PARALLELCHAIN HOTSTUFF CONSENSUS SECURITY ANALYSIS

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Intro

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another party. Any subsequent publication of this report shall be without mandatory consent.

Name	ParallelChain HotStuff Consensus
Website	https://parallelchain.io/
Repository	https://github.com/parallelchain-io/hotstuff_rs
Commit	759b6b766876217633f61d6cd38dd57fa9df8b88
Platform	L1
Network	ParallelChain
Languages	Rust
Methodology	Blockchain Protocol and Security Analysis Methodology
Lead Auditor	s.akermoun@hacken.io
Auditor	n.lipartiia@hacken.io
Auditor	o.haponiuk@hacken.io
Approver	l.ciattaglia@hacken.io
Timeline	15.05.2023 - 15.06.2023
Changelog	29.05.2023 (Draft Report)
Changelog	16.06.2023 (Preliminary Report)
Changelog	27.06.2023 (Final Report)
Changelog	30.06.2023 (Final Report Updated)

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Table of contents

Summary

ParallelChain Lab is a tech company known for its layer-1 blockchain protocol, ParallelChain. This public and private blockchain infrastructure supports high-performance, enterprise-grade applications, providing a secure environment for traditional enterprises and the DeFi community. ParallelChain Mainnet, their latest offering, is a public smart contract platform powered by a proof-of-stake consensus mechanism, ParallelBFT.

The focus of this report is their implementation of the HotStuff consensus protocol. ParallelChain's version adheres to the original protocol's design, demonstrating their dedication to delivering robust and efficient blockchain solutions. This analysis assesses the implementation's documentation quality, code quality, architecture quality, and overall security.

Documentation quality

The project presents comprehensive crate-level documentation, which provides a solid understanding of the library's functionality and its usage. The documentation extensively covers the API and the different modules, making it a valuable resource for developers looking to integrate or work with the HotStuff library.

However, there is room for improvement at the function and data structure level. Many functions, structures and enumerations lack descriptive doc strings, which are necessary for automatically generating detailed API documentation. Enhancing the doc strings would significantly increase the clarity and completeness of the documentation, contributing to better understanding and usage of the library.

The total Documentation Quality score is **6** out of 10.

Code quality

From a compilation perspective, the HotStuff library demonstrates a strong degree of quality. The project compiles and runs without any warnings, indicating an absence of immediately apparent syntax issues or deprecated function usage.

The development team has acknowledged the concern with the error handling mechanism, specifically issue PCH-018. They have outlined a detailed plan to enhance the system's error handling and are committed to addressing this issue, highlighting their proactive approach to improving the project's stability and security.
The team's error handling plan is well-structured and considers various types of errors that could occur:

- For user-causable, synchronous, and recoverable errors, the team plans to return a Result, which will enable the caller to handle the error.
- For user-causable, background, and recoverable errors, the team plans to handle these internally.
- For user-causable and irrecoverable errors, the team plans to panic with a user-oriented message. This will allow the user to quickly identify and understand the error that has occurred.
- For violations of library invariants, the team plans to panic with a library developer-oriented message. This will provide invaluable information for the library developers for debugging and remediation.

In a separate endeavor, the team has also addressed the Rust best practice violations PCH-002 identified in the codebase through a linting process (cargo clippy). This rectification demonstrates the team's dedication to adhering to best coding practices and their determination to continually refine the quality of their code.

Moreover, the team has recognized the concern of unsafe arithmetic operations raised in PCH-014. They've taken a thoughtful and detailed approach to address this issue, acknowledging the potential for arithmetic overflows and underflows.
While the team asserts that many of the highlighted operations won't under/overflow due to established invariants or the large limit of u64, they also identify lines that warrant special guards to prevent potential overflows. These include potential duration overflows and total power overflows. To address this, they have added specifications in the documentation and introduced `checked_*` operations to prevent violations. These measures highlight the team's understanding and commitment to maintaining a high degree of security in their codebase, which is especially critical in the blockchain domain where minor oversights can lead to significant vulnerabilities.
However, it's important to note that ensuring safe arithmetic operations is becoming the default standard in the blockchain and smart

contract realm due to historical instances of vulnerabilities that can be triggered even indirectly. Therefore, while the measures undertaken by the team are commendable, continuous efforts toward implementing and maintaining secure code practices are essential.

Regarding the testing strategy, the team acknowledges the present deficiency in unit tests, even though the project initially reported a test coverage of 71.17%, predominantly from integration tests. During the fix stage, they added more integration tests, effectively raising the coverage to 77.41%. Nevertheless, they have committed to improving the testing suite, recognizing the necessity for a more comprehensive set of unit tests to ensure all aspects of the codebase are adequately tested.

However, it's worth mentioning that the current coverage shortfall does not overshadow the overall good quality of the codebase. Instead, it indicates areas for future improvements and the team's ongoing commitment to refining their project.

The total Code Quality score is **6** out of 10.

Architecture quality

The HotStuff library follows the architectural design outlined in the HotStuff white paper. This design has been widely recognized for its robustness and efficiency in handling consensus in distributed systems and blockchains. It lends the project strong architectural quality and makes it suitable for use in various blockchain applications.

The architecture quality score is **8** out of 10.

Security Level

Our analysis of the HotStuff implementation has revealed a multitude of security issues that warrant immediate attention. However, the development team's response to these concerns has been proactive and effective.

Firstly, critical issues PCH-006 and PCH-007 have been swiftly addressed. These problems pertained to Byzantine behavior in systems with less than 64-bit architecture during leader selection and potential node crashes due to an unsafe function within the logging system respectively. Both these issues have been effectively resolved.

The development team has acknowledged PCH-008 and PCH-009, which pointed out the insufficient validation of PublicKeyBytes and the risk of network-wide compromise due to malicious vote requests. As a countermeasure, the team has declared plans to enhance type safety in the Network trait. The intention is to require it to return a valid Ed25519 public key, thereby increasing security.

Critical issue PCH-016, due to a malfunction in the caching system incorrectly storing future-view messages under the current view, is an open door to a wide range of bugs and potential security vulnerabilities. This has been corrected by the team, enhancing the stability and security of the system.

Critical issue PCH-017, which pointed out system-wide panic due to a malicious vote, has been fixed.

PCH-019, another critical issue, has been acknowledged by the team. It involves the potential for Byzantine nodes to disrupt the blockchain's functionality. The team has outlined a robust plan to address this, which includes a "sync blacklist"

High severity issue PCH-011 uncovers memory exhaustion and DoS vulnerabilities due to unconstrained vector size in the Block structure. The team argues that potential memory exhaustion and DoS vulnerabilities due to unconstrained vector size in the Block structure is an application-level concerns. They posit that Networking and App implementers have the flexibility and responsibility to handle over-large or invalid blocks.

High severity issue PCH-015, highlighting potential node crashes due to unsafe arithmetic operations in time durations and validator powers calculations, has been addressed. The issue was tackled through two measures. First, documentation has been updated to include constraints, offering guidelines for avoiding such errors. Second, checked arithmetic operations have been introduced. These operations, if detecting a violation, will trigger a system response that provides a descriptive panic message according to error management policy to answer PCH-018 issue. These changes together are intended to mitigate the risk associated with this issue.

The low-severity issue PCH-003 has been acknowledged by the team. This concern pointed to vulnerable dependencies in a crate that should be monitored for future patches. The development team has expressed that they will apply a fix once the library is patched.

The development team acknowledged the low-severity issue PCH-013, arguing that the use of chain_id as zero in the genesis quorum certificate is intentional and not a security risk. They elaborate that altering this behavior would demand breaking changes and potentially disrupt existing deployments.

The security score is **9** out of 10.

Total score

Considering all metrics, the total score of the report is **8.3** out of 10.

Findings count and definitions

Severity	Findings	Severity Definition
Critical	7	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation by external or internal actors.
High	3	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation by external or internal actors.
Medium	0	Medium vulnerabilities are usually limited to state manipulations but cannot lead to asset loss. Major deviations from best practices are also in this category.
Low	2	Low vulnerabilities are related to outdated and unused code or minor Gas optimization. These issues won't have a significant impact on code execution but affect code quality.
Total	12	

Scope of the audit

Protocol Audit

Cryptography and Keys

- Cryptography Libraries
- Keys Generation
- Asymmetric (Signing and Verification)

Hotstuff Consensus

- HotStuff implementation review
- Attack scenarios analysis (liveness, finality, eclipse,...)

P2P & Networking

- Network implementation review
- Attack scenarios analysis (defaults, DoS, MiM, overflows, state machine)

Storage

- Storage implementation review
- Security aspects analysis (DoS, corruption, state implosion)

Implementation

Code Quality

- Static Code Analysis
- Tests coverage

Protocol Tests

Node Tests

- Environment Setup
- E2E sync tests
- Consensus tests

Fuzzing

- Consensus Fuzzing

Issues

Byzantine Nodes Can Manipulate View Skipping

The current implementation of the `recv` method in the project allows Byzantine nodes to manipulate other nodes and cause them to skip the current view and disrupt the blockchain's functionality.

ID	PCH-019
Scope	algorithm
Severity	CRITICAL
Vulnerability Type	Denial of Service
Status	Acknowledged

Description

The code snippet below, extracted from the `recv` method, raises concerns:

src/networking.rs:129:

```
if received_qc_from_future {
    return Err(ProgressMessageReceiveError::ReceivedQCFromFuture)
}
```

This code checks if a received message contains a quorum certificate (QC) for a future block. If it does, it implies that the replica has missed some blocks and needs to synchronize with other nodes. Consequently, the code triggers an error (`Err(ShouldSync)`) and exits the `execute_view` function. When this error is received in the `start_algorithm` function, the node initiates synchronization with other nodes:

src/algorithm.rs:92:

```
if let Err(ShouldSync) = view_result {
    sync(&mut block_tree, &mut sync_stub, &mut app, &mut pacemaker)
}
```

As a result, the replica stops processing the current view, attempts to synchronize, and then continues executing the `start_algorithm` function. However, it's important to note that when the next `cur_view` is determined, it is always greater than the previous view:

src/algorithm.rs:88:

```
cur_view = max(cur_view, max(block_tree.highest_view_entered(), block_tree.highest_qc().view)) + 1;
```

Therefore, after synchronization, the replica never returns to executing the view it was processing before.

The vulnerability arises from the fact that the `recv` method lacks checks to verify the correctness of the received quorum certificate. This means that any Byzantine node can send a message containing an incorrect QC with a higher view, manipulating the replicas to skip the current view and proceed to the next.

By repeatedly exploiting this flaw, a Byzantine node can prevent other nodes from producing blocks, ultimately rendering the blockchain non-functional.

Proof of Concept

Let's consider a scenario in which a Byzantine node attempts to disrupt the blockchain by causing other nodes to skip the execution of the current view:

1. The Byzantine node constructs a message, such as a `Proposal`, `Nudge`, or `NewView` message, intentionally including an incorrect quorum certificate. Since the Byzantine node cannot produce a valid QC without the cooperation of other validators, it fabricates a QC with a higher view than the current view. The Byzantine node then broadcast the malicious message to all others replica in the network. Here's an example of such a message:

```
ProgressMessage::NewView {
    chain_id: 0, // Ensure correct chain_id
    view: 12345,
    highest_qc: QuorumCertificate {
        chain_id: 0,
        view: 123456, // Higher than the current view
        ...
    }
}
```

2. Upon receiving the message, it checks the `chain_id` and `view` values in the received QC. However, there are no additional checks to ensure the correctness of the QC. The code snippet below demonstrates the current implementation in the `recv` method:

   src/networking.rs:122:

```
// Inform the caller that we've received a QC from the future.
let received_qc_from_future = match &msg {
    ProgressMessage::Proposal(Proposal { block, .. }) => block.justify.view > cur_view,
    ProgressMessage::Nudge(Nudge { justify, .. }) => justify.view > cur_view,
    ProgressMessage::NewView(NewView { highest_qc, .. }) => highest_qc.view > cur_view,
    _ => false,
};
if received_qc_from_future {
    return Err(ProgressMessageReceiveError::ReceivedQCFromFuture)
}
```

   Since the view in the QC is higher than the current view and correctness isn't checked, the `recv` method returns the error `Err(ProgressMessageReceiveError::ReceivedQCFromFuture)`.

3. The resulting error is passed to the `execute_view` function. Upon receiving this error, the `execute_view` function halts the processing of incoming messages and returns `Err(ShouldSync)` to the higher-level `start_algorithm` function.

4. In the `start_algorithm` function, upon receiving the `ShouldSync` error, the replica initiates synchronization with other nodes to catch up to the latest view.

5. Once the synchronization is complete, a new iteration of the loop in the `start_algorithm` function begins, calculating a new `cur_view` value, which is always higher than the previous view. The `execute_view` function is called again, but this time with the new, incremented `cur_view`.

6. The Byzantine node continues sending incorrect messages, persistently causing steps 1-5 to repeat for subsequent views. Each time, the replica terminates the processing of the current view, performs synchronization, and proceeds to the next view. This repeated disruption corrupts the blockchain, preventing the production of new blocks and rendering the system non-functional.

Recommendation

To address this vulnerability, it is crucial to modify the `recv` method to include checks that validate the correctness of the received quorum certificate before returning the `ShouldSync` error. By ensuring the integrity of the received QC, replicas can make informed decisions about whether to synchronize or continue executing the current view, thereby mitigating the risk of view skipping manipulation by Byzantine nodes.

Malformed Origin of a Vote Request Remotely Crashes Nodes

An attacker can remotely crash network nodes by sending a request with a malformed origin. The resulting crash can lead to a complete stoppage of the network or even a full takeover by the attacker. This could allow a variety of malicious actions, such as majority attacks, double spending, and more.

ID	PCH-009
Scope	Consensus / Cryptography
Severity	CRITICAL
Vulnerability Type	Error handling / Data Validation
Status	Acknowledged

Description

HotStuff library handles network requests by processing them as tuples composed of the request's origin and a message. The origin is defined as a `PublicKeyBytes`, a byte array, while the message is an enum variant conforming to the HotStuff protocol.

Upon receiving a `Vote` message, the `on_receive_vote` function gets called, where `Vote`'s `is_correct` method verifies the vote by trying to convert the `PublicKeyBytes` into a `PublicKey`. However, this operation can potentially cause a node to crash if it fails to convert malformed `PublicKeyBytes`: *src/messages.rs:102*

```
impl Vote {
    /// # Panics
    /// pk must be a valid public key.
    pub fn is_correct(&self, pk: &PublicKeyBytes) -> bool {
        if let Ok(signature) = Signature::from_bytes(&self.signature) {
            PublicKey::from_bytes(pk).unwrap().verify(&(self.chain_id, self.view, self.block, self.phase).try_to_vec().
        } else {
            false
        }
    }
}
```

In this situation, the `pk` parameter is a public key that is received from a remote peer during network communication. The `pk` is in the form of a byte array (`PublicKeyBytes`). The peer, who could be a potential attacker, has control over this parameter.

The `is_correct` method in the `Vote` structure uses the `PublicKey::from_bytes(pk)` function to convert this byte array into a `PublicKey`. However, the issue arises when the byte array does not represent a valid public key according to the cryptographic standards of the library.

For instance, it might not correspond to a valid point on the edwards curve (which is a requirement of the ed25519 public key format), or it might not adhere to other restrictions that valid public keys must meet. If such a malformed byte array is provided as `pk`, the `PublicKey::from_bytes(pk)` function will fail and return an error.

The problem is that the `is_correct` method uses `unwrap()` on the result of `PublicKey::from_bytes(pk)`. The `unwrap()` function in Rust will cause the program to panic and crash if it's called on an error.

So, if an attacker provides a malformed `PublicKeyBytes`, by exploiting the vulnerability found in PCH-008, that cannot be converted into a `PublicKey`, this will cause `PublicKey::from_bytes(pk).unwrap()` to trigger a crash. This is what allows an attacker to remotely crash nodes in the network by simply sending them vote requests with malformed public keys.

This flaw poses a severe threat as it allows an attacker to remotely crash all nodes in the network, leading to a halt in the blockchain. Furthermore, this vulnerability may enable malicious actions such as double-spending or taking control of the blockchain.

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Proof of Concept

Below is a simplified version of the `is_correct` method to demonstrate the crash:

```
/*
This Proof of Concept (PoC) demonstrates a security vulnerability in the HotStuff blockchain library.
The library has a function, `is_correct`, which attempts to convert a `PublicKeyBytes` into a `PublicKey`.
This process is not safe because it doesn't handle conversion errors correctly.
In this PoC, we're demonstrating how a node can be crashed remotely.
The vulnerability occurs when a malformed `PublicKeyBytes` is used as an input for the `is_correct` function.
If the `PublicKeyBytes` are malformed (invalid), the `PublicKey::from_bytes(pk)` call fails and triggers a crash.
An attacker can exploit this vulnerability to crash any node, disrupt the network, or potentially take full control.
*/
use ed25519_dalek::PublicKey;
// Define the Vote structure
pub struct Vote {
    signature: Vec<u8>,
}
// Alias for a public key byte array
pub type PublicKeyBytes = [u8; 32];
impl Vote {
    // Define the is_correct function which checks if a public key is valid
    fn is_correct(&self, pk: &PublicKeyBytes) -> bool {
        // Attempt to convert the PublicKeyBytes into a PublicKey
        match PublicKey::from_bytes(pk) {
            Ok(public_key) => {
                // If successful, print a message and return true
                println!("Valid public key: {:?}", public_key);
                true
            }
            Err(_) => {
                // If conversion fails, print a message and trigger a crash
                println!("Invalid public key, causing a crash...");
                panic!("Invalid PublicKeyBytes: {:?}", pk);
            }
        }
    }
}
// Main function to run the proof of concept
fn main() {
    // Create a new Vote instance
    let vote = Vote { signature: vec![] };
    // Define an invalid public key as an array of zeros
    let invalid_pk = [7u8; 32];
    // Pass the invalid public key to the is_correct function, causing a crash
    vote.is_correct(&invalid_pk);
}
```

The output when executing this proof of concept on the command line will be:

```
% cargo run
   Compiling poc v0.1.0 (/Users/hacken/CodeProjects/poc)
    Finished dev [unoptimized + debuginfo] target(s) in 0.22s
     Running `target/debug/poc`
Invalid public key, causing a crash...
thread 'main' panicked at 'Invalid PublicKeyBytes: [7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Recommendation

To mitigate this vulnerability, nodes should be modified to handle malformed `PublicKeyBytes` gracefully. This can be achieved by fixing the issue PCH-008 (preferred approach), or by adjusting the `is_correct` method in the `Vote` structure:

```
pub fn is_correct(&self, pk: &PublicKeyBytes) -> bool {
    if let Ok(signature) = Signature::from_bytes(&self.signature) {
        match PublicKey::from_bytes(pk) {
            Ok(public_key) => public_key
```

```
                .verify(
                    &(self.chain_id, self.view, self.block, self.phase)
                        .try_to_vec()
                        .unwrap(),
                    &signature,
                )
                .is_ok(),
            Err(_) => false,
        }
    } else {
        false
    }
}
```

`succinct` function is vulnerable to Index Out of Bounds

The `succinct` function in the logging module lacks size checking for its array parameter, which can lead to index out of bounds errors and potential crashes.

ID	PCH-007
Scope	logging
Severity	CRITICAL
Vulnerability Type	Index Ouf of Bounds
Status	Fixed

Description

The succinct function is a crucial part of the logging module in the reviewed software. It is designed to provide a brief, readable representation of a byte sequence by base64 encoding the input and taking the first 7 characters.

src/logging.rs:114:

```
// Get a more readable representation of a bytesequence by base64-encoding it and taking the first 7 characters.
pub(crate) fn succinct(bytes: &[u8]) -> String {
    let encoded = STANDARD_NO_PAD.encode(bytes);
    let mut truncated = encoded[0..7].to_string();
    truncated.push_str("..");
    truncated
}
```

However, the function does not perform a check to ensure the base64 encoded string is long enough before attempting to slice the first 7 characters. This can lead to an index out of bounds exception if the base64-encoded output is shorter than 7 characters, thereby potentially causing the software to crash.

While this function may appear innocent, its misuse could have critical consequences. Given its extensive use throughout the logging operations, this issue's exploitation can disrupt the normal operation of the software and possibly even lead to unexpected crashes and potentially compromising the system's integrity, availability, and consensus mechanism.

Proof of Concept

The problem arises when the byte array passed to the succinct function results in a Base64 string of less than 7 characters. For example:

```
use base64::{engine::general_purpose::STANDARD_NO_PAD, Engine as _};
fn main() {
    succinct("fail".as_bytes());
}
```

```
fn succinct(bytes: &[u8]) -> String {
    let encoded = STANDARD_NO_PAD.encode(bytes);
    let mut truncated = encoded[0..7].to_string();
    truncated.push_str("..");
    truncated
}
```

The output when executing this proof of concept on the command line will be:

```
% cargo run
   Compiling poc v0.1.0 (/Users/hacken/CodeProjects/poc)
    Finished dev [unoptimized + debuginfo] target(s) in 0.24s
     Running `target/debug/poc`
thread 'main' panicked at 'byte index 7 is out of bounds of `ZmFpbA`', src/main.rs:9:25
```

Recommendation

A potential solution is to add a length check on the Base64-encoded string before attempting to slice it. If the string is shorter than 7 characters, the function could return the entire string or it could be padded with appropriate characters to reach a length of 7. This would prevent the index out of bounds exception and handle all byte arrays without causing a system crash.

Here is a suggested fix:

```
pub(crate) fn succinct(bytes: &[u8]) -> String {
    let encoded = STANDARD_NO_PAD.encode(bytes);
    let truncated = if encoded.len() > 7 {
        let mut truncated_string = encoded[0..7].to_string();
        truncated_string.push_str("..");
        truncated_string
    } else {
        encoded.to_string()
    };
    truncated
}
```

Byzantine Behavior Due to Unsafe `u64` to `usize` Conversion in Round-Robin Leader Selection on 32-bit Systems

In the `pacemaker` module, the `DefaultPacemaker` implementation's `view_leader` method performs an unsafe conversion from `u64` to `usize`, resulting in different outcomes for leader selection on 32-bit and 64-bit systems.

ID	PCH-006
Scope	pacemaker
Severity	CRITICAL
Vulnerability Type	Incorrect Type Conversion or Cast
Status	Fixed

Description

The `DefaultPacemaker`'s `view_leader` method contains an unsafe conversion from `ViewNumber`, which is an alias for `u64`, to `usize`. The `usize` data type's size is architecture-dependent, potentially leading to divergent behaviors for leader selection on 32-bit and 64-bit systems.

The problematic code snippet is as follows:

src/pacemaker.rs:51:

```
fn view_leader(&mut self, cur_view: ViewNumber, validator_set: &ValidatorSet) -> PublicKeyBytes {
    let num_validators = validator_set.len();
    *validator_set.validators().skip(cur_view as usize % num_validators).next().unwrap()
}
```

The issue arises from the conversion `cur_view as usize`, where `cur_view` is of type `ViewNumber` (an alias for `u64`).
The size of the `usize` type varies depending on the architecture, behaving as `u32` or `u64`. Thus, `cur_view`'s value when cast to `usize` can differ based on the architecture.

Below is an example of a potentially unsafe conversion from `u64` to `u32`, when the value exceeds the maximum limit of `u32`:

```
// Define ViewNumber as an alias for u64
pub type ViewNumber = u64;
fn main() {
    // Assign the value of MAX u32 + 1 to a variable of type ViewNumber (alias of u64)
    let n: ViewNumber = 4294967296; // MAX of u32 + 1
    // Here, we perform a potentially unsafe conversion from u64 to u32
    // Note that a u32 variable has the same maximum size as usize on a 32-bit system
    let converted = n as u32;
    // We then use an assertion to verify the result of the conversion
    // As 4294967296 exceeds the maximum value of u32 (4294967295), the converted value
    // wraps around to 0
    // Thus, the assertion passes, indicating that the conversion resulted in a
    // value different from the original
    assert_eq!(converted, 0);
}
```

The `view_leader` function plays a critical role in choosing the leader, substantially impacting the execution of a view. This discrepancy could cause ambiguity about a node's status: whether it's a leader meant to broadcast a proposal or a nudge, or a voter designated to receive and process messages.

Specifically, issues arise when the number of views surpasses `u32::MAX` (4294967295). Beyond this point, nodes on 32-bit architectures can't correctly determine leaders in a round-robin manner and start to show Byzantine behavior.

This vulnerability affects nodes running on 32-bit architecture machines using the `DefaultPacemaker` implementation of `Pacemaker`. Such nodes could behave erratically when the view number exceeds `u32::MAX` (4294967295). As the network likely contains a mix of 32-bit and 64-bit architecture nodes, severe single-stage desynchronization can occur, resulting in two groups with differing behaviors. This can become particularly dangerous if the "32-bit group" reaches the security threshold of one third of the network, potentially corrupting the entire network.

Proof of Concept

The following simplified version of the `view_leader` function demonstrates this issue:

```
// This example simulates running on a 32-bit architecture,
// where usize maximum value is u32::MAX.
// We use `Usize` as a type alias to mimic a 32-bit usize
// on a 64-bit system for this demonstration.
type Usize = u32;
// `view_leader` method determines the leader in a round-robin fashion
fn view_leader(cur_view: u64, validator_set: &[u8]) -> u8 {
    let num_validators = validator_set.len();
    // The current view number is converted to `Usize` (simulating a 32-bit system)
    // and used to select the leader.
    *validator_set
        .iter()
        .skip((cur_view as Usize) as usize % num_validators)
        .next()
        .unwrap()
}
fn main() {
```

```
    // This validator set is a list of mock validators for demonstration.
    let validator_set = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
    // When `cur_view` is less or equal to `u32::MAX` (4294967295), `view_leader` acts as expected.
    // Here, we start the cycle with `cur_view` being 4294967292 and iterate until `u32::MAX`.
    for cur_view in (u32::MAX as u64 - 3)..=u32::MAX as u64 {
        assert_eq!(view_leader(cur_view, &validator_set), (cur_view % 10) as u8);
    }
    // When `cur_view` is bigger than `u32::MAX` (4294967295), `view_leader` behaves differently due to
    // truncation in the type conversion.
    // We start the cycle with `cur_view` being 4294967292 and iterate until `u32::MAX + 10`.
    // Here, the assertion will fail, demonstrating the potential issues that can arise from this type
    // conversion on a 32-bit system.
    for cur_view in (u32::MAX as u64 - 3)..=(u32::MAX as u64 + 10) {
        assert_eq!(view_leader(cur_view, &validator_set), (cur_view % 10) as u8);
    }
}
```

The output when executing this proof of concept on the command line will be:

```
% cargo run
   Compiling poc v0.1.0 (/Users/hacken/CodeProjects/poc)
    Finished dev [unoptimized + debuginfo] target(s) in 0.77s
     Running `target/debug/poc`
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `0`,
 right: `6`',
```

The leader selection starts to shift once the number of views exceeds `u32::MAX` and restarts with the first validator as a leader.

On a 64-bit architecture, the `view_leader` function operates in the expected round-robin manner when the maximum value of `usize` is `u64::MAX`:

```
type Usize = u64;
fn view_leader(cur_view: u64, validator_set: &[u8]) -> u8 {
    /* ... */
}
fn main() {
    // Validator set
    let validator_set = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
    // The corresponding results of view_leader are 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5
    for cur_view in u32::MAX as u64 - 3..u32::MAX as u64 + 10 {
        assert_eq!(view_leader(cur_view, &validator_set), (cur_view % 10) as u8);
    }
}
```

Recommendation

This issue can be approached with three different remediation plans:

Downsizing `cur_view` to `u32`

One of the most straightforward fixes is to consider whether `cur_view` can be represented as a `u32` instead of a `u64`. This will effectively eliminate the unsafe conversion from `u64` to `usize` on 32-bit systems. However, this approach may limit the maximum view number to `u32::MAX` (4294967295) which could be a limitation based on the design and use case of your system.

The adjustment would look like this: *src/types.rs:33*:

```
// Change the type alias for ViewNumber to u32
pub type ViewNumber = u32;
```

Handling conversion errors

Instead of forcing a direct conversion from `u64` to `usize`, we could attempt to convert `cur_view` to `usize` and gracefully handle any potential conversion errors. This approach would provide more robust cross-platform compatibility, at the expense of increased code complexity.

The implementation would look like this:

```
fn view_leader(&mut self, cur_view: ViewNumber, validator_set: &ValidatorSet) -> Result<PublicKeyBytes, ConversionError>
    let num_validators = validator_set.len();
    let cur_view_usize = match usize::try_from(cur_view) {
        Ok(value) => value,
        Err(_) => return Err(ConversionError),
    };
    Ok(*validator_set.validators().skip(cur_view_usize % num_validators).next().unwrap())
}
```

This code returns a `Result` object. If the conversion is successful, it proceeds as usual, otherwise it returns a `ConversionError`. You'll need to handle this `Result` wherever `view_leader` is called.

Compiling only to 64-bit systems

This approach ensures that the project compiles and runs exclusively on 64-bit systems. This can be achieved by taking two steps:

1. **Update the *Cargo.toml* file**: This step involves adding metadata to the *Cargo.toml* file, stating explicitly that your package is intended for 64-bit architectures. This can be done by adding the `target-arch` field to the `package.metadata` section:

```
[package]
name = "hotstuff_rs"
version = "0.2.0"
description = "An implementation of the HotStuff consensus algorithm intended for production systems."
homepage = "https://parallelchain.io"
repository = "https://github.com/parallelchain-io/hotstuff_rs"
readme = "README.md"
edition = "2021"
license = "Apache-2.0"
keywords = ["consensus", "hotstuff", "blockchain"]
categories = ["cryptography::cryptocurrencies", "concurrency"]
[package.metadata]
target-arch = ["x86_64"]
[dependencies]
base64 = "0.21"
borsh = "0.10"
ed25519-dalek = "1"
fern = "0.6"
log = "0.4"
rand = "0.7"
sha2 = "0.10"
```

The `target-arch` metadata field informs users that your project is specifically intended for 64-bit architectures.

2. **Add a compiler directive in your Rust source file *src/lib.rs***: This step actively prevents the project from being compiled on non-64-bit architectures. It can be achieved by adding the `compile_error!` directive to the root of your library crate *src/lib.rs*:

```
#[cfg(not(target_pointer_width = "64"))]
compile_error!("Compilation is only allowed for 64-bit targets");
```

By taking these measures, you can ensure that your package metadata clearly states its target architecture, and the Rust compiler actively prevents compilation on non-64-bit systems. Thus, it enforces consistent behavior across different platforms and circumvents the potential issues associated with unsafe conversions between `u64` and `usize`.

In addition to the remediation strategies detailed above, it's recommended to proactively monitor for potential truncation issues by leveraging the Rust Clippy linter. Specifically, enabling the `cast_possible_truncation` lint will help identify such issues early in the development cycle. This can be achieved by executing the following command:

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

```
cargo clippy -- -W clippy::cast_possible_truncation
```

Incorporating this lint into your regular development, code review practices, and integrating it into your CI/CD pipeline will provide early detection and mitigation of unsafe truncation situations. This strategic approach can enhance the reliability and robustness of your code, and maintain a high standard of quality in your development process.

Incorrect Caching of Messages for Future Views

An identified issue in the message handling system involves the `recv` method of the `ProgressMessageStub` struct within the `networking` module, incorrectly caching messages intended for future views under the current view. This misplacement can cause severe disruptions in the overall system behavior.

ID	PCH-016
Scope	Cache system / Networking module
Severity	CRITICAL
Vulnerability Type	Logical Flaw
Status	Fixed

Description

In the context of HotStuff's blockchain protocol, the `recv` function is responsible for correctly handling and storing messages received from the network. However, the current implementation incorrectly caches messages intended for future views under the current view, causing message displacement and potential loss.

The following code excerpt is where the issue occurs:

src/networking.rs:133:

```
// Cache the message if its for a future view.
else if msg.view() > cur_view {
    let msg_queue = if let Some(msg_queue) = self.msg_buffer.get_mut(&cur_view) {
        msg_queue
    } else {
        self.msg_buffer.insert(cur_view, VecDeque::new());
        self.msg_buffer.get_mut(&cur_view).unwrap()
    };
    msg_queue.push_back((sender, msg));
}
```

The above snippet should cache messages intended for future views (where `msg.view() > cur_view`). However, instead of storing these messages with the key corresponding to their future view (`msg.view()`), they are mistakenly cached under the key corresponding to the current view (`cur_view`). This error disrupts the intended operation of the caching system, potentially leading to message loss and other unexpected behavior. This flaw disrupts the caching mechanism, resulting in lost messages that fail to reach their intended future views for processing.

This issue becomes even more critical as the `recv` function is invoked repeatedly within the `execute_view` loop, which can lead to a situation where an incorrectly cached message is retrieved and processed as if it belonged to the current view.

Proof of Concept

```
/*
This Proof of Concept (PoC) demonstrates a logic flaw in the ProgressMessageStub
struct's recv method of HotStuff's implementation, which results in incorrect
caching of messages intended for future views under the current view.
```

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

```
We simulate this behavior by generating a network message for a future view and
then processing this message via the recv method.
Assertions are used to verify the incorrect behavior of the caching system and
comments are included for clear understanding.
*/
use std::collections::{BTreeMap, VecDeque};
// Definition of types used in the structs and enums
pub type PublicKeyBytes = [u8; 32];
pub type ViewNumber = u64;
// ProgressMessage enum that holds different types of messages.
// Here, we only define one for simplicity.
#[derive(Debug)]
pub enum ProgressMessage {
    Message(Message),
}
// Implementation of ProgressMessage to retrieve the view of the message
impl ProgressMessage {
    pub fn view(&self) -> ViewNumber {
        match self {
            ProgressMessage::Message(msg) => msg.view,
        }
    }
}
// Message struct that represents a single message with a view number
#[derive(Debug)]
pub struct Message {
    pub view: ViewNumber,
}
// ProgressMessageStub struct that contains the message buffer used
// for caching messages
pub struct ProgressMessageStub {
    msg_buffer: BTreeMap<ViewNumber, VecDeque<(PublicKeyBytes, ProgressMessage)>>,
}
impl ProgressMessageStub {
    // Constructor for ProgressMessageStub
    pub(crate) fn new() -> ProgressMessageStub {
        Self {
            msg_buffer: BTreeMap::new(),
        }
    }
    // recv method which simulates receiving a message for a certain view
    pub(crate) fn recv(&mut self, cur_view: ViewNumber) {
        // We simulate the generation of a future view message
        match generate_future_view_message() {
            Ok((sender, msg)) => {
                // If the message is for a future view, it should be stored
                // in the buffer for the future view
                if msg.view() > cur_view {
                    // However, due to the bug, it's stored in the current view's buffer
                    let msg_queue = if let Some(msg_queue) = self.msg_buffer.get_mut(&cur_view) {
                        msg_queue
                    } else {
                        // If the current view does not have a message queue yet, create a new one
                        self.msg_buffer.insert(cur_view, VecDeque::new());
                        self.msg_buffer.get_mut(&cur_view).unwrap()
                    };
                    // The message for the future view is added to the current view's queue
                    msg_queue.push_back((sender, msg));
                }
            }
            _ => {}
        }
    }
}
// Simulate the generation of a network message for a future view
fn generate_future_view_message() -> Result<([u8; 32], ProgressMessage), ()> {
    // The generated message has a view number higher than the current view,
    // indicating it's for a future view
    Ok(([0; 32], ProgressMessage::Message(Message { view: 11u64 })))
}
fn main() {
```

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

```
    // Create an instance of ProgressMessageStub
    let mut pr_msg = ProgressMessageStub::new();
    // Define the current view
    let cur_view = 10u64;
    // Define the future view
    let future_view = 11u64;
    // Call recv to process a message from a future view
    pr_msg.recv(cur_view);
    // After recv is called, the message for the future view is
    // incorrectly stored under the current view
    // So we assert that the current view has a message
    assert!(
        pr_msg.msg_buffer.get(&cur_view).is_some(),
        "Expected message in current view, found none"
    );
    // Fetch the message from the current view's queue and
    // verify it's for the future view
    let msg_in_current_view = pr_msg.msg_buffer.get(&cur_view).unwrap().front().unwrap();
    assert_eq!(
        msg_in_current_view.1.view(),
        future_view,
        "Expected message view {} in current view, found message view {}",
        future_view,
        msg_in_current_view.1.view()
    );
    // The message for the future view should not have been stored under
    // the future view due to the bug
    // So we assert that the future view does not have a message
    assert!(
        pr_msg.msg_buffer.get(&future_view).is_none(),
        "Expected no message in future view, but found one"
    );
    println!("PoC completed successfully, the bug has been demonstrated.");
}
```

Recommendation

The caching system's implementation needs to be corrected to ensure that messages intended for future views are cached under the correct view. Here is a proposed fix:

```
// Cache the message if its for a future view.
else if msg.view() > cur_view {
    let msg_queue = if let Some(msg_queue) = self.msg_buffer.get_mut(&msg.view()) {
        msg_queue
    } else {
        self.msg_buffer.insert(msg.view(), VecDeque::new());
        self.msg_buffer.get_mut(&msg.view()).unwrap()
    };
    msg_queue.push_back((sender, msg));
}
```

With this adjustment, messages intended for future views will be correctly cached, ensuring that no messages are lost and the system behaves as expected.

Insufficient Validation of PublicKeyBytes in HotStuff Library

An issue has been identified in the HotStuff library regarding the lack of validation for incoming `PublicKeyBytes` in the `Network` trait. The responsibility of validation is currently left to the implementor. If the implementor does not correctly validate the incoming `PublicKeyBytes`, it can lead to various serious issues, including node crashes, network disruption, and potential blockchain compromises.

ID	PCH-008
Scope	Cryptography / Network

Severity	CRITICAL
Vulnerability Type	Error handling / Data Validation
Status	Acknowledged

Description

The HotStuff library provides the `Network` trait for handling incoming network requests. The implementor of the `Network` trait is responsible for the `recv` method, which returns an `Option` of a tuple composed of the request's origin (`PublicKeyBytes`) and a message.
The `Network` trait is defined as follows: *src/networking.rs:21*:

```
pub trait Network: Clone + Send {
    /// Informs the network provider the validator set on wake-up.
    fn init_validator_set(&mut self, validator_set: ValidatorSet);
    /// Informs the networking provider of updates to the validator set.
    fn update_validator_set(&mut self, updates: ValidatorSetUpdates);
    /// Send a message to all peers (including listeners) without blocking.
    fn broadcast(&mut self, message: Message);
    /// Send a message to the specified peer without blocking.
    fn send(&mut self, peer: PublicKeyBytes, message: Message);
    /// Receive a message from any peer. Returns immediately with a None if no message is available now.
    fn recv(&mut self) -> Option<(PublicKeyBytes, Message)>;
}
```

The key issue arises when the implementor does not correctly validate the incoming `PublicKeyBytes`. This lack of validation can result in various severe difficulties and vulnerabilities, including node crashes and network disruption. Furthermore, it can potentially lead to severe compromises in the blockchain's security.

Currently, the HotStuff library does not enforce the validation of incoming `PublicKeyBytes` within the `Network` trait. This leaves the implementor of the trait with the responsibility of ensuring the incoming `PublicKeyBytes` are valid. This lack of enforced validation opens a window for a variety of exploitable issues, with potential consequences ranging from node crashes to full blockchain compromises.

It's essential to note that this general issue can be exploited in various ways across the codebase, not just in a specific function or module. Importantly, the exploitation of this vulnerability can trigger even more critical issues, such as the one documented in PCH-009, leading to remote node crashes.

Recommendation

To address this vulnerability, the HotStuff library should incorporate a layer of validation for `PublicKeyBytes` within the `Network` trait or in the `networking` module.
Alternatively, it could provide explicit documentation instructing implementors on how to properly validate `PublicKeyBytes`.

Incorporating this validation into the library itself would provide an additional layer of defense, preventing issues stemming from malformed `PublicKeyBytes`. This, in turn, would enhance the security and stability of nodes using the library.

A suggested place to perform this validation check would be within the `start_polling` function in the `networking` module. Here's a proposed modification:

```
pub(crate) fn start_polling<N: Network + 'static>(mut network: N, shutdown_signal: Receiver<()>) -> (
    JoinHandle<()>,
    Receiver<(PublicKeyBytes, ProgressMessage)>,
    Receiver<(PublicKeyBytes, SyncRequest)>,
    Receiver<(PublicKeyBytes, SyncResponse)>,
) {
    let (to_progress_msg_receiver, progress_msg_receiver) = mpsc::channel();
    let (to_sync_request_receiver, sync_request_receiver) = mpsc::channel();
    let (to_sync_response_receiver, sync_response_receiver) = mpsc::channel();
    let poller_thread = thread::spawn(move || {
```

```
        loop {
            match shutdown_signal.try_recv() {
                Ok(()) => return,
                Err(TryRecvError::Empty) => (),
                Err(TryRecvError::Disconnected) => panic!("Poller thread disconnected from main thread"),
            }
            if let Some((origin, msg)) = network.recv() {
                match PublicKey::from_bytes(origin) {
                    Ok(_) => {
                        match msg {
                            Message::ProgressMessage(p_msg) => { let _ = to_progress_msg_receiver.send((origin, p_msg))
                            Message::SyncMessage(s_msg) => match s_msg {
                                SyncMessage::SyncRequest(s_req) => { let _ = to_sync_request_receiver.send((origin, s_r
                                SyncMessage::SyncResponse(s_res) => { let _ = to_sync_response_receiver.send((origin, s_
                            }
                        }
                    }
                    Err(_) => continue,
                }
            } else {
                thread::yield_now()
            }
        }
    });
    (
        poller_thread,
        progress_msg_receiver,
        sync_request_receiver,
        sync_response_receiver,
    )
}
```

In this modification, the loop first fetches the network message with `network.recv()`. If a message is available `(Some((origin, msg)))`, the function then tries to construct a `PublicKey` from `origin` using `PublicKey::from_bytes(origin)`. If this operation is successful `(Ok(_))`, the function continues to handle the message as before. If the operation fails `(Err(_))`, the function continues with the next loop iteration, effectively skipping the handling of the current message due to invalid `PublicKeyBytes`.

This ensures only messages with valid `PublicKeyBytes` are processed, adding a robust layer of defense against any malformed `PublicKeyBytes` and thereby enhancing the system's resilience.

Message Cache Poisoning via Malicious Vote Message Causing a System Panic

An identified critical vulnerability lies in the potential for Message Cache Poisoning within the `networking` module of the system. Specifically, this issue surfaces when dealing with the `on_receive_vote()` function in tandem with an incorrectly implemented caching system in the `recv()` function. An adversarial node can construct and disseminate a malicious vote, exploiting these vulnerabilities to incite a panic in other nodes' vote collection process.

ID	PCH-017
Scope	Voting system / Networking module
Severity	CRITICAL
Vulnerability Type	Message Cache Poisoning / Denial Of Service
Status	Fixed

Description

This vulnerability originates from the way the HotStuff protocol handles errors and manages cache for received votes. Specifically, an adversarial node can exploit the system by constructing and broadcasting a malicious vote that contains a future `view` number. This malicious vote can trigger a system panic due to poor error handling within the `votes.collect()` function (as detailed in PCH-018) and a flaw in the `recv()` function's caching system (as noted in PCH-016).

In the `votes.collect()` function, a check exists to validate if the `view` number and `chain_id` of a received vote matches those used to create the `VoteCollector` instance. If these details do not match, instead of safely discarding the vote, the function triggers a panic that disrupts the network operation.
src/types.rs:351:

```
pub(crate) fn collect(&mut self, signer: &PublicKeyBytes, vote: Vote) -> Option<QuorumCertificate> {
    if self.chain_id != vote.chain_id || self.view != vote.view {
        panic!()
    }
    /* ... */
}
```

This `panic!()` can be triggered by a flaw in `recv()` function, where messages intended for future views are cached under the current view as described in PCH-016.
Permitting vote for future view being processed in the current view by `on_receive_vote` if `recv()` function is called 2 times within the current view.

This vulnerability presents an opportunity for an adversarial node to potentially launch a majority attack, effectively paralyzing the operation of a significant portion of the network.

Proof of Concept

1. The adversarial node broadcasts a malicious vote intended for a view higher than the current one. The only requirement is that this message is correctly signed using any valid private key that corresponds to a legitimate public key.
2. Upon receipt, the `recv()` function stores the malicious vote in the cache as a message for the current view, due to the identified caching flaw.
3. A trigger message, which would incite a second call to `recv`, is received by the node. Various scenarios can instigate this event:
    - A valid nudge is received but does not necessarily lead to a transition to the next view. If the replica is not the leader for the next view, it remains in the current view and could invoke `recv` again.
    - A valid vote is received and added to the collected votes. If the collected votes do not yet form a quorum certificate (QC), the replica remains in the current view and could call `recv` again.
    - New view messages from other validators are received and processed. If a quorum of validators has not sent a new view message yet, the replica stays in the current view and could call `recv` again.
    - A proposal that fails checks and can't be inserted into the block tree is received. The replica would stay in the same view and could call `recv` again.
    - A nudge that fails checks and can't be inserted into the block tree is received. The replica would stay in the same view and could call `recv` again.
4. After the processing of the previous message, `recv` is invoked again within `execute_view`, causing the malicious vote in the cache to be processed and subsequently triggering a crash.

Recommendation

It's crucial to address the vulnerabilities identified in PCH-016 and PCH-018.
Resolving these issues will significantly contribute to the robustness of the system and reduce the potential of similar exploits in the future.

The `votes.collect()` function currently contains a check to ensure the `view` and `chain_id` of an incoming vote match the `view` and `chain_id` of the `VoteCollector`. However, instead of handling the mismatch gracefully, the function triggers a `panic!()`.
This can be adjusted so that instead of inducing a panic, the function discards the non-matching vote and returns immediately. This approach will ensure that the system remains stable and is not thrown into a panic state by a malicious vote.

Here's how the revised function could look:

```
pub(crate) fn collect(
    &mut self,
```

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

```
    signer: &PublicKeyBytes,
    vote: Vote,
) -> Option<QuorumCertificate> {
    // Existing check for matching chain_id and view
    if self.chain_id != vote.chain_id || self.view != vote.view {
        // Instead of inducing a panic, now discards the vote and returns immediately
        return None;
    }
    // Rest of the function...
}
```

Node Crash Potential Due to Unsafe Arithmetic Operations

The use of unsafe arithmetic operations in certain parts of your codebase can introduce substantial security vulnerabilities. These risks can lead to significant failures, including the possibility of node crashes, adversely affecting the system's robustness and dependability.

ID	PCH-015
Scope	Arithmetic calculations
Severity	HIGH
Vulnerability Type	Integer Overflow / Crashes
Status	Fixed

Description

Our audit highlighted some arithmetic operations in your code, as described in PCH-014, that could potentially lead to crashes, despite their probability being relatively low due to the requirement of a large number of blocks. However, certain other operations pose a more significant risk. This issue aims to underline those operations that present the most immediate threats.

Handling time durations

Specific pieces of code in the `algorithm.rs` and `pacemaker.rs` modules involve addition operations with time durations. These operations are a potential vector for overflows that can lead to node crashes.

In the `algorithm.rs` module, the following code is of particular concern:

_src/algorithm.rs:110:

```
let view_deadline = Instant::now() + pacemaker.view_timeout(view, block_tree.highest_qc().view);
```

src/algorithm.rs:481:

```
if let Some(response) = sync_stub.recv_response(*peer, Instant::now() + pacemaker.sync_response_timeout())
```

These operations rely on the implementation of `view_timeout` and `sync_response_timeout`, which are parts of the `Pacemaker` trait and require user implementation. If these methods return large values, overflows may occur, leading to node crashes.

Additionally, in `pacemaker.rs`, the implementation of `Pacemaker` for `DefaultPacemaker` introduces security risks:

src/pacemaker.rs:56:

```
self.minimum_view_timeout + Duration::new(u64::checked_pow(2, exp).map_or(u64::MAX, identity), 0)
```

Handling validator powers

In the `types.rs` module, specific calculations involving validator powers are performed without proper validations or range checks. This lack of validation can result in arithmetic operations that overflow and compromise the correctness of the system.

The following code snippets illustrate this issue:

src/types.rs:99:

```
signature_set_power += power;
```

src/types.rs:369:

```
*power += self.validator_set.power(signer).unwrap();
```

src/types.rs:421:

```
self.accumulated_power += self.validator_set.power(sender).unwrap()
```

Recommendation

To address the identified risks associated with unsafe arithmetic operations, we recommend the following measures:

- Utilize the methods provided by the Rust Standard Library. These methods include `checked_add/sub/mul/div`, `saturating_add/sub/mul/div`, `overflowing_add/sub/mul/div`, and others, to perform arithmetic operations with built-in safety checks;
- Implement proper input validation and bounds checking to prevent potential overflows, ensuring that input values are within acceptable ranges;
- Apply appropriate data type conversions or scaling factors to ensure calculations are performed within safe ranges, considering the expected magnitude of the values involved;
- Implement comprehensive unit tests to identify and rectify any potential vulnerabilities related to arithmetic operations, covering a wide range of input scenarios and edge cases.

By implementing these recommendations, you can significantly reduce the risks of node crashes and associated security vulnerabilities stemming from unsafe arithmetic operations. This would bolster both the robustness and the security of your system.

Unbounded Vector Size in `Block` structure

The `Block` structure in the codebase contains a field named `data` which is defined as `Vec<Vec<u8>>`. However, there are no constraints or limits imposed on the size of vectors. This design choice allows the vector to potentially grow without bounds, leading to memory exhaustion and potential denial-of-service (DoS) vulnerabilities.

ID	PCH-011
Scope	state
Severity	HIGH
Vulnerability Type	Memory exhaustion / DoS
Status	Acknowledged

Description

The `Block` structure is currently defined as following:

```
pub type Data = Vec<Datum>;
pub type Datum = Vec<u8>;
#[derive(Clone, BorshSerialize, BorshDeserialize)]
pub struct Block {
    /* ... */
    pub data: Data,
}
```

The issue arises from the lack of constraints on the size of the inner vectors within `data` field. This means that the vector can grow indefinitely, potentially consuming an excessive amount of memory and leading to resource exhaustion.

This vulnerability can be exploited when a Byzantine replica implements the `produce_block` function from the `App` trait in a way that generates a `Block` with a significantly large `data` vector. Since there are no checks on the length of the vector in the algorithms, other replicas that do not perform their own checks in the implementation of `validate_block` may experience slowdowns and memory exhaustion.

The `on_view_timeout` function could mitigate such an attack by limiting the amount of time spent processing a block. However, it doesn't eliminate the risk, as significant processing slowdowns or repeated timeouts could still disrupt the system's operation.

Besides the aforementioned issue, blocks of an unbounded size could result in the following:

1. Network Congestion: Large blocks can lead to network congestion, making it difficult for the nodes to propagate these blocks across the network.
2. Disk Space Exhaustion: Persisting these large blocks on disk might fill up the available disk space rapidly, leading to potential system crashes.
3. Long Validation Time: The validation of these large blocks could be time-consuming and delay the processing of subsequent blocks.
4. Increased Sync Time: Large blocks can significantly increase the time required for a new node to sync with the existing network state.
5. Blockchain Bloat: Over time, the storage of these large blocks could cause the blockchain to grow excessively, making it difficult to manage.

Proof of Concept

To demonstrate the impact of this issue, a modified integration test can be executed. By changing the produce_block function to generate a block with a large data vector, the subsequent views experience timeouts and block creation stops. The modified integration test code snippet is provided below:

```
impl App<MemDB> for NumberApp {
    // In produce_block we change the data to make it bigger
    fn produce_block(&mut self, request: ProduceBlockRequest<MemDB>) -> ProduceBlockResponse {
        /* ... */
        let data = vec![tx_queue.try_to_vec().unwrap(); 1_000_000];
        /* ... */
    }
     /* ... */
}
// The test is similar to the beginning part of the existing integration tests
// The node submits a transactions, since we modified produce_block the resulting block
// contains a large vector of data
#[test]
fn test_block_data() {
    // Test setup is exactly the same as in integration_test
    // Network is mocked, nodes are created
    // The first node is a validator
    /* ... */
    // Submit an Increment transaction to the initial validator.
    log::debug!("Integration test: submit an Increment transaction to the initial validator.");
    nodes[0].submit_transaction(NumberAppTransaction::Increment);
    // Wait some time to observe if blocks are inserted
    thread::sleep(Duration::from_millis(10_000));
}
```

The output when running this test will be:

```
% cargo test test_block_data
   Running tests/test.rs (target/debug/deps/test-f5b9030eb68633ae)
running 1 test
[ThreadId(2)][DEBUG] ReplacingHighestQc, AAAAAAA.., Generic
[ThreadId(5)][DEBUG] EnteredView, 1
[ThreadId(2)][DEBUG] ReplacingHighestQc, AAAAAAA.., Generic
[ThreadId(2)][DEBUG] ReplacingHighestQc, AAAAAAA.., Generic
[ThreadId(8)][DEBUG] EnteredView, 1
[ThreadId(2)][DEBUG] Integration test: submit an Increment transaction to the initial validator.
[ThreadId(11)][DEBUG] EnteredView, 1
[ThreadId(8)][DEBUG] ReceivedProposal, On/KSRM.., Eba/afE.., 0
[ThreadId(11)][DEBUG] ReceivedProposal, On/KSRM.., Eba/afE.., 0
[ThreadId(5)][INFO] Proposed, 1, Eba/afE.., 0
[ThreadId(5)][DEBUG] ReceivedProposal, On/KSRM.., Eba/afE.., 0
[ThreadId(8)][DEBUG] InsertingBlock, Eba/afE.., 0
[ThreadId(11)][DEBUG] InsertingBlock, Eba/afE.., 0
[ThreadId(5)][DEBUG] InsertingBlock, Eba/afE.., 0
[ThreadId(8)][DEBUG] EnteredView, 2
[ThreadId(11)][DEBUG] EnteredView, 2
[ThreadId(5)][INFO] Voted, 1, Eba/afE.., 0, Generic
[ThreadId(5)][INFO] ViewTimedOut, 1, AAAAAAA.., Generic
[ThreadId(5)][DEBUG] EnteredView, 2
...
[ThreadId(8)][DEBUG] EnteredView, 4
[ThreadId(11)][DEBUG] EnteredView, 4
[ThreadId(5)][INFO] Voted, 3, CXSGugz.., 1, Generic
[ThreadId(5)][INFO] ViewTimedOut, 3, fNGCJyk.., Generic
[ThreadId(8)][INFO] ViewTimedOut, 4, fNGCJyk.., Generic
[ThreadId(11)][INFO] ViewTimedOut, 4, fNGCJyk.., Generic
[ThreadId(11)][DEBUG] EnteredView, 5
[ThreadId(11)][INFO] ViewTimedOut, 5, fNGCJyk.., Generic
test test_data ... ok
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 3 filtered out; finished in 26.73s
```

After the second view, subsequent views experience timeouts, resulting in the failure to insert new blocks and impeding the progress of the system.

Recommendation

To mitigate this issue, introduce a constraint on the size of vectors within the `Block` structure. Establish a maximum size for the `data` field to prevent vectors from growing unbounded and exceeding resource limits. Implement corresponding checks in the `validate_block` function to ensure blocks with data exceeding the allowed size are rejected.

Thoroughly test the changes under various scenarios to ensure that the system behaves as expected and the risk of potential memory exhaustion and DoS attacks is effectively mitigated.

Unsafe arithmetics

During the code audit, several instances of unsafe arithmetic operations were identified. These unattended operations can cause unpredictable and potentially harmful side effects in your application, such as arithmetic overflows.

ID	PCH-014
Scope	Arithmetic
Vulnerability	Integer overflow
Severity	HIGH
Status	Acknowledged

Description

Arithmetic operations that are not properly safeguarded could lead to critical errors such as overflows, which in some cases may even result in node crashes. Although certain calculations might be unlikely to trigger an overflow due to their high threshold, others could pose a more imminent risk. Detailed analyses of these calculations' potential risks are provided in the subsequent section, PCH-015.

To identify all instances of unsafe arithmetic operations within your codebase, execute the following command:

```
cargo clippy -- -W clippy::arithmetic_side_effects
```

The instances of unsafe arithmetic operations identified during the audit, broken down by modules, are as follows:

Algorithms

src/algorithm.rs:88:

```
cur_view = max(cur_view, max(block_tree.highest_view_entered(), block_tree.highest_qc().view)) + 1;
```

_src/algorithm.rs:110:

```
let view_deadline = Instant::now() + pacemaker.view_timeout(view, block_tree.highest_qc().view);
```

src/algorithm.rs:183:

```
(Some(highest_qc.block), block_tree.block_height(&highest_qc.block).unwrap() + 1)
```

src/algorithm.rs:239, 267, 280, 304, 333, 391, 427, 448:

```
let next_leader = pacemaker.view_leader(cur_view + 1, &block_tree.committed_validator_set());
```

src/algorithm.rs:476:

```
start_height: if let Some(height) = block_tree.highest_committed_block_height() { height + 1 } else { 0 },
```

src/algorithm.rs:481:

```
if let Some(response) = sync_stub.recv_response(*peer, Instant::now() + pacemaker.sync_response_timeout())
```

Types

src/types.rs:99:

```
signature_set_power += power;
```

src/types.rs:136:

```
((validator_set_power * 2) / 3) + 1
```

src/types.rs:369:

```
*power += self.validator_set.power(signer).unwrap();
```

src/types.rs:421:

```
self.accumulated_power += self.validator_set.power(sender).unwrap()
```

Pacemaker

src/pacemaker.rs:51:

```
*validator_set.validators().skip(cur_view as usize % num_validators).next().unwrap()
```

src/pacemaker.rs:55:

```
let exp = min(u32::MAX as u64, cur_view - highest_qc_view_number) as u32;
```

src/pacemaker.rs:56:

```
self.minimum_view_timeout + Duration::new(u64::checked_pow(2, exp).map_or(u64::MAX, identity), 0)
```

State

src/state.rs:647:

```
cursor += 1;
```

src/state.rs:898:

```
let mut res = Vec::with_capacity(a.len() + b.len());
```

Networking

src/networking.rs:116:

```
match self.receiver.recv_timeout(deadline - Instant::now())
```

src/networking.rs:195:

```
match self.responses.recv_timeout(deadline - Instant::now())
```

Recommendation

The detection of these unsafe arithmetic operations is a call to action for addressing potential vulnerabilities in your codebase. To mitigate these risks, consider using the Rust Standard Library's built-in methods for safer arithmetic computations.
These include `checked_add/sub/mul/div`, `saturating_add/sub/mul/div`, `overflowing_add/sub/mul/div`, and others. Implementing these safe arithmetical methods will help you manage the potential risks associated with arithmetic overflows effectively.

Unsoundness Issue in Borsh Dependency of HotStuff Library

An unsoundness issue has been discovered in the Borsh dependency used by the HotStuff library, as per a security review conducted via cargo audit.

ID	PCH-003
Scope	Dependencies
Severity	LOW
Status	Acknowledged

Description

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

The borsh crate (version 0.10.3) used by the HotStuff library is identified as having an unsoundness issue, as outlined in the RustSec advisory RUSTSEC-2023-0033. This issue relates to potential unsoundness when parsing borsh messages with Zero-Sized Types (ZSTs) that do not implement `Copy` or `Clone` traits.

Here is the output from the cargo audit for reference:

```
Crate:     borsh
Version:   0.10.3
Warning:   unsound
Title:     Parsing borsh messages with ZST which are not-copy/clone is unsound
Date:      2023-04-12
ID:        RUSTSEC-2023-0033
URL:       https://rustsec.org/advisories/RUSTSEC-2023-0033
Dependency tree:
borsh 0.10.3
└── hotstuff_rs 0.2.0
```

The unsoundness could lead to unexpected program behavior, including memory corruption, and in severe cases, potential security vulnerabilities. This can happen when ZSTs that do not implement `Copy` or `Clone` are involved in the serialization/deserialization processes.

However, an examination of the HotStuff library's codebase indicates that it does not employ any ZSTs that utilize `BorshSerialize` / `BorshDeserialize` without implementing `Copy` or `Clone`. As such, while this issue is present in the borsh dependency, the specific usage in the HotStuff library does not expose it to the associated risks.

Recommendation

Despite the HotStuff library not being directly impacted by this specific issue, it is crucial to monitor updates and potential fixes to the borsh crate. The associated issue can be tracked in the borsh repository at near/borsh-rs#19.

It is recommended to update the borsh dependency in the HotStuff library once a fix is released to eliminate potential future risk. Also, it is advised that the development team of the HotStuff library exercises caution when introducing new ZSTs that involve serialization/deserialization, ensuring they implement `Copy` or `Clone` to maintain soundness.

A proactive stance towards monitoring and resolving dependency vulnerabilities will significantly contribute to the overall security posture of the HotStuff library.

Genesis Block's Quorum Certificate Has Incorrect `chain_id`

In the current implementation of the consensus the genesis block always contains quorum certificate with `chain_id` equal to zero.

ID	PCH-013
Scope	Genesis Configuration
Severity	LOW
Status	Acknowledged

Description

The current implementation of the consensus introduces a flaw in the quorum certificate of the genesis block. Specifically, the `chain_id` value in the genesis quorum certificate is always set to zero, regardless of the actual `chain_id` defined by the `chain_id()` method in the `App` trait:

src/types.rs:121:

```
pub const fn genesis_qc() -> QuorumCertificate {
    QuorumCertificate {
        chain_id: 0,
        view: 0,
        block: [0u8; 32],
        phase: Phase::Generic,
        signatures: SignatureSet::new(),
    }
}
```

This inconsistency poses a potential security risk and could lead to attacks in the future if the code undergoes changes.

During replica initialization, the highest quorum certificate of the newly created `BlockTree` is established using the `genesis_qc` function, which incorrectly sets the `chain_id` to zero. Subsequently, when the first proposal is created in the `propose_or_nudge` function, the new block inherits this flawed quorum certificate, resulting in a proposal that contains a block with a quorum certificate featuring a zero `chain_id`.

While this issue may not immediately lead to security vulnerabilities, it introduces error-prone behavior and opens the door to potential vulnerabilities if the codebase evolves. The `genesis_qc` function is widely utilized, including the `is_genesis_qc` method, which is extensively used to handle scenarios related to the genesis block. Therefore, an incorrect `genesis_qc` could potentially give rise to severe security concerns in the future.

Proof of Concept

To verify this issue in tests, the following code can be added after the proposal is broadcasted:

```
if let ProgressMessage::Proposal(proposal_inner) = proposal.clone() {
    log::debug!(
        "proposal_inner.block.justify.chain_id == {}, proposal.chain_id() == {}",
        proposal_inner.block.justify.chain_id,
        proposal.chain_id()
    );
}
```

By modifying the implementation of the `chain_id()` method in the tests and executing them:

```
fn chain_id(&self) -> ChainID {
    42
}
```

The output will indicate the following:

```
...
[ThreadId(9)][DEBUG] proposal_inner.block.justify.chain_id == 0, proposal.chain_id() == 42
...
```

Recommendation

It is strongly advised to revise the implementation of the `genesis_qc` function to correctly capture the `chain_id` value, as defined by the `chain_id()` method in the `App` trait. This will ensure consistency and eliminate potential security risks associated with an incorrect `genesis_qc`.

HotStuff build

The HotStuff library exhibits an efficient and error-free build process.

ID	PCH-001
Scope	Build Process

Description

The HotStuff library, a Rust implementation of the BFT consensus protocol, builds smoothly without any compiler errors or warnings. The output of the `cargo build --release` command indicates a successful build process:

```
% cargo build --release
Finished release [optimized] target(s) in 25.44s
```

This output signifies adherence to sound Rust coding practices and idiomatic conventions. The absence of compiler warnings and errors suggests a high degree of attention to detail and meticulous code management on the part of the developers.

Recommendation

Given that the build process is currently running optimally and without any issues, no changes are recommended at this time. The development team should continue to uphold the established code quality standards and best practices in future updates and modifications. It is essential to maintain the error-free status of the build process, as it is indicative of the robustness and reliability of the software and minimizes potential risks and issues downstream.

Please note that this issue does not cover the results of linting tools such as `Clippy`, which may provide additional warnings and recommendations for code quality improvement. Those will be addressed in a separate issue.

Inconsistent Code Formatting in HotStuff Library

A `cargo fmt` check reveals inconsistent formatting in the HotStuff library's codebase.

ID	PCH-005
Scope	Code Quality
Status	Fixed

Description

Code formatting is essential for maintaining the readability and maintainability of the codebase. Inconsistent code formatting can lead to unnecessary diffs in the version control system, which can in turn complicate code reviews and make it more difficult to identify substantive changes.

The `cargo fmt -- --check` command was used to run a simulation that identifies parts of the code that would be reformatted. This command does not modify the code but prints out how the files would look after formatting.

Recommendation

We recommend running `cargo fmt` on the entire codebase to ensure that all code adheres to the standard Rust formatting. This can help improve the readability and maintainability of the codebase.

In addition, to prevent the introduction of improperly formatted code in the future, you may want to consider adding a `cargo fmt -- --check` step to your continuous integration (CI) pipeline. This would alert developers to formatting issues in their code before it is merged into the main codebase.

Insufficient Details in Functions and Data Structures Documentation

The documentation for the project can be enhanced, specifically regarding functions and data structures within the codebase.

ID	PCH-012
Scope	Documentation
Status	Acknowledged

Description

The project's crate-level documentation is comprehensive and provides a solid understanding of the library's functionality and usage. It covers every modules extensively, serving as a valuable resource for developers working with or integrating the HotStuff library.

However, there is a need for improvement at the function and data structure level. Many functions and structures lack descriptive doc strings, which are essential for generating detailed API documentation automatically.
By enhancing the clarity and completeness of the documentation through well-crafted doc strings, developers can benefit from the ability to generate API documentation effortlessly using the `cargo doc` command. This would greatly facilitate understanding and utilization of the HotStuff library.

Recommendation

We recommend focusing on enhancing the doc strings for functions and structures throughout the codebase. By providing detailed explanations, parameter descriptions, return value explanations, and relevant examples, we can significantly improve the clarity and comprehensiveness of the documentation.

Furthermore, developers can take advantage of the convenient `cargo doc` command to access the API documentation directly, facilitating their workflow and enhancing the overall developer experience with the HotStuff library.

Insufficient Error Handling Mechanism in HotStuff Library

The HotStuff project exhibits a significant deficiency in the implementation of its error handling system, frequently resorting to panic-induced exits rather than providing insightful error messages.

ID	PCH-018
Scope	Error Handling
Status	Acknowledged

Description

An in-depth examination of the HotStuff project revealed an absence of a proficient error handling system. Instead of presenting users with detailed and explanatory error messages, which can guide effective troubleshooting and debugging, the project commonly relies on `unwrap()`, `panic!()`, and `unreachable!()` macros. This practice does not promote user-friendly interactions with the library, nor does it provide useful insights into the underlying issues when they occur.

The use of these macros is particularly troubling as they can cause unexpected panics, which are extremely undesirable within a library such as HotStuff, designed for state machine replication. Panics can lead to node crashes, bringing forth considerable security concerns.

A distinct instance of this issue is outlined in PCH-009.

While the `unreachable!()` macro may be utilized to validate certain scenarios deemed impossible, it possesses the potential to trigger panics, which, combined with the substantial usage of other panic-prone macros, complicates code maintainability and readability.

Throughout the consensus component, we identified 81 instances of `unwrap()`, 9 instances of `panic!()`, and 2 instances of `unreachable!()` (excluding tests). It is essential to reduce their usage and ensure that no instance can potentially cause node crashes, even under actions performed by a Byzantine node.

Recommendation

To improve error handling, diminish panic risks, and promote better user experience, we propose the following recommendations:

Mitigate panic occurrences	
Explicit documentation and comments	
Avoid `unreachable!()`	
Implement structured error handling	

- Develop a distinct module for error handling that comprises an enum representing all possible errors throughout the project's modules.
- Refactor the codebase to replace panic occurrences with the suitable application of the error enum, returning `Result` types that encapsulate potential errors. This step facilitates structured error handling and graceful error propagation.
- Update method signatures throughout the codebase to reflect the new error handling approach, ensuring that errors are properly propagated up the call stack.
- Consider incorporating the thiserror library, which simplifies the creation of custom error types and allows for customization of error messages and associated data.

Implementing these recommendations establishes a more robust error handling system, reduces the reliance on panics, and promotes the resilience and stability of the project. Structured error handling enhances the usability of the library, provides clearer error information to users, and facilitates better error diagnostics and troubleshooting.

Linter Warnings

`cargo clippy` generates numerous warnings that should be addressed to improve the overall code quality.

ID	PCH-002
Scope	Linters
Status	Acknowledged

Description

During the static analysis process using `cargo clippy`, a significant number of warnings are generated. These warnings can indicate various issues in the codebase, including:

- Unoptimized or inefficient code
- Non-idiomatic Rust patterns
- Redundant or unnecessary code
- Lack of documentation for unsafe functions

- Missing implementations of expected methods
- Potential coding errors or logic flaws

The full list of clippy lints that generated warnings includes:

- collapsible_if
- zero_prefixed_literal
- needless_borrow
- too_many_arguments
- redundant_clone
- len_zero
- needless_bool
- match_like_matches_macro
- new_without_default
- len_without_is_empty
- map_entry
- iter_skip_next
- missing_safety_doc
- let_and_return
- unnecessary_cast
- while_let_loop
- needless_lifetimes
- type_complexity
- comparison_chain

Ignoring these warnings may lead to a harder-to-maintain codebase, potential performance issues, and even security vulnerabilities. It is important to address all of the warnings generated by `cargo clippy` to ensure a high-quality and maintainable codebase.

It is important to note that `cargo clippy` is configured to generate warnings predominantly for the default set of lints. However, there may exist additional issues that could be uncovered by enabling and meticulously examining supplementary lints. These potential issues will be addressed systematically in separate, forthcoming issues.

Recommendation

To ensure a high-quality and maintainable codebase, it is crucial to address all the warnings generated by `cargo clippy`. By addressing these linter warnings, you will enhance the overall code quality, making it easier to maintain, troubleshoot, and potentially improve the performance and security of your Rust project.

We recommend the following steps:

- Review each warning generated by `cargo clippy` and understand its implications on the codebase.
- Apply appropriate code changes to resolve the warnings, following the best practices and idiomatic patterns of Rust programming.
- Document any necessary changes or considerations in the codebase to ensure future developers are aware of the reasoning behind the modifications.
- Regularly run `cargo clippy` as part of the development workflow to catch new warnings and maintain code quality over time.

By proactively addressing the linter warnings, you will not only improve the overall code quality but also foster a culture of continuous improvement and adherence to Rust's best practices.

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Test coverage

The project shows a fair test coverage of **77.41%**, but it's entirely comprised of two integration tests, with no unit tests present.

ID	PCH-004
Scope	Code Quality / Testing
Status	Acknowledged

Description

We recommend utilizing the cargo tarpaulin command to assess code coverage. Running the following command will generate an HTML file with detailed coverage information for each file:

```
cargo tarpaulin --out Html --output-dir ./tarpaulin-report
```

The generated HTML file provides coverage statistics for each file, including the number of lines covered and the percentage of coverage.

Covered: 722 of 933 (77.41%)

File	Coverage
algorithm.rs	144/212 (67.92%)
app.rs	8/12 (66.67%)
logging.rs	31/37 (83.78%)
messages.rs	32/36 (88.89%)
networking.rs	49/81 (60.49%)
pacemaker.rs	9/11 (81.82%)
replica.rs	32/32 (100.00%)
state.rs	241/362 (66.57%)
sync_server.rs	7/14 (50.00%)
types.rs	111/136 (81.62%)

The coverage statistics indicate that the files `algorithm.rs`, `app.rs`, `networking.rs`, `state.rs`, and `sync_server.rs` currently have lower test coverage.

While having integration tests is important to test the consensus as a whole, it is advisable to create unit tests for each module as well. Unit tests provide more flexibility and allow for testing individual functions separately, covering all edge cases and ensuring comprehensive coverage of necessary workflows.

Recommendation

We recommend improving the test coverage in the project by implementing a comprehensive test suite that includes both integration tests and smaller unit tests for each modules. A thorough testing approach is essential for ensuring the security, stability, and maintainability of the project. Having separate small unit tests for each module will further enhance the test coverage and enable thorough testing of individual functionalities.

Unconventional Pattern Matching

The code includes an unusual usage of `match` pattern, leading to unnecessary usage of `unreachable!()`. This unconventional pattern matching reduces code readability and increases the potential for errors.

ID	PCH-010
Scope	Code Quality
Status	Fixed

Description

The following code snippet illustrates the issue:

src/algorithm.rs:231:

```
match () {
    // Produce a proposal.
    () if highest_qc.phase.is_generic() || highest_qc.phase.is_commit() => {
        /* ... */
    },
    // Produce a nudge.
    () if highest_qc.phase.is_prepare() || highest_qc.phase.is_precommit() => {
        /* ... */
    },
     _ => unreachable!(),
}
```

While this code snippet does not introduce any errors, the use of this unconventional match pattern is not idiomatic and unnecessary. It makes the code less readable and harder for other developers to understand.

Furthermore, the presence of `unreachable!()` introduces a potential panic if the code changes. This renders the code more error-prone and could lead to security issues in the future.

Recommendation

We recommend refactoring the code to use a more conventional matching pattern, which improves code readability and eliminates the need for `unreachable!()`:

```
match highest_qc.phase {
    Phase::Generic | Phase::Commit(_) => {
        /* ... */
    },
    Phase::Prepare | Phase::Precommit(_) => {
        /* ... */
    }
}
```

This refactored code is safe and does not leave room for potential panics.

Additionally, it is advisable to conduct a thorough code review to identify any other sections of code that may potentially panic and consider removing them if possible. To assist with this, you can use the clippy lint tool with the `unreachable` warning:

```
cargo clippy -- -W clippy::unreachable
```

Furthermore, we recommend using more conventional matching patterns and idiomatic Rust constructs throughout your codebase. By following idiomatic Rust practices, you can improve code readability, maintainability, and reduce the likelihood of introducing errors.

Adhering to these practices also helps other developers understand the code more easily, promotes consistency, and aligns with community best practices.

Hacken OÜ
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia
support@hacken.io

Disclaimers

Hacken disclaimer

The code base provided for audit has been analyzed according to the latest industry code quality, software processes and cybersecurity practices at the date of this report, with discovered security vulnerabilities and issues the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functional specifications). The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code (branch/tag/commit hash) submitted to and reviewed, so it may not be relevant to any other branch. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits, public bug bounty program and CI/CD process to ensure security and code quality. English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical disclaimer

Protocol Level Systems are deployed and executed on hardware and software underlying platforms and platform dependencies (Operating System, System Libraries, Runtime Virtual Machines, linked libraries, etc.). The platform, programming languages, and other software related to the Protocol Level System may have vulnerabilities that can lead to security issues and exploits. Thus, Consultant cannot guarantee the explicit security of the Protocol system in full execution environment stack (hardware, OS, libraries, etc.)