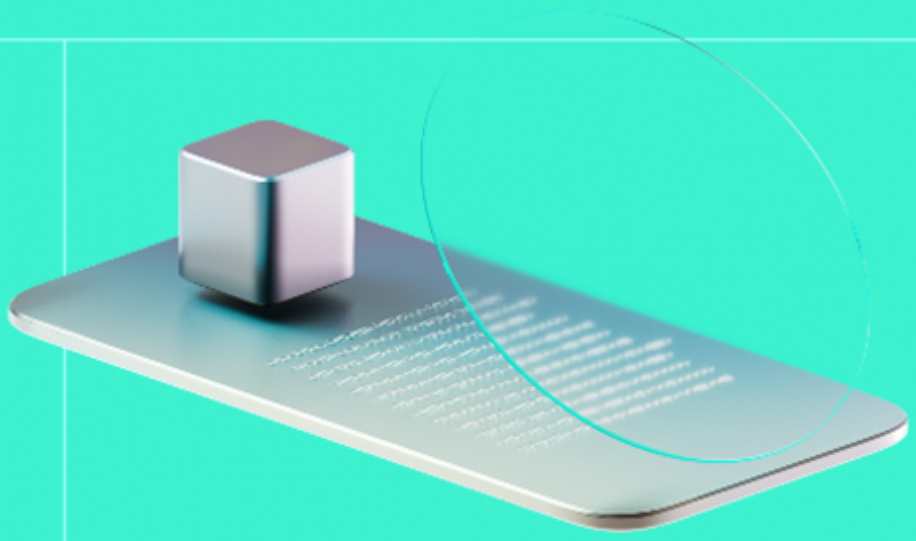
HACKEN

Blockchain Protocol Security Analysis Report

Customer: Analog

Date: 29/03/2024



We express our gratitude to the Analog team for the collaborative engagement that enabled the execution of this Security Assessment.

Analog is an innovative platform that introduces a groundbreaking approach to interoperability in the Web 3.0 ecosystem.
It enables this advanced interoperability through the use of its unique GMP (Generic Message Passing) protocol, facilitating seamless communication and data exchange across various blockchain networks.
Analog secures all operations and data through its Timechain distributed ledger, ensuring a high level of security and trustworthiness. This combination of innovative interoperability and robust security positions Analog at the forefront of developing decentralized solutions and applications for the future web.

Platform: Analog

Language: Rust

Tags: Substrate, Threshold Signature Scheme, Interoperability

Timeline: 22/11/2023 - 29/03/2024

Methodology: Blockchain Protocol and Security Analysis Methodology

Review Scope

Repository	https://github.com/Analog-Labs/testnet/
Commit	3ba97bde46eac298fd61eba7ff5b5ef0078a3ebe

Audit Summary

10/10	9/10	8/10	6/10
Security Score	Code quality score	Architecture quality score	Documentation quality score

Total 9.3/10

The system users should acknowledge all the risks summed up in the risks section of the report

15	13	1	0
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Critical	0
High	1
Medium	1
Low	13

Vulnerability	Status
F-2024-0447 - Panic in Shards Pallet due to Insufficient Error Handling in Group Commitment Computation	Mitigated
F-2024-0515 - Integration of Custom P2P Networking Library	Accepted
F-2023-0158 - Utilization of Non-Cryptographically Secure fastrand Crate	Fixed
F-2023-0176 - Replay Attack and D.O.S. Vulnerability in submit_error Extrinsic	Fixed
F-2023-0213 - Predictability Concern in random_signer Function Due to Hybrid Random-Round Robin Approach	Fixed
F-2023-0216 - Validators Influence on Random Signer Selection	Fixed
F-2023-0241 - Absence of Task State Validation	Fixed
F-2023-0244 - Lack of Shard Validation in Task Execution	Fixed
F-2023-0301 - Lack of Phase Validation When Submitting Task Results/Errors	Fixed
F-2023-0318 - Inconsistent Management of `TaskPhaseState` Across Task Cycles	Fixed
F-2024-0445 - Unencrypted Storage of Chronicle's Secret Share	Fixed
F-2024-0448 - Logical Inconsistencies in Shard Status Handling	Fixed
F-2024-0507 - Validation Gap for Pending Nodes in ROAST Protocol	Fixed
F-2024-0509 - Lack of Size Limitations on Error Messages in TaskState Storage Map for Failed Transactions	Fixed
F-2024-0512 - Chronicle Crate Panic Caused by Mishandled `PeerId`	Fixed

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

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

Document

Name	Blockchain Protocol Code Review and Security Analysis Report for Analog
Audited By	Sofiane Akermoun, Nino Lipartiia, Nataliia Balashova
Approved By	Sofiane Akermoun
Website	https://www.analog.one/
Changelog	23/01/2024 - Preliminary Report
Changelog	29/03/2024 - Final Report

Table of Contents

System Overview

Timechain, based on the Substrate framework, is tasked with settling transactions received from Chronicle nodes. These transactions are subsequently processed using a Threshold Signature Scheme (TSS) among the participants, which ensures secure and efficient transaction handling.
The core components are:

1. The Timechain Node
2. The Chronicle crate
3. The TSS crate

All these components are within the scope of this audit.

Executive Summary

This report presents an in-depth analysis and scoring of the customer's blockchain protocol project. Detailed scoring criteria can be referenced in the corresponding section of the Blockchain Protocol and Security Analysis Methodology.

Documentation quality

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

- Adequate supplementary documentation was available.
- Developers offered useful explanations during the audit process.
- Additional source code documentation is needed for critical functions and core components for enhanced clarity.

Code quality

The total Code Quality score is **9** out of **10**.

- Exceptional quality standards are evident in the Rust code.
- Substrate code maintains a notably high level of quality.
- Weight and Benchmarks are implemented effectively.
- The Mocked Runtime implementation exhibits comprehensive code coverage.
- Presence of TODO comments in the code.

Architecture quality

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

- Employment of the Substrate framework as the foundational infrastructure for the blockchain.
- Effective interaction between Chronicle nodes and Timechain nodes, facilitated by a Threshold Signature Scheme.
- Approach to achieving interoperability, characterized by its scalability through the addition of protocols and ease of upgrade.
- Centralization aspects of the Chronicle node, while reducing the attack surface, raise considerations regarding the overall system's robustness.

Security score

Upon auditing, the code was found to contain **0** critical, **1** high, **1** medium, and **13** low severity issues.
All security challenges were effectively resolved, securing a top-notch security rating of **10** out of **10**. Two minor issues were identified but accepted and mitigated, as they present well-defined, low risks, aligning with rigorous risk management standards.

All identified issues are detailed in the “Findings” section of this report.

Summary

The comprehensive audit of the customer's blockchain protocol yields an overall score of **9.3** out of **10**. This score reflects the combined evaluation of documentation, code quality, architecture quality, and security aspects of the project.

Findings

Vulnerability Details

F-2023-0176 - Replay Attack and D.O.S. Vulnerability in submit_error Extrinsic - High

Description:

The `submit_error` extrinsic within the `tasks` pallet presents a significant security risk. This vulnerability originates from the absence of validation against the resubmission of identical error signatures.
As a result, it opens up the potential for a replay attack, where the same error signature can be repeatedly submitted by malicious actors.
This repetitive submission not only undermines the integrity of the system but also triggers a Denial of Service (D.O.S.) by forcing tasks into a failure state prematurely.

Replay attack:
The signature verification process, as it currently stands, does not utilize a nonce or any other method to guarantee the uniqueness of each submission.
This absence of a uniqueness check allows the same signature to be used repeatedly without being invalidated after its first use as shown in the following code:

```
/// Submit Task Error
#[pallet::call_index(4)]
#[pallet::weight(T::WeightInfo::submit_error())]
pub fn submit_error(
origin: OriginFor<T>,
task_id: TaskId,
cycle: TaskCycle,
error: TaskError,
) -> DispatchResult {
ensure_signed(origin)?;
ensure!(Tasks::<T>::get(task_id).is_some(), Error::<T>::UnknownTask);
Self::validate_signature(
task_id,
cycle,
error.shard_id,
schnorr_evm::VerifyingKey::message_hash(error.msg.as_bytes()),
error.signature,
)?;
// ...
```

In the implementation, the signature is created based on public parameters such as the task ID, task cycle, error shard ID, and the error message, which are all visible when the extrinsic is called.
However, crucially, this process does not incorporate a nonce or any other unique identifier to ensure the uniqueness of each transaction.
The absence of a nonce in the signature generation means that the signature remains valid for multiple transactions as long as the public parameters remain the same.
In a blockchain environment, where transaction details are publicly visible on the chain, this aspect of transparency becomes a double-edged sword. While it upholds the blockchain's principle of openness, it also means that any user can view the details of transactions, including those that call the `submit_error` extrinsic.

Denial of Service:
By exploiting the replay attack vulnerability, as previously described, a malicious actor can repeatedly submit the same `submit_error` extrinsic with a valid signature.
Each successful execution of the `submit_error` extrinsic increments the `TaskRetryCounter` for the specified task. This counter tracks the number of times an error has been submitted for a task.
In the runtime configuration, `type MaxRetryCount = ConstU8<3>` specifies that the maximum retry count for a task is set to 3. This is a critical threshold value in the

context of task execution.
When the `TaskRetryCounter` reaches the maximum retry count of 3, as defined in the runtime configuration, the task's state is automatically set to `Failed`. This is enforced by the line `TaskState::<T>::insert(task_id, TaskStatus::Failed { error: error.clone() });` in the `submit_error` extrinsic.
Once the retry counter hits this limit, the task is marked as failed regardless of the actual validity or severity of the reported errors.

Consequences::

- **Premature Task Failure**: A malicious actor can cause a task to fail prematurely by artificially inflating the retry count. This is achieved by repeatedly submitting the same error signature until the retry count threshold is reached.
- **Disruption of Normal Operations**: Such premature failures of tasks can disrupt the normal operations of the blockchain system, leading to a denial of service for those particular tasks.
- **Undermining System Reliability**: Repeated occurrences of such incidents can erode trust in the system's reliability, as legitimate tasks may be unjustly terminated due to this vulnerability.

Assets:

- Runtime & Pallets

Status: Fixed

Classification

Severity: High

Impact: 5/5

Likelihood: 3/5

Recommendations

Recommendation: To address the Denial of Service (D.O.S.) vulnerability in the `submit_error` extrinsic, incorporating the `TaskRetryCounter` as part of the signature presents a promising solution.
This approach effectively uses the `TaskRetryCounter` as a nonce, adding a unique element to each transaction and thereby mitigating the risk of replay attacks. Here's a detailed recommendation:

Incorporate `TaskRetryCounter` into Signature:

- **Unique Transaction Identifier**: Modify the signature generation process to include the current value of the `TaskRetryCounter` for the task. Each time an error is reported and the `submit_error` extrinsic is called, the `TaskRetryCounter` is incremented. By including this incremented value in the signature, each submission becomes unique.
- **Preventing Replay Attacks**: This change ensures that a signature used in a previous submission cannot be reused for a new submission, as the `TaskRetryCounter` part of the signature would differ. This effectively prevents the possibility of replaying the same transaction.

Evidences

Proof of Concept

Reproduce:

The following Rust code is a Proof of Concept (PoC) specifically designed to demonstrate a replay attack vulnerability in the `submit_error` extrinsic.
This PoC is based on a test scenario originally developed by the Analog team to demonstrate the vulnerability discovered during the audit process.
It effectively simulates the scenario where a malicious actor can repeatedly submit the same error, leading to an unwarranted task failure. This PoC is instrumental in illustrating the vulnerability and confirming its presence in the system.

To run this PoC, follow these steps:

1. Copy the entire PoC script provided below.
2. Paste the script at the bottom of the `pallets/tasks/src/tests.rs` file. This is the appropriate location in the Substrate pallet where the `submit_error` extrinsic is defined and tested.
3. Execute the test using the following command in your terminal:
   cargo test -p pallet-tasks poc_submit_task_error_replay_sig
4. If the test passes (i.e., the assertions in the test are true), it confirms the existence of the vulnerability. The test is crafted to pass if it successfully shows that the task's status is incorrectly set to `Failed` after the repetitive submission of the same error.

```
#[test]
fn poc_submit_task_error_replay_sig() {
// Start a new test environment. This is a simulated blockchain environment
// provided by the testing framework.
new_test_ext().execute_with(|| {

// Step 1: Initialize the environment and create a task.
// This step involves bringing a shard online and creating a task
// within the Ethereum network context.
Tasks::shard_online(1, Network::Ethereum);
assert_ok!(Tasks::create_task(
RawOrigin::Signed([0; 32].into()).into(),
mock_task(Network::Ethereum, 1)
));

// Step 2: Prepare a mock error and submit it
```

See more

Results:

Output:

```
running 1 test
test tests::poc_submit_task_error_replay_sig ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 37 filtered out; fini
shed in 0.01s
```

F-2024-0448 - Logical Inconsistencies in Shard Status Handling - Medium

Description:

The existing implementation encounters several challenges in managing shard states, particularly concerning the `ShardStatus::Committed` state.

The primary issue arises when all members commit to the shard, transitioning it to the `Committed` state. In this scenario, if at least one member fails to call `ready`, the shard remains indefinitely in the committed state, hindering its progression to an online status. Additionally, the shard does not initiate an offline state, leaving other members trapped in a dysfunctional shard.

Another significant concern involves the absence of state changes in `ShardStatus::Committed` when members go online or offline. The functions responsible for these transitions, namely `online_member` and `offline_member`, lack the necessary logic to ensure accurate reflections of individual member states in the overall shard status.

The relevant code snippets for these functions are provided below:

primitives/src/shard.rs:107

```
pub fn online_member(&self) -> Self {
match self {
ShardStatus::PartialOffline(count) => {
let new_count = count.saturating_less_one();
if new_count.is_zero() {
ShardStatus::Online
} else {
ShardStatus::PartialOffline(new_count)
}
},
_ => *self,
}
}

pub fn offline_member(&self, max: u16) -> Self {
match self {
ShardStatus::PartialOffline(count) => {
let new_count = count.saturating_plus_one();
if new_count > max {
ShardStatus::Offline
} else {
ShardStatus::PartialOffline(new_count)
}
},
// if a member goes offline before the group key is submitted,
// then the shard will never go online
ShardStatus::Created(_) => ShardStatus::Offline,
ShardStatus::Online => {
if max.is_zero() {
ShardStatus::Offline
} else {
ShardStatus::PartialOffline(1)
}
},
_ => *self,
}
}
```

It is crucial to highlight the presence of logic monitoring the number of offline members in `ShardStatus::PartialOffline(u16)`. When this count reaches a certain threshold, the shard transitions to an offline state. However, a notable deficiency exists in this logic. Specifically, if a member goes offline when the shard is `ShardStatus::Committed`, the `shards` pallet fails to capture this change.

This scenario poses a vulnerability: a member can invoke the `ready` function on a committed shard and later go offline while the shard remains committed. Consequently, the shard may transition to `ShardStatus::Online` even when the member is offline, causing a mismatch in the count of offline members reflected in the shard's state. Subsequent transitions to `PartialOffline(n)` will yield inaccurate values due to members going offline during the committed state. This inconsistency delays the

shard's move to the offline state and hinders a precise assessment of the functional member count, leading to notable disruptions in the pallet's logic.

Assets:

- Runtime & Pallets

Status: Fixed

Classification

Severity: Medium

Impact: 2/5

Likelihood: 2/5

Recommendations

Recommendation: To address this issue effectively, it is advisable to reevaluate the logic governing shard status, introducing a validation step when the shard is in the committed state. One potential solution involves preventing a shard from transitioning to the online state if any member goes offline by immediately making the shard go offline. Alternatively, a systematic mechanism for tracking the count of offline members during the committed state could be implemented. This count could then be leveraged to facilitate a transition to either `ShardStatus::Online` or `ShardStatus::PartialOffline(n)`, ensuring an accurate representation of disconnected members.

This refinement aims to fortify security measures by guaranteeing a precise depiction of shard states. It mitigates potential disruptions in the pallet's logic arising from discrepancies in member counts during state transitions.

F-2023-0158 - Utilization of Non-Cryptographically Secure fastrand Crate - Low

Description:

The `fastrand` crate, currently used in the codebase, is explicitly stated as non-cryptographically secure in its GitHub documentation.
This is an issue for blockchain applications, where cryptographic security is non-negotiable.
The README.md of the `fastrand` crate found here clearly mentions the use of `Wyrand`, which, while efficient and fast, does not meet cryptographic security standards.
Therefore, the use of `fastrand` in any blockchain-related codebase is inappropriate and poses a security risk.

Assets:

- Cryptography and Keys
- Dependencies

Status: Fixed

Classification

Severity: Low

Impact: 1/5

Likelihood: 1/5

Recommendations

Recommendation:

Before exploring external crates for secure random number generation, it is advisable to first consider a Substrate idiomatic way for generating randomness. Substrate's framework offers tailored solutions for randomness within blockchain environments, particularly aligning with the deterministic and consensus-driven nature of blockchains.

Substrate provides the `Randomness` trait, which is central to both generating and consuming randomness in a blockchain context. When implementing randomness in a Substrate-based blockchain, there are two primary options within the Substrate framework to consider:

- **Insecure Randomness Pallet**: This pallet offers a function to generate pseudo-random values based on the block hashes of the previous 81 blocks. While this method provides efficient performance, it is important to note that it is not secure. This type of randomness is suitable for scenarios with low security requirements or for testing purposes in randomness-consuming applications. It is not recommended to use this pallet in a production environment where security is a concern.
- **BABE Pallet for Randomness**: The BABE (Blind Assignment for Blockchain Extension) pallet offers a more secure alternative by using verifiable random functions (VRFs) for randomness generation. This pallet provides production-grade randomness and is utilized in the Polkadot network. Opting for the BABE pallet implies that your blockchain should employ the BABE slot-based consensus mechanism for block production. This method is recommended for environments where security and trust are paramount.

In summary, when integrating randomness into a Substrate-based blockchain, it is crucial to align the method of randomness generation with the overall security requirements and architecture of your blockchain.
For secure, production-grade environments, the BABE pallet is the recommended

choice, while the insecure randomness pallet can be used for lower-security applications or testing stages.
For more information please read: https://docs.substrate.io/build/randomness/

External References:

- fastrand Github
- Substrate randomness

F-2023-0213 - Predictability Concern in random_signer Function Due to Hybrid Random-Round Robin Approach - Low

Description:

The `random_signer` function in `shards` pallet selects a signer from a pool of members using a hybrid approach combining random selection and a round-robin method.

While this approach ensures that all members are eventually selected, it introduces a bias in the probability of selection, especially noticeable when only a few members remain unchosen.

The function iteratively checks each member from left to right in the `members` array to find an eligible signer.

An eligible signer is defined as a member who has not yet signed, and therefore, is not listed in the `PastSigners` storage value.

The starting point for this check is determined by a randomly generated index (`signer_index`).
pallets/shards/src/lib.rs:340:

```
let mut rng = fastrand::Rng::with_seed(seed);
let members = Self::get_shard_members(shard_id);
let mut signer_index = rng.usize(..members.len());
let mut signer = T::Members::member_public_key(&members[signer_index].0)
.expect("All signers should be registered members");
if members.len() == 1 {
// only one possible signer for shard size 1
return signer;
}
if PastSigners::<T>::iter_prefix(shard_id).count() < members.len() {
while PastSigners::<T>::get(shard_id, &signer).is_some() {
signer_index = if signer_index == members.len() - 1 { 0 } else { signer_index + 1
};
signer = T::Members::member_public_key(&members[signer_index].0)
.expect("All signers should be registered members");
}
}
PastSigners::<T>::insert(shard_id, &signer, ());
signer
```

- The `signer_index` is incremented to move to the next member in the array. If it reaches the end of the array (`members.len() - 1`), it wraps around to the beginning (index 0). This ensures that all members are considered.
- For each new index, the public key of the member at that index is fetched to check if they are in `PastSigners`.
- The loop continues until it finds a member who is not in `PastSigners`, making them an eligible signer.

This iteration process, while ensuring that all members are eventually considered, introduces a bias.
Members who have a greater number of already selected members (listed in `PastSigners`) preceding their position in the `members` array tend to have a higher probability of being selected.
This bias could potentially impact the fairness of the selection process, particularly in scenarios where an unbiased and evenly distributed probability of selection is required.

Assets:

- Runtime & Pallets

Status: Fixed

Classification

Severity: Low

Impact: 1/5

Likelihood: 1/5

Recommendations

Recommendation: To address the bias in the selection process of the `random_signer` function and enhance its randomness, a more effective approach would be to create a separate array of eligible signers.
This array should include only those members who have not yet signed and are not present in the `PastSigners` storage value.
Once this array of eligible signers is prepared, the randomness can be applied directly to this array. Here's how this can be implemented:

```
fn random_signer(shard_id: ShardId) -> PublicKey {
let seed = u64::from_ne_bytes(
frame_system::Pallet::<T>::parent_hash().encode().as_slice()[0..8]
.try_into()
.expect("Block hash should convert into [u8; 8]")
);
let mut rng = fastrand::Rng::with_seed(seed);
let members = Self::get_shard_members(shard_id);

// Handle the case where there is only one member
if members.len() == 1 {
return T::Members::member_public_key(&members[0].0)
.expect("Single member should have a valid public key");
}

let eligible_signers = members
.iter()
.filter(|member| !PastSigners::<T>::contains_key(shard_id, member))
.collect::<Vec<_>>();

if !eligible_signers.is_empty() {
let signer_index = rng.usize(..eligible_signers.len());
let signer = T::Members::member_public_key(&eligible_signers[signer_index].0)
.expect("All signers should be registered members");

// Add the selected signer to PastSigners
PastSigners::<T>::insert(shard_id, &eligible_signers[signer_index].0, ());

signer
} else {
// Handling case based on decision from observation F-2023-0214
// If all members have already signed, the approach to handle this situation need
s to be defined
// based on the decision from observation F-2023-0214
// This could involve selecting a random member from the entire list or returning
a specific error/default value
panic!("Based on observation F-2023-0214, handling of no eligible signers needs t
o be defined");
}
}
```

Key Points in This Revision:

- **Single Member Check**: The function first checks if there is only one member in the shard. If so, it directly returns this member as the signer, bypassing any need for random selection.
- **Random Selection for Multiple Members**: If more than one member exists, the function then constructs an array of eligible signers (those not in `PastSigners`) and randomly selects one of them.
- **Updating** `PastSigners`: After a signer is selected, they are added to `PastSigners`. This ensures that members are not repeatedly chosen before others have had a chance to sign.
- **Handling No Eligible Signers**: The function now includes a placeholder for handling the case where there are no eligible signers left. The specific behavior in this scenario should be defined based on the decision from observation **F-2023-0214.**

This could involve repeating the selection from all members or handling the situation in a different manner, as per the system's requirements.

This revision aligns the function's behavior with the observations and recommendations, ensuring a balanced and fair signer selection process while also providing a clear direction for handling edge cases.

F-2023-0216 - Validators Influence on Random Signer Selection - Low

Description:

The `random_signer` function within `shards` pallet demonstrates a potential vector for influencing the random number generator (RNG) seed, which could affect the fairness of signer selection.
This influence stems from the ability of validators to choose and order transactions within a block they produce.
pallets/shards/src/lib.rs:325

```
fn random_signer(shard_id: ShardId) -> PublicKey {
let seed = u64::from_ne_bytes(
frame_system::Pallet::<T>::parent_hash().encode().as_slice()[0..8]
.try_into()
.expect("Block hash should convert into [u8; 8]"),
);
let mut rng = fastrand::Rng::with_seed(seed);
...
}
```

In this function, the RNG seed is derived from the hash of the parent block.
Validators, when creating a new block, can manipulate the set and order of transactions.
Since these transactions are part of the data that generates the block hash, they indirectly influence the RNG seed used in `random_signer`.

The potential for exploiting this issue is generally low under the BABE consensus mechanism, given its inherent randomness in selecting validators.
However, this likelihood may increase with other consensus models, especially those where validators have greater predictability or control in producing blocks.
This consideration is particularly relevant for the Timechain blockchain if it opts to use alternative consensus mechanisms that give more weight to the intentions and actions of validators in the block creation process.

Assets:

- Cryptography and Keys
- Runtime & Pallets

Status: Fixed

Classification

Severity: Low

Impact: 1/5

Likelihood: 1/5

Recommendations

Recommendation:

To mitigate the risk of validators influencing the random number generator (RNG) in the `random_signer` function, it is crucial to utilize a more robust and less manipulable source of randomness.
The Substrate framework, particularly the BABE consensus mechanism, offers features for generating on-chain randomness that can serve this purpose more effectively than relying solely on the previous block's hash.

The BABE pallet includes mechanisms to accumulate randomness over time, combining data from various blocks to produce a more unpredictable and secure random number.
This method reduces the potential impact any single validator can have on the RNG

outcome. Integrating this enhanced source of randomness into the `random_signer` function would increase its fairness and resilience against potential manipulation.

Refer to the recommendations in issue **F-2023-0158** for additional details on implementing and integrating these features. This approach aligns with best practices for ensuring the integrity of random number generation within blockchain networks, especially in scenarios where unbiased and unpredictable outcomes are critical.

F-2023-0241 - Absence of Task State Validation - Low

Description:

The tasks pallet is susceptible to a vulnerability arising from the absence of essential checks to ensure that a task is not in a stopped, failed, or completed state before any data pertaining to the task is submitted.

Within the `tasks` pallet, the `TaskState` map is responsible for storing task statuses, including options such as:

```
pub enum TaskStatus {
Created,
Failed { error: TaskError },
Stopped,
Completed,
}
```

The identified issue stems from a lack of status validation in key extrinsic functions like `submit_result`, `submit_error`, `submit_hash`, and `submit_signature`. Despite the pivotal role these functions play in the task lifecycle, the absence of a task status check allows operations on tasks marked as `Stopped`, `Failed`, and `Completed`. This not only renders the stopping and resuming of tasks ineffective but also undermines the notion of a completed task, as any task can be perpetually executed even after being marked as `Completed`. This disruption significantly impacts the overall integrity of the task lifecycle.

This vulnerability is compounded by the absence of shard validation, allowing shards not assigned to a specific task to invoke `submit_result` and `submit_error` (see *F-2023-0244*). Although these functions still necessitate a valid shard signature, as verified by the `validate_signature` function, the likelihood of misbehaviors due to the lack of task state validation is significantly increased.

A noteworthy issue is the absence of signer validation in the `submit_signature` function. This allows anyone to submit a signature for a task in the `Sign` phase without the necessity of a valid shard signature or being a member of the shard. This flawed logic heightens the risk of potential malicious activities, increasing the security concern related to the absence of task state validation.

Assets:

- Runtime & Pallets

Status: Fixed

Classification

Severity: Low

Impact: 2/5

Likelihood: 2/5

Recommendations

Recommendation:

To address this vulnerability effectively, it is strongly recommended to implement a robust validation check within the specified functions. This check should ensure that the task is in an active state before permitting any further operations. Such a measure will significantly enhance the security and reliability of the tasks pallet, contributing to the overall stability of the blockchain network.

Evidences

Reproduce:

A series of tests has been conducted to illustrate that submissions on tasks are allowed when they should not be. These tests demonstrate potential vulnerabilities across various scenarios.

```
// The test checks that `submit_result` and `submit_error` can be called on the s
topped task
#[test]
fn poc_task_status_vulnerability_stopped() {
new_test_ext().execute_with(|| {
assert_ok!(Tasks::create_task(
RawOrigin::Signed([0; 32].into()).into(),
mock_task(Network::Ethereum, 1)
));

// Stop the task
assert_ok!(Tasks::stop_task(RawOrigin::Signed([0; 32].into()).into(), 0));
assert_eq!(TaskState::<Test>::get(0), Some(TaskStatus::Stopped));

// submit_result and submit_error don't fail, even though the task is stopped
assert_ok!(Tasks::submit_result(
RawOrigin::Signed([0; 32].into()).into(),
0,
0,
mock_result_ok(1, 0, 0)
));
assert_ok!(Tasks::submit_error(
RawOrigin::Signed([0; 32].into()).into(),
0,
1,
mock_error_result(1, 0, 1)
));
});
}

// The test checks that `submit_result` and `submit_error` can be called on the f
ailed task
#[test]
fn poc_task_status_vulnerability_failed() {
let mock_error = mock_error_result(1, 0, 0);
new_test_ext().execute_with(|| {
assert_ok!(Tasks::create_task(
RawOrigin::Signed([0; 32].into()).into(),
mock_task(Network::Ethereum, 1)
));
Tasks::shard_online(1, Network::Ethereum);

// Fail the task
for _ in 1..=10 {
assert_ok!(Tasks::submit_error(
RawOrigin::Signed([0; 32].into()).into(),
0,
0,
mock_error.clone()
));
}
assert_eq!(Tasks::task_state(0), Some(TaskStatus::Failed { error: mock_error }));

// submit_result and
```

See more

F-2023-0244 - Lack of Shard Validation in Task Execution - Low

Description:

The `tasks` pallet exhibits a logical flaw, allowing any shard to invoke extrinsic functions on tasks for which it was not originally assigned.

The core issue resides in the implementation of the `submit_result` and `submit_error` functions. While the pallet includes logic for assigning tasks to specific shards by storing them in the `ShardTasks` and `TaskShard` maps, these extrinsics neglect to validate whether the executing shard is the one originally designated for the task.

Despite the implementation of checks in the `submit_result` and `submit_error` functions to ensure the correctness of the shard's signature, a crucial validation is absent, enabling any shard to call these functions for tasks not assigned to it. Consequently, a shard can submit a result or error with `TaskResult` or `TaskError` and the correct signature, leading to the execution of these transactions without error.

This flaw presents a concern for the effective management of tasks, casting doubt on the overall reliability of the system and compromising the integrity of the task execution logic. While the current centralized nature of nodes may mitigate immediate security threats, the significance of this vulnerability is expected to increase as the system transitions to a decentralized model. In a decentralized environment, where nodes can potentially act maliciously, and when considered alongside other existing flaws, this issue has the potential to significantly impact the overall integrity of the task lifecycle (see *F-2023-0241*). However, the probability of exploitation is diminished by the necessity for controlling the entire shard to carry out the attack, rendering it less feasible.

Assets:

- Runtime & Pallets

Status: Fixed

Classification

Severity: Low

Impact: 1/5

Likelihood: 1/5

Recommendations

Recommendation: To address this issue, it is strongly recommended to implement stringent checks, ensuring that each shard can only execute tasks specifically assigned to it. This proactive measure will fortify the integrity of task execution, contributing to the overall resilience and dependability of the system.

F-2023-0301 - Lack of Phase Validation When Submitting Task Results/Errors - Low

Description:

The tasks pallet reveals a logical flaw wherein the extrinsics `submit_result` and `submit_error` lack any checks on the phase of the task. This deficiency permits a shard to execute these functions during the `Write` or `Sign` phase, potentially disrupting the intended task lifecycle.

The issue stems from an inconsistency in the implementation of the task phase logic. In a typical progression, a payable task is expected to traverse the `Write` phase (with an additional `Sign` phase for tasks featuring the `SendMessage` function) before advancing to the `Read` phase through hash submission. However, this crucial sequence can be circumvented by directly executing the `submit_result` or `submit_error` functions, as these functions lack validation to ensure the task is in the correct phase.

This flaw allows a malicious shard to execute tasks without submitting the hash of the corresponding transaction.
To maintain the robustness of the implemented logic, it is crucial to have the capability to verify the accuracy and validity of the submitted hash.
Skipping the `submit_hash` step significantly disrupts the task logic and may lead to improper validation of executed tasks, jeopardizing the proper handling of tasks and casting doubt on the overall reliability of the system.

The vulnerability is further exacerbated by the absence of shard validation, enabling shards not assigned to a specific task to invoke `submit_result` and `submit_error` (see *F-2023-0244*). While these functions still require a valid shard signature, as verified by the `validate_signature` function, the likelihood of this issue being exploited is significantly increased.

It is noteworthy that the current risk level is relatively low, hinging on the centralized nature of the chronicles, where all shard members are presumed non-malicious. However, with the project's transition to a decentralized model, the gravity of this issue escalates, rendering it a more pronounced vulnerability.

Assets:

- Runtime & Pallets

Status: Fixed

Classification

Severity: Low

Impact: 3/5

Likelihood: 1/5

Recommendations

Recommendation:

To address this issue, it is strongly recommended to implement checks for the task's phase in the `submit_result` and `submit_error` extrinsics, allowing these functions to be called only when the task is in the `Read` phase. This proactive measure will fortify the integrity of task execution, ensuring tasks progress through the designated phases as intended.

Evidences

Reproduce:

The test below highlights the issue:

```
#[test]
fn poc_submit_result_error_on_write_phase() {
let a: AccountId = A.into();
new_test_ext().execute_with(|| {
// Create a new payable task
assert_ok!(Tasks::create_task(
RawOrigin::Signed(a.clone()).into(),
mock_payable(Network::Ethereum)
));
System::assert_last_event(Event::<Test>::TaskCreated(0).into());
Tasks::shard_online(1, Network::Ethereum);
// Check that the task is in the Write phase
assert_eq!(
Tasks::get_shard_tasks(1),
vec![TaskExecution::new(0, 0, 0, TaskPhase::Write(pubkey_from_bytes(A)))]
);
// submit_error doesn't return an error when it's called on this task
let task_error = mock_error_result(1, 0, 0);
assert_ok!(Tasks::submit_error(RawOrigin::Signed(a.clone()).into(), 0, 0, task_er
ror));
// The task is still in the Write phase
assert_eq!(
Tasks::get_shard_tasks(1),
vec![TaskExecution::new(0, 0, 1, TaskPhase::Write(pubkey_from_bytes(A)))]
);
// submit_result doesn't return an error when it's called on this task
let task_result = mock_result_ok(1, 0, 0);
assert_ok!(Tasks::submit_result(RawOrigin::Signed(a).into(), 0, 0, task_result.cl
one()));
System::assert_last_event(Event::<Test>::TaskResult(0, 0, task_result).into());
});
}
```

F-2023-0318 - Inconsistent Management of `TaskPhaseState` Across Task Cycles - Low

Description:

The `tasks` pallet exhibits a logical flaw in managing task phases, preventing the submission of the hash for a recurring task after its initial submission in the first cycle of task execution.

According to the `tasks` pallet logic, each payable task is expected to transition from the `Write` phase (preceded by an additional `Sign` phase for tasks incorporating the `SendMessage` function) to the `Read` phase by submitting a hash during each task cycle, as regulated by `TaskCycleState`. However, a notable bug has been observed: after the initial completion of this process, the phase remains stuck at `TaskPhase::Read`. This occurs because the `TaskPhaseState` doesn't change when either a result or error is submitted.

Consequently, in subsequent cycles, the task is incapable of reinitiating this process, constraining its functionality solely to submitting results/errors, while attempts to execute `submit_hash` and `submit_signature` prove futile. This issue disrupts the logical continuity of the task lifecycle significantly and introduces potential vulnerabilities for future disruptions.

Assets:

- Runtime & Pallets

Status: Fixed

Classification

Severity: Low

Impact: 1/5

Likelihood: 5/5

Recommendations

Recommendation:

It is strongly recommended to implement a robust logic that ensures the synchronization of the task phase with its actual state. This entails introducing mechanisms that update the `TaskPhaseState` to accurately reflect the ongoing phase of the task. Such an implementation will enhance the reliability and functionality of the tasks pallet, mitigating the observed bug and fostering a seamless progression through task cycles.

F-2024-0445 - Unencrypted Storage of Chronicle's Secret Share - Low

Description:

A vulnerability is present in the chronicle's security architecture due to the unencrypted storage of its secret share, making it susceptible to potential unauthorized access and misuse.

During our security review, a weakness has been identified in the storage mechanism of the chronicle's secret share. This secret share is currently stored in an unencrypted format within a file, raising concerns about data security. The functions responsible for interacting with this file, specifically `write_key_to_file` and `read_key_from_file`, operate without applying any encryption measures. This configuration introduces a significant risk, providing an avenue for unauthorized parties to access the secret share, potentially leading to severe security breaches.

The absence of encryption leaves the stored data vulnerable to exploitation, particularly if an attacker exploits system or OS vulnerabilities or identifies weaknesses in other software running on the server. Unauthorized access to the file containing the unencrypted secret share empowers malicious actors to execute harmful actions and potentially impersonate the chronicle. Beyond the immediate threat, this vulnerability has the potential to erode stakeholder confidence in the overall security of the system.

Assets:

- Cryptography and Keys

Status: Fixed

Classification

Severity: Low

Impact: 4/5

Likelihood: 1/5

Recommendations

Recommendation:

To rectify this vulnerability, the following measures are recommended:

Encryption at Boot: Introduce a robust mechanism to facilitate the secure decryption of the chronicle's secret share during the distributed key generation process. Adopt industry-recognized cryptographic algorithms such as *AES-256-GCM* or *ChaCha20-Poly1305* for storing the secret share within the file.

Utilize Zeroize Crate: Ensure that any unencrypted representation of the secret share in memory is thoroughly scrubbed using tools like the zeroize crate. This crate is specifically designed to securely zero out sensitive data from memory, thereby enhancing the overall security posture of the system.

F-2024-0447 - Panic in Shards Pallet due to Insufficient Error Handling in Group Commitment Computation - Low

Description:

The `shards` pallet exhibits a potential panic during the computation of group commitments, arising from inadequate error handling within the `commit` function.

The problematic section of the code lies within the `commit` function of the `shards` pallet:

pallets/shards/src/lib.rs:147:

```
for c in &commitment {
ensure!(VerifyingKey::from_bytes(*c).is_ok(), Error::<T>::InvalidCommitment);
}
/* ... */
let commitment = ShardMembers::<T>::iter_prefix(shard_id)
.filter_map(|(_, status)| status.commitment().cloned())
.reduce(|mut group_commitment, commitment| {
for (group_commitment, commitment) in
group_commitment.iter_mut().zip(commitment.iter())
{
*group_commitment = VerifyingKey::new(
VerifyingKey::from_bytes(*group_commitment).unwrap().to_element()
+ VerifyingKey::from_bytes(*commitment).unwrap().to_element(),
)
.to_bytes()
.unwrap();
}
group_commitment
})
.unwrap();
```

While each commitment undergoes individual validation as a valid `VerifyingKey`, the aggregation of all commitments may potentially yield `VerifyingKey::new(ProjectivePoint::IDENTITY)`. These specific scenarios pose a risk of encountering an error during the subsequent `to_bytes` operation, ultimately leading to panic during unwrapping.

It is noteworthy that while the occurrence of obtaining `IDENTITY` as the sum of valid commitments is considered improbable due to the large order of the underlying elliptic curve group secp256k1, it remains a potential risk that could compromise the overall security of the system.

Importantly, the current low risk is contingent on the centralized nature of the chronicles, where all shard members are assumed non-malicious. However, as the project transitions to a decentralized model, this issue becomes critical. Malicious members could exploit the situation by constructing intentional malicious commitments, leading to panics in the `shards` pallet.

Assets:

- Runtime & Pallets

Status: Mitigated

Classification

Severity: Low

Impact: 5/5

Likelihood: 1/5

Recommendations

Recommendation:

To address this issue effectively, it is highly advisable to improve the error-handling mechanisms within the commit function. Specifically, the function should be modified to return an appropriate error in cases where the result of the cumulative sum is `VerifyingKey::new(ProjectivePoint::IDENTITY)`. This adjustment aims to replace the current scenario leading to panics with a more controlled and informative response. By implementing a more robust error-handling approach, the potential for panics during commitment computation can be significantly reduced.

Moreover, it is advisable to implement an escape mechanism to gracefully handle situations in which the commitment sum results in the IDENTITY value. Potential options for mitigation include committing again, forming a new shard, and/or implementing measures to slash misbehaving nodes.

Evidences

Reproduce:

The accompanying test underlines this issue:

```
#[test]
fn poc_malicious_commitment_causes_panic() {
let shard = shard();
let public_key = public_key();
new_test_ext().execute_with(|| {
Shards::create_shard(Network::Ethereum, shard.to_vec(), 1);

// Compute a malicious commitment that will be submitted by the last member
// The sum of all commitments is IDENTITY
let public_key_element = VerifyingKey::from_bytes(public_key).unwrap().to_element
();
let malicious_commitment =
VerifyingKey::new(ProjectivePoint::IDENTITY - public_key_element - public_key_ele
ment);

let shard_id = 0;
for (member_number, account) in shard.iter().enumerate() {
if member_number != shard.len() - 1 {
// All members submit their commitments
assert_ok!(Shards::commit(
RawOrigin::Signed(account.clone()).into(),
shard_id as _,
vec![public_key],
[0; 65]
));
} else {
// The last member submits `malicious_commitment`
// This causes the panic
assert_ok!(Shards::commit(
RawOrigin::Signed(account.clone()).into(),
shard_id as _,
vec![malicious_commitment.to_bytes().unwrap()],
[0; 65]
));
}
// The members 0 and 1 submitted successfully
// The panic is during the last member call
println!("Member number {} has committed successfully", member_number);
}
});
}
```

Results:

This test results in panic:

```
Member number 0 has committed successfully
Member number 1 has committed successfully
thread 'tests::poc_malicious_commitment_causes_panic' panicked at pallets/shards/
src/lib.rs:164:30:
called `Result::unwrap()` on an `Err` value: InvalidPublicKey
```

F-2024-0507 - Validation Gap for Pending Nodes in ROAST Protocol - Low

Description:

The identified issue stems from the ROAST implementation in the tss crate, specifically the coordinator's failure to verify that a node submitting a commitment is not concurrently in a pending state within an existing signing session. This lack of verification contradicts the fundamental logic of ROAST.

The root cause lies in the implementation of the `on_commit` function for `RoastCoordinator`, as depicted below:

tss/src/roast.rs:112:

```
fn on_commit(&mut self, peer: Identifier, commitment: SigningCommitments) {
self.commitments.insert(peer, commitment);
}
```

Upon receiving a ROAST request containing a commitment, the coordinator directly appends it to the map storing commitments for future sessions, neglecting to confirm whether the signer is currently involved in any ongoing signing session. As the coordinator proceeds to initiate a new session upon accumulating a sufficient number of commitments, the existing flaw permits a node already pending in one signing session to commit once again and engage in other sessions.

This deviation from the fundamental logic of the ROAST protocol is critical, as the protocol's robustness relies on the premise that each signer is pending in at most one session. The protocol anticipates a finite and limited number of sessions to conclude. However, the identified flaw introduces a vulnerability, enabling a malicious node to secure a position in every session by consistently dispatching commitments to the coordinator. Subsequently, the malicious node can impede session termination by withholding its signature share.

This compromised scenario severely undermines the robustness of the ROAST protocol, providing an avenue for any malicious member to obstruct the process and obstruct the computation of a valid signature.

It is essential to underscore that the current risk level is relatively low, contingent upon the centralized nature of the chronicles, where all shard members are presumed non-malicious. However, as the project transitions to a decentralized model, this issue becomes more critical, representing a vulnerability susceptible to exploitation by any shard member. Such exploitation could compromise the performance and overall security of the system.

Assets:

- Runtime & Pallets

Status: Fixed

Classification

Severity: Low

Impact: 4/5

Likelihood: 1/5

Recommendations

Recommendation: To rectify this issue, it is strongly recommended to introduce logic that verifies whether the signer is pending in any existing signing session before storing its commitment for a future session. This validation mechanism will enhance the integrity of the ROAST protocol, mitigating the potential exploitation of the identified vulnerability by malicious nodes.

F-2024-0509 - Lack of Size Limitations on Error Messages in TaskState Storage Map for Failed Transactions - Low

Description:

The tasks pallet in the Substrate runtime utilizes a storage map named `TaskState`, which plays a key role in tracking the state of tasks . Within `TaskState`, when a transaction fails, in particular *pallets/tasks/src/lib.rs:278* `pub fn submit_error` the `TaskStatus::Failed` variant is employed to preserve detailed error information encapsulated in a message of type `String` . However, there exists a significant vulnerability due to the absence of appropriate size limitations on these error messages.

The relevant code sections are as follows:

primitives/src/task.rs:72

```
pub enum TaskStatus {
Created,
Failed { error: TaskError },
Stopped,
Completed,
}
```

```
pub struct TaskError {
pub shard_id: ShardId,
pub msg: String,
pub signature: TssSignature,
}
```

This issue is briefly mentioned in "F-2023-0313 | Absence of Task Deletion Mechanism", and here is a thorough exposition of this concern, pertaining to the lack of size constraints on error messages within the `TaskState` storage map.

Currently, there are no constraints on the size of the `msg` parameter within the `TaskError` struct, allowing for the storage of arbitrary-length strings in failed transactions. This lack of limitation poses a potential risk of memory exhaustion or a Denial-of-Service (DoS) attack, as malicious actors could intentionally submit exceptionally large error messages during failed transactions.

Assets:

- Runtime & Pallets

Status: Fixed

Classification

Severity: Low

Impact: 3/5

Likelihood: 1/5

Recommendations

Recommendation:

While the current absence of size limitations on the error messages in the `TaskState` storage map may not pose a severe threat in the current centralized model, it becomes a more significant concern when transitioning to a decentralized model, where malicious actors could exploit this vulnerability to potentially execute memory exhaustion or Denial-of-Service (DoS) attacks by intentionally submitting excessively large error messages during failed transactions.

To address this vulnerability, it is imperative to implement size constraints on the msg parameter before storing it in the `TaskState` storage map. This can be achieved by incorporating appropriate validations within the `TaskStatus::Failed` variant, ensuring that error messages adhere to predefined size limits.

Additionally, consider documenting the maximum allowable size for error messages and communicate this information to network participants and developers. Thoroughly test the proposed changes to ensure they effectively mitigate the vulnerability without introducing regressions.

F-2024-0512 - Chronicle Crate Panic Caused by Mishandled `PeerId` - Low

Description:

A malicious member within a shard has the potential to induce a panic in the `chronicle` by registering an incorrect `PeerId`.

The vulnerability in question arises from the `Tss::new` implementation in the `chronicle` crate. Specifically, the issue stems from the lack of error handling and the use of `unwrap()` in the code snippet below:

chronicle/src/shards/tss.rs:28:

```
let members: BTreeSet<_> = members
.into_iter()
.map(|peer| p2p::PeerId::from_bytes(&peer).unwrap().to_string())
.collect();
```

The problem lies in the absence of proper error handling, where `unwrap()` is utilized without considering potential errors. It's noteworthy that, in this context, `peer` is represented as `[u8; 32]`, and `p2p::PeerId::from_bytes` can return an error if the passed bytes do not represent a valid `ed25519` curve point.

This issue is traceable to the `on_finality` function, where shard members' data is queried from the pallet using `get_shard_members` and `get_member_peer_id`. The `get_member_peer_id` function retrieves the `peer_id` of the member from the `MemberPeerId` storage map in the `members` pallet. It is crucial to observe that each member submits its `peer_id` by calling the `register_member` extrinsic of the members pallet, and it is stored as a `[u8; 32]` array without additional validations.

This scenario opens the possibility for a malicious node to submit an invalid `peer_id` during registration. Consequently, when this malicious node enters a shard, other members of the shard may encounter a panic during the `unwrap` operation, triggered by the actions of this one malicious shard member.

An attacker could potentially escalate the impact by registering multiple malicious nodes. If these nodes are distributed across several shards, they could induce panics in the members of these shards almost simultaneously. This situation has the potential to disrupt the block generation process, posing a serious security concern.

It is crucial to emphasize that the current level of risk is relatively low, relying on the centralized nature of the chronicles where all shard members are assumed to be non-malicious. Nonetheless, as the project shifts towards a decentralized model, this concern gains significance, portraying a vulnerability that could be exploited by any node, potentially compromising the system's performance and overall security.

Assets:

- Cryptography and Keys
- Runtime & Pallets

Status: Fixed

Classification

Severity: Low

Impact: 4/5

Likelihood: 1/5

Recommendations

Recommendation: To address this identified concern and further fortify the project's security, it is recommended to replace the mentioned `unwrap` instance, along with other occurrences of `unwrap` and `expect`, with robust error-handling mechanisms. This approach involves returning an error gracefully instead of triggering a panic, unless such behavior aligns with intentional design decisions. This step will not only contribute to heightened security but also enhance the overall reliability and resilience of the system.

F-2024-0515 - Integration of Custom P2P Networking Library - Low

Description:

In our security audit of the `Chronicle` crate, we have identified a significant concern regarding its peer-to-peer (P2P) communication functionality. The crate employs an external library for P2P communication, referenced as follows:

```
p2p = { git = "https://github.com/dvc94ch/p2p" }
```

The reliance on this external, non-standard library for a crucial feature like P2P communication introduces several risks. The primary concern is the library's limited exposure and validation within the broader development community. Unlike well-established libraries, which benefit from extensive community scrutiny, continuous updates, and widespread usage, this particular library has not been as extensively vetted or adopted. This situation inherently increases the potential for undiscovered bugs and vulnerabilities, especially in a domain as complex and security-sensitive as P2P communication.

Assets:

- Dependencies

Status: Accepted

Classification

Severity: Low

Impact: 2/5

Likelihood: 2/5

Recommendations

Recommendation:

Given the risks identified, we recommend the following actions:

1. **Enhanced Monitoring and Regular Security Audits:** If the decision is to continue using the current `p2p` library, it is crucial to implement rigorous and continuous monitoring measures. This should include comprehensive security audits at regular intervals to promptly identify and address any emerging vulnerabilities or anomalies.
2. **Consideration of an Industry-Standard P2P Library:** We strongly recommend evaluating the feasibility of replacing the current `p2p` library with an established and widely-recognized P2P library like `libp2p`. The `libp2p` framework is renowned in the industry for its robustness, having been extensively tested in diverse environments. Its adoption would not only reduce the likelihood of encountering unknown vulnerabilities but also align the `Chronicle` crate with industry best practices for secure P2P communication.

Observation Details

F-2023-0313 - Memory Exhaustion Risk Due to Absence of Task Deletion Mechanism - Low

Description:

The existing on-chain storage infrastructure lacks a systematic approach for the removal of data associated with tasks, thereby introducing potential vulnerabilities leading to memory exhaustion and impeding the performance of nodes.

The fundamental issue resides in the life cycle management of tasks. Following their inclusion in storage maps such as `Tasks`, `TaskState`, `TaskPhaseState`, `TaskSignature`, `TaskRetryCounter`, `TaskCycleState`, and `TaskResults`, tasks endure indefinitely without a dedicated deletion mechanism. While certain key-value pairs may exert negligible storage pressure, others possess the capacity to accrue more substantial data volumes. Malicious entities could exploit this lacuna by deliberately submitting data of larger magnitudes, resulting in a discernible escalation of on-chain data.

A notable illustration is the `TaskPhaseState` storage map, wherein `TaskPhase::Read(Some(hash))` has the ability to store a submitted hash for a payable task without being subject to proper size constraints. This lack of constraints is concerning because it solely depends on block size, lacking size constraint and weight calculations, as discussed in detail in *F-2023-0173*.

Another pivotal storage map is `TaskState`, wherein, for failed transactions, `TaskStatus::Failed {error}` is preserved, encapsulating a message of type `String` devoid of appropriate size limitations (see **F-2024-0509**).

Furthermore, the `Tasks` storage map encompasses fields devoid of meticulous size management and weight validations (see *F-2023-0159*), thereby facilitating the genesis of tasks exhibiting storage consumption surpassing optimal thresholds. It is imperative to acknowledge that, unlike preceding maps populated with data supplied by shards or assigned shard members, the `create_task` function can be invoked by any account, amplifying the likelihood of such behavior.

These vulnerabilities empower malicious shards or accounts to intentionally introduce transactions with voluminous data, precipitating a gradual accumulation leading to memory depletion and consequent node deceleration. Even in the absence of malevolent intent, the elimination of redundant and superfluous data is a fundamental best practice for fortifying overall system stability and dependability.

It is prudent to acknowledge that a task's state can manifest as one of the following enum options:

```
pub enum TaskStatus {
Created,
Failed { error: TaskError },
Stopped,
Completed,
}
```

Upon achieving the `Completed` state, a task ceases to be processed by any shard, rendering its continued storage purposeless. While completed tasks do not exert influence on logic governing task scheduling and shard assignment, they may yet harbor the potential for memory consumption.

Tasks in a `Failed` or `Stopped` state may undergo resumption, making immediate deletion impractical. Nevertheless, the adoption of validation strategies, such as introducing a temporal threshold for resumption, offers a feasible avenue. In this

context, tasks would be expunged subsequent to surpassing a predefined temporal constraint.

Assets:

- Runtime & Pallets

Status: Accepted

Classification

Impact: 2/5

Likelihood: 2/5

Recommendations

Recommendation: To address this issue, the following recommendations are posited:

- Ensure that all on-chain data is subjected to judicious size constraints. Consider using `frame_support::BoundedVec` instead of regular vectors and enforce maximum lengths for `String` values.
- Enhance correct weight logic to render manipulations of data size economically prohibitive.
- Instigate a comprehensive logic for the deletion of tasks from associated maps upon task completion. Ensure elimination of all superfluous data, retaining only necessary values of limited size, such as task results or signatures.
- Explore the prospect of implementing a structured logic for the deletion of tasks that have lingered in a `Failed` or `Stopped` status for a prolonged period. Alternatively, enforce stringent limitations on memory consumption to preclude potential risks in the foreseeable future.
- If implementing the task deletion logic proves suboptimal for the project, take proactive measures to monitor the volume of stored on-chain data. This proactive monitoring will not only ensure the data size remains reasonable but also enable the detection of any malicious attempts to overflow the storage, thereby mitigating potential risks.

F-2023-0181 - Test coverage - Info

Description:

The project currently boasts a commendable overall test coverage exhibiting strong coverage across most packages. However, there is a noticeable dip in the chronicle and tss package, warranting attention.

The table below contains coverage information for crates tss and chronicle:

Name	Coverage
chronicle	399/608 (65,6%)
tss	387/545 (71..0%)

When examining the coverage for pallets , the following results can be observed:

Name	Coverage
tasks	179/223 (80.3%)
shards	108/145 (74.5%)
members	43/55 (78.2%)
elections	37/38 (97.4%)

Assets:

- Test Coverage

Status: Pending Fix

Recommendations

Recommendation:

Although the existing test coverage is robust, there is room for improvement. We propose augmenting the test suite for the tss and chronicle packages,
These actions will not only enhance the overall test coverage but also bring it closer to the industry-recommended standard of 80% coverage.

F-2023-0214 - Behavior of random_signer Function with All Members in PastSigners - Info

Description:

Upon reviewing the `random_signer` function in your pallet, we've identified a noteworthy behavior pattern when all members have already signed and are thus listed in the `PastSigners` storage.

In such a scenario, where every member of the shard has previously signed, the function's mechanism for selecting a signer undergoes a subtle shift.
Normally, the function aims to select a member who hasn't signed yet, as determined by their absence in `PastSigners`.
It begins by randomly generating an index within the members array and checks if the member at this index is in `PastSigners`.
However, when all members are in `PastSigners`, the function's pre-loop condition — `if PastSigners::<T>::iter_prefix(shard_id).count() < members.len()` — is not met, and as a result, it skips the subsequent loop that would typically filter out previously signed members.

Consequently, the function defaults to selecting a member purely based on the initially generated random index, effectively making the selection random from the entire set of members, irrespective of their signing history.
This means that in such cases, any member, regardless of their previous signing status in the current cycle, could be chosen again.

It's important to note that this behavior is not inherently problematic but is a characteristic of the function's design. Depending on the specific requirements of your application, you might consider this behavior as aligning with your expectations or decide to modify the function for a different approach.
For instance, if a unique signer is required in each round until all members have signed, additional logic may be necessary to reset `PastSigners` or to handle the scenario differently.
However, if the intention is to allow repeat signers in subsequent rounds regardless of their past participation, the current implementation meets this criterion.

This observation is crucial for understanding how the `random_signer` function operates under all possible scenarios and ensures that its behavior aligns with the intended logic of your blockchain system.

Assets:

- Runtime & Pallets

Status: Fixed

F-2023-0221 - TODO comments in code - Info

Description:

The prevailing issue pertains 5 `TODO` comments interspersed within the codebase, each functioning as markers for areas necessitating meticulous attention due to potential security implications, or specific segments meriting enhanced scrutiny for refinement.

This confluence of comments underscores the dynamic and iterative nature inherent in the ongoing software development process, wherein the identification and resolution of these flagged aspects assume a crucial role in fortifying the robustness and security posture of the software project.

`TODO` comments to be implemented/considered/removed:

- *pallets/shards/src/lib.rs:150* verify proof of knowledge
- *pallets/tasks/src/lib.rs:328* update bench, weights
- *tss/src/lib.rs:342* make rts asynchronous
- *tss/src/rts.rs:31* delta share probably needs to be encrypted/authenticated
- *tss/src/rts.rs:272* handle failure somehow. maybe try a different set of random peers?

Assets:

- Documentation and Comments

Status: Pending Fix

Recommendations

Recommendation:

We strongly advise addressing all `TODO` comments within your codebase. These annotations are more than simple reminders; they often contain crucial functionality or improvements that have been deferred for later implementation.
Neglecting these areas could potentially make your system vulnerable.

F-2023-0222 - Documentation Lacks Comprehensive Coverage - Info

Description:

The project's current documentation, along with a supplemental video guide, offers a basic overview of various modules but falls short in providing comprehensive detail and clarity.
While the video is a helpful tool, it underscores the existing gaps in the written documentation and in-code comments.
We appreciate the Analog team's responsiveness and helpful communication, which has been instrumental in enhancing our understanding of the code base during the audit process.

Clear and thorough documentation is vital in blockchain projects to assist developers, contributors, auditors, and users in comprehending and navigating the system's functionalities. An earnest effort to refine the written documentation and comments will considerably enhance the project's accessibility and usability, making it more approachable for all involved parties.

Assets:

- Documentation and Comments

Status: Pending Fix

Recommendations

Recommendation:

To improve the understanding and accessibility of the code base, we recommend enhancing the documentation with detailed comments covering:

- crates
- extrinsics
- hooks
- storage values

This effort will facilitate internal code comprehension, simplifies debugging, and can improve security by clearly outlining the intent of each functions.

Developers and contributors can leverage the `cargo doc` tool to generate a comprehensive API documentation. This documentation can be viewed in a web browser, offering an in-depth look at the project's structure and functionalities.

Enhanced documentation and comments will not only make the code base more approachable for current developers but also more welcoming for new contributors, providing a valuable overview of the project and its operational mechanics.

F-2023-0223 - Employment of Sudo Pallet - Info

Description:

The runtime configuration incorporates the `sudo-pallet`.
runtime/src/lib.rs:1187:

```
construct_runtime!(
pub struct Runtime
{
/* ... */
Sudo: pallet_sudo,
/* ... */
}
);
```

The root account, initially set at genesis, is defined as follows:

node/src/chain_spec.rs:318:

```
generate_analog_genesis(
wasm_binary,
// Sudo account
hex!["1260c29b59a365f07ac449e109cdf8f95905296af0707db9f3da0254e5db5741"].into(),
/* ... */
)
```

The root user possesses the authority to call the following extrinsics, irrespective of ownership:

- `set_shard_config` *pallets/elections/src/lib.rs:93*
- `create_task` *pallets/tasks/src/lib.rs:186*
- `stop_task` *pallets/tasks/src/lib.rs:215*
- `resume_task` *pallets/tasks/src/lib.rs:229*

While the use of the `Sudo` pallet is often perceived as a challenge to decentralization, in the context of this project, its deployment during the development phase is standard practice.
This approach allows for streamlined management and testing until the application reaches a stable state.

Assets:

- Runtime & Pallets

Status: Pending Fix

Recommendations

Recommendation:

Short-Term: At present, no immediate action is required regarding the Sudo pallet. The project team has already acknowledged in their `README.md` file the plan for "Enabling governance, and removing the sudo pallet" as an upcoming feature. This indicates a proactive approach towards evolving the project's governance structure.

Long-Term: In line with the project's roadmap, the implementation of a comprehensive governance system for protocol management is advisable. This step will be crucial for transitioning from a development-focused framework to a more decentralized and community-driven model, aligning with the long-term objectives of the project.

External References:

- Sudo removal procedure outlined by Polkadot

F-2024-0513 - Unsafe arithmetics - Info

Description:

During the security audit of the Analog project, a comprehensive list of unsafe arithmetic operations was compiled. These operations, as they currently stand, do not pose any known issues, as all potential scenarios have been thoroughly checked. This list is provided for informational purposes, ensuring transparency and awareness of these arithmetic operations within the project.

Run the following command to view the complete list of unsafe arithmetic operations:

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

Members pallet

pallets/members/src/lib.rs:89

```
if heart.is_online && n.saturating_sub(heart.block) >= T::HeartbeatTimeout::get()
{
Heartbeat::<T>::insert(&member, heart.set_offline());
Self::member_offline(&member);
writes += 1;
}
```

Shards pallet

pallets/shards/src/lib.rs:160

```
*group_commitment = VerifyingKey::new(
VerifyingKey::from_bytes(*group_commitment).unwrap().to_element()
+ VerifyingKey::from_bytes(*commitment).unwrap().to_element(),
)
```

pallets/shards/src/lib.rs:210

```
if n.saturating_sub(created_block) >= T::DkgTimeout::get() {
Self::remove_shard_offline(shard_id);
Self::deposit_event(Event::ShardKeyGenTimedOut(shard_id));
writes += 5;
}
```

pallets/shards/src/lib.rs:311

```
fn create_shard(network: Network, members: Vec<AccountId>, threshold: u16) {
let shard_id = <ShardIdCounter<T>>::get();
<ShardIdCounter<T>>::put(shard_id + 1);
```

pallets/shards/src/lib.rs:343

```
if signer_index == members.len() - 1 { 0 } else { signer_index + 1 };
```

Tasks pallet

pallets/tasks/src/lib.rs:357

```
if let Some(shard_id) = TaskShard::<T>::get(task_id) {
Self::start_write_phase(task_id, shard_id);
writes += 2;
}
```

TSS

tss/src/dkg.rs:92:36

```
if self.round2_packages.len() != self.members.len() - 1 {
return None;
}
```

tss/src/dkg.rs:105:23

```
SigningShare::new(acc.to_scalar() + e.to_scalar())
```

tss/src/roast.rs:129:3

```
let session_id = self.session_id;
self.session_id += 1;
```

tss/src/rts.rs:107:24

```
if deltas.len() != self.threshold as usize - 1 {
anyhow::bail!("invalid deltas");
}
```

tss/src/rts.rs:151:22

```
if deltas.len() != self.helpers.len() - 1 {
return false;
}
```

tss/src/rts.rs:240:3

```
self.session_id += 1;
```

tss/src/lib.rs:197:33

```
let coordinators: BTreeSet<_> =
members.iter().copied().take(members.len() - threshold as usize + 1).collect();
```

Chronicle

chronicle/src/shards/service.rs:352

```
let heartbeat_time = (self.substrate.get_heartbeat_timeout().unwrap() / 2) * min_
block_time;
```

chronicle/src/shards/service.rs:355

```
let mut heartbeat_tick =
interval_at(Instant::now() + heartbeat_duration, heartbeat_duration);
```

Assets:

- Runtime & Pallets

Status: Pending Fix

Recommendations

Recommendation:

As part of our commitment to maintaining high standards in Rust programming, we recognize the current stability and effectiveness of the arithmetic operations within the Analog project. At present, no immediate action is required as the operations are performing reliably.

However, to further enhance the robustness of the code and mitigate any potential issues in the future, we suggest considering the adoption of safer arithmetic methods provided by the Rust Standard Library. These methods include those in the categories of `checked_*`, `saturating_*`, and `overflowing_*`, arithmetic.

Appendix 1. Severity Definitions

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

Appendix 2. Scope

The scope of the project includes the following components from the provided repository:

Scope Details	
Repository	https://github.com/Analog-Labs/testnet/
Commit	3ba97bde46eac298fd61eba7ff5b5ef0078a3ebe
Whitepaper	https://www.analog.one/Analog-Timepaper.pdf

Components in Scope

Cryptography and Keys

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

Substrate fork review

- Review of all code changes and missing updates since Substrate clone date

Substrate client configuration review

- Genesis & chain spec review
- Consensus configuration
- Substrate FRAME pallets usage review
- chronicle crate review
- tss crate review
- Standard attacks review (replay, malleability,...)

Runtime & Pallets

- Runtime implementation review
- pallet-elections review
- pallet-members review
- pallet-shards review
- pallet-tasks review
- Attack scenarios analysis (Weight, race, stack, DoS, state implosion, access control bypass, overflow...)

Weights & Benchmarks

- Weight values & benchmarks review

Substrate RPC

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

Testing

- Environment Setup
- E2E sync tests
- Fuzz tests