HACKEN

POLKADEX PARACHAIN SECURITY ANALYSIS

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

Intro

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

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

Name	Polkadex Parachain
Website	https://polkadex.trade/
Repository	https://github.com/Polkadex-Substrate/parachain
Commit	684fac00a125f45344b29bbd4003a2ec9af10f6b
Platform	L1 / Parachain
Network	Polkadex
Languages	Rust
Methods	Automated Code analysis, Manual review, Issues simulation
Lead Auditor	s.akermoun@hacken.io
Auditor	n.lipartiia@hacken.io
Approver	l.ciattaglia@hacken.io
Timeline	08.03.2023 - 17.05.2023
Changelog	17.04.2023 (Preliminary Report)
Changelog	24.08.2023 (Final Version)

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

Table of contents

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

Summary

Polkadex is a fully non-custodial, peer-to-peer orderbook-based cryptocurrency exchange designed for the decentralized finance (DeFi) ecosystem. Built on the Substrate framework, Polkadex aims to provide a scalable and secure trading platform with high throughput and low latency.

The Polkadex parachain is a key component of the Polkadex project, enabling seamless integration with the Polkadot ecosystem. Through the parachain, Polkadex can access and support assets from other parachains, significantly improving interoperability and broadening its scope within the Polkadot network. The parachain leverages Cross-Chain Message Passing (XCMP) and XCM (Cross-Consensus Message) to facilitate communication and asset transfers between Polkadex and other parachains.

By utilizing the advanced features of the Polkadot ecosystem, Substrate framework, and XCM communication, the Polkadex parachain aims to offer a robust, high-performance trading platform that caters to the evolving needs of the DeFi community.

Documentation quality

The Polkadex parachain's documentation is robust and comprehensive, especially with the transition to a production-ready state. Crate-level documentation, coverage of all extrinsics, errors, event variants, and data storage has made the documentation accessible and navigable.

However, some utility functions remain without documentation, and there is still a notable concern with the lack of documentation for the `on_initialize hook`. This hook executes code at each block, and without proper documentation, its behavior and potential side effects may be challenging to understand, presenting a risk.

Despite these areas for improvement, the overall enhancements have contributed to the project's stability and success, raising the total documentation quality.

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

Code quality

The Polkadex parachain's code quality has seen substantial improvements, reflecting its transition from active development to a production-ready state. Key areas of enhancement include alignment with Rust best practices, the introduction of comprehensive testing coverage, and the implementation of benchmarking using the Substrate framework. Error management has been refined, and all incomplete sections have been addressed.

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

Architecture quality

The Polkadex parachain is built on the robust Substrate framework, which is a modular and highly customizable blockchain development platform. By using the Substrate framework, Polkadex benefits from a solid foundation, allowing them to focus on implementing specific functionality tailored to their unique requirements.

Furthermore, Polkadex has employed the Cumulus node template to bootstrap their code, which is a widely-accepted and well-tested starting point for developing parachains. This approach ensures a strong, reliable, and proven base for the Polkadex parachain, significantly reducing the risk of architectural issues.

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

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

Security score

In our analysis of the Polkadex parachain, we have identified a number of security issues that warrant attention, including 1 critical, 2 high, 5 medium, and 3 low severity issues.
After the remediation process, all issues have been fixed.
The security score is **10** out of 10.

Total score

Considering all metrics, the total score of the report is **9.6** out of 10.

Findings count and definitions

Severity	Findings	Severity Definition
Critical	1	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	2	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	5	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	3	Low severity vulnerabilities are minor flaws in the blockchain protocol that might not have a direct impact on security but could cause minor inefficiencies in transaction processing or slight delays in block propagation. They might include vulnerabilities that allow attackers to cause nuisance-level disruptions or are only exploitable under extremely rare and specific conditions. These vulnerabilities should be corrected but do not represent an immediate threat to the system.
Total	11	

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

Scope of the audit

Protocol Audit

Substrate fork review

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

Cryptography and Keys

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

Substrate client configuration review

- Parachain configuration & chain spec review
- Consensus
- Substrate FRAME pallets usage review
- Standard attacks review (replay, maullability,...)

Runtime & Pallets

- Runtime implementation review
- Pallet xcm-handler review
- Pallet thea-council review
- Attack scenarios analysis (Gas, race, stack,DoS, state implosion, access control bypass...)

RPC

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

Implementation

Code Quality

- Static Code Analysis
- Tests coverage
- Benchmark

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

Protocol Tests

Node Tests

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

Runtime Tests

- Fuzz tests

Issues

Local Nonce Dependency and Potential DoS Attack via Locking Withdrawals

This issue highlights a potential logic bug in the `withdraw_asset` extrinsic in `xcm-helper` pallet, where a transaction with a valid signature might fail due to a nonce mismatch at the time of execution in a decentralized network. Coupled with Issue Issue PDX-009, a malicious actor can potentially exploit these vulnerabilities to launch a denial-of-service (DoS) attack, locking withdrawals.

ID	PDX-010
Scope	Assets Management
Severity	**CRITICAL**
Vulnerability Type	Data Validation / DoS Attack
Status	Fixed

Details

The `withdraw_asset` extrinsic uses a separate, local nonce to validate transactions. However, in a decentralized network, the order in which transactions are executed and added to blocks is not deterministic. This could lead to nonce mismatches and transaction failures, even if the signature is valid.

The nonce check is performed in the following part of the code in `xcm-helper` pallet:
pallets/xcm-helper/src/lib.rs:372:

```
let current_withdraw_nonce = <WithdrawNonce<T>>::get();
ensure!(withdraw_nonce == current_withdraw_nonce, Error::<T>::NonceIsNotValid);
```

When combined with Issue PDX-009, where the nonce is incremented even on transaction failure, a malicious actor can potentially submit invalid transactions that rapidly increase the nonce, causing a mismatch for any new legitimate withdrawal transactions. This would effectively lock withdrawals, resulting in a denial-of-service attack.

In addition, multiple good actors might submit withdrawal transactions simultaneously, each using the current nonce value. Due to the asynchronous nature of the network, the order in which these transactions are executed may differ from the order in which they were submitted. Consequently, nonce mismatches can still occur, leading to transaction failures.

Exploit scenario:

1. The malicious actor identifies the current nonce value `WithdrawNonce` in the system.
2. The attacker crafts multiple withdrawal transactions with invalid signatures but with the correct current nonce value.
3. The attacker submits these transactions to the network.
4. The `withdraw_asset` extrinsic increments the nonce for each failed transaction due to Issue PDX-009
5. As a result, the nonce value increases rapidly, causing a mismatch for any new legitimate withdrawal transactions.
6. Due to the nonce mismatch new legitimate withdrawal transactions will fail to pass the verification process, effectively locking withdrawals and resulting in a denial-of-service attack.

Recommendation

Resolving Issue PDX-009 will lower the severity of this vulnerability as only good actors will be able to pass the signature verification and trigger a nonce incrementation.

However, even when only good actors, who can pass the signature verification, can increment the nonce, there may still be nonce mismatches due to the inherent unpredictability of transaction execution order in a decentralized network.
As there is no limit enforcing a unique Polkadex Sovereign Account, in addition to resolving Issue PDX-009, consider implementing nonce management per account. This can be done using a `StorageMap` to store the nonce value associated with each account and modify the `withdraw_asset` function to use this new storage and verify the nonce before incrementing it.

```
/* ... */
// Add a new storage map to store nonces for each account
#[pallet::storage]
#[pallet::getter(fn account_nonces)]
pub(super) type AccountNonces<T: Config> = StorageMap<_, Blake2_128Concat, T::AccountId, u32, ValueQuery>;
/* ... */
#[pallet::call]
impl<T: Config> Pallet<T> {
    /// Transfers Assets from Polkadex Sovereign Account to Others on native/non-native parachains using XCMP.
    ///
    /// # Parameters
    ///
    /// * `payload`: List of Assets and destination.
    /// * `withdraw_nonce`: Current Nonce of Withdrawal.
    /// * `signature`: Payload signed using Thea Public Key.
    #[pallet::weight(Weight::from_ref_time(10_000) + T::DbWeight::get().writes(1))]
    pub fn withdraw_asset(
        origin: OriginFor<T>,
        payload: BoundedVec<
            (
                sp_std::boxed::Box<VersionedMultiAssets>,
                sp_std::boxed::Box<VersionedMultiLocation>,
            ),
            ConstU32<10>,
        >,
        withdraw_nonce: u32,
        signature: sp_core::ecdsa::Signature,
    ) -> DispatchResultWithPostInfo {
        let origin = ensure_signed(origin.clone())?;
        // Get the current nonce for the account
        let current_account_nonce = AccountNonces<T>::get(&origin);
        ensure!(withdraw_nonce == current_account_nonce, Error::<T>::NonceIsNotValid);
        let public_key = <ActiveTheaKey<T>>::get().ok_or(Error::<T>::PublicKeyNotSet)?;
        let encoded_payload = Encode::encode(&payload);
        let payload_hash = sp_io::hashing::keccak_256(&encoded_payload);
        // Perform the signature verification
        if Self::verify_ecdsa_prehashed(&signature, &public_key, &payload_hash)? {
            // Increment the account nonce after successful verification
            AccountNonce::<T>::insert(&origin, current_account_nonce.saturating_add(1));
            let withdrawal_execution_block: T::BlockNumber =
                <frame_system::Pallet<T>>::block_number()
                    .saturated_into::<u32>()
                    .saturating_add(
                        T::WithdrawalExecutionBlockDiff::get().saturated_into::<u32>(),
                    )
                    .into();
            for (asset, dest) in payload {
                let pending_withdrawal =
                    PendingWithdrawal { asset, destination: dest, is_blocked: false };
                <PendingWithdrawals<T>>::try_mutate(
                    withdrawal_execution_block,
                    |pending_withdrawals| {
                        pending_withdrawals
                            .try_push(pending_withdrawal)
                            .map_err(|_| Error::<T>::PendingWithdrawalsLimitReached)
                    },
                )?;
            }
        } else {
            return Err(Error::<T>::SignatureVerificationFailed.into())
        }
        Ok(().into())
    }
}
```

This updated code adds a `StorageMap` to manage the nonces for individual accounts and modifies the `withdraw_asset` function to utilize this new storage. The nonce is verified and incremented only after successful signature verification, mitigating the risk of nonce mismatches and potential DoS attacks.

Loss of Treasury Funds

Trapped treasury funds due to improper account management, leading to potential liquidity issues and challenges in managing system finances.

ID	PDX-021
Scope	Assets Management / XCM configuration
Severity	HIGH
Vulnerability Type	Misconfiguration / Invalid Logic
Status	Fixed

Details

The issue lies in the way treasury funds are managed within the system. The current implementation uses the `xcm-helper` pallet as the treasury account, and there is no mechanism in place to withdraw or transfer the revenue collected in this account.
Additionally, the revenue is minted and swapped, potentially causing abnormal behavior in the AMM (Automated Market Maker) and affecting liquidity.

In the `xcm-helper` pallet's configuration, the `AssetHandlerPalletId` is defined as a constant:
pallets/xcm-helper/src/lib.rs:208:

```
#[pallet::constant]
type AssetHandlerPalletId: Get<PalletId>;
```

In the runtime configuration, the `AssetHandlerPalletId` is set to the XcmHelper pallet: *runtime/src/lib.rs:467*:

```
parameter_types! {
    pub const AssetHandlerPalletId: PalletId = PalletId(*b"XcmHandl");
    /* ... */
}
impl xcm_helper::Config for Runtime {
    /* ... */
    type AssetHandlerPalletId = AssetHandlerPalletId;
    /* ... */
}
```

The revenue destination is set to the `xcm-helper` pallet's account in the `take_revenue` function implemented for `RevenueCollector`, and then the revenue is minted and swapped to this account: *runtime/src/xcm_config.rs:357*:

```
fn take_revenue(revenue: MultiAsset) {
    if let AssetId::Concrete(location) = revenue.id {
        if let (Some(asset_id), Fungibility::Fungible(amount)) =
            (AC::convert_location_to_asset_id(location), revenue.fun)
        {
            let asset_handler_account = AssetHandlerPalletId::get().into_account_truncating(); //TODO: Change account
            if let (Ok(asset_id_associated_type), Ok(amount_associated_type)) =
                (AssetConv::convert_ref(asset_id), BalanceConv::convert_ref(amount))
            {
                AM::mint_into(
                    asset_id_associated_type,
```

```
                    &asset_handler_account,
                    amount_associated_type,
                );
                AMM::swap(&asset_handler_account, (asset_id, NativeCurrencyId::get()),amount);
            }
        }
    }
}
```

A TODO comment in the code indicates the need to address this issue: *runtime/src/xcm_config.rs:362*:

```
//TODO: Change account
```

As a result, the funds collected as revenue are trapped in the `xcm-helper` pallet's account, with no way to retrieve them. This issue could have an impact on the overall user experience, market liquidity, and the management of system finances.

Recommendation

1. Modify the implementation to use the Treasury pallet for managing funds earned from `take_revenue`, as suggested by the TODO comment. This would allow proper governance and access control over the collected funds.
2. As a less ideal alternative, implement extrinsic functions to withdraw or transfer the funds from the `xcm-helper` pallet's account, with controlled access to ensure only authorized users can perform these operations.

Proposals Resource Exhaustion

In the `thea-council` pallet's proposal management, an attacker can generate a large number of fake proposals, leading to resource exhaustion.

ID	PDX-006
Scope	Governance
Severity	HIGH
Vulnerability Type	Resource Exhaustion
Status	Fixed

Details

In the thea-council pallet's proposal management, an attacker can generate a large number of fake proposals, leading to resource exhaustion and making it difficult for legitimate users to manage pending proposals. This can be exploited as a Denial of Service (DoS) attack.

The following test simulates this attack scenario:

```
#[test]
fn test_potential_dos_attack() {
    new_test_ext().execute_with(|| {
        setup_council_members();
        let (first_council_member, second_council_member, _) = get_council_members();
        let attacker = 100;
        // Attacker becomes a council member
        assert_ok!(TheaCouncil::add_member(RuntimeOrigin::signed(first_council_member), attacker));
        assert_ok!(TheaCouncil::add_member(RuntimeOrigin::signed(second_council_member), attacker));
        assert_ok!(TheaCouncil::claim_membership(RuntimeOrigin::signed(attacker)));
        // Attacker generates fake proposals
        for i in 10..100010 {
            assert_ok!(TheaCouncil::add_member(RuntimeOrigin::signed(attacker), i));
```

```
        }
        // Count the number of proposals in the storage
        let number_of_proposals = Proposals::<Test>::iter().count();
        // Check if the number of proposals equals the number of fake proposals created by the attacker
        assert_eq!(number_of_proposals, 100000);
    })
}
```

In this simulated attack, an attacker can become a council member and then generate numerous fake proposals, filling up the Proposals storage. This makes it difficult for legitimate users to manage pending proposals, as the pending proposals can remain in the storage indefinitely if they do not reach the required number of votes.

Currently, pending proposals are deleted in the `evaluate_proposal` function when the required number of votes is reached. Here is the relevant code snippet:

```
fn evaluate_proposal(
    proposal: Proposal<T::AccountId>,
    sender: T::AccountId,
) -> DispatchResult {
    // ...
    let mut remove_proposal = false;
    <Proposals<T>>::try_mutate(proposal.clone(), |votes| {
        // ...
        if current_votes(votes) >= expected_votes() {
            Self::execute_proposal(proposal.clone())?;
            remove_proposal = true;
        }
        Ok::<(), sp_runtime::DispatchError>(())
    })?;
    if remove_proposal {
        Self::remove_proposal(proposal);
    }
    Ok(())
}
```

The `evaluate_proposal` function uses the `try_mutate` method to atomically update the vote count for a proposal and check if the proposal has received enough votes to be executed. If the proposal has received enough votes, the function executes the proposal and sets the `remove_proposal` flag to true. After executing the proposal, if the `remove_proposal` flag is true, the `remove_proposal` function is called to remove the proposal from storage.

There is no mechanism in place to clean up or remove pending proposals that have not reached the required number of votes, which can lead to storage bloat and management issues.

Recommendation

To mitigate this issue, it is recommended to implement a mechanism to limit the number of proposals that can be created by a single member within a certain time frame. This will prevent an attacker from flooding the storage with fake proposals. Additionally, consider implementing a mechanism to automatically remove stale proposals from the storage after a certain period or under specific conditions, which will help maintain the storage and ease proposal management.

Council Locking Issue in the Voting System

An identified vulnerability in the voting system can lead to a locked council effectively disabling its functionality and compromising the decision-making process.

ID	PDX-005
Scope	Governance
Severity	**MEDIUM**

Vulnerability Type	Invalid Logic
Status	Fixed

Details

In the current Thea Council system, implemented in `thea-council` pallet, a significant risk has been identified wherein if the last active council member removes themselves from the council, it would be left without any active members.
This action would lock the council, as there would be no remaining active council members to add new members, which could disrupt the overall functionality of the system.
However, it is essential to note that pending members can still join the council by claiming their membership. While this mitigates the risk of a completely locked council, the possibility of having a council with a single member still poses a risk to the integrity of the decision-making process.

Recommendation

To address this vulnerability, it is advised to implement a minimum council size greater than two members (see PDX-004) or prevent the last active council member from removing themselves.

No transaction weighting for extrinsics in router pallet

in `router` pallet, all extrinsic weights are tagged with 0

ID	PDX-003
Scope	AMM
Severity	**MEDIUM**
Vulnerability Type	DoS
Status	Fixed

Details

To protect blockchain resources from being drained or overloaded, you need to manage how they are made available and how they are consumed. The two of the most important mechanisms available to block authors are weights and transaction fees.

in `router` pallet, all extrinsic weights are tagged with:

```
#[pallet::weight(0)]
```

Weights are used to manage the time it takes to validate a block. In general, weights are used to characterize the time it takes to execute the calls in the body of a block. By controlling the execution time that a block can consume, weights set limits on storage input and output and computation.

Some of the weight allowed for a block is consumed as part of the block's initialization and finalization. The weight might also be used to execute mandatory inherent extrinsic calls. To help ensure blocks don’t consume too much execution time—and prevent malicious users from overloading the system with unnecessary calls—weights are used in combination with transaction fees.

By setting it in 0 a block has no execution time limit and transactions fees can be miscalculated.

Recommendation

Use Substrate tools and helper functions for determining the weight of an extrinsics.
If you need to relax restrictions or requirements during development or testing processes, you can enable Dev Mode for a pallet.

No transaction weighting for extrinsics in swap pallet

in `swap` pallet, all extrinsic weights are tagged with 0

ID	PDX-002
Scope	AMM
Severity	**MEDIUM**
Vulnerability Type	DoS
Status	Fixed

Details

To protect blockchain resources from being drained or overloaded, you need to manage how they are made available and how they are consumed. Two of the most important mechanisms available to block authors are weights and transaction fees.

in `swap` pallet, all extrinsic weights are tagged with:

```
#[pallet::weight(0)]
```

Weights are used to manage the time it takes to validate a block. In general, weights are used to characterize the time it takes to execute the calls in the body of a block. By controlling the execution time that a block can consume, weights set limits on storage input and output and computation.

Some of the weight allowed for a block is consumed as part of the block's initialization and finalization. The weight might also be used to execute mandatory inherent extrinsic calls. To help ensure blocks don’t consume too much execution time—and prevent malicious users from overloading the system with unnecessary calls—weights are used in combination with transaction fees.

By setting it in 0 a block has no execution time limit and transactions fees can be miscalculated.

Recommendation

Use Substrate tools and helper functions for determining the weight of an extrinsics.
If you need to relax restrictions or requirements during development or testing processes, you can enable Dev Mode for a pallet.

Only 2 active council members allows a weaker consensus

An edge case may allow a weaker consensus in thea council voting system to 1/2 majority rule instead of 2/3.

ID	PDX-004
Scope	Governance
Severity	**MEDIUM**
Vulnerability Type	Invalid Logic
Status	Fixed

Details

In *thea-council* pallet the voting system requires a 2/3 majority for adding or removing members in the thea council.
The function `evaluate_proposal` check if 2/3 majority is reached, and if it is reached execute the `Proposal`:

pallets/thea-council/src/lib.rs::

```
fn evaluate_proposal(
    proposal: Proposal<T::AccountId>,
    sender: T::AccountId,
) -> DispatchResult {
    let current_votes =
        |votes: &BoundedVec<Voted<T::AccountId>, ConstU32<100>>| -> usize { votes.len() };
    // ---> Expected_votes is 2/3 of total council members.
    let expected_votes = || -> usize {
        let total_active_council_size = <ActiveCouncilMembers<T>>::get().len();
        total_active_council_size.saturating_mul(2).saturating_div(3)
    };
    let mut remove_proposal = false;
    <Proposals<T>>::try_mutate(proposal.clone(), |votes| {
        ensure!(!votes.contains(&Voted(sender.clone())), Error::<T>::SenderAlreadyVoted);
        votes.try_push(Voted(sender)).map_err(|_| Error::<T>::StorageOverflow)?;
        // ---> If current nb votes is greater or equal than expected_votes then execute proposal
        if current_votes(votes) >= expected_votes() {
            Self::execute_proposal(proposal.clone())?;
            remove_proposal = true;
        }
        Ok::<(), sp_runtime::DispatchError>(())
    })?;
    if remove_proposal {
        Self::remove_proposal(proposal);
    }
    Ok(())
}
```

There is an edge case when the number of total active council members is 2, which goes against the intended 2/3 majority rule. If the number of total active council members is 2, then only a 1/2 majority rule is applied.

```
// The council contains only 2 members
let total_active_council_size = 2;
// Applying the formula, expected_votes will be equal to 1, which is 1/2 of the majority
let expected_votes = total_active_council_size.saturating_mul(2).saturating_div(3);
// Only 1 vote , which is 1/2 of total council members
let nb_vote = 1
// Always true
assert!(nb_vote >= expected_votes);
```

If a malicious individual or a group of malicious actors can secure a position among the last two council members, they can unilaterally remove the other member without any discussion, effectively gaining control over the council.
After doing so, they can add new members at their discretion.
According to game theory, there is no incentive to delay the removal of the other remaining member.

Recommendation

To address this vulnerability, it is advised to implement a minimum council size greater than two members (>= 3).
Else a decent solution would be to require a unanimous decision when the voter set contains only two voters, as it ensures a high level of agreement in the decision-making process (100% or 2/2 of voters). This maintains consistency with the intended purpose of the 2/3 majority rule, which is to achieve a strong consensus.

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

Council Members can have more than one seat

No protections are in place to prevent a council member from occupying more than one slot in the governance system. This situation can lead to a denial of service (DoS) if the council cannot reach consensus.

ID	PDX-017
Scope	Governance
Severity	**MEDIUM**
Vulnerability Type	Data Validation, DoS
Status	Fixed

Details

An active council member can be added to the council multiple times. While this allows the council member to occupy more than one slot, it does not increase their voting power. There is no check to prevent a council member from rejoining the pending council member list and claiming membership to gain another slot in the council: *pallets/thea-council/src/lib.rs:140*

```
#[pallet::weight(10_000 + T::DbWeight::get().writes(1).ref_time())]
pub fn add_member(origin: OriginFor<T>, new_member: T::AccountId) -> DispatchResult {
    let sender = ensure_signed(origin)?;
    // 1.  Only check if proposer is a council member
    ensure!(Self::is_council_member(&sender), Error::<T>::SenderNotCouncilMember);
    Self::do_add_member(sender, new_member)?;
    Ok(())
}
```

The following proof of concept (PoC) demonstrates how a council member can be added to the council multiple times, occupying three slots:

```
#[test]
fn test_add_existing_member() {
    new_test_ext().execute_with(|| {
        // Initially, there are 3 council members:
        // [first_council_member, second_council_member, third_council_member]
        setup_council_members();
        let (first_council_member, second_council_member, third_council_member) =
            get_council_members();
        // The new member to add is the first_council_member
        let new_member = first_council_member;
        // Loop 2 times for adding the first_council_member again in the council.
        // At first loop we will reach 2/3 majority
        // At second loop we abuse a flaw in the consensus threshold as only 1/2 majority is need
        // when there are 4 members in the council. (see PDX-18)
        for _ in 0..2 {
            println!("IN LOOP");
            // first_council_member proposes himself
            assert_ok!(TheaCouncil::add_member(
                RuntimeOrigin::signed(first_council_member),
                new_member
            ));
            // second_council_member approves
            assert_ok!(TheaCouncil::add_member(
                RuntimeOrigin::signed(second_council_member),
                new_member
            ));
            // As consensus is reached, first_council_member
            //can claim membership in the council again
            assert_ok!(TheaCouncil::claim_membership(RuntimeOrigin::signed(new_member)));
        }
        // Get current council set
```

```
        let active_set = <ActiveCouncilMembers<Test>>::get();
        // Set that we are expecting to have now
        let council = BoundedVec::<u64, ConstU32<10>>::try_from(vec![
            first_council_member,
            second_council_member,
            third_council_member,
            first_council_member,
            first_council_member,
        ])
        .unwrap();
        // Check if first_council_member is 3 times in the council
        assert_eq!(active_set, council)
    })
}
// Below 2 utility functions
fn setup_council_members() {
    let (first_council_member, second_council_member, third_council_member) = get_council_members();
    let council = BoundedVec::try_from(vec![
        first_council_member,
        second_council_member,
        third_council_member,
    ])
    .unwrap();
    <ActiveCouncilMembers<Test>>::put(council);
}
fn get_council_members() -> (u64, u64, u64) {
    let first_council_member = 1;
    let second_council_member = 2;
    let third_council_member = 3;
    (first_council_member, second_council_member, third_council_member)
}
```

Impact of DoS due to consensus not being reached

When a council member occupies multiple slots in the council, it doesn't increase their voting power. However, it can negatively impact the consensus process. If the council cannot reach consensus due to the member's multiple slots, it can lead to a denial of service (DoS) situation in the governance system.

In such a scenario, the council may struggle to make decisions and approve proposals, as the consensus threshold may not be met. This can significantly hinder the system's governance process and may cause delays or disruption in its operation.

Recommendation

Implement a check within the add_member function to verify if the proposed new member is already an active council member. If so, raise the `AlreadyMember` error and prevent the addition. Here's the modified `add_member` function with the check:

```
#[pallet::weight(10_000 + T::DbWeight::get().writes(1).ref_time())]
pub fn add_member(origin: OriginFor<T>, new_member: T::AccountId) -> DispatchResult {
    let sender = ensure_signed(origin)?;
    // 1. Only check if the proposer is a council member
    ensure!(Self::is_council_member(&sender), Error::<T>::SenderNotCouncilMember);
    // 2. Check if the new member is already an active council member
    ensure!(!Self::is_council_member(&new_member), Error::<T>::AlreadyMember);
    Self::do_add_member(sender, new_member)?;
    Ok(())
}
```

By implementing the recommended check in the `add_member` function, you can prevent these issues from occurring and maintain the proper functioning of the governance system.

Incorrect Nonce Incrementation in withdraw_asset Function

The `withdraw_asset` extrinsic in the `xcm-helper` pallet exhibits improper nonce incrementation that occurs even if the transaction fails, potentially leading to undesirable consequences.

ID	PDX-009
Scope	Nonce Management
Severity	**LOW**
Vulnerability Type	Data Validation
Status	Fixed

Details

In the current implementation of the `withdraw_asset` extrinsic within the `xcm-helper` pallet, the `WithdrawNonce` is incremented before the signature's validity is checked. This approach contradicts the Substrate design principle "verify first, write last.".

pallets/xcm-helper/src/lib.rs:372:

```
let current_withdraw_nonce = <WithdrawNonce<T>>::get();
ensure!(withdraw_nonce == current_withdraw_nonce, Error::<T>::NonceIsNotValid);
// WithdrawNonce is incremented before signature verification
<WithdrawNonce<T>>::put(current_withdraw_nonce.saturating_add(1));
let pubic_key = <ActiveTheaKey<T>>::get().ok_or(Error::<T>::PublicKeyNotSet)?;
let encoded_payload = Encode::encode(&payload);
let payload_hash = sp_io::hashing::keccak_256(&encoded_payload);
// Signature verification is done here
if Self::verify_ecdsa_prehashed(&signature, &pubic_key, &payload_hash)? {
    /* ... */
} else {
    return Err(Error::<T>::SignatureVerificationFailed.into());
}
```

The existing implementation permits users to repeatedly call the function and increment the nonce without proper validation, potentially causing the nonce value to be higher than expected and leading to issues with subsequent valid transactions.
Although this issue has a minor severity on its own, it can be exploited to trigger a critical vulnerability that could lock all withdrawals (see PDX-010).

Recommendation

To address this issue, move the nonce incrementation to a point in the code after all validations have been performed and the transaction is confirmed to be valid.
This ensures that the nonce is only incremented for valid transactions, adhering to the "verify first, write last" principle.
The updated code snippet is provided below:

```
if Self::verify_ecdsa_prehashed(&signature, &pubic_key, &payload_hash)? {
    let current_withdraw_nonce = <WithdrawNonce<T>>::get();
    ensure!(withdraw_nonce == current_withdraw_nonce, Error::<T>::NonceIsNotValid);
    // Incrementation is done after signature verification
    <WithdrawNonce<T>>::put(current_withdraw_nonce.saturating_add(1));
    // ... rest of the code
} else {
    return Err(Error::<T>::SignatureVerificationFailed.into());
}
```

Implementing this change will mitigate potential issues arising from improper nonce incrementation, enhancing the security and robustness of the extrinsic.

Another way to address this issue is to include the nonce as part of the payload that is signed. This approach ensures that only the sovereign account can increment the nonce, adding another layer of security.
Update the payload type to include the nonce:

```
pub type WithdrawalPayload = (
    u32, // Nonce
    BoundedVec<
        (
            sp_std::boxed::Box<VersionedMultiAssets>,
            sp_std::boxed::Box<VersionedMultiLocation>,
        ),
        ConstU32<10>,
    >,
);
```

Update the `withdraw_asset` extrinsic to include the nonce in the payload and verify the signature accordingly:

```
#[pallet::call]
impl<T: Config> Pallet<T> {
    /// Transfers Assets from Polkadex Sovereign Account to Others on native/non-native parachains using XCMP.
    ///
    /// # Parameters
    ///
    /// * `payload`: Nonce and list of assets and destinations.
    /// * `signature`: Payload signed using Thea Public Key.
    #[pallet::weight(Weight::from_ref_time(10_000) + T::DbWeight::get().writes(1))]
    pub fn withdraw_asset(
        origin: OriginFor<T>,
        payload: WithdrawalPayload,
        signature: sp_core::ecdsa::Signature,
    ) -> DispatchResultWithPostInfo {
        let origin = ensure_signed(origin.clone())?;
        let (withdraw_nonce, transfers) = payload;
        let public_key = <ActiveTheaKey<T>>::get().ok_or(Error::<T>::PublicKeyNotSet)?;
        let encoded_payload = Encode::encode(&payload);
        let payload_hash = sp_io::hashing::keccak_256(&encoded_payload);
        // Perform the signature verification
        if Self::verify_ecdsa_prehashed(&signature, &public_key, &payload_hash)? {
            let current_withdraw_nonce = <WithdrawNonce<T>>::get();
            ensure!(withdraw_nonce == current_withdraw_nonce, Error::<T>::NonceIsNotValid);
            // Incrementation is done after signature verification
            <WithdrawNonce<T>>::put(current_withdraw_nonce.saturating_add(1));
            let withdrawal_execution_block: T::BlockNumber =
                <frame_system::Pallet<T>>::block_number()
                    .saturated_into::<u32>()
                    .saturating_add(
                        T::WithdrawalExecutionBlockDiff::get().saturated_into::<u32>(),
                    )
                    .into();
            for (asset, dest) in transfers {
                let pending_withdrawal =
                    PendingWithdrawal { asset, destination: dest, is_blocked: false };
                <PendingWithdrawals<T>>::try_mutate(
                    withdrawal_execution_block,
                    |pending_withdrawals| {
                        pending_withdrawals
                            .try_push(pending_withdrawal)
                            .map_err(|_| Error::<T>::PendingWithdrawalsLimitReached)
                    },
                )?;
            }
        } else {
            return Err(Error::<T>::SignatureVerificationFailed.into())
        }
        Ok(().into())
    }
}
```

By implementing both recommendations, you will ensure that only valid transactions can increment the nonce, and only the sovereign account can influence it. These changes will enhance the security and robustness of the extrinsic and help prevent potential exploitation of the nonce incrementation issue.

Inaccurate Consensus Threshold Calculation

ID	PDX-018
Scope	Governance
Severity	**LOW**
Vulnerability Type	Arithmetic Errors
Status	Fixed

Details

The current implementation of the consensus threshold calculation uses a "round down" approach, which can lead to a lower consensus threshold than intended.
As a result, in some cases, the actual consensus threshold is below the desired 2/3 majority (**66.67%**). This issue occurs because the floor division used in the code does not round up the resulting value, effectively rounding down the threshold and potentially making it easier for a proposal to be approved.
The code snippet responsible for this calculation is: *pallets/thea-council/src/lib.rs:221*:

```
let expected_votes = || -> usize {
    let total_active_council_size = <ActiveCouncilMembers<T>>::get().len();
    total_active_council_size.saturating_mul(2).saturating_div(3)
};
```

Here, `total_active_council_size` is multiplied by 2 and then divided by 3. However, because both values are `usize`, the division operation will round down the result, which can lead to a lower consensus threshold in certain cases.

This rounding down behavior is demonstrated in the table from the previous report, where the actual consensus threshold for some council sizes deviates significantly from the intended 2/3 majority (**66,67%**).

Council Size	Votes Needed	Actual Consensus Threshold
1	1	100%
2	1	50%
3	2	66.67%
4	2	50%
5	3	60%
6	4	66.67%
7	4	57.14%
8	5	62.5%
9	6	66.67%
10	6	60%

Recomendation

To address this issue, we recommend updating the consensus threshold calculation to use a "round up" approach, ensuring that the actual consensus threshold remains at or above the intended 2/3 majority (66.67%).
A possible implementation using the `ceil` function from the `f64` type in Rust is as follows:

```
let expected_votes = || -> usize {
    let total_active_council_size = <ActiveCouncilMembers<T>>::get().len();
    let threshold = (total_active_council_size as f64 * 2.0 / 3.0).ceil() as usize;
    threshold
};
```

By converting the `total_active_council_size` to an `f64` and using the `ceil` function, the calculation will round up the result, ensuring that the consensus threshold remains at or above the desired 2/3 majority.

After implementing this change, the consensus threshold should be closer to the intended value for all council sizes, providing a more robust governance mechanism.

Max Pending Members Can Limit Council Expansion

There is no mechanism to delete unclaimed membership or incentivize claiming membership within a certain period. This can limit the ability to add new members to the council, thus restricting the expansion of the governance system.

ID	PDX-019
Scope	Governance
Severity	**LOW**
Vulnerability Type	DoS
Status	Fixed

Details

in `thea-council` pallet code, there is no limit or criteria set for the pending council members to claim their membership. This can lead to a situation where the maximum number of pending members is reached, preventing new members from being added to the council, thus limiting the expansion of the governance system.
The issue arises in the `execute_add_member` function:
pallets/thea-council/src/libs.rs:253:

```
fn execute_add_member(new_member: T::AccountId) -> DispatchResult {
    let mut pending_council_member = <PendingCouncilMembers<T>>::get();
    pending_council_member
        .try_push(new_member.clone())
        .map_err(|_| Error::<T>::StorageOverflow)?;
    <PendingCouncilMembers<T>>::put(pending_council_member);
    Self::deposit_event(Event::<T>::NewPendingMemberAdded(new_member));
    Ok(())
}
```

As evident from the code, pending council members are added to the `PendingCouncilMembers` storage, but there is no mechanism to remove them if they do not claim their membership within a specific time or block limit.

Recommendation

To resolve this issue, introduce a mechanism to remove unclaimed pending council memberships after a certain number of blocks have passed. This will prevent the council from being locked and allow new members to be added.

1. Add a storage item to store the block number when a member is added to the pending council members list.
2. Implement a function to check if the unclaimed pending membership has passed the block limit and remove it if necessary.
3. Modify the `do_claim_membership` function to check and remove expired pending memberships before processing the claim. This can also be done in an `on_initialize` hook.

By implementing these changes, you can ensure that the council can continue to expand by removing unclaimed memberships and making room for new members.

Absence of Refund Mechanism for Unused Weight in WeightTrader Implementation

The current implementation of the `WeightTrader` trait lacks a refund mechanism for unused weight during XCM execution, which can impact efficiency, fairness, and customization of the blockchain.

ID	PDX-008
Scope	Code Quality
Status	Fixed

Details

The runtime configuration for XCM reveals that the `WeightTrader` trait implementation for the `ForeignAssetFeeHandler` struct, does not include the `refund_weight` method.
runtime/src/xcm_config.rs:261:

```
impl<T, R, AMM, AC> WeightTrader for ForeignAssetFeeHandler<T, R, AMM, AC>
where
    T: WeightToFeeT<Balance = u128>,
    R: TakeRevenue,
    AMM: support::AMM<AccountId, u128, Balance, BlockNumber>,
    AC: AssetIdConverter,
{
    fn new() -> Self {
        /* ... */
    }
    fn buy_weight(
        &mut self,
        weight: u64,
        payment: Assets,
    ) -> sp_std::result::Result<Assets, XcmError> { /* ... */}
    // Missing refund_weight implementation
}
```

The `refund_weight` method is responsible for returning any unused weight after the execution of XCM messages, ensuring users only pay for the resources they actually consume.

A lack of refund mechanism for unused weight can lead to the following issues:

- Inefficiency: Users who pay for more weight than they actually need during XCM execution won't receive any refund for the unused portion, resulting in higher costs and reduced efficiency.
- Fairness: Charging users for the full amount of weight even when only a portion of it is used can be considered unfair, as it does not reflect the actual resources consumed by the user.

Recommendation

To address the issues arising from the absence of a refund mechanism for unused weight, it is recommended to implement the `refund_weight` method in the `WeightTrader` trait as follows:

1. Add the `refund_weight` method to the `ForeignAssetFeeHandler` implementation of the `WeightTrader` trait.
2. Determine the appropriate logic for refunding the unused weight based on the desired fee structure and use case.
3. Implement the refund logic within the `refund_weight` method to ensure users only pay for the resources they actually consume.

Compilation and Linter Warnings

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

ID	PDX-001
Scope	Code Quality
Status	Fixed

Details

During the process of verifying the code compilation with `cargo check`, as well as performing static analysis using `cargo clippy`, it has been observed that a significant number of warnings are generated. These warnings could potentially indicate various issues in the code, such as:

- Unoptimized or inefficient code
- Deprecated functions or methods
- Unnecessary or unused variables and imports
- Non-idiomatic Rust patterns
- Potential coding errors or logic flaws

Ignoring these warnings may lead to a harder-to-maintain codebase, potential performance issues, and even security vulnerabilities.

Recommendation

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

Error variants Improvement in `thea-council`

In the `thea-council` pallet, some error variants are unused and should be removed, while others should be added or used for better clarity and error handling.

ID	PDX-016
Scope	Code Quality
Status	Fixed

Details

The `Error` enum definition in `thea-council` pallet is as follow: *pallets/thea-council/src/lib.rs:112*

```
// Errors inform users that something went wrong.
#[pallet::error]
pub enum Error<T> {
```

```
    /// Storage Overflow
    StorageOverflow,
    /// Not a Valid Sender
    BadOrigin,
    /// Already Council Member
    AlreadyMember,
    /// Not Pending Member
    NotPendingMember,
    /// Sender not council member
    SenderNotCouncilMember,
    /// Sender Already Voted
    SenderAlreadyVoted,
    /// Not Active Member
    NotActiveMember,
}
```

We observe that the `StorageOverflow` variant is a generic name. More meaningful names should be used to specify which storage is overflowing: `ActiveCouncilMembers`, `PendingCouncilMembers`, or `Proposals`. This would result in more helpful error messages and improved monitoring and reporting.

Additionally, the `AlreadyMember` variant is declared but never used. We have identified that there is no check to prevent an active council member from having more than one seat in the council. (see PDX-017).

Furthermore, the `BadOrigin` variant is defined but never used in the pallet.

Recommendation

1. Improve the granularity of storage overflow error management by introducing separate error variants for each storage type that could overflow (e.g., `ActiveCouncilMembersOverflow`, `PendingCouncilMembersOverflow`, and `ProposalsOverflow`).
2. Use `AlreadyMember` error variant by addressing PDX-017.
3. Remove `BadOrigin` as it is never used and is already managed by substrate's internal code.

Inaccurate Error Handling in xcm-helper Pallet

Inaccurate error handling with the use of a generic `InternalError` variant leading to less informative error messages and reduced ease of debugging.

ID	PDX-023
Scope	Code Quality
Status	Fixed

Details

The `xcm-helper` pallet's Error enum uses a generic `InternalError` variant for multiple error situations. This approach can make it difficult for developers to diagnose specific issues and understand the root cause of an error.

The `InternalError` variant is used in the following functions:

- `handle_deposit`
- `deposit_native_token`
- `deposit_non_native_token`

Using a generic error variant reduces the granularity and informativeness of the error messages, making it harder to identify the specific issues that occurred during the execution.

For example, in the `handle_deposit` function, the `InternalError` variant is returned in three different situations:

1. When the conversion of the destination into a `MultiLocation` fails.
2. When the `get_destination_account` function returns None.
3. When the conversion of asset into `MultiAssets` fails.

In these cases, it would be more helpful to use separate, more descriptive error variants for each situation, allowing developers to pinpoint the cause of the problem more easily.

Recommendation

1. Refactor the Error enum by replacing the generic InternalError variant with more specific and descriptive error variants for each situation in which the error is currently used.
2. Update the handle_deposit, deposit_native_token, and deposit_non_native_token functions to return the new, more specific error variants when appropriate.
3. Ensure that error messages provide enough context to help developers identify and understand the cause of the problem.

By improving error handling and providing more informative error messages, developers will be able to diagnose and resolve issues more efficiently, leading to better overall code quality and maintainability.

Inefficient Use of BoundedVec Size Limit

While using BoundedVec ensures that storage is not wasted, the current implementation in the `thea-council` pallet has an inconsistency in the declaration of the size limit for voters in proposals.

ID	PDX-014
Scope	Code Quality
Status	Fixed

Details

The current implementation in `pallets/thea-council/src/lib.rs` uses a BoundedVec with an upper limit of 100 items, as defined by the `ConstU32<100>` type. This means that each proposal can have up to 100 council member votes.
However, the maximum number of council members allowed is only 10. This inconsistency between the number of allowed voters and the actual number of council members can cause confusion and lead to potential logic errors.
pallets/thea-council/src/lib.rs:92

```
/// Proposals
#[pallet::storage]
#[pallet::getter(fn proposal_status)]
pub(super) type Proposals<T: Config> = StorageMap<
    _,
    frame_support::Blake2_128Concat,
    Proposal<T::AccountId>,
    BoundedVec<Voted<T::AccountId>, ConstU32<100>>,
    ValueQuery,
>;
```

Recommendation

To improve code clarity and maintainability, we recommend adjusting the BoundedVec size limit for voters in proposals to match the maximum number of council members (10). This can be done by updating the `ConstU32<100>` type to `ConstU32<10>`:

```
/// Proposals
#[pallet::storage]
#[pallet::getter(fn proposal_status)]
pub(super) type Proposals<T: Config> = StorageMap<
    _,
    frame_support::Blake2_128Concat,
    Proposal<T::AccountId>,
    BoundedVec<Voted<T::AccountId>, ConstU32<10>>, // match the max council members
    ValueQuery,
>;
```

Additionally, consider implementing the recommendations from PDX-013 to make the maximum number of voters per proposal configurable at runtime, which would provide flexibility and allow for easier tuning of the system's behavior.

Magic Numbers As Constants

Usage of magic numbers as constant, directly hardcoded in pallets code, can lead to maintainability, readability, and potential security issues.

ID	PDX-013
Scope	Code Quality
Status	Fixed

Details

Hardcoding values in the code makes it harder to maintain and can introduce bugs, as the same value may need to be updated in multiple places. It may also make the code less readable, as the meaning behind the hardcoded values might not be immediately clear to other developers working on the project.

In addition, hardcoded values can lead to security vulnerabilities if these values are related to security-sensitive parameters, such as timeouts, limits, or key sizes. An attacker may exploit these vulnerabilities by taking advantage of hardcoded limits or other constraints in the system.

Here is a list of all usage of hardcoded values in the project:

pallets/thea-council/src/lib.rs:73:

```
/// Active Council Members
#[pallet::storage]
#[pallet::getter(fn get_council_members)]
pub(super) type ActiveCouncilMembers<T: Config> =
    StorageValue<_, BoundedVec<T::AccountId, ConstU32<10>>, ValueQuery>;
```

pallets/thea-council/src/lib.rs:79:

```
/// Pending Council Members
#[pallet::storage]
#[pallet::getter(fn get_pending_council_members)]
pub(super) type PendingCouncilMembers<T: Config> =
    StorageValue<_, BoundedVec<T::AccountId, ConstU32<10>>, ValueQuery>;
```

pallets/thea-council/src/lib.rs:79:

```
/// Proposals
#[pallet::storage]
#[pallet::getter(fn proposal_status)]
pub(super) type Proposals<T: Config> = StorageMap<
```

```
    _,
    frame_support::Blake2_128Concat,
    Proposal<T::AccountId>,
    BoundedVec<Voted<T::AccountId>, ConstU32<100>>,
    ValueQuery,
>;
```

pallets/thea-council/src/lib.rs:219:

```
fn evaluate_proposal(
    proposal: Proposal<T::AccountId>,
    sender: T::AccountId,
) -> DispatchResult {
    let current_votes =
        |votes: &BoundedVec<Voted<T::AccountId>, ConstU32<100>>| -> usize { votes.len() };
    /* ... */
}
```

pallets/xcm-helper/src/lib.rs:148

```
impl Get<u32> for IngressMessageLimit {
    fn get() -> u32 {
        20 // TODO: Arbitrary value
    }
}
```

pallets/xcm-helper/src/lib.rs:241

```
/// Pending Withdrawals
#[pallet::storage]
#[pallet::getter(fn get_pending_withdrawls)]
pub(super) type PendingWithdrawals<T: Config> = StorageMap<
    _,
    Blake2_128Concat,
    T::BlockNumber,
    BoundedVec<PendingWithdrawal, ConstU32<100>>,
    ValueQuery,
>;
```

pallets/xcm-helper/src/lib.rs:252

```
/// Failed Withdrawals
#[pallet::storage]
#[pallet::getter(fn get_failed_withdrawls)]
pub(super) type FailedWithdrawals<T: Config> = StorageMap<
    _,
    Blake2_128Concat,
    T::BlockNumber,
    BoundedVec<PendingWithdrawal, ConstU32<100>>,
    ValueQuery,
>;
```

pallets/xcm-helper/src/lib.rs:261

```
/// Thea Assets, asset_id(u128) -> (network_id(u8), identifier_length(u8),
/// identifier(BoundedVec<>))
#[pallet::storage]
#[pallet::getter(fn get_thea_assets)]
pub type TheaAssets<T: Config> =
    StorageMap<_, Blake2_128Concat, u128, (u8, u8, BoundedVec<u8, ConstU32<1000>>), ValueQuery>;
```

pallets/xcm-helper/src/lib.rs:311

```
#[pallet::hooks]
impl<T: Config> Hooks<BlockNumberFor<T>> for Pallet<T> {
    fn on_initialize(n: T::BlockNumber) -> Weight {
        let mut failed_withdrawal: BoundedVec<PendingWithdrawal, ConstU32<100>> =
            BoundedVec::default();
        /* ... */
    }
    /* ... */
}
```

pallets/xcm-helper/src/lib.rs:683

```
/// Get Asset Info for given AssetId
pub fn get_asset_info(
    asset: AssetId,
) -> sp_std::result::Result<(u8, BoundedVec<u8, ConstU32<1000>>, usize), DispatchError> { /* ... */ }
```

pallets/xcm-helper/src/lib.rs:691

```
let asset_identifier = BoundedVec::try_from(asset_identifier.encode())
    .map_err(|_| Error::<T>::IdentifierLengthMismatch)?;
```

Recommendation

To mitigate these issues and improve the code quality, we recommend to move configurable constants to the pallet configuration, allowing them to be adjusted at runtime. this provides flexibility and allows for easier tuning of the system's behavior.
For example in `thea-pallet`, maximum active council members, maximum pending council members and maximum voters per proposal can be configuration parameters to the `Config` trait in `pallets/thea-council/src/lib.rs`:

```
// pallets/thea-council/src/lib.rs
/// Configure the pallet by specifying the parameters and types on which it depends.
#[pallet::config]
pub trait Config: frame_system::Config + xcm_helper::Config {
    /// Because this pallet emits events, it depends on the runtime's definition of an event.
    type RuntimeEvent: From<Event<Self>> + IsType<<Self as frame_system::Config>::RuntimeEvent>;
    /// Maximum number of active council members
    #[pallet::constant]
    type MaxActiveMembers: Get<u8>
    /// Maximum number of pending council members
    #[pallet::constant]
    type MaxPendingMembers: Get<u8>
    /// Maximum number of voter per proposal
    #[pallet::constant]
    type MaxVotersPerProposal: Get<u8> // See PDX-014 for this value
}
```

By adding the configurable constants to the pallet configuration, you can use these parameters in the declaration of the storage values.

```
// pallets/thea-council/src/lib.rs
/// Active Council Members
#[pallet::storage]
#[pallet::getter(fn get_council_members)]
pub(super) type ActiveCouncilMembers<T: Config> =
    StorageValue<_, BoundedVec<T::AccountId, T::MaxActiveMembers>, ValueQuery>
/// Pending Council Members
#[pallet::storage]
#[pallet::getter(fn get_pending_council_members)]
pub(super) type PendingCouncilMembers<T: Config> =
    StorageValue<_, BoundedVec<T::AccountId, T::MaxPendingMembers>, ValueQuery>;
/// Proposals
#[pallet::storage]
#[pallet::getter(fn proposal_status)]
pub(super) type Proposals<T: Config> = StorageMap<
    _,
```

```
    frame_support::Blake2_128Concat,
    Proposal<T::AccountId>,
    BoundedVec<Voted<T::AccountId>, T::MaxVotersPerProposal>,
    ValueQuery,
>;
```

And configure the pallet in the runtime as follows:

```
// runtime/src/lib.rs
parameter_types! {
    pub const MaxActiveMembers: u8 = 10;
    pub const MaxPendingMembers: u8 = 10;
    pub const MaxVotersPerProposal: u8 = 100; // See PDX-014 for this value
}
impl thea_council::Config for Runtime {
    type RuntimeEvent = Event;
    type MaxActiveMembers = MaxActiveMembers;
    type MaxPendingMembers = MaxPendingMembers;
    type MaxVotersPerProposal = MaxVotersPerProposal;
}
```

With these changes, it will be possible to configure the maximum number of active council members, pending council members, and voters per proposal at runtime. This approach makes the code more maintainable, flexible, and secure by avoiding the use of hardcoded values. Review and update other pallets in the project to follow this pattern and ensure a consistent codebase.

Missing Benchmark Calculations for Extrinsic Weights

The project currently lacks benchmark calculations for the extrinsic weights of all pallets, which is crucial for ensuring the efficiency and stability of the runtime.

ID	PDX-020
Scope	Code Quality / Performance
Status	Fixed

Details

A thorough examination of the project revealed that there are no benchmark calculations present for the extrinsic weights of all pallets. Extrinsics represent the various actions or operations that can be executed within the runtime environment. Accurate weight calculations are essential for maintaining a well-balanced and efficient runtime.

Failing to include benchmark calculations for extrinsic weights can result in:

- Unoptimized runtime performance
- Inaccurate or ineffective transaction fee calculations
- Unpredictable or undesirable behavior under high loads
- Imbalanced resource consumption across different extrinsics
- Potential vulnerabilities or attack vectors, such as denial-of-service (DoS) attacks

Recommendation

To ensure that the project benefits from a performant and stable runtime environment, it is recommended to include benchmark calculations for extrinsic weights of all pallets by following these steps:

- Identify all extrinsics within the project's pallets and group them accordingly.
- Set up a benchmarking framework, such as frame-benchmarking, to automate the process of calculating extrinsic weights.

- Create benchmarking tests for each extrinsic, focusing on various input parameters and potential edge cases.
- Run the benchmark tests on a range of hardware configurations to account for varying performance capabilities of potential nodes.
- Analyze the results to determine accurate weight values for each extrinsic, considering the expected usage patterns and resource consumption.
- Integrate the calculated extrinsic weights into the project's pallets and runtime configuration.
- Regularly review and update the benchmark calculations as the project evolves to maintain performance and efficiency.

By implementing benchmark calculations for extrinsic weights, you will ensure a more efficient and stable runtime environment, leading to optimized resource usage, accurate transaction fees, and overall improved performance in your project.

No Code Test Coverage

At commit `684fac00a125f45344b29bbd4003a2ec9af10f6b` (Scope of audit), the xcm configuration and the pallets have **0%** unit tests coverage.

ID	PDX-011
Scope	Code Quality / Testing
Status	Fixed

Details

At commit `684fac00a125f45344b29bbd4003a2ec9af10f6b` all the 5 pallets of the project: `asset-handler`, `router`, `swap`, `thea-council` and `xcm-helper` are not covered by unit tests.
`thea-council` pallet contains unit tests that can't compile due to a flawed implementation of the mocking runtime.

```
cargo test -p thea-council
...
error: could not compile `thea-council` (lib test) due to 49 previous errors; 2 warnings emitted
```

Pallet `asset-handler` contains a copy of the flaw unit tests related to `thea-council`.

```
diff -q pallets/asset-handler/src/tests.rs pallets/thea-council/src/tests.rs && diff -q pallets/asset-handler/src/mock.r
```

Pallets `router`, `swap`, and `xcm-helper` don't contain a single unit test.

We have noticed integration testing with the xcm-simulator, but it does not compile too:

```
cargo test -p xcm-simulator@0.1.0
error: could not compile `xcm-simulator` (lib test) due to 12 previous errors; 17 warnings emitted
```

Given the current status of the project, we were unable to test any functionality of the pallets and XCM with the provided tests.

Recommendation

We strongly recommend addressing the low test coverage in the pallets of the project. Implementing a comprehensive test suite consisting of unit tests is essential to ensure the security, stability, and maintainability of the project. Our recommendations are as follows:

1. **Fix existing unit tests**: Resolve the compilation issues in the `thea-council` pallet unit tests. Correct the flawed implementation of the mocking runtime to ensure the tests can be executed successfully.
2. **Implement unit tests for all pallets**: Develop unit tests for the `asset-handler`, `router`, `swap`, and `xcm-helper` pallets. Each pallet should have a thorough set of unit tests covering its functionality, edge cases, and potential error conditions.
3. **Fix existing xcm-simulator tests**: Resolve the compilation issues in the `xcm-simulator` tests.

4. **Use xcm-emulator**: `xcm-simulator` is a good tool for testing without the need for deploying to a live network by using mock relay chain and parachain runtime, but `xcm-emulator` uses production relay chain and parachain runtime and provides the ability to verify if specific XCM messages work in live networks.
5. **Establish a testing culture**: Encourage a testing culture within the development team, emphasizing the importance of thorough testing for the project's overall quality, reliability, and maintainability. Implement test-driven development (TDD) practices to ensure that new code is adequately tested before being integrated into the project.
6. **Automate testing and reporting**: Set up continuous integration (CI) systems to automate the execution of the test suite and generate regular code coverage reports. This will help the team identify any gaps in test coverage, regressions, or areas in need of improvement.
7. **Monitor and improve code coverage**: Continuously monitor the code coverage metrics and set specific goals for improvement. Regularly review the test suite and address any gaps or areas with low coverage to maintain a high level of code quality and security.

By implementing these recommendations, the project can significantly improve its test coverage, ensuring that the codebase is more secure, reliable, and maintainable in the long term.

Typographical Errors in the Project

This report highlights various typographical errors found throughout the project.

ID	PDX-024
Scope	Code Quality
Status	Fixed

Details

- Fix `Paracdhain` to `Parachain` and `foregin` to `foreign`. *pallets/xcm-helper/src/lib.rs:42*:

```
//! - **ParachainAsset** Type using which native Paracdhain will identify assets from foregin Parachain.
```

- Fix `Exector` to `Executor`. *pallets/xcm-helper/src/lib.rs:48*:

```
//! -[`TransactAsset`]: Used by XCM Exector /* ... /
```

- Fix `Convetor` to `Convertor`. *pallets/xcm-helper/src/lib.rs:198*:

```
/// Multilocation to AccountId Convetor
```

- Fix `PendingWithdrawals` getter name: `get_pending_withdrawls` to `get_pending_withdrawals`. *pallets/xcm-helper/src/lib.rs:235*:

```
#[pallet::storage]
#[pallet::getter(fn get_pending_withdrawls)]
pub(super) type PendingWithdrawals<T: Config> = /* ... */
```

- Fix `FailedWithdrawals` getter name: `get_failed_withdrawls` to `get_failed_withdrawals`. *pallets/xcm-helper/src/lib.rs:246*:

```
#[pallet::storage]
#[pallet::getter(fn get_failed_withdrawls)]
pub(super) type FailedWithdrawals<T: Config> = /* ... */
```

- Fix all 4 occurrences of `pubic_key` to `public_key` in `withdraw_asset` and `change_thea_key` extrinsics implemented in `xcm-helper` pallets.

- Fix `Non-Naitve` to `Non-Native`. *pallets/xcm-helper/src/lib.rs:620*:

```
/// Deposits Non-Naitve Token to Destination Account
```

- Fix `pending_withdrwal` to `pending_withdrawal`.
  pallets/xcm-helper/src/lib.rs:736-738:

```
let pending_withdrwal: &mut PendingWithdrawal =
    pending_withdrawals.get_mut(index as usize).ok_or(Error::<T>::IndexNotFound)?;
pending_withdrwal.is_blocked = true;
```

-

Recommendation

We recommend fixing the typographical errors identified and consider using a spell checker extension or tool to catch any further mistakes.

Unnecessary Checks and Wrong Error Variant in `do_claim_membership` function

The `do_claim_membership` function in the *pallets/thea-council/src/lib.rs* contains unnecessary checks and uses an incorrect error variant. This can be refactored to improve the code quality and avoid potential confusion.

ID	PDX-015
Scope	Code Quality
Status	Fixed

Details

The `do_claim_membership` function in the `thea-Council` pallet performs unnecessary checks for verifying the presence of a pending council member before adding them as an active council member (1). Additionally, the iteration over all pending council members handles a wrong error variant `NotActiveMember` that could never happen as a previous check confirms the presence or absence of the potential pending council member (2).

pallets/thea-council/src/lib.rs:275

```
fn do_claim_membership(sender: &T::AccountId) -> DispatchResult {
    let mut pending_members = <PendingCouncilMembers<T>>::get();
    // 1. Check for the presence of the pending member in PendingCouncilMembers storage value.
    ensure!(pending_members.contains(sender), Error::<T>::NotPendingMember);
    let index = pending_members
        .iter()
        .position(|member| *member == *sender)
        .ok_or(Error::<T>::NotActiveMember)?; //2. A wrong Error variant, that can never be triggered.
        /* ... */
}
```

Recommendation

`do_claim_membership` can be refactor to only verify the presence of the pending council member while iterating over `PendingCouncilMembers` storage value and dispatch a good error variants `NotPendingMember` if the potential pending council member is not found:

```
fn do_claim_membership(sender: &T::AccountId) -> DispatchResult {
    let mut pending_members = <PendingCouncilMembers<T>>::get();
    let index = pending_members
        .iter()
        .position(|member| *member == *sender)
        .ok_or(Error::<T>::NotPendingMember)?;
        /* ... */
}
```

This modification will simplify the logic and improve efficiency.

Unresolved TODO macros

Unfinished code and potential panics due to remaining `todo!()`.

ID	PDX-022
Scope	Code Quality
Status	Fixed

Details

The first instance is in the `can_withdraw` function: *pallets/asset-handler/src/lib.rs:126*:

```
fn can_withdraw(
    asset: Self::AssetId,
    who: &T::AccountId,
    amount: Self::Balance,
) -> WithdrawConsequence<Self::Balance> {
    return if asset != T::NativeCurrencyId::get() {
        let consequences = T::MultiCurrency::can_withdraw(
            asset.saturated_into(),
            who,
            amount.saturated_into(),
        );
        return consequences.into()
    } else {
        todo!()
    }
}
```

This function is responsible for checking whether a withdrawal can be performed. However, the `todo!()` macro is used as a placeholder for handling the case when the asset is the native currency. If this case is encountered in production, it will result in a panic.

The second instance is in the `Convert` trait implementation for the pallet `xcm-helper`: *pallets/xcm-helper-src/lib.rs:487*:

```
impl<T: Config> Convert<MultiLocation, Option<u128>> for Pallet<T> {
    fn convert(a: MultiLocation) -> Option<u128> {
        todo!()
    }
}
```

This function is supposed to convert a `MultiLocation` type into an `Option<u128>` type. However, it is not yet implemented and has a `todo!()` macro, which will cause a panic if called.

Recommendation

1. Replace the `todo!()` macro in the `can_withdraw` function with the appropriate logic for handling withdrawals involving the native currency. Ensure that the function works correctly for both native and non-native assets.

2. Implement the conversion logic in the `Convert` trait implementation for the pallet `xcm-helper`. Ensure that the conversion function works correctly and can handle various cases related to the `MultiLocation` type.

Unused function insert_pending_withdrawal

The `insert_pending_withdrawal` function in the `xcm-helper` pallet is currently unused.

ID	PDX-007
Scope	Code quality
Status	Fixed

Details

The `insert_pending_withdrawal` function is defined within the `xcm-helper` pallet, but it is not being utilized in the codebase. *pallets/xcm-helper/src/libs.rs:749*

```
/// Inserts new pending withdrawals
pub fn insert_pending_withdrawal(
    block_no: T::BlockNumber,
    pending_withdrawal: PendingWithdrawal,
) {
    let mut pending_withdrawals = <PendingWithdrawals<T>>::get(block_no);
    pending_withdrawals.try_push(pending_withdrawal).unwrap();
    <PendingWithdrawals<T>>::insert(block_no, pending_withdrawals);
}
```

This function is designed to insert a pending withdrawal into the `PendingWithdrawals` storage. However, pending withdrawals are currently being inserted through the `withdraw_asset` function, which directly mutates the `PendingWithdrawals` storage map instead of using the `insert_pending_withdrawal` function.

pallets/xcm-helper/src/libs.rs:359

```
pub fn withdraw_asset(/* .. */) -> DispatchResultWithPostInfo {
    /* ... */
    for (asset, dest) in payload {
        let pending_withdrawal =
            PendingWithdrawal { asset, destination: dest, is_blocked: false };
        <PendingWithdrawals<T>>::try_mutate(
            withdrawal_execution_block,
            |pending_withdrawals| {
                pending_withdrawals
                    .try_push(pending_withdrawal)
                    .map_err(|_| Error::<T>::PendingWithdrawalsLimitReached)
            },
        )?;
    }
    /* ... */
}
```

Recommendation

To improve code quality, maintainability, and future testing capabilities, we recommend either utilizing the `insert_pending_withdrawal` function where appropriate or removing it if deemed unnecessary.
Using the `insert_pending_withdrawal` function would enhance encapsulation and ensure that all pending withdrawals are inserted consistently.
If the function is not needed, removing it would help to declutter the code and eliminate potential confusion for developers working on the project.

Usage of sudo pallet with a root account

sudo FRAME pallet is used for thea key management with a root account.

ID	PDX-012
Scope	Code Quality / Decentralization
Status	Acknowledged (Team will remove sudo pallet after launch)

Details

The runtime configuration includes the `sudo-pallet`. *runtime/src/lib.rs:612*:

```
// Create the runtime by composing the FRAME pallets that were previously configured.
construct_runtime!(
    pub enum Runtime where
        Block = Block,
        NodeBlock = opaque::Block,
        UncheckedExtrinsic = UncheckedExtrinsic,
    {
        /* ... */
        Sudo: pallet_sudo::{Pallet, Call, Storage, Event<T>, Config<T>} = 45,
        /* ... */
    }
);
```

The root account is set at genesis in the following section: *node/src/chain_spec.rs:241*:

```
let root_key: AccountId =
        hex!["10d063a8244f2bce1f34e973891bc3b115bbd552d4f163e731047ace72e59d5f"].into();
```

This root account is used to arbitrarily set the thea key in `xcm-helper` pallet:

```
/// Initializes Thea Key.
///
/// # Parameters
///
/// * `thea_key`: Key which will be initialized.
#[pallet::weight(Weight::from_ref_time(10_000) + T::DbWeight::get().writes(1))]
pub fn set_thea_key(
    origin: OriginFor<T>,
    thea_key: [u8; 64],
) -> DispatchResultWithPostInfo {
    ensure_root(origin)?;
    <ActiveTheaKey<T>>::put(thea_key);
    <IngressMessages<T>>::try_mutate(|ingress_messages| {
        ingress_messages.try_push(TheaMessage::TheaKeySetBySudo(thea_key))
    })
    .map_err(|_| Error::<T>::IngressMessagesFull)?;
        Ok(().into())
}
```

While it is understood that this extrinsic is necessary to set an initial Thea key because an existing Thea key is required for using the `change_thea_key` extrinsic, which doesn't require root privileges, the `set_thea_key` function can still be used at any time to arbitrarily set the Thea key by a root account, even after the first Thea key setting.

Recommendation

We recommend addressing the potential security and decentralization concerns that arise from the usage of the sudo pallet for managing the Thea key. Our recommendations are as follows:

1. **Set the initial Thea key at genesis**: Instead of relying on the sudo pallet and root account to set the initial Thea key, include it in the genesis configuration. This way, the Thea key will be available from the beginning, without the need for any privileged actions.
2. **Limit the usage of the root account**: Modify the `set_thea_key` function to restrict the root account's ability to arbitrarily set the Thea key after the initial setting. This can be achieved by introducing an additional storage item to track whether the initial Thea key has been set, and only allowing the root account to set the key if it hasn't been set previously.
3. **Implement governance**: Consider implementing a governance mechanism, such as a multisig, committee, or on-chain governance, to manage the initialization of Thea key. This approach would increase decentralization and reduce reliance on a single root account.
4. **Audit and monitoring**: Regularly audit and monitor the use of the sudo pallet and the root account to ensure that they are not being misused or compromising the security of the system.
5. **Documentation and transparency**: Clearly document the intended use of the sudo pallet and the root account within the project. Ensure that both the development team and users are well-informed about potential risks and limitations associated with its usage. If there are plans to disable the sudo functionality after the network goes live, provide thorough documentation of this process, similar to how Polkadot outlined its sudo removal.

The current implementation does not pose an immediate security risk; however, it does raise potential centralization concerns and is a less desirable design choice. If the root account's private key were to be compromised, this issue could potentially escalate into a vulnerability, leading to unauthorized access to the privileged action of arbitrarily setting the Thea key, which could disrupt, lock, or grant unauthorized ownership of the entire asset withdrawal process.

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

Disclaimers

Hacken disclaimer

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

Technical disclaimer

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