rs/consensus`

Difficulty: High
Finding ID: TOB-DCA-003
Status: Fixed

Description
The consensus protocol's implementation depends on a number of crates with known vulnerabilities. By using `cargo-audit`, we identified the following vulnerable dependencies. (All of the identified crates except for `tiny_http` are transitive dependencies of the consensus protocol's implementation.)

Dependency	Version	ID	Description
`chrono`	0.4.19	RUSTSEC-2020-0159	Potential segfault in `localtime_r` invocations
`tiny_http`	0.7.0	RUSTSEC-2020-0031	HTTP Request smuggling through malformed `Transfer Encoding` headers
`wasmtime`	0.29.0	RUSTSEC-2021-0110	Multiple Vulnerabilities in `wasmtime`
`anymap`	0.12.1	RUSTSEC-2021-0065	`anymap` is unmaintained
`difference`	2.0.0	RUSTSEC-2020-0095	`difference` is unmaintained
`memmap`	0.7.0	RUSTSEC-2020-0077	`memmap` is unmaintained

Other than `chrono`, all the dependencies can simply be updated to their newest versions to fix the vulnerabilities. The `chrono` crate issue has not been mitigated and remains problematic. A specific sequence of calls must occur to trigger the vulnerability, which is discussed in this GitHub thread in the `chrono` repository.

Exploit Scenario
An attacker exploits a known vulnerability in a dependency of the consensus layer and performs a denial-of-service attack on the node.

Recommendations
Short term, update the project dependencies to their newest versions. Monitor the referenced GitHub thread regarding the `chrono` crate segfault issue.

Long term, run `cargo-audit` as part of the CI/CD pipeline and ensure that the team is alerted to any vulnerable dependencies that are detected.

3. Inconsistent handling of duplicate shares

Severity: Informational
Type: Data Validation
Target: `consensus/src/consensus/utils.rs`

Difficulty: N/A
Finding ID: TOB-DCA-004
Status: Fixed

Description

The `utils::aggregate` function aggregates threshold and multisignature shares into combined signatures. The function groups artifact shares into a `BTreeMap` to map message content into a list of signature shares. For each message content, the function checks that the node has received enough signature shares before attempting to aggregate the shares into a combined signature. However, the function does not take duplicate shares into account, which means that the signature shares could be forwarded to `crypto::aggregate` even if some shares are duplicates.

```
    pub fn aggregate<
        Message: Eq + Ord + Clone + std::fmt::Debug + HasHeight + HasCommittee,
        CryptoMessage,
        Signature,
        KeySelector: Copy,
        CommitteeSignature,
        Shares: Iterator<Item = Signed<Message, Signature>>,
    >(
        log: &ReplicaLogger,
        membership: &Membership,
        crypto: &dyn Aggregate<CryptoMessage, Signature, KeySelector, CommitteeSignature>,
        selector: Box<dyn Fn(&Message) -> Option<KeySelector> + '_>,
        artifact_shares: Shares,
    ) -> Vec<Signed<Message, CommitteeSignature>> {
        group_shares(artifact_shares)
            .into_iter()
            .filter_map(|(content_ref, shares)| {
                // ... <redacted>
                let threshold = match membership
                    .get_committee_threshold(content_ref.height(), Message::committee())
                {
                    Ok(threshold) => threshold,
                    Err(err) => {
                        error!(log, "MembershipError: {:?}", err);
                        return None;
                    }
                };
                if shares.len() < threshold {
                    return None;
                }
                let shares_ref = shares.iter().collect();
                crypto
                    .aggregate(shares_ref, selector)
```

```
                    .ok()
                    .map(|signature| {
                        let content = content_ref.clone();
                        Signed { content, signature }
                    })
            })
            .collect()
    }
```

Figure 3.1: The `utils::aggregate` function checks that the number of shares is higher than the threshold, but does not consider duplicate shares. (consensus/src/consensus/utils.rs)

For notarization shares and finalization shares, duplicate shares are filtered out by the `Validator::validate_notary_signed` and `Validator::dedup_change_actions` methods.

For random beacon shares and CUP shares, all valid shares are added to the validated pool, which means that duplicate shares may end up being passed to `utils::aggregate`. The same is true for random tape shares as long as there is no random tape for the corresponding height in the validated pool. This is not currently an issue, as duplicate threshold shares are caught in `crypto::combine_signatures` after the signer node IDs are mapped to the corresponding inputs used for Lagrange interpolation.

Recommendations
Short term, add comments to each function detailing the preconditions required by each function and any postconditions that are guaranteed to hold by each function.

Long term, refactor the message validation mechanism to ensure that duplicate messages do not enter the validated pool.

4. Misbehaving nodes are not reported or punished by the consensus layer

Severity: Undetermined
Type: Auditing and Logging
Target: `consensus/src/consensus`

Difficulty: High
Finding ID: TOB-DCA-005
Status: Risk accepted

Description

The consensus protocol implemented by DFINITY assumes that at most *(N - 1) / 3* nodes are Byzantine, where *N* is the size of the committee. (To be able to aggregate threshold signature shares for new CUPs, a committee of *(2N - 1) / 3* honest nodes is required.)

Typically, economic incentives are used to encourage participating nodes to follow the protocol and to punish Byzantine nodes. However, the current implementation of the protocol does not take any action when potentially malicious messages are detected. This is true even for messages that are certain to be malicious, like equivocations from the lowest ranked block maker. This gives Byzantine nodes more freedom to act maliciously without being detected or punished by the honest nodes in the network.

```
    fn process_changes(
        &self,
        time_source: &dyn TimeSource,
        artifacts: Vec<UnvalidatedArtifact<ConsensusMessage>>,
    ) -> (Vec<AdvertSendRequest<ConsensusArtifact>>, ProcessingResult) {
        // ... <redacted>
        for change_action in change_set.iter() {
            // ... <redacted>

             match change_action {
                // ... <redacted>
                ConsensusAction::HandleInvalid(artifact, s) => {
                    self.invalidated_artifacts.inc();
                    warn!(self.log, "Invalid artifact {} {:?}", s, artifact);
                }
            }
        }
        // ... <redacted>
        self.consensus_pool
            .write()
            .unwrap()
            .apply_changes(time_source, change_set);

        (adverts, changed)
    }
```

Figure 4.1: `ConsensusProcessor<PoolConsensus, PoolIngress>::process_changes` logs invalid artifacts. (artifact_manager/src/processors.rs)

When an invalid artifact is detected by the artifact validation mechanism, a `ChangeAction::HandleInvalid` action is generated. This is handled by the `ConsensusProcessor<PoolConsensus, PoolIngress>::process_changes` method, which simply logs invalid artifacts, and the `ConsensusPoolImpl::apply_changes` method, which removes any invalid artifacts from the unvalidated pool.

```
    fn apply_changes(&mut self, time_source: &dyn TimeSource, change_set: ChangeSet) {
        for change_action in change_set {
            match change_action {
                // ... <redacted>
                ChangeAction::HandleInvalid(to_remove, _) => {
                    unvalidated_ops.remove(to_remove.get_id());
                }
            }
        }
        // ... <redacted>
        self.apply_changes_unvalidated(unvalidated_ops);

        // ... <redacted>
    }
```

Figure 4.2: `ConsensusPoolImpl::apply_changes` removes invalid artifacts from the unvalidated pool. (artifact_pool/src/consensus_pool.rs)

Equivocating block proposals are not detected by the validation mechanism at all.

The worst case scenario for an attacker is that the network ignores her malicious messages; this is not a sufficient disincentive. Furthermore, since the current implementation does not report malicious behavior, honest nodes will never learn about such behavior directed against other nodes.

Exploit Scenario
A malicious node operator executes a denial-of-service attack against a number of nodes in the network. Some of the malicious messages are detected as malformed by honest nodes, but since there is no built-in mechanism to report or punish the attacker, other honest nodes are not alerted, and the malformed messages are simply dropped.

Recommendations
Short term, improve the validation of incoming messages and revise the validation mechanism so that it returns a different error type for messages that are malicious. Ensure that the system broadcasts any malicious behavior across the network so that it becomes known to all nodes.

Long term, consider implementing a slashing mechanism to disincentivize nodes from acting maliciously.

5. Invalid notarizations cause the validator to skip block validation

Severity: High
Type: Data Validation
Target: `consensus/src/consensus/validator.rs`

Difficulty: High
Finding ID: TOB-DCA-006
Status: Fixed

Description
When the `Validator::validate_blocks` function validates a block proposal, it first checks the unvalidated pool for a valid notarization for the block. If it finds a valid notarization, it moves both the block and the notarization to the validated pool. However, if the function finds an invalid notarization for the block, it removes the notarization and leaves the block proposal in the unvalidated pool.

```
    for proposal in pool_reader
        .pool()
        .unvalidated()
        .block_proposal()
        .get_by_height_range(range)
    {
        if proposal.check_integrity() {
            // Attempt to validate the proposal through a notarization
            if let Some(notarization) = pool_reader
                .pool()
                .unvalidated()
                .notarization()
                .get_by_height(proposal.height())
                .find(|notarization|
                    &notarization.content.block == proposal.content.get_hash()
            ) {
                // Verify notarization signature before checking block validity.
                let verification = self.verify_signature(pool_reader, &notarization);
                if let Err(ValidationError::Permanent(e)) = verification {
                    change_set.push(ChangeAction::HandleInvalid(
                        notarization.into_message(),
                        format!("{:?}", e),
                    ));
                    continue;
                } else if verification.is_ok() {
                    // ... <redacted>
                }
                // Note that transient errors on notarization signature
                // verification should cause fall through, and the block
                // proposals proceed to be checked normally.
            }
        // ... <redacted>
        }
    }
```

Figure 5.1: If an invalid notarization is found in the unvalidated pool, the corresponding block proposal is not validated. (consensus/src/consensus/validator.rs)

Exploit Scenario
The current block leader publishes a new block proposal P, which is sent to all nodes. A malicious node sees the proposal and immediately starts sending invalid notarizations on P to all nodes. This stops a number of nodes from validating the proposal, which means the lowest ranked block cannot be notarized. After some time passes, the nodes start accepting proposals from block makers with higher ranks. As a result, the malicious node controls which node gets to create the next block.

Recommendations
Short term, remove the `continue` statement highlighted in figure 5.1 to allow block validation to continue when an invalid notarization is found in the unvalidated pool.

Long term, review any control flow choices that are based on untrusted inputs from the network.

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
Testing	Related to test methodology or test coverage
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. Code Quality Recommendations

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

1. The inline documentation for `CatchupContentT` (line 27 in `types/types/src/consensus/catchup.rs`) claims that the catchup package (CUP) height is given by the distributed key generation (DKG) summary, but the implementation of `HasHeight` for `CatchupContentT` uses the random beacon to compute the height.
2. The return value from `Purger::purge_validated_pool_by_catch_up_package` (in `consensus/src/consensus/purger.rs`) is never used and could be removed.
3. The threshold signature aggregation and verification code uses panics to identify unexpected states during Boneh–Lynn–Shacham (BLS) signature aggregation. This could needlessly expose the node to denial-of-service attacks if the code is refactored and new code paths are introduced.
4. The `BlockMaker::get_stable_registry_version` method (in `consensus/src/consensus/block_maker.rs`) returns `None` if `self.registry_client.get_version_timestamp` returns `None` inside the loop. The function should most likely continue the loop instead of exiting early and returning `None`.

D. Fix Log

After the initial assessment, the DFINITY team provided Trail of Bits with a detailed fix review document and links to the corresponding Jira issues and GitLab merge requests. The Trail of Bits audit team reviewed each fix to ensure that the underlying issue was correctly addressed.

#	Title	Severity	Status
1	Maliciously crafted catchup package shares could cause memory resource exhaustion	High	Fixed
2	The consensus protocol uses vulnerable dependencies	Low	Fixed
3	Inconsistent handling of duplicate shares	Informational	Fixed
4	Misbehaving nodes are not reported or punished by the consensus layer	Undetermined	Risk accepted
5	Invalid notarizations cause the validator to skip block validation	High	Fixed

For additional information for each fix, please refer to the detailed fix log below.

Detailed Fix Log

Finding 1: Maliciously crafted catchup package shares could cause memory resource exhaustion
Fixed. Catchup package (CUP) shares are now fetched only if the package height is greater than the current CUP height and less than the current finalized height.

Finding 2: The consensus protocol uses vulnerable dependencies
Fixed. All vulnerable dependencies apart from `chrono` have been updated. Since there is no upgrade path for `chrono`, this vulnerable dependency remains. The corresponding risk is accepted by the DFINITY team.

Finding 3: Inconsistent handling of duplicate shares
Fixed. The signature share aggregation is now implemented using a `BTreeSet`, which ensures proper signature share de-duplication, rather than a `BTreeMap`.

Finding 4: Misbehaving nodes are not reported or punished by the consensus layer
Risk accepted. This issue affects the performance of the protocol as long as the number of malicious nodes is less than *N / 3* (where *N* is the size of the committee). Malicious behavior is currently logged locally, and when any performance degradation or misbehavior is observed, the offending node can be removed from the network by a Network Nervous System (NNS) proposal. In the long term, the team has opted to move forward and implement a number of mechanisms to further strengthen the protocol against malicious behavior. The high-level goals of this project are detailed in the recently accepted NNS proposal on malicious party security.

Finding 5: Invalid notarizations cause the validator to skip block validation
Fixed. The DFINITY team dropped the call to `continue` when an invalid notarization for the block is found in `Validator::validate_blocks`.