HACKEN

PARALLELCHAIN SECURITY ASSESSMENT

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 Full Node
Website	https://parallelchain.io/
Repository	https://github.com/parallelchain-io/fullnode
Commits	50e04772508f3212744976f33b1c42d3a18c75ba
Repository	https://github.com/parallelchain-io/pchain-world-state
Commits	28f683405a988dd5321da7c4955591b106500e1e
Repository	https://github.com/parallelchain-io/pchain-runtime
Commits	9f3209144743b889b8723889252ffb4afd53bf4b
Repository	https://github.com/parallelchain-io/pchain-types-rust
Commits	1dc39d5d942156a2e468bcb09dcf1afe41c0a11f
Repository	https://github.com/parallelchain-io/pchain-network
Commits	90367aa1cd08824197cdf8f7d1d34a9c80661ac4
Platform	L1
Network	Parallel Chain
Languages	Rust
Methodology	Blockchain Protocol and Security Analysis Methodology
Lead Auditor	s.akermoun@hacken.io
Auditor	n.lipartiia@hacken.io
Approver	l.ciattaglia@hacken.io
Timeline	12.06.2023 - 21.08.2023
Changelog	22.08.2023 (Preliminary Report)
Changelog	18.09.2023 (Final Report)

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

Table of contents

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

Summary

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

Central to ParallelChain's appeal is its robust SDK that offers developers a powerful platform to create their own applications, including smart contracts, using the Rust programming language. The choice of Rust reflects the company's commitment to leveraging cutting-edge technologies, offering a balance of performance, safety, and sophistication. Applications and smart contracts developed using ParallelChain's SDK are compiled to WebAssembly (Wasm), a binary instruction format that ensures seamless execution, flexibility, and security.

The focus of this report will be a comprehensive analysis of the ParallelChain full node, including its associated modules and crates, runtime, network, world state, and custom types and data structures. This evaluation aims to provide insight into the project's quality, security, and potential.

Documentation quality

ParallelChain offers a complete set of documentation that spans across various components of the system, such as the network, runtime, worldstate, and pchain-types. These associated crates and modules are well-documented, providing developers with a clear understanding of the inner mechanisms and functionalities.

The documentation of the node, in comparison to other components, is slightly less comprehensive but remains at a good level. This slight disparity doesn't diminish the overall quality, but additional focus on detailing the node's documentation would harmonize it with the extensive documentation found in the associated components.

Moreover, developers and reviewers can benefit from the external documentation available in the repository ParallelChain Protocol. This outside resource further describes the protocol and its inner workings, acting as a valuable guide to those working with the ParallelChain system.

The total Documentation Quality score is **8** out of 10.

Code quality

ParallelChain's code quality is commendable, displaying a strong adherence to Rust's best practices. This commitment to excellence is demonstrated by the minimal presence of linter warnings, reflecting the codebase's alignment with well-established programming standards.

The project exhibits an impressive test coverage that speaks to its robustness.

The development team has acknowledged the concern regarding the use of catch_unwind as a substitute for traditional try/catch error handling, particularly referencing issue PCN-018. They have concurred on the importance of reducing reliance on this mechanism and have expressed their commitment to exploring alternative means for detecting unforeseen exceptions, including the implementation of fuzz tests.

The development team has indeed acknowledged the issue concerning unsafe arithmetic operations. They have approached this matter with thorough consideration and attention to detail, recognizing the inherent risk of arithmetic overflows. While the team has justified that certain instances of unsafe arithmetic usage are not susceptible to overflows, they have also taken proactive steps to address other potential concerns. Specifically, they have successfully resolved a critical issue by introducing a comprehensive error-handling mechanism in conjunction with the utilization of `checked_*` arithmetic operations. This prudent action enhances the overall safety and reliability of the codebase.

The attention to code quality is evident, and the existing areas for improvement are resolvable. By addressing these identified concerns, ParallelChain can further bolster the reliability and safety of its codebase.

The total Code Quality score is **8** out of 10.

Architecture quality

The architecture of ParallelChain's full node is both elegant and efficient, reflecting a simplicity that is a strong asset rather than a limitation. By adopting a previously audited BFT consensus mechanism that is an implementation of the HotStuff consensus protocol, the design attains a blend of robustness and efficiency.

The utilization of wasmer runtime for WebAssembly (Wasm) execution offers modern and versatile execution capabilities. Simultaneously, it supports the streamlined architecture, providing flexibility and performance benefits for smart contract execution.

Furthermore, the network stack within ParallelChain is well-defined and thoughtfully structured. This clear design leads to a robust peer-to-peer network, facilitating efficient communication between nodes without unnecessary complexity.

One area of concern that merits attention is the synchronization of nonces, particularly in the handling of transactions within the mempool. An observed behavior where transactions sent in quick succession are dropped due to a `Nonce inaccessible` error reveals a complexity in the nonce handling logic. This situation doesn't pose an immediate security risk but is considered a design flaw that requires careful consideration to maintain system robustness and integrity.The development team has acknowledged this flaw and has outlined their intention to incorporate a new mempool design in a future protocol version, addressing this concern comprehensively.

This balance of simplicity and sophistication in the architecture contributes to the system's stability, scalability, and ease of understanding. It's an exemplary demonstration of how effective design doesn't have to be overly complex but can achieve its goals through clarity and precision.

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

Security score

In our analysis of the Parallelchain fullnode and its components, as defined within the scope of this audit, we have uncovered multiple security issues that required immediate attention. These comprise two critical, one high, two medium, and seven low severity concerns. Nevertheless, the development team has responded proactively and effectively to rectify these issues.

Critical issue PCN-001 which identified the potential for a Denial of Service (DoS) attack stemming from the utilization of an outdated version of the `libp2p` library, has been promptly addressed.

Furthermore, the critical issue PCN-008, pinpointing a vulnerability where submitting a transaction with a high-priority fee could trigger a runtime panic, has been successfully rectified by the team through the implementation of error handling mechanisms employing safe arithmetic operations.

The development team has also demonstrated diligence in providing the requisite checks and mechanisms to resolve issue PCN-020, which highlighted the absence of adequate validation measures for transactions received through the broadcast mechanism.

Medium severity issue PCN-013, revealing a potential avenue for authorized users to induce a runtime panic by submitting a transaction with the `NextEpoch` command, has likewise been effectively addressed by the development team.

Medium severity issue PCN-016 received prompt and effective correction, mitigating the possibility of a malicious node executing whitelisted commands that should not be accepted from this signer.

Low issue PCN-009 was resolved, which eliminated the risk of runtime panics related to unsafe arithmetic during balance management.

Low issue PCN-012 has been effectively managed, ensuring that vulnerable dependencies are updated, thereby reducing potential risks.

Low issue PCN-014 underwent correction, removing the previously identified typo that compromised the effectiveness of whitelisting the `SetPoolSettings` command.

Low issue PCN-015 has received attention and resolution, preventing the possibility of a malicious validator circumventing the initiation of the next epoch, thus preserving the integrity of the epoch transition process.

Low issue PCN-017 has been addressed effectively, rectifying the vulnerability associated with the unencrypted storage of private keys and mitigating the potential security risk.

Low issue PCN-019 has been duly recognized, drawing attention to a vulnerability that has the potential to impact specific Apple devices operating on the ARM64 architecture. This vulnerability may result in node crashes in the event of Wasmer VM errors. Notably, it has been acknowledged that the preferred environment for running nodes is on Linux. As part of the resolution process, the documentation has been revised to provide users with updated information regarding the recommended hardware requirements.

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

Total score

Considering all metrics, the total score of the report is **9.4** out of 10.

Findings count and definitions

Severity	Findings	Severity Definition
Critical	2	Vulnerabilities that can lead to a complete breakdown of the blockchain network's security, privacy, integrity, or availability fall under this category. They can disrupt the consensus mechanism, enabling a malicious entity to take control of the majority of nodes or facilitate 51% attacks. In addition, issues that could lead to widespread crashing of nodes, leading to a complete breakdown or significant halt of the network, are also considered critical along with issues that can lead to a massive theft of assets. Immediate attention and mitigation are required.
High	1	High severity vulnerabilities are those that do not immediately risk the complete security or integrity of the network but can cause substantial harm. These are issues that could cause the crashing of several nodes, leading to temporary disruption of the network, or could manipulate the consensus mechanism to a certain extent, but not enough to execute a 51% attack. Partial breaches of privacy, unauthorized but limited access to sensitive information, and affecting the reliable execution of smart contracts also fall under this category.
Medium	2	Medium severity vulnerabilities could negatively affect the blockchain protocol but are usually not capable of causing catastrophic damage. These could include vulnerabilities that allow minor breaches of user privacy, can slow down transaction processing, or can lead to relatively small financial losses. It may be possible to exploit these vulnerabilities under specific circumstances, or they may require a high level of access to exploit effectively.
Low	6	Low severity vulnerabilities are minor flaws in the blockchain protocol that might not have a direct impact on security but could cause minor inefficiencies in transaction processing or slight delays in block propagation. They might include vulnerabilities that allow attackers to

		cause nuisance-level disruptions or are only exploitable under extremely rare and specific conditions. These vulnerabilities should be corrected but do not represent an immediate threat to the system.
Total	11	

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

Scope of the audit

Protocol Audit

Cryptography and Keys

- Cryptography Libraries
- Keys Generation
- Keystore storage
- Asymmetric (Signing and Verification)
- Cryptographic attacks analysis (Analytic Attack, Implementation Attack, Statistical Attack, ...)

Accounts

- Accounts implementation review
- Wallet implementation review
- Security vectors analysis (data availability, nonce,..)

Chain

- Tx implementation review (defaults, timestamps, assembly)
- Bootstrap review (genesis, seed peers)
- Mempool review (defaults, timestamps)
- Standard attacks review (replay, malleability,...)

Consensus

- ParallelChain Hotstuff integration review

VM

- VM implementation review
- Known VM Vulnerabilities review
- Attack scenarios analysis (Gas, race, stack, DoS, state implosion...)

P2P

- P2P implementation review
- Attack scenarios analysis (defaults, DoS, MiM, overflows, state machine)

RPC

- RPC implementation review
- Attack scenarios analysis (defaults, DoS, overflows, ..)

Storage

- ParallelChain Worldstate integration review

Implementation

Code Quality

- Static Code Analysis
- Tests coverage

Protocol Tests

Node Tests

- Environment Setup
- E2E sync tests
- Consensus tests
- E2E transaction tests

Fuzz Tests

- Chain fuzz tests
- VM fuzz tests
- Serialization tests

Issues

Arithmetic Overflow Due to High Priority Fee

The runtime encounters an arithmetic overflow when a transaction with a high `priority_fee_per_gas` is processed, leading to a panic.

ID	PCN-008
Scope	Mempool/Runtime
Severity	CRITICAL
Vulnerability Type	Integer overflow
Status	Fixed (b6d86f6)

Description

During the thorough code audit, a line of code in the pre-charge phase of the runtime has raised concerns:

pchain-runtime/src/execution/phase.rs:48:

```
if (gas_limit * (base_fee + priority_fee)) > origin_balance
```

The variable `priority_fee_per_gas` lacks bounds, making it susceptible to the creation and submission of a transaction where the sum of the base fee and priority fee exceeds the maximum value of `u64::MAX`. This triggers an arithmetic overflow, leading to a panic. This would cause a panic due to the overflow, which would be caught using `catch_unwind` in the fullnode.

It's important to acknowledge that if the node was compiled with

```
[profile.release]
panic = "abort"
```

the panic will result in an immediate crash of the node.

Furthermore, it's noteworthy that the transaction does not get included in the blockchain, not even as a failed transaction. Additionally, the sender's account does not incur any fee for this transaction.

Proof of Concept

To cause a single overflow in runtime, an attacker can submit a transaction with correct data (including commands, nonce, and base fee) and set `priority_fee_per_gas` to `u64::MAX`:

```
{
  "commands": [
    {
      "Transfer": {
        "recipient": "NBbktZ2Ed0GHDH0jCbtYzD4ICbZ7USv_bT0z-27yHjc",
        "amount": 1
      }
    }
  ],
  "nonce": 0,
  "gas_limit": 500000,
  "max_base_fee_per_gas": 10,
```

```
  "priority_fee_per_gas": 18446744073709551615
}
```

This transaction will not be included in the blockchain. However, if you run the blockchain locally and check logs from mempool, you can observe that the transaction is included there. Yet, when `execute_from_mempool` is called, it logs something like this:

```
thread '<unnamed>' panicked at 'attempt to add with overflow', /Users/nino/.cargo/git/checkouts/pchain-runtime-c0b1e987
```

Recommendation

Addressing potential overflow scenarios in a thoughtful manner is of utmost importance. It is advised to utilize the secure arithmetic computation methods provided by the Rust Standard Library.

Transactions that trigger overflows should be meticulously recorded on the blockchain, marked with a "failed" status, and the corresponding fee should be charged accordingly.

To bolster the application's security and stability, it is vital to substitute risky arithmetic operations with reliable alternatives. This proactive approach will greatly enhance the overall resilience of the system.

DoS Risk from Vulnerable Dependency

The node utilizes an outdated version of the `libp2p` library, which introduces a vulnerability that can lead to Denial-of-Service (DoS) attacks.

ID	PCN-001
Scope	Dependencies
Severity	CRITICAL
Vulnerability Type	Denial-of-Service (DoS)
Status	Fixed (5b0b2c3)

Description

The current implementation of the peer-to-peer networking module relies on `libp2p` version `0.43.0`, which has been identified as vulnerable due to its lack of resource management, as detailed in this advisory and in libp2p's Github issue page. The vulnerability allows any remote peer to exploit resource exhaustion, potentially causing a victim node to run out of memory. This vulnerability is of critical severity, as executing the attack requires no special privileges or user interaction, and is not mitigated by fee and gas mechanisms since it operates at the network level, below the application layer. This allows for potential disruption across the entire blockchain network, thereby significantly amplifying the impact of the exploit.

Furthermore, it's important to note that earlier versions of `libp2p` utilize `owning_ref`, which itself has unresolved soundness issues. These issues have not been addressed in the affected versions. Subsequent versions of `libp2p` have removed the usage of `owning_ref` to mitigate the associated security concerns.

Proof of Concept

Several methods can be employed to carry out the attack. One option is for a malicious node to open new streams on a single connection using a stream multiplexer. However, this attack can be mitigated by setting strict per-connection stream and connection limits.

Another option involves sending partial payloads on various protocol levels, forcing the victim node to buffer the partial payloads for a period of time. The attacker can also trick the victim into pre-allocating buffers for messages that are never sent, consuming additional resources.

These attacks cause the victim node to allocate numerous small memory chunks, which can ultimately lead to the victim's process running out of memory and being terminated by the operating system. When executed continuously, this type of attack can result in a denial of service, particularly when targeting multiple nodes within a `libp2p` based network.

Recommendation

To address these concerns and maintain a secure codebase, as recommended by `libp2p` developers, we strongly recommend to update the dependency of `pchain_network` to utilize the latest version of `libp2p`. The issue with resource management has been fixed since version `0.45.1`. However, it is advisable to use the most recent version available, as it also eliminates the security issues generated by the usage of `owning_ref`. By updating to the latest version, you ensure that your codebase benefits from the most recent bug fixes, improvements, and security enhancements.

To avoid using vulnerable or unmaintained dependencies, incorporate the usage of `cargo audit` into your development workflow. This tool checks for any new vulnerabilities or outdated packages in your project's dependencies. Performing regular checks using `cargo audit` helps you stay updated on potential security issues and enables you to address them promptly.

Additionally, we recommend following best practices by consulting the DoS Mitigation page. This resource provides valuable information on how to incorporate effective mitigation strategies, monitor your application's behavior, and respond to potential attacks.

By updating the `libp2p` dependency, regularly auditing your dependencies with `cargo audit`, and following the best practices outlined in the DoS Mitigation page, you take proactive measures to strengthen the security of your codebase and protect your network from potential vulnerabilities and attacks.

Unchecked Broadcasted Transactions

Broadcasted transactions are unchecked upon reception, potentially allowing a Byzantine node to cause denial of service and network congestion.

ID	PCN-020
Scope	Transactions
Severity	HIGH
Vulnerability Type	Data Validation
Status	Fixed (f3caf2d)

Description

When transactions are broadcasted within the network, no validation checks are performed at reception. However, when transactions are sent from an external account through the RPC API, several checks are conducted before adding them to the mempool and broadcasting them.
The `submit_transaction` function, executes a series of tests to ensure that transactions meet certain criteria before being accepted. Here is the function in question:
src/rpc/transaction.rs:27:

```
pub(crate) async fn submit_transaction(
    request: SubmitTransactionRequest,
    tx_size: usize,
    sync_mempool: SyncMempool<NonceStore>,
) -> SubmitTransactionResponse {
    let tx = request.transaction;
    // Ensure the transaction has a minimum gas limit higher than the minimum cost for the transaction.
    if tx.gas_limit < pchain_runtime::gas::tx_inclusion_cost(tx_size, tx.commands.len()) {
        return SubmitTransactionResponse { error: Some(SubmitTransactionError::Other)}
    }
    // Ensure the transaction doesn't exceed the current block gas limit.
```

```
    if tx.gas_limit > BLOCK_GAS_LIMIT as u64{
        return SubmitTransactionResponse { error: Some(SubmitTransactionError::Other)}
    }
    // Ensure the transaction base fee is greater than or equal to the minimum
    if tx.max_base_fee_per_gas < MIN_BASE_FEE{
        return SubmitTransactionResponse { error: Some(SubmitTransactionError::Other)}
    }
    // Ensure the transaction size is smaller than the block size limit
    if tx_size > BLOCK_SIZE_LIMIT {
        return SubmitTransactionResponse { error: Some(SubmitTransactionError::Other)}
    }
    // Ensure the transaction can be converted from pchain_types::blockchain::Transaction
    // to pchain_types::blockchain::Transaction
    if tx.is_cryptographically_correct().is_err(){
        return SubmitTransactionResponse { error: Some(SubmitTransactionError::Other)}
    }
    // Return error if mempool is full or wrong nonce is provided
    if let Err(error) = sync_mempool.insert(tx) {
        return SubmitTransactionResponse {
            error: Some(match error {
                InsertTxError::NonceTooLow => SubmitTransactionError::UnacceptableNonce,
                InsertTxError::MempoolFull => SubmitTransactionError::MempoolFull,
            }),
        };
    }
    SubmitTransactionResponse { error: None }
}
```

The cryptographic correctness of a transaction is verified within another function, `is_cryptographically_correct`, defined in the `pchain-types` crate:
src/blockchain.rs:145:

```
/// Check whether the Transaction's:
/// 1. Signer is a valid Ed25519 public key.
/// 2. Signature is a valid Ed25519 signature.
/// 3. Signature is produced by the signer over the intermediate transaction.
/// 4. Hash is the SHA256 hash over the signature.
pub fn is_cryptographically_correct(&self) -> Result<(), CryptographicallyIncorrectTransactionError> {
    // 1.
    let public_key = PublicKey::from_bytes(&self.signer)
        .map_err(|_| CryptographicallyIncorrectTransactionError::InvalidSigner)?;
    // 2.
    let signature = ed25519_dalek::Signature::from_bytes(&self.signature)
        .map_err(|_| CryptographicallyIncorrectTransactionError::InvalidSignature)?;
    // 3.
    let signed_msg = {
        let intermediate_txn = Transaction {
            signature: [0u8; 64],
            hash: [0u8; 32],
            ..self.to_owned()
        };
        Serializable::serialize(&intermediate_txn)
    };
    public_key.verify(&signed_msg, &signature).map_err(|_| CryptographicallyIncorrectTransactionError::WrongSignature)?
    // 4.
    if self.hash != sha256(ed25519_dalek::ed25519::signature::Signature::as_bytes(&signature)) {
        return Err(CryptographicallyIncorrectTransactionError::WrongHash)
    }
    Ok(())
}
```

These checks include validations for minimum gas limit, block gas limit, base fee, block size limit, and cryptographic correctness. Unfortunately, these checks are not applied to transactions when they are received as a broadcast from other nodes. This omission allows malformed or incorrect transactions to be inserted into the mempool.
Even though further checks may occur later in the execution of the transaction, the lack of immediate validation creates a vulnerability.

Recommendation

To resolve the vulnerability identified, it is essential to introduce the same series of validations applied in the `submit_transaction` function to transactions when they are received through broadcast. This includes:

- **Minimum Gas Limit Check:** Ensure the transaction has a minimum gas limit higher than the required minimum for the transaction size and commands.
- **Block Gas Limit Check:** Ensure the transaction doesn't exceed the current block gas limit.
- **Minimum Base Fee Check:** Ensure the transaction's base fee is at or above the minimum required.
- **Transaction Size Check:** Ensure the transaction size is smaller than the block size limit.
- **Cryptographic Correctness Check:** Utilize the cryptographic checks performed by the `is_cryptographically_correct` function, ensuring the validity of the signature, hash, and other cryptographic elements.
- **Mempool Constraints Check:** Return appropriate errors if the mempool is full or if the nonce provided is too low.

By implementing these checks at the reception stage for broadcasted transactions, the network can prevent malformed or incorrect transactions from being inserted into the mempool. This approach aligns the handling of broadcasted transactions with the checks performed on transactions sent from external accounts through the RPC API, thereby enhancing the overall integrity and robustness of the transaction processing system.

Execution of Unauthorized Command in Epoch Transactions

This issue concerns the absence of checks during the execution of an epoch transaction to ascertain whether all commands within the transaction are indeed accepted by the corresponding signer. When combined with the vulnerability described in PCN-015, it introduces the possibility for a node to execute a command for which it lacks the authorization.

ID	PCN-016
Scope	Engine
Severity	**MEDIUM**
Vulnerability Type	Logic Error
Status	Fixed (f3caf2d)

Description

Expanding on the details outlined in PCN-015, the validator responsible for creating an epoch transaction possesses the capability to execute a command other than `NextEpoch`. This transaction, upon validation by other validators, undergoes simulation within the `execute_epoch_transaction` method. Notably, the checks conducted by this method deviate slightly from those in `execute_transactions`. Specifically, there is a lack of scrutiny on whether all commands within the transaction are sanctioned for the signer executing it.

Although the absence of these checks may not inherently present an issue when epoch transactions are ensured to exclusively feature the `NextEpoch` command, the lack of such assurance creates an avenue for validators to execute commands, even in scenarios where the configurations of other nodes explicitly prohibit the acceptance of those commands from the respective signer.

Recommendation

The recommended course of action mirrors the advice provided in the preceding issue. During block validation, comprehensive checks must be implemented to ensure that an epoch transaction solely contains the `NextEpoch` command.

This measure will not only address the current concern but also contribute to the broader problem of inappropriate commands within epoch blocks.

Runtime Panic Due to Untimely `NextEpoch` Transaction

A runtime panic occurs when a transaction containing a single `NextEpoch` command is submitted by a whitelisted signer.

ID	PCN-013
Scope	Runtime
Severity	**MEDIUM**
Vulnerability Type	Error Handling
Status	Fixed (996cd5d and f3caf2d)

Description

The vulnerability originates from the usage of the `unwrap()` within the `next_epoch` function of the runtime. The specific line in question is:

pchain-runtime/src/execution/protocol.rs:34:

```
let block_performance = state.bd.validator_performance.clone().unwrap();
```

Analyzing how the fullnode generates blocks, we observe that the `produce_block` implementation incorporates distinct logic for situations where the current block is intended to be the epoch block. Specifically, if the block is an epoch, a new epoch transaction is crafted. Conversely, if the block is not an epoch, the subsequent transaction is executed from the mempool. Nonetheless, it remains straightforward to generate and present a transaction containing a lone `NextEpoch` command. This transaction will eventually undergo processing via `execute_from_mempool`.

`execute_from_mempool` relies on `params_from_blockchain`, obtained from the `produce_block` function:

```
let params_from_blockchain = BlockchainParams {
    /* Other fields */
    validator_performance: pacemaker::validator_performance(
        self.kv_store.snapshot(),
        self.config.blocks_per_epoch as u32,
        this_block_number,
        prev_block_hash,
    ),
};
```

The `params_from_blockchain` variable is employed in the `execute_from_mempool` in order to simulate the result of transition function. Significantly, `pacemaker::validator_performance` yields `None` when dealing with non-epoch block numbers. Consequently, when a transaction with a single `NextEpoch` command traverses the runtime's `transition` function alongside `params_from_blockchain`, it triggers a panic due to the attempt to `unwrap()` a `None` value.

The success of this exploit pivots on whether the signer of the transaction is whitelisted for the `NextEpoch` command. Configuration within the nodes' settings allows for the whitelisting of the execution of the `NextEpoch` command:

fullnode/config/config.toml:

```
[[engine.executor.tx_whitelist]]
command = "NextEpoch"
accept = ["malicious_signer_addr"]
```

Conversely, if nodes do not enforce limitations on the execution of the `NextEpoch` command within their configuration, any individual can transmit a transaction that will result in a runtime panic.

The panic is captured by `catch_unwind` within the fullnode, which leads to the exclusion of the transaction from the blockchain. Furthermore, the sender's account evades any deduction of fees for this transaction.

It is important to acknowledge that nodes compiled with the configuration:

```
[profile.release]
panic = "abort"
```

will immediately crash upon encountering the panic.

Recommendation

To mitigate this issue, consider the following steps:

- **Error-Handling Strategy Enhancement:** Implement an advanced error-handling strategy within the runtime's codebase. Replace occurrences of methods that lead to panic, such as `unwrap` and `panic`, with error-returning mechanisms. This transition to returning and appropriately handling error results will contribute to the overall robustness of the system.
- **Refinement of NextEpoch Logic:** Conduct a meticulous review of the `NextEpoch` command's logic within the fullnode. Consider options to either restrict its inclusion in the mempool or confine its addition solely to epoch blocks. This strategic refinement will serve to mitigate the risk of triggering panics in the runtime due to the presence of the `NextEpoch` command.
- **Comprehensive Testing:** Establish a comprehensive testing framework that covers various scenarios involving the `NextEpoch` command and its interactions with the runtime. This should encompass cases where the command is executed within different block types, as well as instances of authorized and unauthorized usage. Rigorous testing can help identify and rectify any remaining vulnerabilities.

Private Key Stored Unencrypted

The node operator's private key is stored without encryption, exposing it to potential unauthorized access and misuse.

ID	PCN-017
Scope	Cryptography
Severity	**LOW**
Vulnerability Type	Insecure Data Storage
Status	Fixed (92b5e2c)

Description

During our security review, it was identified that the node operator's private key is stored in an unencrypted format within a text file. Specifically, the private key is stored in base64 encoding but lacks any form of cryptographic protection. The file in question is named `keypair.json` in default node configuration settings.
The presence of an unprotected private key poses a potential risk, as anyone with access to this file can potentially utilize the private key for unauthorized activities.

If an attacker exploits network vulnerabilities, system/OS flaws, or weaknesses in other software running on the server, they could gain access to the *keypair.json* file containing the unencrypted private key. With this key, they can execute unauthorized transactions on the blockchain and even impersonate the node operator for various malicious activities. This vulnerability not only poses a direct threat but may also erode stakeholders' confidence in the system's security measures.

Recommendation

To remediate this vulnerability, we recommend the following:

1. **Encryption at Boot:** Implement a mechanism to securely request the decryption of the node operator's private key during the node's boot process. The private key should be stored encrypted using strong, industry-recognized cryptographic algorithms such as `AES-256-GCM` or `ChaCha20-Poly1305` within the *keypair.json* file.
2. **Use of Zeroize Crate:** Ensure that any unencrypted representation of the private key in memory is properly scrubbed using tools like the `zeroize` crate, which is designed to securely zero out sensitive data from memory.
3. **Education:** Provide documentation and training materials to node operators, emphasizing the importance of securing their private keys and the potential risks associated with storing keys unencrypted.

Although the vulnerability is rated as low, the fundamental importance of private key security in a blockchain environment cannot be overstated. Given the straightforward nature of the recommended remediations, it is advisable to address this vulnerability in a timely manner to uphold best security practices.

Misaligned Pointer Dereference in Wasmer-VM Crate

A misaligned pointer dereference vulnerability has been identified in the `wasmer-vm` crate affecting some apple device running with ARM64 architecture.

ID	PCN-019
Scope	Runtime/Dependencies
Severity	**LOW**
Vulnerability Type	Misaligned Pointer Dereference
Status	Acknowledged

Description

When running `wasmer` on certain Apple devices with ARM64 architecture, a misaligned pointer dereference occurs when a trap is handled for managing `wasmer` runtime errors. This is due to an incorrect cast towards the `libc::ucontext_t` type, as described in this issue on the `wasmer` repository. This issue triggers a panic with the message:

```
thread panicked at 'misaligned pointer dereference: address must be a multiple of 0x10 but is 0x101a4fc18', /Users/hacke
```

Recommendation

To mitigate this issue, the following actions should be taken:

1. **Upgrade to Version 4.1.1 or Later:** As stated in the changelog, the fix for this misalignment issue has been applied starting from version **4.1.1** of the `wasmer` crate with pull request #4120. Upgrading to this version or a later one will resolve the issue but can introduce breaking changes.
2. **Test on Affected Architectures:** Ensure that any changes are thoroughly tested on the affected Apple devices with ARM64 architecture, in addition to other platforms, to confirm that the fix is effective and does not introduce new issues.
3. **Inform Users about Supported Architectures:** If a decision is made not to patch the issue, it is essential to communicate to users which architectures are supported and which may encounter this problem. Clear documentation on this matter will help users understand the limitations and make informed decisions about their use of the product.

Non-Epoch Commands in Epoch Transaction

This issue pertains to the epoch block's intended purpose of containing transactions with only the `NextEpoch` command. However, it has been observed that the epoch block can unintentionally accommodate transactions with different commands.

ID	PCN-015
Scope	Engine
Severity	**LOW**
Vulnerability Type	Logic Error
Status	Fixed (f3caf2d)

Description

The pivotal process responsible for generating epoch transactions lies within the `epoch_transaction` method:

paralellchain-fullnode/src/engine/executor.rs:228:

```
pub(in crate::engine) fn epoch_transaction(
    &self,
    world_state: &WorldState<SpeculativeStore>,
) -> Transaction {
    let nonce = world_state.nonce(self.public_address());
    Transaction::new(&self.keypair, nonce, vec![Command::NextEpoch], 0, 0, 0)
}
```

This method forms the foundation for creating the necessary transactions utilized by the `produce_block` function. The purpose is to simulate execution outcomes and potential validator changes.

While the `epoch_transaction` method is designed with appropriate logic, a vulnerability arises when malicious nodes exploit the code by creating alternative transactions. This can be accomplished by modifying the code, for instance:

```
let command = Command::SetPoolSettings(SetPoolSettingsInput { commission_rate: 42 });
Transaction::new(&self.keypair, nonce, vec![command], 500000000, 10, 0)
```

Nodes with modified code, when tasked with producing an epoch block, generate a block that incorporates an erroneous transaction. The absence of validation for this scenario leads to the acceptance of such transactions by other nodes, followed by progression to the subsequent non-epoch block.

While this issue may not necessarily lead to direct attack scenarios, it disrupts the blockchain's functioning. It allows a situation in which all nodes skip the transition to the next epoch, thereby bypassing validator changes and reward allocations to stakers.

Additionally, this loophole could potentially assist a node in evading certain restrictions, as epoch transactions undergo a slightly different validation process compared to other transactions. This specific aspect will be addressed in PCN-016.

Recommendation

To rectify this concern, it is advisable to incorporate an additional validation check within the `validate_block` function. This check would ensure that epoch blocks exclusively contain transactions featuring the singular `NextEpoch` command.

Implementing this safeguard guarantees that all nodes can validate this requirement, subsequently only permitting the `NextEpoch` command for epoch blocks.

Potential Panic Due to Unsafe Balance Management

In the `pchain-runtime`, several instances of unsafe arithmetic operations were identified, potentially leading to panics in the code execution.

ID	PCN-009
Scope	Arithmetic
Severity	**LOW**
Vulnerability Type	Integer overflow
Status	Fixed (b6d86f6)

Description

Several unsafe operations, discussed in more detail in PCN-007, pose a risk of integer overflow. While the probability of such overflows is relatively low due to the requirement of a large number of blocks, certain operations related to balance transfers, stake management, and deposits could present a higher potential risk.

Although the occurrence of these panics is impossible with the current total amount of tokens in existence, it is crucial to address these concerns proactively to prevent any future vulnerabilities. The overflows would trigger panics in the runtime, which would be caught using `catch_unwind` in the fullnode. It's important to note that if the node was compiled with `panic = "abort"` in the release profile, the panic would lead to an immediate crash of the node.

The specific operations that have the potential for dangerous overflows are listed below:

- **Transfer balances:**
  - Between accounts: *pchain-runtime/src/execution/account.rs:88*:

    ```
    state.set_balance(recipient, recipient_balance + amount);
    ```

  - When account calls a smart contract:
    pchain-runtime/src/execution/account.rs:120:

    ```
    state.set_balance(target, target_balance + amount);
    ```

  - When invoking a contract from another contract:
    pchain-runtime/src/execution/internal.rs:59:

    ```
    let to_address_new_balance = to_address_prev_balance + value;
    ```

  - From a contract:
    pchain-runtime/src/execution/internal.rs:138:

    ```
    let to_address_new_balance = to_address_prev_balance + amount;
    ```

- **Deposit management:**
  - When topping up a deposit:
    pchain-runtime/src/execution/staking.rs:243:

    ```
    deposits.set_balance(deposit_balance + amount);
    ```

- When withdrawing a deposit:
  pchain-runtime/src/execution/staking.rs:309:

```
state.set_balance(owner, owner_balance + deposit_balance - new_deposit_balance);
```

- When increasing stake power:
  pchain-runtime/src/execution/staking.rs:567, 576:

```
power: stake_power + stake_power_to_increase,
```

Recommendation

To safeguard against potential Denial-of-Service (DoS) attacks resulting from arithmetic overflow in the runtime, it is essential to utilize the Rust Standard Library's built-in methods, as recommended in PCN-007, to prevent arithmetic overflows and panics in the runtime.

You should implement robust error handling and graceful recovery for potential overflow cases. Transactions involving these operations should be marked as "failed".

Taking these precautions will enhance the security and stability of the application, reducing the risk of potential DoS attacks caused by unsafe balance management. It is crucial to prioritize addressing these issues promptly to prevent any potential vulnerabilities in the codebase.

Unrestricted `SetPoolSettings` Transactions Despite Whitelist Configuration

Despite configuring the node to whitelist the `SetPoolSettings` command for specific accepted signers, a vulnerability exists wherein the command remains unrestricted, allowing any signer to submit transactions with this command.

ID	PCN-014
Scope	Engine
Severity	LOW
Vulnerability Type	Logic Error
Status	Fixed (f3caf2d)

Description

The ability to whitelist any command is provided for the purpose of limiting transaction submission rights exclusively to specified signers. This configuration is executed in the following manner:

fullnode/config/config.toml:

```
[[engine.executor.tx_whitelist]]
command = "SetPoolSettings"
accept = ["addr1", "addr2", "addr3"]
```

However, even in scenarios where nodes are configured as aforementioned, the flaw lies in the implementation of the `is_whitelisted_command` method, due to a typographical error:

paralellchain-fullnode/src/engine/executor.rs:366:

```
Command::SetPoolSettings { .. } => self.command.as_str() == "setpoolpettings",
```

The inadvertent typo of `"setpoolpettings"` instead of the correct `"setpoolsettings"` within the aforementioned line results in the `is_whitelisted_command` method constantly returning `false` for the `SetPoolSettings` command. Consequently, the method `is_not_accepted` also returns `false`. This misconfiguration negates the intended purpose of the command as a whitelisted item, allowing the creation of `SetPoolSettings` transactions without proper restriction.

Recommendation

To rectify this issue, it is important to correct the typographical error as follows:

```
Command::SetPoolSettings { .. } => self.command.as_str() == "setpoolsettings",
```

Additionally, it is advised to rigorously test all functionalities to proactively identify and resolve similar errors, ensuring the integrity of the system's security mechanisms.

Unsoundness Issue in Borsh Dependency

An unsoundness issue has been discovered in the Borsh dependency.

ID	PCN-012
Scope	Dependencies
Severity	**LOW**
Vulnerability Type	Undefined Behavior
Status	Fixed (8903746, b6d86f6, fcc6b4b, 7275672)

Description

The borsh crate (version 0.10.2) used by Parallelchain node and its dependencies 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:   0.10.2
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
```

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 parallelchain node codebase and the associated crates 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 parallelchain node and its dependencies does not expose it to the associated risks.

Worth noting is that within the *Cargo.toml* of the Parallelchain node and its associated crates (runtime, worldstate, and types), the borsh version remains affixed to `0.10.2`.

Recommendation

As of the latest information, the aforementioned issue in the borsh crate, specifically described in near/borsh-rs#19, has been addressed and merged into the master branch with the pull request near/borsh-rs#145. However, it's worth noting that, as of the time of writing this report, the fix has not been included in a stable release, as mentioned in the comments of the pull request.

This means that, while a resolution exists for the unsoundness problem, the fix is not readily available for direct consumption through traditional dependency update mechanisms. Organizations relying on the borsh crate should be cognizant of this delay and plan their integration strategies accordingly.

For the way forward, we propose:

1. **Monitoring Borsh Crate Iterations**: Regularly check the borsh crate's release updates, focusing on versions succeeding `0.10.2`. This ensures the integration of the fix as soon as it's rolled out in a stable release.
2. **Strengthen Internal Audits**: Maintain rigorous internal code audits to ascertain that ZSTs, which bypass the `Copy` or `Clone` traits yet lean on `BorshSerialize`/`BorshDeserialize`, aren't inadvertently introduced. This preemptive measure will cushion against potential risks while we await the borsh fix.
3. **Consider Version Unpinning**: Reevaluate the strategy of pinning the borsh version at `0.10.2`. By adjusting this, the team can seamlessly tap into the latest patches and fixes borsh offers.

By adhering to these strategies and fostering a keen awareness around dependency management, coupled with internal coding best practices, the team can maximize the borsh crate's capabilities without compromising the system's integrity.

Compilation error due to dependency hotstuff_rs

The FullNode repository fails to build.

ID	PCN-011
Scope	Build Process
Status	Fixed (b24dcc0)

Description

The FullNode repository, an implementation of the Parallelchain node, fails to build.

The `cargo build --release` command fails with the following error:

```
   Compiling fullnode v0.4.2 (/Users/hacken/Projects/L1Parallelchain/paralellchain-fullnode)
error[E0599]: no method named `set_highest_entered_view` found for struct `BlockTree` in the current scope
   --> src/main.rs:223:24
    |
223 |             block_tree.set_highest_entered_view(view);
    |
```

The missing method is from the `hotstuff_rs` library, and it was renamed from `set_highest_entered_view` to `set_highest_view_entered` in version 0.2.1.

This breaks the build chain, because the `hotstuff_rs` version is set to **0.2.0** in the *Cargo.toml* file, which allows for any versions `0.2.x` following the cargo versioning system.

Recommendation

This issue arises due to a dependency on a specific version of the `hotstuff_rs` library that has undergone a breaking change. Here are some measures that can mitigate and prevent such issues in the future:

1. **Dependency Pinning:** Pin the `hotstuff_rs` library to the specific version that your code is compatible with. To do this, instead of `hotstuff_rs = "0.2.0"` which allows for any `0.2.x` versions, use `hotstuff_rs = "=0.2.0"`. This ensures that your code always builds with the correct version of the dependency, eliminating build failures due to incompatible library updates.

2. **Dependency Updation:** Regularly update dependencies to their latest versions and accordingly modify your codebase. This not only prevents potential security issues but also allows you to take advantage of the latest features and improvements. In this case, it would mean updating the FullNode repository to use the `set_highest_view_entered` method from the `hotstuff_rs` library.

3. **Semantic Versioning:** Enforce strict semantic versioning (semver) for the `hotstuff_rs` library and all other dependencies. For `0.x.x` versions, increment the minor version in the event of potentially breaking changes. For versions `1.x.x` and beyond, increment the major version.

4. **Continuous Integration:** Implement a robust Continuous Integration (CI) pipeline that includes building the application with every change that is pushed to the repository. This helps to identify such build errors at an early stage.

By following these steps, the risk of build failures due to changes in dependencies can be minimized.

Correct transactions can be dropped

Transactions sent in a batch with a small time gap between them are being dropped.

ID	PCN-006
Scope	Mempool / Nonce Synchronization
Status	Acknowledged

Description

During testing of transaction behavior, it was observed that if a sender submits a batch of transactions with a small time gap between them (approximately 6 seconds in our test), starting from a certain transaction, all subsequent transactions are dropped with a status code indicating an `Nonce inaccesible` error.

The cause of this behavior can be traced to the implementation of the `insert` function in the `Mempool<N: NonceProvider>`:

```
if let Some(txns) = self.txns.get_mut(&txn.signer) {
    /* Logic for the case when there already are some transactions */
} else {
    // There are no Transactions in the Mempool that come from the signing Account.
    if txn.nonce == self.committed_nonces.get(txn.signer) {
        /* Logic for adding the transaction to the mempool if it contains the correct nonce */
        Ok(())
    } else {
        Err(UnacceptableNonceError::Inaccessible)
    }
}
```

This logic may not work correctly if all previous transactions from the same signer were removed from the mempool, but the nonce has not yet been updated.

Transactions are popped out of the mempool during the execution of the `execute_from_mempool` function, which is called to produce a block. As a result, the list of transactions in the mempool from the signer can be emptied as soon as blocks with these transactions are produced.

However, the nonce, calculated using `self.committed_nonces.get(txn.signer)`, is only changed after the block is committed.

The `self.committed_nonces` variable has the type:

```
pub(crate) struct NonceStore(pub BlockTreeDB);
```

and the calculation of the nonce in the `get` method relies on the internal `BlockTreeDB`.

This `BlockTreeDB` is initialized when the node starts:

```
let block_tree_db = block_tree::open(&block_tree_config);
```

Later, it is cloned and asynchronously updated in multiple threads. It is also used for the HotStuff consensus algorithm:

```
// start consensus engine
let _replica = Replica::start(
    engine,
    my_keypair,
    node_network,
    block_tree_db.clone(),
    engine::pacemaker::PaceMaker::new(pacemaker_config),
);
```

It is important to note that the `BlockTreeDB` uses a reference-counting pointer to store the database, allowing HotStuff to modify the BlockTree stored by this pointer.

In the HotStuff implementation, when a proposal is received and `validate_block()` returns a correct block, the block is inserted into the BlockTree:

```
let validator_set_updates_because_of_commit = block_tree.insert_block(
    &proposal.block,
    app_state_updates.as_ref(),
    validator_set_updates.as_ref(),
);
```

This insertion ultimately leads to a change in the nonce.

While this behavior does not pose an immediate and serious security risk, dropping correct transactions that should have been included in the block is considered a design flaw. It is inconvenient and counterintuitive.

However, there is a potential security risk if a third party gains knowledge about the dropped transaction. If the nonce, signature, and details of the commands are known, this third party can resend the transaction. The transaction would be accepted because it contains correct data and a valid signature, even though the original sender did not resend the transaction and may not want it to be processed again. This consideration, combined with PCN-007, opens a window for potential misbehavior, leading to a security issue.

Recommendation

It is imperative to address the nonce synchronization issue among all relevant modules, including the mempool, Worldstate, and consensus. Resolving this discrepancy may necessitate a change in the software architecture, potentially adding to its complexity. Proper evaluation and meticulous planning will be essential to ensure that while making these changes, the system remains robust, efficient, and easy to maintain.

Global Mutable Static Variable within Unsafe Block in `rpc::state::view` Function

Global mutable static variable `STATIC_BLOCKTREEDB` used within an unsafe block is non-idiomatic Rust, posing potential challenges in future development.

ID	PCN-010
Scope	RPC State View
Status	Acknowledged

Description

The `view` function within the RPC State component employs a global mutable static variable `STATIC_BLOCKTREEDB` within an `unsafe` block. This method of programming is not idiomatic Rust and bypasses the safety features that Rust provides.

The function in question is as follows, found in *src/rpc/state.rs:251*:

```
pub(crate) unsafe fn view(
    request: ViewRequest,
    state_hash: CryptoHash,
    block_tree_db: BlockTreeDB,
    sc_cache: &Option<PathBuf>,
    gas_limit: u64,
) -> ViewResponse {
    static mut STATIC_BLOCKTREEDB: MaybeUninit<BlockTreeDB> = MaybeUninit::uninit();
    let block_tree_snapshot = unsafe {
        STATIC_BLOCKTREEDB.as_mut_ptr().write(block_tree_db); // update block_tree_db
        (*STATIC_BLOCKTREEDB.as_ptr()).snapshot() // snapshot inherits static life time from the caller (STATIC_BLOCKTRI
    };
    // remaining code
}
```

Currently, `STATIC_BLOCKTREEDB` is only accessed within the context of a single thread, avoiding potential race conditions. However, this usage of a global mutable static variable within an `unsafe` block can lead to potential maintenance difficulties, or create bugs that are hard to trace in the future.

The use of `unsafe` should be minimized in Rust code. In cases where it's necessary, adequate justification and explanation should be provided. In this particular case, there are safer alternatives available instead of using a mutable static variable, such as passing the `block_tree_db` directly to the `view` function.

Recommendation

Refactoring the code to pass `block_tree_db` directly to the `view` function would eliminate the need for the `unsafe` block and the mutable static variable `STATIC_BLOCKTREEDB`. This would simplify the code, improve readability, and maintain Rust's safety guarantees.

When shared state or mutability is required, Rust provides safer alternatives such as Mutex or RwLock, which are preferred over `unsafe` constructs. Utilizing these safe constructs can mitigate the risk of race conditions and undefined behaviors.

When `unsafe` blocks and mutable statics are deemed necessary, they should be accompanied by a detailed explanation, justifying their usage and explaining why they're considered safe. This would provide important context to reviewers and future maintainers of the code.

As part of the ongoing code quality assurance process, it is recommended to include specific tests for concurrency and mutation. These tests can help prevent the introduction of data races or undefined behaviors as the codebase evolves.

Inconsistent Code Formatting

A `cargo fmt` check reveals inconsistent formatting in codebase and all crates in scope of the audit.

ID	PCN-003
Scope	Code Formatting
Status	Acknowledged

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 of the ParallelChain node and all crates within the scope of the audit to ensure that all code adheres to the standard Rust formatting conventions. 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.

Linter Warnings

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

ID	PCN-002
Scope	Linters
Status	Acknowledged

Description

The codebase has been analyzed using `cargo clippy`, which has generated multiple warnings related to the Fullnode and Network crates. These warnings highlight potential issues in the code that should be addressed to improve overall code quality. The warnings can indicate various problems, including unoptimized or inefficient code, non-idiomatic Rust patterns, redundant or unnecessary code, and potential coding errors or logic flaws.

The specific warnings generated by `cargo clippy` are as follows:

Fullnode

9 warnings were generated:

- module_inception
- 4 occurrences of needless_borrow
- enum_variant_names
- useless_conversion
- unnecessary_cast
- expect_fun_call

Network

The Network crate generated one warning:

- new_without_default

On the other hand, the code of the Runtime, World State and Types crates did not generate any clippy warnings.

Ignoring these warnings may result in a codebase that is harder to maintain, potentially leading to performance issues and even security vulnerabilities. To ensure a high-quality and maintainable codebase, it is essential to address all the warnings generated by `cargo clippy`.

It is worth noting that `cargo clippy` is currently configured to generate warnings primarily for the default set of lints. However, there might be additional issues that can be discovered by enabling and carefully examining supplementary lints. These potential issues will be systematically addressed in separate future issues.

Recommendation

To maintain a high-quality and easily maintainable codebase, it is crucial to address all the warnings generated by `cargo clippy`. This can be achieved by following these steps:

- Address all warnings generated by `cargo clippy`.
- Apply appropriate code changes following Rust's best practices.
- Regularly run cargo clippy to catch new warnings and maintain code quality.

By proactively addressing linter warnings, you improve the overall code quality and foster adherence to Rust's best practices, leading to a more robust and secure project.

RPC submit_transaction Inadequate Logging Detail

Insufficient detail provided upon transaction submission failure.

ID	PCN-005
Scope	RPC Auditing and Logging
Status	Acknowledged

Description

The `submit_transaction` handler of the Transaction-related API does not provide adequate information in the event of a transaction failure. The current level of detail in logging can pose a challenge for issue identification and rectification.

The associated function in question is presented below in full node crate:
src/rpc/transaction.rs:27:

```
pub(crate) async fn submit_transaction(
    request: SubmitTransactionRequest,
    tx_size: usize,
    sync_mempool: SyncMempool<NonceStore>,
) -> SubmitTransactionResponse {
    let tx = request.transaction;
    // Ensure the transaction has a minimum gas limit higher than the minimum cost for the transaction.
    if tx.gas_limit < pchain_runtime::gas::tx_inclusion_cost(tx_size, tx.commands.len()) {
        return SubmitTransactionResponse { error: Some(SubmitTransactionError::Other)}
    }
    // Ensure the transaction doesn't exceed the current block gas limit.
    if tx.gas_limit > BLOCK_GAS_LIMIT as u64{
        return SubmitTransactionResponse { error: Some(SubmitTransactionError::Other)}
    }
    // Ensure the transaction base fee is greater than or equal to the minimum
    if tx.max_base_fee_per_gas < MIN_BASE_FEE{
        return SubmitTransactionResponse { error: Some(SubmitTransactionError::Other)}
    }
    // Ensure the transaction size is smaller than the block size limit
    if tx_size > BLOCK_SIZE_LIMIT {
        return SubmitTransactionResponse { error: Some(SubmitTransactionError::Other)}
```

```
    }
    // Ensure the transaction can be converted from pchain_types::blockchain::Transaction
    // to pchain_types::blockchain::Transaction
    if tx.is_cryptographically_correct().is_err(){
        return SubmitTransactionResponse { error: Some(SubmitTransactionError::Other)}
    }
    // Return error if mempool is full or wrong nonce is provided
    if let Err(error) = sync_mempool.insert(tx) {
        return SubmitTransactionResponse {
            error: Some(match error {
                InsertTxError::NonceTooLow => SubmitTransactionError::UnacceptableNonce,
                InsertTxError::MempoolFull => SubmitTransactionError::MempoolFull,
            }),
        };
    }
    SubmitTransactionResponse { error: None }
}
```

The `SubmitTransactionError::Other` error is returned in various instances. This practice lacks specificity and can complicate troubleshooting efforts.

Recommendation

We recommend improving the granularity of the `SubmitTransactionError` enum in the `pchain-types` crate. This can be achieved by assigning unique and descriptive identifiers to each potential error condition. Here's an updated version of the `SubmitTransactionError` enum with new error variants:

```
#[derive(Debug, Clone, BorshSerialize, BorshDeserialize)]
pub enum SubmitTransactionError {
    UnacceptableNonce,
    MempoolFull,
    GasLimitTooLow,
    GasLimitTooHigh,
    BaseFeeTooLow,
    TransactionSizeTooLarge,
    FailedCryptographicCheck,
    Other,
}
```

With these modifications, each unique error in the `submit_transaction` function will return a distinct and descriptive error variant, enhancing error interpretability, and contributing to more efficient problem identification and resolution.

std::panic::catch_unwind() usage

Usage of `std::panic::catch_unwind()` as a general try/catch mechanism.

ID	PCN-018
Scope	Error Handling
Status	Acknowledged

Description

All calls to `state_transition::simulate` for the execution of transactions are made within a `std::panic::catch_unwind` block. This usage is indicative of a code quality issue, as it may lead to improper error handling, potentially masking underlying issues or bugs. Notably, this catch-all error handling made it challenging to uncover issue PCN-008 that leads to a runtime panic.

Recommendation

1. **Replace with Specific Error Handling:** Instead of using `std::panic::catch_unwind` as a catch-all mechanism, identify the specific exceptions or errors that might occur, and handle them explicitly using Rust's standard error handling techniques.
2. **Utilize Proper Logging:** Implement comprehensive logging that captures errors and exceptions, facilitating effective monitoring, troubleshooting, and analysis.
3. **Adopt Code Quality Standards:** Encourage adherence to best practices for error handling, which can lead to a more robust and maintainable codebase.

While this issue may not pose a direct threat to the security of the system, it reflects an area where code quality can be improved. Addressing it will contribute to more transparent and proper error handling, enhancing the overall maintainability and integrity of the software.

Test coverage

The current test coverage of the project is commendable, adequately addressing the critical sections. Nevertheless, there are opportunities for minor refinements in specific areas to meet the industry recommended standards.

ID	PCN-004
Scope	Code Quality / Testing
Status	Fixed

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 crate and file, including the number of lines covered and the percentage of coverage.

Full Node

Covered: 671 of 919 (73.01%)

File / Directory	Coverage
engine	558 / 697 (80.06%)
rpc	269 / 467 (57.60%)
server	194 / 228 (85.09%)
storage	99 / 114 (86.84%)
config.rs	77 / 82 (93.90%)
consensus.rs	15 / 24 (62.50%)
genesis.rs	38 / 40 (95.00%)
main.rs	92 / 131 (70.23%)
mempool.rs	86 / 103 (83.50%)
network.rs	27 / 32 (84.38%)

The coverage statistics indicate that the component `rpc`, file `consensus.rs` and file `main.rs` currently have lower test coverage.

pchain-runtime

Covered: 1885 of 2252 (83.70%)

File / Directory	Coverage
contract	407 / 503 (80.91%)
execution	860 / 939 (91.59%)
wasmer	211 / 376 (56.12%)
cost.rs	19 / 19 (100.00%)
error.rs	12 / 15 (80.00%)
formulas.rs	5 / 6 (83.33%)
gas.rs	87 / 97 (89.69%)
read_write_set.rs	189 / 201 (94.03%)
transition.rs	84 / 85 (98.82%)
types.rs	11 / 11 (100.00%)

The coverage statistics indicate that the component `wasmer` currently has lower test coverage, particularly `non_determinism_filter.rs` has a test coverage of just 17.49%.

pchain-world-state

Covered: 671 of 919 (73.01%)

File / Directory	Coverage
network	332 / 406 (81.77%)
error.rs	3 / 7 (42.86%)
keys.rs	33 / 54 (61.11%)
states.rs	151 / 284 (53.17%)
storage.rs	71 / 72 (98.61%)
trie.rs	81 / 96 (84.38%)

The coverage statistics indicate that the files `error.rs`, `keys.rs`, and `states.rs` currently have lower test coverage.

pchain-types-rust

No test coverage.

pchain-network

No test coverage.

Recommendation

While the current test coverage is commendable, there's always room for enhancement. We recommend introducing tests for `pchain-network` and `pchain-types` crates, and further improving the test coverage in other crates to strive towards the industry-recommended benchmark of **80%** coverage.

Unsafe arithmetics

Multiple instances of unsafe arithmetic operations were discovered in the codebase. These unchecked operations have the potential to cause unpredictable and potentially harmful side effects in your application, particularly concerning arithmetic overflows.

ID	PCN-007
Scope	Arithmetic
Status	Acknowledged

Description

Arithmetic operations that are not properly safeguarded could lead to critical errors such as overflows, which in some cases During the code audit, multiple instances of unsafe arithmetic operations were identified in the codebase. These unchecked operations have the potential to cause unpredictable and potentially harmful side effects, such as arithmetic overflows, in your application. These issues can lead to critical errors, crashes, and even security vulnerabilities.

While certain calculations might seem unlikely to trigger an overflow due to their high threshold, others may pose a more imminent risk. To thoroughly identify all instances of unsafe arithmetic operations in your codebase, you can execute the following command:

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

When you run this command on different parts of the project, it generates several warnings related to the use of potentially unsafe arithmetic:

- 63 warnings from pchain-runtime
- 23 warnings from paralellchain-fullnode
- 19 warnings from pchain-world-state
- 2 from pchain-network

Note that pchain-types-rust does not exhibit any issues with unsafe arithmetic.

Additionally, within `paralellchain-fullnode/src/engine/engine.rs`, three instances of `checked_add` immediately followed by `unwrap` have been identified. The use of safe arithmetic methods is rendered ineffective by unwrapping the result.

Some of these instances may have a more significant impact and pose security risks, which will be addressed in the subsequent issue.

Recommendation

To mitigate potential vulnerabilities in your codebase, it is strongly recommended to use 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 arithmetic methods will help you effectively manage the potential risks associated with arithmetic overflows and maintain a more secure and stable application.

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.)