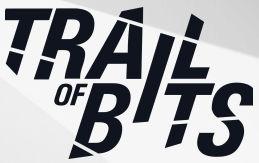

ChromaWay
Consensus & Directory Chain

Security Assessment

June 18, 2024

Prepared for:
August Botsford and Ludvig Oberg
ChromaWay AB

Prepared by: **Benjamin Samuels, Simone Monica, and Sam Alws**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Benjamin Samuels, Consultant
benjamin.samuels@trailofbits.com

Simone Monica, Consultant
simone.monica@trailofbits.com

Sam Alws, Consultant
sam.alws@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
April 25, 2024	Pre-project kickoff call
May 6, 2024	Status update meeting #1
May 13, 2024	Status update meeting #2
May 17, 2024	Status update meeting #3
May 28, 2024	Status update meeting #4
June 3, 2024	Status update meeting #5
June 10, 2024	Delivery of report draft
June 10, 2024	Report readout meeting
June 18, 2024	Delivery of comprehensive report

Executive Summary

Engagement Overview

ChromaWay engaged Trail of Bits to review the security of multiple components of the ChromaWay blockchain. This includes ChromaWay's EBFT consensus system, its networking stack, and its Directory Chain implementation.

A team of three consultants conducted the review from April 29 to June 7, 2024, for a total of 13 engineer-weeks of effort. Our testing efforts focused on the properties of the system's consensus, denial-of-service (DoS) attacks against the networking stack, and various malicious attacks against the system's Directory Chain. With full access to source code and documentation, we performed static and dynamic testing of the targets, using automated and manual processes.

Observations and Impact

EBFT component

We found two high-severity issues relating to the EBFT component. The first is a weak subjectivity issue (TOB-CHROMAWAY-2) that could allow nodes to sync to an incorrect chain in the future. The second (TOB-CHROMAWAY-7) is a theoretical consensus issue that affects the network's liveness properties, and is exploitable without the presence of an active attacker.

Networking component

We found one medium-severity issue in the networking component that could lead to a Slowloris-style DoS attack (TOB-CHROMAWAY-8). No other findings were identified for this component.

Directory chain

We found four high-severity issues relating to the directory chain. First, conversions between US dollars and CHR tokens are done incorrectly, resulting in conversions that are incorrect by orders of magnitude (TOB-CHROMAWAY-15). Another issue (TOB-CHROMAWAY-16) allows for action points to be stolen by transferring a negative amount of points to another user. The third issue (TOB-CHROMAWAY-23) allows for a blockchain to be transferred to another chain with permission from only one member of its deployer set. The final issue (TOB-CHROMAWAY-25) allows for a DoS attack in which a dapp provider creates a large number of extra dapp providers.

Lastly, we found a medium-severity issue (TOB-CHROMAWAY-20) that enables a user to avoid paying the upgraded cost of a container. This highlights a pattern that can happen when sending a message and the handler function uses a variable that can be modified by another operation.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that ChromaWay take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of direct remediation or as part of any refactoring that may occur when addressing other recommendations.

- **Improve Kotlin test coverage.** At the time of the engagement, EBFT's unit test coverage was measured at 46% branch coverage, and the networking component was measured at 15% branch coverage. This is relatively low coverage for a protocol with value-at-stake. We recommend at least 60% branch coverage at the absolute minimum, with 70-95% coverage recommended for mature projects. Guidance on how to improve ChromaWay's test coverage is provided in appendix D.

- **Add test coverage measurement to Rell.** At the time of the engagement, Rell does not have a mechanism to measure test coverage, making it much harder to objectively measure how comprehensive the Directory Chain's test suite is. Given Rell's operational similarity to a smart contract programming language, we recommend aiming for 90%-100% unit test coverage, the same recommendation we provide for Solidity smart contracts.

- **Migrate to a gossip-based messaging system.** EBFT's use of a poll-based messaging system makes it much easier for entities to attack its consensus using balancing and timing attacks, as described in TOB-CHROMAWAY-7. We recommend migrating to libp2p to allow validators to gossip about the states of other validators in the network, drastically increasing the complexity of balancing attacks. Migration to libp2p would also encrypt ChromaWay's gossip.

- **Document centralization risks.** ChromaWay's documentation should include a page about the system's centralization risks, going into detail about what the system providers can and cannot do using proposals and discussing the economy chain's admin user.

- **Develop a standard event logging system.** Dapp developers should have an interface to easily access the log messages being produced by their dapps. ChromaWay developers could use this same system to more easily view logs produced by the directory chain.

- **Add fuzz testing to Rell.** This would greatly improve the quality and coverage of the directory chain's test code. Property-based fuzzing is considered the de-facto standard for testing smart contract languages like Solidity, and lacking this tool may put Rell at a disadvantage compared to other blockchains.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	6
Medium	6
Low	5
Informational	6
Undetermined	3

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Data Validation	8
Denial of Service	3
Timing	1
Undefined Behavior	13

Project Goals

The engagement was scoped to provide a security assessment of the ChromaWay EBFT Consensus and Directory Chain. Specifically, we sought to answer the following non-exhaustive list of questions:

- Do nodes correctly revolt when a revolting consensus is formed?
- Does the poll-based messaging model used in EBFT introduce new attack vectors?
- Do syncing nodes check for consensus properly?
- Do validators correctly discard invalid proposed blocks?
- Are there any synchronicity issues present that may cause validators to stall, or subsequently cause the chain to halt?
- Are there viable DoS attacks against the system's networking stack?
- Is it possible for a proposal to pass without the correct number of legitimate votes?
- Are proposals applied correctly, without unintended side effects?
- Are proposals' preconditions checked correctly?
- Are the rewards correctly computed?
- Are the operations correctly rate limited by action points?
- Is it possible to steal action points?
- Is it possible to underpay or overpay for a lease?

Project Targets

The engagement involved a review and testing of the targets listed below.

ChromaWay EBFT

Repository	https://gitlab.com/chromaway/postchain
Path	`postchain-base/src/main/kotlin/net/postchain/ebft`
Commit Hash	`efdc42fe91738fd70833059be2304ae705b570f8`
Type	Kotlin

ChromaWay Networking

Repository	https://gitlab.com/chromaway/postchain
Path	`postchain-base/src/main/kotlin/net/postchain/network`
Commit Hash	`efdc42fe91738fd70833059be2304ae705b570f8`
Type	Kotlin

ChromaWay Directory Chain

Repository	https://gitlab.com/chromaway/core/directory-chain
Paths	`src/cm_api`
	`src/common`
	`src/common_proposal`
	`src/economy_chain`
	`src/economy_chain_claim_tchr`
	`src/economy_chain_in_directory_chain`
	`src/management_chain_common`
	`src/model`
	`src/proposal`
	`src/proposal_blockchain`
	`src/proposal_blockchain_move`
	`src/proposal_cluster`
	`src/proposal_cluster_anchoring`
	`src/proposal_container`
	`src/proposal_provider`
	`src/proposal_voter_set`
	`src/roles`
	`src/signer_list_update`
Commit Hash	`2f051d48021550e9e60fefbe055143e448828a82`
Type	Rell

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **ChromaWay EBFT:** This component implements the blockchain's consensus algorithm, EBFT. EBFT is a Byzantine fault-tolerant consensus algorithm that uses a poll-based messaging system in lieu of the more common gossip-based system.
    - **Synchronization:** There are two synchronization modes—a fast sync for initial synchronization and a slow sync for read-only nodes to follow the chain head. We reviewed each mode to ensure that it correctly checks EBFT consensus rules, checks for block validity, and checks that blocks are correctly chained together. We also assessed the system's ability to tolerate weak subjectivity attacks and recover from a peer connection failure.
    - **Consensus:** The system's consensus is based on two-round PBFT (practical Byzantine fault tolerance) with instant finality. We compared the consensus against two relatively recent implementations of PBFT (Tendermint and HotStuff) to identify potential weaknesses that may be exclusive to EBFT. We also reviewed the EBFT's state machine and state transition criteria for potential attacks, bugs, and issues that may result in a consensus halt. This review included the "revolt" feature, EBFT's implementation of how to handle PBFT view changes.
    - **Messaging:** EBFT uses a poll-based messaging system, where all validators connect directly to the other validators in the active set and only relay information about their own individual states. We reviewed this system to determine whether poll-based messaging introduces any consensus weaknesses or makes certain consensus attacks easier to execute. We also reviewed this system to identify potential cross-peer polling attacks, where one peer would send an unprompted response to a victim validator.
- **ChromaWay Networking:** This component implements the blockchain's networking stack, which is used to facilitate consensus and communication between master nodes and sidechain containers. It is implemented using Netty, and we reviewed it to ensure that messages are signed and validated properly. This system was also reviewed for potential DoS attacks.

- **ChromaWay Directory Chain:** This component, written in Rell, is responsible for managing the ChromaWay economy and orchestrating the various containers and blockchains involved in the ChromaWay system.
    - **Economy chain:** This component manages the economy part of the system, including staking, rewards, and paying the container's lease. We looked for ways to bypass the staking requirements and still get the rewards. We also analyzed how the rewards are computed and if it is possible to manipulate them to get more. We also assessed whether container leases can be undercharged or overcharged.
    - **Proposals:** Proposals allow users to suggest and vote on administrative actions, such as creating and deleting blockchains and containers. We reviewed the directory chain's proposal system to ensure that proposals could not be passed or rejected without receiving the proper number of legitimate votes. In addition, we reviewed the directory chain's specific implementation of different actions that can be proposed. We mainly checked that these proposals are assigned to the correct voter sets, that they are applied correctly, and that all necessary preconditions are checked before applying them.
    - The directory chain also included some auxiliary/helper code, which we also reviewed.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- Some components within the audit's scope (EBFT, Directory Chain) are critically dependent on other ChromaWay constructions outside of the audit's scope (e.g., ft3, GTV, Postchain Base, Rell Interpreter). Since we did not review these components during this engagement, they may contain issues that critically impact the operation of components in this engagement's scope.
- We did not review the system's various hashing constructions, such as block hash, transaction hash, and other critical hashes, because they were out of scope.
- We did not review the system's various cryptography implementations because they were out of scope. As such, issues related to signature malleability, signatory duplication, signature verification, nonces, and other cryptography-specific attacks were not analyzed.

- The Rell interpreter and language specification were not in the scope of the review. As such, issues related to runtime determinism, correctness, and DoS in the interpreter were not analyzed.
- Block construction, transaction verification, and mempool components were not in the scope of the review. As such, issues related to block verification, transaction verification, mempool admission, DoS, and maximal extractable value (MEV) were not analyzed.
- The Directory Chain implements special authorized operations that may be called only by a block proposer. These operations are highly privileged, and misuse can lead to a loss of funds. We did not review the code that validators use to verify the correctness of calls to authorized operations because it is out of scope.
- The ft4 library, used to handle token ownership and transfers, and used in some cases to handle authentication, was out of scope for this review.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

EBFT Consensus & Networking

Category	Summary	Result
Auditing	The ChromaWay EBFT and Networking components adequately log critical events that occur within their scope.	**Satisfactory**
Authentication / Access Controls	Peer-to-peer consensus messages are adequately authenticated using signatures.	**Satisfactory**
Complexity Management	The EBFT code has some challenges around readability, coupling of data and code, and the overuse of first-class functions. This should be expected to some extent, as Kotlin purposefully intermingles the properties of object-oriented languages with those of functional languages. However, we believe some of these patterns have been a detriment to the complexity management and testability of the codebase. Guidance on how to improve the codebase's complexity management is provided in appendix C and appendix D.	**Moderate**
Cryptography and Key Management	The system's cryptography and key management capabilities were out of the scope of the engagement.	**Not Applicable**
Decentralization	ChromaWay's EBFT implements a PBFT that is robust to scenarios where a malicious actor controls less than ⅓ of the validator set. However, ChromaWay's use of a messaging system where validators cannot gossip about the state of other validators creates a scenario where a single validator can trivially perform balancing attacks against the network, such as the attack demonstrated in TOB-CHROMAWAY-7.	**Weak**

	This messaging system does not directly contradict the EBFT's safety or asynchrony assumptions, but does make it much more practical to perform complex balancing attacks. In contrast, in a gossip-based messaging system, a balancing attack requires the attacker to have extraordinary control over the global network conditions, drastically increasing the complexity of such attacks. ChromaWay's criteria for a validator to become part of its validator set, along with the validator set's expected composition, are out of the scope of this engagement and excluded from this analysis.	
Documentation	The documentation for the EBFT module was high-quality and thorough; however, the networking stack documentation could be improved. Migrating to libp2p would help alleviate this issue.	**Satisfactory**
Low-Level Manipulation	Kotlin does not expose low-level runtime primitives.	**Not Applicable**
Memory Safety and Error Handling	ChromaWay's use of Kotlin does not have any memory safety concerns, and its error handling is satisfactory.	**Satisfactory**
Testing and Verification	The EBFT and networking component's unit test coverage was measured at 46% and 15% at the time of the engagement. Our general guidance for blockchain nodes is for engineers to target 50% branch coverage at the absolute minimum, with a steadily increasing target of up to 70-95% depending on the project's maturity. See appendix D for guidance on how to improve the project's testing maturity.	**Weak**
Transaction Ordering	The system's block construction components were outside of the engagement's scope. However, given ChromaWay's use of a priority system instead of gas, additional investigation is recommended to determine the system's resilience to transaction ordering attacks.	**Further Investigation Required**

Directory Chain

Category	Summary	Result
Arithmetic	Arithmetic is not done often in the directory chain. When it is, decimals and big ints are typically used, decreasing the chance of overflows or rounding errors. In one case, we found that a division by zero is possible (TOB-CHROMAWAY-22). In another case, we found that conversions between CHR tokens and US dollars are done incorrectly (TOB-CHROMAWAY-15).	**Moderate**
Auditing	The directory chain codebase often emits log messages when important events happen. ChromaWay developers can view these log messages because they run their own nodes. However, this is a fairly ad hoc solution; ideally, a standardized interface for viewing log messages should be developed.	**Moderate**
Authentication / Access Controls	Directory chain operations authenticate users by checking the operation context's signer list, or by calling the `ft4.auth.authenticate` function. This is done correctly throughout the codebase, with one exception (TOB-CHROMAWAY-17). There is no standard convention used to authenticate operations; if such a convention were developed, it would be easier to check that operations authenticate correctly. Many important operations require proposals to be voted on before they can be applied. We found that this system is implemented securely, with the exception of some time-of-check/time-of-use discrepancies (TOB-CHROMAWAY-19).	**Moderate**
Complexity Management	The in-scope code that involves the economy chain and proposals is well-structured overall. The functions are clearly scoped and have a single job to do. Although there appears to be a naming convention, it is not explicitly documented; documenting it would allow for a quicker understanding of functions' responsibilities.	**Satisfactory**
Decentralization	The directory chain system is fairly centralized; the economy chain's admin user can unilaterally set transaction fees above 100% (TOB-CHROMAWAY-24), and system providers have various ways of bringing the	**Moderate**

	system to a halt if enough of them collude to vote on a proposal. ChromaWay's documentation mentions system providers, but not the economy chain's admin user. The documentation should have a page that specifically addresses the system's centralization risks.	
Documentation	The documentation provides a high-level explanation of the system, and the more technical documentation is focused on explaining each possible operation with a brief description, roles needed for that operation, and arguments' types. However, it would be beneficial to have a more thorough description of the arguments' types and to document the possible values for each argument so it is easier to implement the appropriate validation in the code. Additionally, it could be beneficial to perform an analysis that documents the interaction and dependencies of operations to avoid issues like TOB-CHROMAWAY-20.	**Satisfactory**
Testing and Verification	There is currently no way to measure Rell test coverage; the ChromaWay team has stated that they are currently working on this. Certain issues included in this report indicate that test coverage is lacking and does not sufficiently cover the code's edge cases (TOB-CHROMAWAY-16, TOB-CHROMAWAY-22, TOB-CHROMAWAY-24, TOB-CHROMAWAY-15). There is currently no system in place to fuzz test Rell code. Such a system should be developed to make Rell competitive with Solidity.	**Weak**
Transaction Ordering	We have not found any issues where transaction ordering can be used to extract value from users. Because of the types of operations available, it is unlikely that there is any way for this to happen.	**Satisfactory**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Honest nodes may not revolt during a consensus revolt	Undefined Behavior	Medium
2	Syncing nodes may be hijacked using weak subjectivity attacks	Data Validation	High
3	Revolts may succeed without the necessary quorum in asynchronous conditions	Undefined Behavior	Low
4	Validators are miscounted when determining whether to transition from HaveBlock to Prepared	Undefined Behavior	Low
5	Wrong condition when setting to fetch a block in the wait state	Undefined Behavior	Low
6	packBlockRange returns the packet is not full even when there are more blocks to be added	Undefined Behavior	Informational
7	Liveness violation under asynchronous network conditions	Timing	High
8	Slowloris denial-of-service attack	Denial of Service	Medium
9	Incorrect check in require_blockchain function	Data Validation	Undetermined
10	Missing after_provider_updated when updating a provider's state	Undefined Behavior	Undetermined
11	Provider state proposals can create duplicate entities with the same keys	Undefined Behavior	Informational
12	Insufficient check allows for voter sets to be made unusable accidentally	Data Validation	Informational

13	Potential out-of-bounds array index due to insufficient require statement	Data Validation	**Informational**
14	Incorrect staking_requirement_dapp_provider_total_stake_usd default value	Undefined Behavior	**Low**
15	Incorrect conversion of CHR to USD	Undefined Behavior	**High**
16	Action points can be stolen	Data Validation	**High**
17	Incorrect logic in propose_minting allows for one vote to be forged	Data Validation	**Medium**
18	Users are forced to pay for the container expired time when renewing it	Undefined Behavior	**Medium**
19	Time of check / time of use issues for proposals	Data Validation	**Low**
20	Users can avoid paying the upgraded container cost for a certain duration	Undefined Behavior	**Medium**
21	Required staking amounts are in USD rather than CHR	Undefined Behavior	**Informational**
22	Division by zero when calculating rewards	Undefined Behavior	**Informational**
23	Blockchain move proposals only need permission from destination, not source	Access Controls	**High**
24	Missing input validation on setter functions	Data Validation	**Medium**
25	Dapp providers can circumvent rate limits by creating more dapp providers	Denial of Service	**High**
26	Rate Limits may be insufficient to prevent Rell Denial-of-Service attacks	Denial of Service	**Undetermined**

Detailed Findings

1. Honest nodes may not revolt during a consensus revolt	
Severity: **Medium**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-CHROMAWAY-1
Target: EBFT	

Description

Honest nodes check for a consensus among revolting nodes only if the honest node itself is revolting, as shown in figure 1.1. Revolts represent a consensus view-change, where there is a problem with the network and a new leader must be selected. Since the nature of the problem is unknown, honest nodes must assume that the issue may be related to the honest node's own operation in addition to potentially being caused by a malicious leader node.

This means that an honest node must constantly check for a consensus among revolters, even if the honest node itself does not see any reason for a revolt to be triggered.

```
if (myStatus.revolting) {
    when (potentiallyRevolt()) {
        FlowStatus.Break -> return false
        FlowStatus.Continue -> return true
        FlowStatus.JustRunOn -> Unit // nothing, just go on
    }
}
```

Figure 1.1: In the `shouldRecomputeStatusAgain` function, nodes will only check whether other nodes are revolting when the node itself is also revolting. (`postchain/postchain-base/src/main/kotlin/net/postchain/ebft/BaseStatusManager.kt#587–593`)

Exploit Scenario

Consider a leader node connected to the network over an unreliable connection. The leader has a block available, but only a minority of nodes on the network are able to query the leader for the block. The majority that did not receive a block begin a revolt and have enough votes to succeed in revolting. However, the minority of nodes that *did* receive the block will stall since they have no way of observing that a revolt occurred.

Recommendations

Short term, modify the code so nodes will always check whether a revolt is occurring and whether said revolt has consensus support, even if the node itself is not revolting.

Long term, consider running advanced fuzzing or chaos testing against multi-node networks to test for edge conditions in the system's consensus.

2. Syncing nodes may be hijacked using weak subjectivity attacks	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CHROMAWAY-2
Target: EBFT Synchronization	

Description

ChromaWay's chain synchronization mechanism is vulnerable to weak subjectivity attacks: a category of attacks where an attacker constructs an alternate fork of the blockchain using purchased or leaked validator keys that once formed a Byzantine majority of validators.

Subjectivity is a property of blockchains that refers to the amount of social information required to agree on the current state of the blockchain with no knowledge other than the hash of the genesis block. Proof-of-stake blockchains universally exhibit weak subjectivity, meaning that some level of social interaction is required to synchronize a node from genesis to the latest block head. If a user tries to sync a proof-of-stake blockchain without the required social interaction, or if the chain does not support the required social interaction, they expose themselves to the possibility of weak subjectivity attacks.

Weak subjectivity attacks take advantage of the fact that proof-of-stake blockchains need a way to add and remove validators from their validator sets. This means an entity that once controlled a Byzantine majority of the system's voting power may later withdraw from the system, reducing their own risk but retaining the ability to control blockchain's history at the point in time when they held the majority of the system's stake.

The entity would then construct a competing blockchain, forking from the main chain at the point when they controlled the Byzantine majority of voting power. The entity would then gossip their competing blockchain to naive syncing nodes that do not have any way to identify the malicious fork as invalid. Since BFT-based chains do not have a notion of forks or chain-weight, there is no way for nodes following the malicious chain to re-org to the honest chain.

In ChromaWay, there is no slashing, so this issue may be used to orchestrate malicious forks even for validators that are still present in the validator set.

Exploit Scenario

An attacker purchases or steals validator keys corresponding to a Byzantine majority of the chain's voting power at a specific time in the chain's history. They use the validator keys to construct an alternate chain history, and use it to exclude certain transactions from the malicious chain's history, essentially creating a double-spend attack.

Recommendations

Short term, add a checkpoint parameter to ChromaWay node's sync containing a specific recent block height and blockRID. If the node syncs up to the specified block height and the blockRID does not match, halt synchronization and alert the user.

Long term, establish a robust weak-subjectivity checkpointing system to ensure that syncing clients always arrive at the correct chain head. These checkpoints would allow a syncing client to be certain it arrives at the correct chain head as long as the checkpoint is not older than the "weak subjectivity period": a protocol-chosen duration directly correlated to the amount of time allocated to the "exit period."

The "exit period" is an additional mitigation that prevents validators from leaving the validator set and unbonding their stake in block `N`, then immediately constructing alternate forks for blocks `n < N`. When validators are demoted or plan to leave the validator set, their stake remains locked for the exit period to allow one or more weak subjectivity checkpoints to protect against the rewriting of history. For this mitigation to be effective, slashing must also be implemented to allow for punishing validators who construct alternate chain histories, especially during their exit period.

References

- Weak Subjectivity (ethereum.org)

3. Revolts may succeed without the necessary quorum in asynchronous conditions

Severity: **Low**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-CHROMAWAY-3
Target: EBFT	

Description

Poorly synchronized validators in a revolting state fail to remove their revolting status when a revolt occurs that they did not observe directly. This vulnerability allows an attacker to orchestrate a revolt against a proposer without the required quorum, since the non-synchronized validator will erroneously continue to revolt into the following round.

The root vulnerability is caused by a bug in `potentiallyDoSync()`: when a revolt occurs too quickly for a validator to notice, the code in figure 3.1 handles the scenario and increments the validator's round number. However, this code fails to clear the validator's revolting status, allowing the revolt for a previous round to unintentionally "carry over" into the next round.

```
} else if (sameHeightHigherRounds.size >= this.quorum) {
    myStatus.serial += 1
    myStatus.round = sameHeightHigherRounds.sortedDescending()[this.quorum - 1]
    if (myStatus.state == NodeBlockState.HaveBlock) {
        logger.info("Resetting block in HaveBlock state due to new round")
        resetBlock()
    }
    return FlowStatus.Continue
```

Figure 3.1: A revolt occurred as indicated by the number of nodes at a higher height exceeding the quorum, but the node's revolting status is not reset. (postchain/postchain-base/src/main/kotlin/net/postchain/ebft/BaseStatusManager.kt#427–434)

Exploit Scenario

This attack can theoretically allow an attacker controlling a non-quorum number of nodes on the network (say, two out of four), to launch a revolt against an honest primary node. Using this attack, a malicious actor may work to deny the primary node block proposal rewards or censor the proposed block's contents.

However, it should be noted that an attacker who controls 50% of the network can also

choose to attack the network's liveness to achieve the same goals instead of orchestrating a complex revolt.

There is additional risk to this finding in a scenario where EBFT adds slashing in a future update: if a validator accidentally performs a revolt due to network issues that cause its stake to get slashed, the impact of the finding is much greater.

Recommendations

Short term, modify `potentiallyDoSync()` so that it clears the validator's revolting flag in scenarios where the round is changed without direct observation of the revolt.

Long term, add additional tests and consider investing in property-based testing to ensure that node status transitions are always valid.

4. Validators are miscounted when determining whether to transition from HaveBlock to Prepared

Severity: **Low**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-CHROMAWAY-4
Target: EBFT	

Description

The `countNodes()` function, shown in figure 4.1, fails to account for the round number when counting nodes in the specified state. The `countNodes()` function is used by `handleHaveBlockState()` to determine whether the validator should transition from the `HaveBlock` state to the `Prepared` state. Since it fails to pass the current validator's current round, it is possible for the function to count validators that are in the `HaveBlock` state with the same `blockRID`, but different rounds. This will cause a safety violation since the validator will commit to a `blockRID` at a specific round that does not meet the consensus threshold.

```
private fun countNodes(state: NodeBlockState, height: Long, blockRID: ByteArray?):
Int {
    var count = 0
    for (ns in nodeStatuses) {
        if (ns.height == height && ns.state == state) {
            if (blockRID == null) {
                if (ns.blockRID == null) count++
            } else {
                if (ns.blockRID != null && ns.blockRID.contentEquals(blockRID))
                    count++
            }
        }
    }
    return count
}
```

Figure 4.1: The `countNodes()` function fails to account for nodes that are on a different round. (postchain/postchain-base/src/main/kotlin/net/postchain/ebft/BaseStatusManager.kt#63–76)

Recommendations

Short term, add a `roundId` parameter to `countNodes()` to allow the function to account for the node's round ID.

Long term, consider improving the naming scheme or documentation of certain functions to help establish an understanding of what the function is actually trying to accomplish. In addition, add positive and negative unit tests to help establish the properties of various functions.

5. Wrong condition when setting to fetch a block in the wait state

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-CHROMAWAY-5
Target: EBFT	

Description

The `handleWaitBlockState` function has the job of executing the appropriate action when a node is in a wait state. If it is a primary node, it sets the intent to `BuildBlockIntent`; otherwise, it sets the current node's intent to `FetchUnfinishedBlockIntent`, which will try to fetch the block from the primary node. However, the condition to set the intent to `FetchUnfinishedBlockIntent` is wrong because it checks if the current intent is equal to `FetchUnfinishedBlockIntent` and is waiting for the `myStatus.blockRID` (at that point, this is always `null`). However, this condition should check if the current intent is waiting for the `primaryBlockRID`.

```
fun handleWaitBlockState(): Boolean {
     if (isMyNodePrimary()) {
         if (intent !is BuildBlockIntent) {
             intent = BuildBlockIntent
             return true
         }
     } else {
         val primaryBlockRID = this.nodeStatuses[this.primaryIndex()].blockRID
         if (primaryBlockRID != null) {
             val _intent = intent
             if (!(_intent is FetchUnfinishedBlockIntent &&
_intent.isThisTheBlockWeAreWaitingFor(myStatus.blockRID))) {
                 intent = FetchUnfinishedBlockIntent(primaryBlockRID)
                 return true
             }
         } else {
             return if (intent == DoNothingIntent) {
                 false
             } else {
                 intent = DoNothingIntent
                 true
             }
         }
     }
     return false
 }
```

Figure 5.1: The `handleWaitBlockState()` function (postchain/postchain-base/src/main/kotlin/net/postchain/ebft/BaseStatusManager.kt#L549-574)

While this issue does not compromise the correctness of execution, the major consequence is that when a non-primary node is in a wait state, it always enters the `if` branch and sets the intent to `FetchUnfinishedBlockIntent` while returning `true`, meaning the current state has changed. The `handleWaitBlockState` function is called in the `shouldRecomputeStatusAgain` function when the current state is waiting, and the latter is called in a loop of 1,000 times in the `recomputeStatus` function (figure 5.2) until the current state does not change. Given that it incorrectly always returns that the state has changed, the loop is always iterated unnecessarily 1,000 times.

```
fun recomputeStatus() {
     for (i in 0..1000) {
         if (!shouldRecomputeStatusAgain()) break
     }
 }
```

Figure 5.2: The `recomputeStatus()` function (postchain/postchain-base/src/main/kotlin/net/postchain/ebft/BaseStatusManager.kt#L353-357)

Exploit Scenario

Node A is in a wait state. Instead of iterating a few times in the `recomputeStatus` function, it iterates 1,000 times, consuming unnecessary resources.

Recommendations

Short term, correct the condition to check the `primaryNode`'s `blockRID` `_intent.isThisTheBlockWeAreWaitingFor(primaryBlockRID)`.

Long term, consider checking how many resources each integration test uses (e.g., for profiling) and analyze for possible discrepancies from the expected behavior.

6. packBlockRange returns the packet is not full even when there are more blocks to be added

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-CHROMAWAY-6
Target: EBFT	

Description

The `packBlockRange` function builds the blocks list for the `GetBlockRange` message and returns `true` or `false` depending on whether the blocks fit in the packed list. The list has a limit of 10 blocks (`MAX_BLOCKS_IN_PACKAGE`); however, if the number of blocks exceeds the maximum, the function still returns `true`, which incorrectly signals that all blocks can fit in the blocks list.

```
/**
  * Packs blockchain blocks into the "blocks" list so that it does not go over
package size.
  *
  * @return true if all blocks we had could fit in the block
  */
 fun packBlockRange(
         peerId: NodeRid,
         peerEbftVersion: Long,
         startAtHeight: Long,
         myHeight: Long,
         getBlockFromHeight: (height: Long) -> BlockDataWithWitness?, // Sending
this to simplify testing
         buildFromBlockDataWithWitness: (height: Long, blockData:
BlockDataWithWitness) -> CompleteBlock, // Sending this to simplify testing
         packedBlocks: MutableList<CompleteBlock> // Holds the "return" list
 ): Boolean {
     logger.debug { "GetBlockRange from peer $peerId, starting at height
$startAtHeight, myHeight is $myHeight" }
     var totByteSize = 0
     var blocksAdded = 0
     while (blocksAdded < MAX_BLOCKS_IN_PACKAGE) {
       ...
     }
     return true
 }
```

Figure 6.1: The `packBlockRange()` function (postchain/postchain-base/src/main/kotlin/net/postchain/ebft/syncmanager/common/BlockPacker.kt#L25-68)

Since this information is not currently used, this issue is informational.

Recommendations

Short term, when exiting the while loop, the `packBlockRange` function should not always return `true`, but instead should check if `startAtHeight + MAX_BLOCKS_IN_PACKAGE` is equal to `MyHeight;` if it is equal, the `packBlockRange` function should return `true`, and if it is not, the function should return `false`.

Long term, improve the testing suite to check all the possible paths that a function can take at least once.

7. Liveness violation under asynchronous network conditions	
Severity: **High**	Difficulty: **Low**
Type: Timing	Finding ID: TOB-CHROMAWAY-7
Target: EBFT	

Description

ChromaWay's EBFT is vulnerable to a liveness violation under specific asynchronous network conditions, which ultimately leads to a victim node erroneously committing a block that will be orphaned from the chain. Note that the following attack is *not* Byzantine; it may occur under the normal semi-synchronous network conditions that are standard assumptions for BFT algorithms.

Attack 1

Consider a network of four validator nodes of equal voting power, A, B, C, and D, where there is a proposal for block X at round 1. Suppose that nodes B, C, and D all receive the block from the proposer and change their state from `WaitBlock` to `HaveBlock`, but node A does not receive the block due to network asynchrony.

Each node broadcasts its status updates, but only node D receives all of the status updates. Nodes A, B, and C receive all of the status updates except node D's update. Node D observes that three out of four nodes are in `HaveBlock`, so it changes its state to `Prepared`. This effectively commits the block for node D, as it generates and serves a commit signature for the block.

Meanwhile, nodes A, B, and C cannot form a quorum of `HaveBlock` without D's status update, so they trigger a revolt and the revolt succeeds. Nodes A, B, and C now proceed in round 2, proposing block Y, forming a quorum, and committing the block. Since node D committed to block X, it cannot respect the revolt and continue participating in consensus. If node D were to create a new commitment to block Y, it would be committing a safety violation because it is committed to two different blocks at the same block height.

Attack 2

If we assume this safety violation is benign and allow D to commit to a different block at the same height, a new attack against the system's safety is created using a balancing attack. Consider a different network of four validator nodes where node A has 1% of voting power, and nodes B, C, and D each have 33% of voting power. Node A is malicious and wants to cause a chain split. Block X in round 1 is proposed, all the nodes receive the block, and they all transition to the `HaveBlock` state. However, node B's `HaveBlock` status update is not

received by any nodes. Since nodes A, C, and D are all in `HaveBlock`, they still have enough voting power to transition to `Prepared`. However, malicious node A withholds its transition to the `Prepared` state, which causes nodes C and D to commit to block X, but they are unable to form a consensus with only 66% of the network's voting power. A revolt is executed, moving to round 2 with a proposal for block Y. No other malicious action is taken, and block Y is committed to by all nodes.

At this point, malicious node A may take the commit signatures it obtained from nodes C and D in round 1, add its own signature, form a 67% quorum, and build a valid witness for block X at the same height as block Y. This proves that simply allowing a validator to sign pre-commits for two different blocks for the same block height causes a non-benign safety violation that can be exploited by validators with arbitrary voting power.

Attack 3

It should be noted that if the malicious node is the block proposer for round 1, it can trigger the above attack without any asynchronous network conditions. It does this by sharing the block with only 66% of the network's voting power, withholding its commit signature, and then signing the block after the revolt passes.

Exploit Scenario

An attacker may use a doubly-signed block for a variety of nefarious purposes. The highest-impact attack would be to attack a third-party exchange; the attacker would create a deposit transaction and include it in block X (the block to be orphaned). When the exchange's nodes try to synchronize, the attacker ensures that they receive block X before they can receive the canonical block Y, thus creating a double-spend.

Recommendations

Short term, the protocol's gossip should be updated in such a way that validators can gossip about the known states of other validators in the set. This can help prevent attack 3 by allowing nodes to obtain the proposed block from other validators, and can partially prevent attack 2 by allowing honest nodes to gossip about other nodes' `HaveBlock` messages, preventing malicious nodes from withholding `HaveBlock` messages. However, solving attack 1 and comprehensively solving attack 2 will require a more thorough, multi-faceted mitigation.

Long term, ChromaWay should migrate its BFT to a three-step round, where a new "pre-vote" phase preempts the pre-commit phase. This new phase provides a way for validators to "lock" their pre-commit votes to a specific block, and to "unlock" their pre-commit votes only when a consensus is formed. An explanation of the pre-vote phase and vote locking/unlocking can be found in section 3.2.2 of Buchman (2016)[0] and Buchman et. al (2018)[1]. A formal treatment of generic Byzantine fault-tolerant algorithms used to research this finding may be found in Rütti et. al (2010)[2].

References

- [0] Tendermint: Byzantine Fault Tolerance in the Age of Blockchains
- [1] The latest gossip on BFT consensus
- [2] Generic construction of consensus algorithms for benign and Byzantine faults

8. Slowloris denial-of-service attack	
Severity: **Medium**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-CHROMAWAY-8
Target: Networking code	

Description

Postchain's networking code is vulnerable to a Slowloris DoS attack by an unprivileged network attacker. The attack would work as follows:

- The attacker opens a connection to a postchain node, claiming to be a trusted node with a known public key and promising to send a 29-MB message. This is below the 30 MB limit.
- The attacker sends the first 20 MB of data. So far, this data cannot be verified, since signature verification can only happen once the full 29 MB of data is received.
- The attacker then sends one byte of data every 30 seconds in order to keep the connection alive and avoid triggering the idle timeout of 60 seconds.
- While keeping the first connection alive, the attacker creates many more connections in the same way.
- Eventually, these connections will either consume all of the memory in the target node, causing an out-of-memory crash, or reach a maximum connection count, causing legitimate nodes to be unable to communicate with the target node.

This attack is possible due to two weaknesses in Postchain's networking system:

- Signatures can only be verified once a full message has been delivered.
- There is no timeout for messages, aside from the idle timeout. This idle timeout still allows messages to be sent extremely slowly.

Recommendations

Short term, enforce a message timeout. Messages that are not sent before the timeout should be dropped.

Long term, send messages over mutual TLS (mTLS). This would require every TCP packet to be signed, causing the Slowloris attack to be caught much earlier.

9. Incorrect check in require_blockchain function

Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-CHROMAWAY-9
Target: `directory-chain/src/common/require.rell`	

Description

The check shown in figure 9.1 is incorrect. This check is used in the require_blockchain function to determine whether a blockchain is running or not, with an optional `require_running` parameter. When `require_running` is set to `true`, the blockchain's state should only be allowed to be `RUNNING`, rather than `PAUSED`. However, a paused blockchain will still pass this check, even when `require_running` is set to `true`. This is because the expression `(require_running and .state == blockchain_state.RUNNING) or .state in [blockchain_state.RUNNING, blockchain_state.PAUSED]` will simplify to `(true and false) or true`, which further simplifies to `true`.

```
function require_blockchain(
    blockchain_rid: byte_array,
    require_running: boolean = false,
    include_removed: boolean = false
) = require(
    blockchain @? {
        blockchain_rid,
        include_removed or
        (require_running and .state == blockchain_state.RUNNING) or
        .state in [blockchain_state.RUNNING, blockchain_state.PAUSED]
    }, "Unknown blockchain " + blockchain_rid
);
```

Figure 9.1: The `require_blockchain` function (`directory-chain/src/common/require.rell:25-36`)

Recommendations

Short term, fix this check so that it correctly accounts for `require_running`. An example of how to do this is shown in figure 9.2.

```
function require_blockchain(
    blockchain_rid: byte_array,
    require_running: boolean = false,
```

```
    include_removed: boolean = false
) = require(
    blockchain @? {
        blockchain_rid,
        include_removed or
        (require_running and .state == blockchain_state.RUNNING) or
        ((not require_running) and .state in [blockchain_state.RUNNING,
blockchain_state.PAUSED])
    }, "Unknown blockchain " + blockchain_rid
);
```

Figure 9.2: Proposed replacement for `require_blockchain` function

10. Missing after_provider_updated when updating a provider's state

Severity: **Undetermined**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-CHROMAWAY-10
Target: `directory-chain/src/proposal_provider/proposal_provider_state.rell`	

Description

The `propose_provider_state` operation directly updates the state of a provider instead of creating a proposal when its tier is DAPP_PROVIDER. However, the operation is missing a call to the `after_provider_updated` function that would send a message to the economy chain to notify the change.

```
operation propose_provider_state(my_pubkey: pubkey, provider_pubkey: pubkey, active:
boolean, description: text = "") {
    val me = require_provider(my_pubkey);
    require_provider_auth_with_rate_limit(me);

    val other_prov = require_provider(provider_pubkey);

    // Only SP and NP can enable/disable providers
    require_node_access(me);

    if (roles.has_node_access(me) and other_prov.tier ==
provider_tier.DAPP_PROVIDER) {
        update_provider_state(other_prov, active);
    } else {
      ...
```

Figure 10.1: Snippet of the `propose_provider_state()` operation (directory-chain/src/proposal_provider/proposal_provider_state.rell#L33-44)

Recommendations

Short term, when updating the state of a provider with the DAPP_PROVIDER tier, add a call to `after_provider_updated(other_prov)`.

11. Provider state proposals can create duplicate entities with the same keys

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-CHROMAWAY-11
Target: `directory-chain/src/proposal_provider/{proposal_provider_state,proposal_provider_is_system}.rell`	

Description

In the directory chain's `proposal_provider` module, there is no safeguard preventing a proposal that activates an already-active provider. If this provider is a system provider, this will result in the creation of a duplicate `voter_set_member` entity, which has the same key as an existing `voter_set_member` entity (see figure 11.1).

This issue is rated as informational since the entity creation would result in a revert with a confusing error message, rather than allowing for any attacks.

```
function update_provider_state(provider, active: boolean, proposal: proposal? =
null) {
    provider.active = active;
    if (active == false) {
        // ...
    } else { // enable
        // If enabled provider is a system provider, update SYSTEM_P voter set
        if (roles.has_system_access(provider)) {
            create voter_set_member(voter_set @ { voter_sets.system_p }, provider);
        }
    }
}
```

Figure 11.1: Creation of `voter_set_member` entity. This entity will be a duplicate if the provider was already active.
(`directory-chain/src/proposal_provider/proposal_provider_state.rell:53-91`)

In addition, it is also possible to create and apply a proposal that gives system permissions to an existing system provider. This would cause the `enroll.system` function to be called on this provider, which would result in the creation of duplicate `voter_set_member` and `cluster_provider` entities.

Recommendations

Short term, when applying a `pending_provider_state` proposal, check that the provider's `active` property is the opposite of the proposal's `active` property. When applying a `pending_provider_is_system` proposal, check that the provider's `system` property is the opposite of the proposal's `system` property.

12. Insufficient check allows for voter sets to be made unusable accidentally

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CHROMAWAY-12
Target: `directory-chain/src/economy_chain/economy_chain_proposal_voter_set.rell`	

Description

The check highlighted in figure 12.1 is used to ensure that voter sets do not accidentally set their threshold too high; this would render the voter set unusable, since it would be unable to meet its threshold. However, the threshold can be changed while at the same time removing members from the voter set. If this happened, the threshold could potentially be set too high, since the check does not account for this case.

```
function _propose_update_ec_voter_set(my_pubkey: pubkey, voter_set_name: text,
new_threshold: integer?, new_member: list<pubkey>, remove_member: list<pubkey>) {
    require_is_signer(my_pubkey);
    val voter_set = require_common_voter_set(voter_set_name);
    require(voter_set.name != voter_sets.system_p, "Cannot update system voter set.
Update this by proposing system provider role");
    require_common_voter_set_governor(voter_set, my_pubkey);
    require(empty(common_proposal @* { voter_set, common_proposal_state.PENDING }),
"Cannot have more than one pending proposal involving this voter set.");

    val prop = create common_proposal(
        op_context.last_block_time,
        common_proposal_type.ec_voter_set_update,
        my_pubkey,
        voter_set,
        "Update %s voter set".format(voter_sets.chromia_foundation)
    );
    val update_prop = create pending_ec_voter_set_update(prop, voter_set);
    if (new_threshold != null) {
        require(new_threshold >= -1 and new_threshold <= (common_voter_set_member @*
{ voter_set }).size(),
        "Invalid threshold level, must be in range [-1, voter_set.size()]");
        create ec_voter_set_update.threshold(update_prop, new_threshold);
    }
    for (m in new_member) {
        create ec_voter_set_update.new_member(update_prop, m);
    }
    for (m in remove_member) {
        create ec_voter_set_update.remove_member(update_prop, m);
```

```
    }
    internal_common_vote(my_pubkey, prop, true);
}
```

Figure 12.1: Check on voter set threshold (directory-chain/src/economy_chain/economy_chain_proposal_voter_set.rell:62-89)

Recommendations

Short term, modify this check so that it accounts for the `remove_member` and `new_member` lists. The maximum threshold should be `(common_voter_set_member @* { voter_set }).size() + new_member.size() - remove_member.size()`.

13. Potential out-of-bounds array index due to insufficient require statement

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CHROMAWAY-13
Target: `directory-chain/src/lib/iccf/module.rell`	

Description

The `extract_operation_arg` function, used for the Inter-Chain Confirmation Facility, contains a `require` statement meant to prevent out-of-bounds array access (see figure 13.1). However, the `require` statement does not trigger if an index to `args[args.size()]` is attempted. This can lead to an out-of-bounds array indexing.

This issue is rated as informational because this out-of-bounds array indexing would trigger a revert anyway, but with a more confusing error message.

```
function extract_operation_arg(
    gtx_transaction,
    op_name: text,
    arg: integer = 0,
    verify_signers: boolean = true,
    require_anchored_proof: boolean = false
): gtv {
    val args = extract_operation_args(gtx_transaction, op_name, verify_signers,
require_anchored_proof);
    require(
        args.size() >= arg,
        "Argument number %d not found on operation %s. %d arguments found"
            .format(arg, op_name, args.size())
    );
    return args[arg];
}
```

Figure 13.1: Insufficient `require` statement (`directory-chain/src/lib/iccf/module.rell:23-37`)

Recommendations

Short term, change the greater-than-or-equals operator shown in figure 13.1 to a greater-than operator. This will cause the `require` statement to trigger if `args.size()` equals `arg`.

14. Incorrect staking_requirement_dapp_provider_total_stake_usd default value

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-CHROMAWAY-14
Target: `directory-chain/src/economy_chain/economy_chain_model.rell`	

Description

The `staking_requirement_dapp_provider_total_stake_usd` default value is set to 10k instead of 100k. As a result, a dapp provider could stake a much lower amount than the requirement to receive the rewards.

```
mutable staking_requirement_dapp_provider_total_stake_usd: integer = 10000; // 100k
usd
```

Figure 14.1: The `staking_requirement_dapp_provider_total_stake_usd()` default value (`directory-chain/src/economy_chain/economy_chain_model.rell#L35`)

Exploit Scenario

Alice, a dapp provider, stakes a total of 10k USD instead of 100k while still receiving rewards.

Recommendations

Short term, correct the `staking_requirement_dapp_provider_total_stake_usd` default value to `100_000`.

Long term, improve the testing suite with tests for the default staking values with at least one positive and one negative test.

15. Incorrect conversion of CHR to USD

Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-CHROMAWAY-15
Target: `directory-chain/src/economy_chain/economy_chain_reward.rell`	

Description
The `get_chr_in_usd` function is used when computing the user's amount of CHR in USD to decide if the staking requirements are satisfied; however, the function multiplies the CHR in input for the `chr_per_usd` value instead of dividing it.

```
function get_chr_in_usd(chr: integer): decimal =
    chr * economy_constants.chr_per_usd;
```

Figure 15.1: The `get_chr_in_usd()` function (`directory-chain/src/economy_chain/economy_chain_reward.rell#L281-282`)

`chr_per_usd` represents how many CHR one can get for 1 USD, so multiplying for it is incorrect. For example, if we have 100 CHR and the `chr_per_usd` is 5, the correct result would be 100 / 5 = 20 instead of 100 * 5 = 500.

Exploit Scenario
Assuming the `chr_per_usd` is set to 5, every user can stake 25 times less CHR than required while receiving rewards.

Recommendations
Short term, make the `get_chr_in_usd` function divide the CHR by the `chr_per_usd` value.

Long term, improve the testing suite to validate the appropriate amount of CHR that users are required to stake. Additionally, consider validating the staking requirements using the CHR amount directly instead of the amount converted in USD (see TOB-CHROMAWAY-21).

16. Action points can be stolen

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CHROMAWAY-16
Target: `directory-chain/src/common/operations/common_operations_provider.rell`	

Description

The `transfer_action_points` function does not validate that the amount passed is not negative; as a consequence, by passing a negative value, it is possible to steal action points from the `to` address.

```
operation transfer_action_points(from: pubkey, to: pubkey, amount: integer) {
    val _from = require_is_provider_with_rate_limit(from);
    val _to = require_provider(to);
    require(provider_rl_state @ { _from } .points >= amount, "Not enough action points to
transfer from.");
    update provider_rl_state @ { _from } ( .points -= amount );
    update provider_rl_state @ { _to } ( .points += amount );
}
```

Figure 16.1: The `transfer_action_points()` function (directory-chain/src/common/operations/common_operations_provider.rell#L42-48)

The system uses action points to limit the actions each provider can execute through the `require_provider_auth_with_rate_limit` and `require_is_provider_with_rate_limit` functions. A non-exhaustive list of actions that are limited by action points includes: register a new provider, update a new provider, add and remove a blockchain replica, propose a new container, propose new cluster limits, and propose a provider to be a system.

A malicious provider could cause a DoS by executing many actions in quick succession or remove all action points from other providers.

Exploit Scenario

Eve, a provider, exploits this vulnerability to get a lot of points and execute many actions, causing a DoS to the system.

Recommendations

Short term, validate that `amount` is greater than 0.

Long term, improve the unit testing of functions that take a signed integer with tests for both positive and negative integers and verify they have the expected behavior.

17. Incorrect logic in propose_minting allows for one vote to be forged

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CHROMAWAY-17
Target: `directory-chain/src/economy_chain/economy_chain_mint.rell`	

Description

The `propose_minting` operation, shown in figure 17.1, is used to create a proposal to mint CHR tokens. It creates an initial "yes" vote for the user-provided `proposal_by` value. However, it does not verify that this value is one of the signers of the transaction that called the operation. This means that one extra "yes" vote can be maliciously gained in favor of a minting proposal.

```
operation propose_minting(proposal_by: byte_array, amount: integer, account_id: byte_array)
{

    val mint_chr_voter_set = require_mint_chr_voter_set();
    require_signer_member_of_voter_set(mint_chr_voter_set);
    require(amount > 0, "Amount must be greater than 0");
    ft4.accounts.Account(account_id);

    val proposal = create common_proposal(
        op_context.last_block_time,
        common_proposal_type.ec_mint,
        proposal_by,
        mint_chr_voter_set,
        "Minting %d assets to account id %s".format(amount, account_id));

    create pending_minting(
        proposal,
        amount,
        account_id
    );

    internal_common_vote(proposal.proposed_by, proposal, true);
}
```

Figure 17.1: `propose_minting` operation (`directory-chain/src/economy_chain/economy_chain_mint.rell:43-64`)

Exploit Scenario

A group of nine voters wants to mint tokens. However, a threshold of 10 voters is required for the vote to pass. One of the voters calls the `propose_minting` function, and provides someone else's public key as the `proposal_by` argument. The nine voters then vote "yes" on this proposal, and it passes.

Recommendations

Short term, add a statement to the `propose_minting` operation requiring that `proposal_by` was a signer of the transaction.

Long term, create a standard convention for how to authenticate operations, and create automated tests that ensure that this convention is always followed. For example, this convention may be: “The first argument to the operation must be named `my_pubkey`, and the operation must always contain the statement `require_is_signer(my_pubkey)`.” This convention can be checked by simple text parsing or syntax tree inspection, without need for advanced static analysis tooling.

18. Users are forced to pay for the container expired time when renewing it

Severity: **Medium**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-CHROMAWAY-18
Target: `directory-chain/src/economy_chain/economy_chain_container.rell`	

Description

When a user renews a container's lease with the `renew_container_lease` function, the user must also repay for the time when the container expired and stopped as if it was actually running during that time.

```
function renew_container_lease(ft4.accounts.account, lease, duration_weeks: integer) {
     val tag = lease.cluster.tag;
     val cost = calculate_container_cost(duration_weeks, lease.container_units,
lease.extra_storage_gib, tag);
     ft4.assets.Unsafe.transfer(account, get_pool_account(), get_asset(), cost);
     lease.duration_millis += duration_weeks * millis_per_week;
 }
```

Figure 18.1: The `renew_container_lease()` function (`directory-chain/src/economy_chain/economy_chain_container.rell#L264-269`)

A container has a lease associated with a `start_time` and `duration_millis` fields. A container is considered expired if the current time is greater than `start_time` + `duration_millis`; if it is expired, it will be stopped. Given that the `start_time` is not updated in the `renew_container_lease` function, the user is effectively paying for the time when the container expired and stopped.

Exploit Scenario

Alice makes her container's lease expire, but after three weeks she wants to renew the lease. She renews it for four weeks; however, after one week, it unexpectedly expires again.

Recommendations

Short term, update the lease `start_time` in `renew_container_lease` when the lease is expired.

Long term, improve the documentation and testing suite for this specific scenario to clarify and check for the expected behavior, respectively.

19. Time-of-check/time-of-use issues for proposals

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CHROMAWAY-19
Target: Directory chain, various locations	

Description

In the directory chain codebase, many properties are checked at the time that proposals are created, but not at the time that they are applied. These properties may become false while the proposal is being voted on, leading to problems when applying the proposal. Instead, properties should be checked both when creating and when applying proposals.

This issue details various locations where time-of-check/time-of-use discrepancies may cause problems when applying proposals.

Cluster CPU and storage capabilities are checked when creating cluster limit proposals, rather than when applying them. This can cause clusters to have limits that are lower than their current usage.

Quotas are checked when creating container proposals, rather than when applying them. This means that someone with a quota of 10 containers can propose 100 containers, and then these proposals can be applied later on in quick succession, circumventing the quota.

Two proposal handlers in the `proposal_blockchain_import` directory check that a blockchain is in an `IMPORTING` state when a proposal is suggested, but not when it is applied. In particular, this happens for the `configuration_import` and `foreign_blockchain_blocks_import` proposals. The blockchain could finish importing in the meantime, leading to various unexpected actions being applied on running blockchains.

When a proposal to remove a blockchain is suggested, there is a check to ensure that no other blockchains depend on it. This check is not performed again when applying the proposal. However, this will only result in the proposal failing to be applied, because the Rell interpreter's constraint checker would detect a `blockchain_dependency` entity with a dangling pointer to the deleted blockchain.

Recommendations

Short term, remediate the individual issues described above by adding relevant checks to proposal-applying functions.

Long term, for each proposal, create a function that checks all of the properties that must hold for the proposal to be applied, and add calls to this function in the code for creating and for applying the proposal.

20. Users can avoid paying the upgraded container cost for a certain duration

Severity: **Medium**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-CHROMAWAY-20
Target: `directory-chain/src/economy_chain/economy_chain_container.rell`	

Description

Users can upgrade the resources a container uses while still paying the cost of the old, non-upgraded container.

When a container is upgraded, the cost paid by the user first considers the new resources used, as shown in figure 20.1, after which a ticket is created and sent as a message.

```
function upgrade_container_impl(
    container_name: text,
    upgraded_container_units: integer,
    upgraded_extra_storage_gib: integer,
    upgraded_cluster_name: text
) {
    ...
    val cost = calculate_container_cost(lease.duration_millis / millis_per_week,
upgraded_container_units, upgraded_extra_storage_gib, cluster.tag)
    - if (lease.expired) 0 else calculate_remaining_lease_value(lease,
op_context.last_block_time);
    ft4.assets.Unsafe.transfer(account, get_pool_account(), get_asset(), cost);

    val ticket = create ticket(type = ticket_type.UPGRADE_CONTAINER, account);
    create upgrade_container_ticket(ticket,
        container_name = container_name,
        container_units = upgraded_container_units,
        extra_storage_gib = upgraded_extra_storage_gib,
        cost = cost,
        cluster_name = cluster.name
    );
    send_message(upgrade_container_topic, upgrade_container_message(
        ticket_id = ticket.rowid.to_integer(),
        container_name = container_name,
        container_units = upgraded_container_units,
        extra_storage = 1024 * upgraded_extra_storage_gib,
        cluster_name = upgraded_cluster_name
    ).to_gtv());
}
```

Figure 20.1: The `handleWaitBlockState()` function (core/directory-chain/src/economy_chain/economy_chain_container.rell#L117-159)

The message is then handled by the `receive_ticket_container_result_message` function (figure 20.2). If successful, it creates a new lease. The problem is that the `duration_millis` used in that new lease is taken directly from the current lease, which can be increased by renewing the lease while still paying for the old resources used. This means that users can effectively increase the lease duration with the new resources while paying for the old (smaller) resources.

```
function receive_ticket_container_result_message(body: gtv) {
    val message = ticket_container_result_message.from_gtv(body);
    val ticket = ticket @? { .rowid == rowid(message.ticket_id) };
    if (ticket == null) {
        log("ticket_id %s not found".format(message.ticket_id));
        return;
    }
    ticket.state = if (message.error_message == null) ticket_state.SUCCESS else
ticket_state.FAILURE;
    ticket.error_message = message.error_message ?: "";

    when (ticket.type) {
        ticket_type.CREATE_CONTAINER -> {
          ...
        }
        ticket_type.UPGRADE_CONTAINER -> {
            val specific_ticket = upgrade_container_ticket @ { ticket };
            val cluster = cluster @ { .name == specific_ticket.cluster_name };
            when (ticket.state) {
                ticket_state.SUCCESS -> {
                    // Note: When we implement "move" the container name might change so we
should check what name D1 returns
                    val upgraded_container_name = message.container_name!!;
                    val upgraded_cluster_name = message.cluster_name!!;
                    log("Successfully upgraded container %s for ticket
%s".format(upgraded_container_name, message.ticket_id));
                    val current_lease = lease @ { .container_name ==
specific_ticket.container_name };
                    val currently_auto_renewed = current_lease.auto_renew;
                    val current_duration = current_lease.duration_millis;
                    val bridge_leases = delete_bridge_leases(current_lease);
                    delete current_lease;
                    val new_lease = create lease(
                        container_name = upgraded_container_name,
                        account = ticket.account,
                        container_units = specific_ticket.container_units,
                        extra_storage_gib = specific_ticket.extra_storage_gib,
                        cluster = cluster,
                        start_time = op_context.last_block_time,
                        duration_millis = current_duration,
                        auto_renew = currently_auto_renewed
                    );
                    transfer_bridge_leases_to_lease(new_lease, bridge_leases);
                }
                ticket_state.FAILURE -> {
                    ...
                }
            }
        }
    }
```

```
}
```

Figure 20.2: The `recomputeStatus()` function (`core/directory-chain/src/economy_chain/economy_chain_ticket.rell#L6-78`)

Exploit Scenario

Eve has a container with 15 `container_units` and 10 `extra_storage_gib`. She calls the `upgrade_container` operation with 50 container_units and 40 `extra_storage_gib`. She then immediately calls the `renew_container` operation to extend the lease duration by 12 weeks and pays the cost for the old resources (15 `container_units` and 10 `extra_storage_gib`). The ticket message is then handled and she gets a new lease with 50 `container_units` and 40 `extra_storage_gib` with a duration extended by 12 weeks.

Recommendations

Short term, make the `renew_container` operation not callable when a container upgrade is ongoing.

Long term, analyze all possible operations that can be called and that could affect expected behavior when a message is sent and they are called before it is handled.

21. Required staking amounts are in USD rather than CHR	
Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-CHROMAWAY-21
Target: Directory chain	

Description

The minimum amount of money that must be held by stakers is specified in US dollars, rather than in CHR tokens. However, stakers hold their value in CHR. This means that a decrease in the price of CHR can make existing stakers ineligible. In addition, specifying the staking amount in USD gives the price oracle an undue amount of influence over the staking system.

This issue is rated as informational, since staking eligibility currently affects only who receives awards.

Recommendations

Short term, specify staking amounts in CHR, rather than USD. Doing so would also solve TOB-CHROMAWAY-15, since the `get_chr_in_usd` function would no longer be needed.

22. Division by zero when calculating rewards

Severity: **Informational**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-CHROMAWAY-22
Target: `directory-chain/src/economy_chain/economy_chain_reward.rell`	

Description

The following calculations, performed while determining reward payouts for dapp clusters, may perform divisions by zero. A cluster with malicious owners can force these divisions by zero to happen by adjusting the number of available nodes and container units in the cluster.

```
val max_dapp_provider_reward_per_node = ((dapp_cluster_value * (1 -
economy_constants.dapp_provider_risk_share) + dapp_cluster_value * occupancy_rate *
economy_constants.dapp_provider_risk_share) / number_of_nodes_per_dapp_cluster) *
economy_constants.chr_per_usd * units_per_asset;


function occupancy_rate(cluster): decimal = decimal(occupied_scus(cluster)) /
(total_available_scus(cluster.cluster_units) -
standard_cluster_unit.system_container_units);
```

Figure 22.1: Calculations that can result in division by zero (`directory-chain/src/economy_chain/economy_chain_reward.rell:70,204`)

This division by zero will result in a revert, and will prevent the cluster from receiving rewards. However, the revert will not interfere with rewards given to other clusters, so this issue is rated as informational.

Recommendations

Short term, add a check before these calculations that causes the calculations to exit without reverting if they will result in a division by zero.

23. Blockchain move proposals only need permission from destination, not source

Severity: **High**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-CHROMAWAY-23
Target: `directory-chain/src/proposal_blockchain_move/proposal_blockchain_move.rell`	

Description

Blockchain move proposals are voted on by the destination container, and not by the source container. Only one member of the source container's voter set is needed to create this proposal. This means that a member of a container's voter set can take over a blockchain by moving it to a malicious container. A TODO comment in the code mentions this problem.

```
operation propose_blockchain_move(my_pubkey: pubkey, blockchain_rid: byte_array,
destination_container: text, description: text = "") {
    // ...
    val src_container = container_blockchain @ { blockchain } .container;
    require_container_deployer(src_container, me);
    // ...
    val prop = create_proposal(proposal_type.blockchain_move_start, me,
dst_container.deployer, description); // TODO: POS-961: src + dst?
    create pending_blockchain_move(prop, blockchain, dst_container);
    internal_vote(me, prop, true);
}
```

Figure 23.1: The `propose_blockchain_move()` function. The first highlighted piece of code requires a single member of the source container's voter set to create the proposal. (`directory-chain/src/proposal_blockchain_move/proposal_blockchain_move.rell:7-37`)

Exploit Scenario

One of the members of a container's voter set goes rogue and creates a blockchain move proposal to transfer the blockchain to a container entirely under his control. The other members of the source container's voter set are unable to stop him.

Recommendations

Short term, change the blockchain move proposal code so that it creates two proposals that must happen in succession: one that must be approved by the source of the move, and one that must be approved by the destination.

24. Missing input validation on setter functions

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CHROMAWAY-24
Target: `directory-chain/src/economy_chain/economy_chain_operations.rell`	

Description

The operations `update_economy_constant` and `update_system_provider_economy_constants` update certain system variables that represent a percentage, such as `staking_reward_fee_share` and `system_provider_fee_share`. However, the system does not validate that the new value is between 0 and 1, with the latter being 100%.

For the `update_economy_constants` operation (figure 24.1), the following arguments are missing validations: `staking_reward_fee_share`, `chromia_foundation_fee_share`, `resource_pool_margin_fee_share`, and `dapp_provider_risk_share`.

```
operation update_economy_constants(
     min_lease_time_weeks: integer?,
     max_lease_time_weeks: integer?,
     staking_reward_fee_share: decimal?,
     chromia_foundation_fee_share: decimal?,
     resource_pool_margin_fee_share: decimal?,
     dapp_provider_risk_share: decimal?
 ) {
     val provider = require_system_provider_signer_entity();

     val proposal = create_system_p_proposal(common_proposal_type.ec_constants_update,
provider, "Update economy constants");
     create pending_economy_constants(
         proposal,
         min_lease_time_weeks = get_integer_or_default(min_lease_time_weeks, -1)!!,
         max_lease_time_weeks = get_integer_or_default(max_lease_time_weeks, -1)!!,
         staking_reward_fee_share = get_decimal_or_default(staking_reward_fee_share, -1)!!,
         chromia_foundation_fee_share = get_decimal_or_default(chromia_foundation_fee_share,
-1)!!,
         resource_pool_margin_fee_share =
get_decimal_or_default(resource_pool_margin_fee_share, -1)!!,
         dapp_provider_risk_share = get_decimal_or_default(dapp_provider_risk_share, -1)!!
     );

     internal_common_vote(provider.pubkey, proposal, true);
 }
```

Figure 24.1: The `update_economy_constants()` operation (core/directory-chain/src/economy_chain/economy_chain_operations.rell?#L66-88)

For the `update_system_provider_economy_constants` operation (figure 24.2), the following arguments are missing validations: `system_provider_fee_share`, and `system_provider_risk_share`.

```
operation update_system_provider_economy_constants(
     total_cost_system_providers: integer? = null,
     system_provider_fee_share: decimal? = null,
     system_provider_risk_share: decimal? = null
 ) {
     require_admin();
     economy_constants.total_cost_system_providers = total_cost_system_providers ?:
economy_constants.total_cost_system_providers;
     economy_constants.system_provider_fee_share = system_provider_fee_share ?:
economy_constants.system_provider_fee_share;
     economy_constants.system_provider_risk_share = system_provider_risk_share ?:
economy_constants.system_provider_risk_share;
 }
```

Figure 24.2: The `update_system_provider_economy_constants()` operation (core/directory-chain/src/economy_chain/economy_chain_operations.rell?#L31-40)

Possible issues can arise when one of those variables are set to greater than 1, such as an underflow when computing the `dapp_provider_fee_share`. Additionally, an underflow can happen even if the sum of the three subtracted values is greater than 1.

```
function dapp_provider_fee_share(): decimal = 1 -
economy_constants.chromia_foundation_fee_share -
economy_constants.resource_pool_margin_fee_share -
economy_constants.system_provider_fee_share;
```

Figure 24.3: The `dapp_provider_fee_share()` function (core/directory-chain/src/economy_chain/economy_chain_reward.rell#L196)

Exploit Scenario

Alice, the protocol admin, incorrectly updates the `system_provider_fee_share` to a value greater than 1. The `dapp_provider_fee_share` function underflows and returns a large value, which compromises the system's intended behavior.

Recommendations

Short term, validate in the `update_economy_constant` and `update_system_provider_economy_constants` operations that the new values are not greater than 1 and, where necessary, that their sum is also not greater than 1. For example, the sum of `chromia_foundation_fee_share`, `resource_pool_margin_fee_share` and `system_provider_fee_share` should not be greater than 1.

Long term, improve the system test suite by checking that the setter functions respect the specifications and do not allow setting the corresponding variable to an invalid value.

25. Dapp providers can circumvent rate limits by creating more dapp providers

Severity: **High**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-CHROMAWAY-25
Target: Directory chain	

Description
Dapp providers are rate limited to prevent DoS attacks. However, they can circumvent these rate limits by creating more dapp providers. Each dapp provider has an independent rate limit, so by creating more dapp providers, an attacker will be allowed to perform more actions during the same amount of time.

It is worth noting that node providers or system providers can delete these newly created dapp providers, although this would be difficult to do if the attacker creates an extremely large number of providers.

Exploit Scenario
An attacker has a single dapp provider and wishes to perform one million operations per day, but his dapp provider has a rate limit of 1,000 actions per day. He uses his dapp provider to create 1,000 more dapp providers. Each of these 1,000 providers can perform 1,000 actions per day, so the attacker can now perform one million actions per day.

The attacker also continuously creates more dapp providers to make it more difficult for a node or system provider to delete them all.

Recommendations
Short term, create a separate rate limit for dapp provider creation that is tied to a lease, rather than to a specific dapp provider. Alternately, create a veto period when dapp providers are created; during this veto period, prevent the newly created provider from taking any actions, and allow node and system providers to delete the newly created dapp provider if they deem it to be malicious.

26. Rate limits may be insufficient to prevent Rell denial-of-service attacks

Severity: **Undetermined**	Difficulty: **N/A**
Type: Denial of Service	Finding ID: TOB-CHROMAWAY-26
Target: Rell	

Description

ChromaWay does not use a traditional gas-based metering system to prevent DoS attacks, and instead uses a role-bound rate-limiting system. We believe that over the long term, this rate-limiting system will cause existential risks to the stability of ChromaWay, preventing ChromaWay dapps from being able to operate without using KYC (know-your-customer) restrictions.

Blockchains with arbitrary code execution capabilities always require a way to determine if and when a program will halt. Without this capability, it is impossible to construct a block containing transactions that will terminate before the block proposal deadline. Traditionally, this problem is solved by using either languages that are not Turing-complete (Bitcoin Script), or a runtime environment that can be metered and constrained using gas limits (Ethereum Virtual Machine, or EVM). These two solutions allow the following properties:

1. The execution time of a transaction can be estimated without running the program itself.
2. Transaction execution costs remain accurate even when the blockchain is highly utilized.
3. Block proposers are protected against Sybil attacks by the presence and willingness of a transaction to spend its balance to pay for fees according to its execution time.

ChromaWay's solution to this problem is the points system: accounts are allocated a certain amount of points that refresh periodically, and dapp developers estimate how many points a specific operation should cost based on its complexity. We believe that this system only superficially solves the blockchain-halting problem, and does not exhibit any of the three properties outlined above, for the following reasons.

1. The execution time of a Rell transaction has to be estimated ahead of time by dapp developers, likely through benchmarking. However, if the parameters of a function affect its execution time, the benchmarking process must consider only the worst case when determining how many points a transaction should cost. The worst-case

execution time likely takes many orders of magnitude more time to execute than the average case, drastically reducing the throughput of the blockchain. If the average transaction cost is used instead of the worst case, an attacker may launch a DoS attack against the chain by creating worst-case transactions.

2. Since the point-cost of a Rell operation is static, there is no way to account for the increase in execution time when there are more rows in the database to scan. This means that dapp developers may have to periodically update their dapp's point costs based on the dapp's utilization.

3. Since points and rate limits are refreshed periodically, an attacker's potential attack throughput is limited only by the number of accounts they are able to create. A theoretical attacker may leverage this to fill every block with transactions up to their limit, preventing users from using the dapp. Without a monetary cost associated with points, dapp developers would have to implement some other kind of anti-sybil mechanism, such as KYC.

   Other kinds of anti-sybil mechanisms, such as IP-detection, browser fingerprinting, Captcha, email verification, and SMS verification, are unlikely to offer meaningful protection, as malicious entities can bypass these measures at a trivial cost.

Since this finding is technically out of scope for this engagement, its severity is rated as undetermined.

Recommendations

Short term, ensure that point estimates are generated using a benchmarking of the worst case scenario for each operation. In addition, ensure that the point limit for each block ensures that the block execution time consumes only a fraction of the consensus's round time—preferably one tenth of the round time. For a blockchain that produces one block every ten seconds, this means that block execution should never take longer than one second in the average case.

Long term, introduce a method through which Rell transactions may be metered and paid for using a fee token with monetary value. The metering of Rell code does not present a challenge in itself since it is run in an interpreter; it is the metering of Postgres queries that is a problem to estimate. Several options for metering Postgres resource consumption are outlined below:

1. Use Postgres query plans to estimate the time complexity of a given SQL query, along with the number of rows that must be scanned in the worst case.

   This is how many cloud providers handle billing for their hosted database products. Google's BigQuery bills based on the amount of data scanned to execute a query,

charging a static amount per terabyte of I/O consumed by the query.

2. Compile Postgres and the Rell interpreter into Arbitrum Stylus, and use its ink-metering system to determine transaction cost. Stylus is a blockchain-based interpreter for WASM programs. While Stylus is intended to be used for Arbitrum's optimistic rollup offering, there is no reason it cannot be forked for ChromaWay's needs. It should be noted that at this time, Arbitrum Stylus is a very early-stage project, and if pursuing this option, ChromaWay may have to make substantial changes to Stylus to allow Postgres to run within it.

3. Remove Postgres as a dependency and implement the Rell SQL API as a custom, levelDB-based database. This should be considered an option of last resort. Implementing even a feature-constrained database is no small feat, but doing so would allow the execution of arbitrary queries to be metered accurately.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

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

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Issues

This appendix contains issues that do not have immediate or obvious security implications. However, they may facilitate exploit chains targeting other vulnerabilities or may become easily exploitable in future releases.

- **Unused `provider_null` variable in `get_null_provider_account` function** (see figure C.1). This variable should be used or removed.

```
function get_null_provider_account() {
    // Relies on code that runs at initialisation
    val ft4_null = ft4.accounts.account @ { NULL_ACCOUNT_ID };
    val provider_null = provider @ {.pubkey == NULL_ACCOUNT_ID};
    return provider_account @ { .account == ft4_null };
}
```

Figure C.1: `get_null_provider_account()` function (`directory-chain/src/economy_chain/economy_chain_staking_helpers.rell:14-19`)

- **Overcomplicated database query in `cm_get_cluster_info` query** (see figure C.2). In particular, the `cluster.name == name` expression will always evaluate to `true`. This query can be simplified to `cluster_node @* { cluster } ( … )`.

```
query cm_get_cluster_info(name): cm_cluster_info {
    val cluster = require_cluster(name);
    val cac = require(cluster_anchoring_chain @? { cluster }, "Cluster anchoring chain not
found for cluster " + name);
    val cluster_peer = cluster_node @* {
        cluster_node.cluster == cluster,
        cluster.name == name
    } (
        peer = .node.pubkey,
        peer_api_url = .node.api_url
    );
```

Figure C.2: Snippet of `cm_get_cluster_info()` query; overcomplicated database query is highlighted (`directory-chain/src/src/cm_api/module.rell:21-30`)

- **Unused EBFT message topics**. SIG(3) and BLOCKDATA(5).

- **Important TODOs**.

```
private fun handleTransaction(message: Transaction) {
    // TODO: reject if queue is full
    CompletableFuture.runAsync {
        withLoggingContext(loggingContext) {
            val tx =
blockchainConfiguration.getTransactionFactory().decodeTransaction(message.data)
            workerContext.engine.getTransactionQueue().enqueue(tx)
```

```
        }
    }
}
```

Figure C.3: Snippet of `handleTransaction()` function (core/postchain/postchain-base/src/main/kotlin/net/postchain/ebft/syncmanager/validator/ValidatorSyncManager.kt#L273-281)

- **Conditions are executed unnecessarily**. The `if` conditions should be swapped so the first one is executed only if needed.

```
if (this.maybeLegacy != isLegacy) {
    if (logger.isDebugEnabled) {
        logger.debug("Setting new fast sync peer status maybeLegacy: $isLegacy")
    }
}
```

Figure C.4: Snippet of `maybeLegacy()` function (core/postchain/postchain-base/src/main/kotlin/net/postchain/ebft/syncmanager/common/KnownState.kt#L156-160)

- **Incorrect comments.**
  - "@return true if all blocks we had could fit in the block" should be "@return true if all blocks we had could fit in the ***packet***".
  - "Unsuccessful response to [MsQueryRequest] or [MsBlockAtHeightRequest]." should be "Unsuccessful response to [MsQueryRequest], [MsBlockAtHeightRequest] ***or [MsBlocksFromHeightResponse]***."
- `MAX_QUEUED_PACKETS` constant is unused.
- **Statements try to check the same thing but use different cardinalities**.

```
require(voter_set_member @? { proposal.voter_set, provider }, provider.pubkey + " must be a
member of the voter set");
```

Figure C.5: Snippet of `internal_vote()` function (core/directory-chain/src/proposal/voting/apply.rell#L3)

```
require(exists(voter_set_member @* { voter_set, provider}), "Provider is not a member of
voter set " + voter_set.name);
```

Figure C.6: Snippet of `require_voter_set_member()` function (core/directory-chain/src/common/require.rell?#L61-63)

- **`after_replica_node_removed_from_cluster` function is called before removing the node**. Swap the statements.

```
function _remove_replica_node_from_cluster_internal(cluster, node) {
    val crn = cluster_replica_node @? { cluster, node };
    if (exists(crn)) {
        after_replica_node_removed_from_cluster(node, cluster);
        delete crn;
    }
}
```

Figure C.7: `_remove_replica_node_from_cluster_internal()` function (core/directory-chain/src/common/cluster.rell#L96-102)

- **Call to `try_apply_proposal` function in `retract_vote` not needed**. The proposal result cannot change when retracting a vote, but only when making a vote.
- **Unused variable**. The `vs` local variable is never used.

```
function remove_container_and_voter_set(container) {
    if (empty(is_container_available_for_removal(container))) {
        val vs = container.deployer;
        remove_container_impl(container);
    }
}
```

Figure C.8: `remove_container_and_voter_set()` function (core/directory-chain/src/common/container.rell#L62-67)

D. Testing Guidance and Recommendations

This appendix provides general recommendations on improving processes and enhancing the quality of the ChromaWay's Kotlin test suite.

Identified Testing Deficiencies

We identified several issues during the engagement that could have been prevented by a more thorough test suite (TOB-CHROMAWAY-1, TOB-CHROMAWAY-4, TOB-CHROMAWAY-6).

In addition, the unit test branch coverage of the EBFT and Networking components was measured at 46% and 15%, respectively, which is unusually low for a blockchain node project. For blockchain node projects, we normally recommend 60% branch coverage at absolute minimum, with 75-95% coverage recommended for critical code paths and more mature projects.

While reviewing the project, we also identified some issues with code complexity management that may make high quality tests much harder to write.

Plan for Remediation

1. Refactor the system to make it more testable.

Kotlin is a unique language in that it combines the properties of object-oriented languages with functional programming. However, this can make it much harder to write easily testable code. We recommend that ChromaWay refactors its Kotlin codebase to respect the following coding standards to help keep the code easily testable and maintainable.

Use pure functions wherever possible, even at the expense of OOP patterns

"Pure" functions are functions that do not mutate or view any external state; instead, they simply accept parameters and return values. Pure functions are extremely easy to test because it is very clear what data has to be mocked to test them. The use of pure functions often comes at the expense of OOP design patterns; functions within a class are encouraged to be tightly coupled with the class's state and to be hidden from external callers.

One good design rule that can be used to weigh whether a function should be pure is that if a function does not write to the class's state, it should be converted to a pure function. This is doubly applicable if the function reads only from publicly accessible class variables. One good example is the `countNodes` function, shown in figure D.1.

```
private fun countNodes(state: NodeBlockState, height: Long, blockRID: ByteArray?):
Int {
```

```
    var count = 0
    for (ns in nodeStatuses) {
        if (ns.height == height && ns.state == state) {
            if (blockRID == null) {
                if (ns.blockRID == null) count++
            } else {
                if (ns.blockRID != null && ns.blockRID.contentEquals(blockRID))
                    count++
            }
        }
    }
    return count
}
```

Figure D.1: The `countNodes` function does not mutate any class-internal state, and is thus a good candidate for conversion to a pure function. (postchain/postchain-base/src/main/kotlin/net/postchain/ebft/BaseStatusManager.kt#63–76)

Avoid using lambda expressions unless necessary

Lambda expressions are often very challenging to test since they are not directly accessible from the test suite, and they are often declared in parts of the codebase that are "far away" from where they are actually used (and need to be tested). We observed that in some cases, ChromaWay uses lambda expressions for code organization, making the code harder to test and less readable with no discernible benefit, as shown in the `shouldRecomputeStatusAgain` function in figure D.2.

By breaking lambda expressions into separate second-class functions, they can be directly tested by the test suite and even be converted into pure functions in some scenarios.

```
fun shouldRecomputeStatusAgain(): Boolean { // Not private bc unit test

    fun resetBlock() {
          [...]
    }

    fun potentiallyDoSync(): FlowStatus {
          [...]
    }

    fun potentiallyRevolt(): FlowStatus {
          [...]
    }

    fun handleHaveBlockState(): Boolean {
          [...]
    }

    fun handlePreparedState(): Boolean {
```

```
            [...]
    }

    fun handleWaitBlockState(): Boolean {
            [...]
    }

    if (myStatus.state != NodeBlockState.Prepared) {
        when (potentiallyDoSync()) {
            FlowStatus.Break -> return false
            FlowStatus.Continue -> return true
            FlowStatus.JustRunOn -> Unit // nothing, just go on
        }
    }
    [...]
}
```

Figure D.2: The `shouldRecomputeStatusAgain` function relies heavily on lambdas to implement the EBFT state machine, but these individual functions cannot be individually tested because they are lambda expressions.
(postchain/postchain-base/src/main/kotlin/net/postchain/ebft/BaseStatusManager.kt#371–601)

2. Create function-level unit tests that test the behavior of each individual function

These tests fulfill the role of traditional unit tests: they ensure that your CI screams if you make a change that critically breaks something. When writing these tests, the goal is to obtain a breadth of coverage, preferably at least 60% branch coverage.

When writing these unit tests, you should always test the "bad case" wherever possible. For example, if you have a function that verifies signatures, you should have at least two test cases for it: one that verifies a correct signature, and one that verifies the function will catch an incorrect signature.

This may be especially challenging for the networking component. If the networking component was not written with testability in mind, it may be challenging to refactor to make it testable. Depending on how soon ChromaWay will migrate to libp2p, it may make more sense to skip breadth-of-testing coverage for the networking component, since much of the code will be removed for libp2p.

3. Create end-to-end tests that verify the properties and expected behavior of the protocol as a whole

This task helps ensure the stability of the protocol overall, not just test coverage. One example of an end-to-end test could be a regression test for TOB-CHROMAWAY-7: a regression test for this finding will require setting up multiple nodes with very specific configurations to test the scenario.

It is important to avoid accidentally inflating test coverage measurements with end-to-end tests. It is very easy to obtain a very high coverage using end-to-end tests, when in reality, they adequately test very few code paths.