HACKEN

POLKADEX ORDERBOOK 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 Orderbook
Website	https://polkadex.trade/orderbook
Repository	https://github.com/Polkadex-Substrate/orderbook
Commit	1f7dc6138f4c2e930852c193ac113d7c78918c43 and 7dc43b82503dead5ee01c05983060cf586e24ee2
Platform	L2-based exchange, which includes: AWS cloud functions, standalone binaries and L1-protocol client.
Network	Polkadex
Languages	Rust, Go
Methodology	Blockchain Protocol and Security Analysis Methodology
Lead Auditor	g.pakharenko@hacken.io
Approver	l.ciattaglia@hacken.io
Timeline	17.04.2023 - 19.05.2023
Changelog	29.05.2023 - Preliminary Report
Changelog	04.08.2023 - Final Report

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

Table of contents

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

Summary

Orderbook v2 is the next implementation of trading logic in the Polkadex network. Its documentation page describes it as a fully decentralized Layer 2 based exchange built on top of the Polkadot Network and featuring an order book. It implements a Layer 2 Trusted Execution Environment (TEE) on top of Polkadex. This mechanism allows an operator to maintain the Orderbook and controls that the operator cannot cheat since the results produced by the TEE contain cryptographic proof. The second version of Orderbook (v2) is an improvement of some components from the first version (v1). V2 documentation specifies the following components in its scope:

- AWS infrastructure (S3 buckets, SQS, Appsync, Dynamodb, Timestream database, etc.) They are used by other components to exchange data about trades and facilitate user registration.
- OCEX pallet. The OCEX pallet is the foundation for the fund security of the Polkadex Orderbook. This pallet handles all the critical operational tasks of the Orderbook via extrinsics.
- Engine. This program performs critical calculations like order-matching, updating balances of users etc, and thus is the most important program in the entire flow.
- State Change Handler. Coordinates data exchange between S3, timestream, DynamoDB
- Orderbook Worker. Is a client-side logic that runs along all the validators and full nodes of Polkadex solochain, and it works similarly to the BABE and Grandpa logic; it leverages a Substrate's gossip machinery and off-chain storage APIs to manage its state.
- Http wrapper. Uploads files to S3.
- Chain follower. Responsible for sending finalized blocks via SNS.

There are several Lambda functions in the scope:

- Chain-notifier. Listens for the blocks on the blockchain and publishes them to S3.
- Authoriser. Appsync uses this lambda to authorize requests sent by the user.
- User-actions. Handles user requests allowed by the authorizer lambda.
- Candlestick. Creates candlestick bars graph for trades.
- Ticker. Creates ticker analytics data for trades.

Other important components that were reviewed during this engagement:

- BLS primitives. Handle BLS FastAggregate signatures and based on the blst library
- Orderbook primitives. Include different types and utility functions to work with orders, trades, etc.

The code was in continuous development, but we worked mostly on the following commits:

- 1f7dc6138f4c2e930852c193ac113d7c78918c43 for Orderbook v2
- https://github.com/Polkadex-Substrate/Polkadex.git for Polkadex

Documentation Quality

The documentation supplied for the system covers key aspects of the various system components. However, it does have certain identifiable limitations:

- There's an absence of a detailed description of the order/trade business logic, which is critical for users and developers to understand the functioning of the system. Furthermore, the lifecycle, business rules, and possible state transitions of the system are also not adequately covered. These elements are essential to ensure effective use and maintenance of the system.

- The documentation lacks sufficient information regarding some critical security controls. Adequate documentation of these controls is fundamental to maintain the system's security and to educate users about secure usage practices.

- The initial network parameters and the system setup process have not been explained. Clear instructions on these aspects are essential for users and developers who wish to install, deploy, or further develop the system.

Despite these shortcomings in the documentation, the Polkadex team has been quite helpful, providing the necessary information and assistance for the audit.

Given the current state of the documentation, the Documentation Quality Score is assessed at **7** out of 10.

Code Quality

While the code has seen continuous improvements and refactorings, our review has identified some areas of concern:

- There's a notable absence of certain tests. Adequate testing is vital for any codebase as it validates the code's functionality and helps identify potential bugs or vulnerabilities early on.
- We've identified the use of certain ineffective programming patterns. These can lead to code that is harder to understand, maintain, and could potentially introduce bugs or vulnerabilities.
- There's an inappropriate use of mathematical operations in certain instances. This could lead to unexpected behaviors, especially in edge cases, and could potentially introduce security vulnerabilities.
- The lack of synchronization among different parts of the code was also noted. This could lead to race conditions, potential deadlocks, and could impact the overall performance and reliability of the system.
- Finally, the lack of sufficient input validation and limits, while not posing a significant risk, can still lead to potential bugs and security vulnerabilities.

It's important to note that even if the code currently functions without any apparent issues, the identified quality issues could increase the likelihood of future bugs and the required maintenance efforts if they are not addressed promptly.

Considering these factors, the Code Quality score is assessed to be **7** out of 10.

Architecture quality

This blockchain protocol leverages the strengths of cloud-based technology and Substrate's native blockchain architecture. It therefore embodies the core qualities of scalability and decentralization that define these technologies. However, the absence of a comprehensive integration test to examine the performance of all system components concurrently, coupled with a lack of documented state transitions, presents limitations. Without this information, it becomes challenging to fully predict the potential risks of bottlenecks, race conditions, and deadlocks. Such issues might surface as the user base grows in size.

In response to these anticipated challenges, the development team intends to keep a close watch on performance metrics and enforce necessary limits or restrictions when they identify a substantial risk of system performance degradation. The Orderbook is interconnected with other networks via another Polkadex component - Thea bridge, an element not considered in this evaluation (hence, the report excludes the interoperability aspect with other networks).

While users are expected to interact with the Orderbook components indirectly via the web interface, it's worth noting that this aspect fell outside the scope of our audit. Additionally, there's an evident lack of documented disaster recovery plans, an issue flagged in the findings.

Given these considerations, the architecture quality score has been evaluated as **7** on a scale of 10.

Security score

Our assessment identified **1** High, **1** Medium, and **1** Low severity security-related issues. These findings have been shared with the Orderbook development team for acknowledgment and subsequent investigation. All issues have been partially or fully resolved by the development team, with suitable explanations provided.

Post these fixes, the Security Score stands at a commendable **10** out of 10.

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

Total score

Considering all metrics, the total score of the report is **9.1** out of 10.

Findings count and definitions

Severity	Findings	Severity Definition
Critical	0	Vulnerabilities that can lead to a complete breakdown of the blockchain network's security, privacy, integrity, or availability fall under this category. They can disrupt the consensus mechanism, enabling a malicious entity to take control of the majority of nodes or facilitate 51% attacks. In addition, issues that could lead to widespread crashing of nodes, leading to a complete breakdown or significant halt of the network, are also considered critical along with issues that can lead to a massive theft of assets. Immediate attention and mitigation are required.
High	1	High severity vulnerabilities are those that do not immediately risk the complete security or integrity of the network but can cause substantial harm. These are issues that could cause the crashing of several nodes, leading to temporary disruption of the network, or could manipulate the consensus mechanism to a certain extent, but not enough to execute a 51% attack. Partial breaches of privacy, unauthorized but limited access to sensitive information, and affecting the reliable execution of smart contracts also fall under this category.
Medium	1	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	1	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	3	

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

Issues

Dynamic SQL

The candlestick/ticker lambdas accept the CandleStickResolverEvent structure and use its members to dynamically create an SQL query to the Timeseries DB (PostgreSQL under the hood). This can be used to trigger SQL injections. Dynamic SQL code is embedded in the state/change/timestream_client.go as well.

ID	PDXO-104
Scope	Candlestick/Ticker lambdas, state-change service
Vulnerability Type	CWE-89: SQL Injection
Severity	HIGH
Status	Partially Fixed (3caf926)

Description

The candlestick/ticker lambdas accept the CandleStickResolverEvent structure and use its members to dynamically create an SQL query to the Timeseries DB (PostgreSQL under the hood). This can be used to trigger SQL injections. Dynamic SQL code is embedded in the state/change/timestream_client.go as well.

The customer has added multiple validations in the following commits after the initial finding and explained that the use of prepared statements is complicated for the timeseries database. In addition, the vulnerable lambda functions accept data only from internal sources and the data itself is not critical. Taking into account that SQL injections can lead not only to the violation of target data integrity but in some cases even to remote code execution we are marking this finding as partially fixed.

```
//main.go
........
type CandleStickResolverEvent struct {
        From     string `json:"from"`
        To       string `json:"to"`
        Interval string `json:"interval"`
        Market   string `json:"market"`
}
.......
.......
func HandleRequest(ctx context.Context, bytes json.RawMessage) (CandleStickResolverResponse, error) {
        log.Infof("lambda invoked! event: %s", string(bytes))
        var event CandleStickResolverEvent
        err := json.Unmarshal(bytes, &event)
        log.Infof("event params %v!", event)
        if err != nil {
                return CandleStickResolverResponse{}, err
        }
        candles, err := client.QueryCandles(event)
        if err != nil {
                return CandleStickResolverResponse{}, err
        }
........
```

We see that CandleStickResolverEvent is passed to the client.QueryCandles, which in turn calls to CandlesQuery(market string, interval string, from string, to string) and uses fmt.Sprintf to dynamically create the SQL string:

```
// the candlestick code in the file timeStreamClient.go
 {
```

```
        dbName := os.Getenv("TIMESTREAM_DB")
        tableName := os.Getenv("TIMESTREAM_TABLE")
        return fmt.Sprintf(`WITH market_table AS (
    SELECT * FROM "%s"."%s" WHERE market='%s' AND time BETWEEN from_iso8601_timestamp('%s') AND from_iso8601_timestamp(
),
//the ticker lambda code in the timeStreamClient.go
func TickerQuery(market string, from string, to string) string {
        dbName := os.Getenv("TIMESTREAM_DB")
        tableName := os.Getenv("TIMESTREAM_TABLE")
        return fmt.Sprintf(`SELECT
    MAX(price) as high,
    MIN(price) as low,
    MAX_BY(price, time) as close,
    MIN_BY(price, time) as open,
    SUM(quantity) as volume,
        SUM(quote_qty) as quote_volume
FROM "%s"."%s"
WHERE market = '%s' AND time BETWEEN from_iso8601_timestamp('%s') AND from_iso8601_timestamp('%s')`,
                dbName, tableName, market, from, to)
}
```

We have created a simple unit test proof of concept to demonstrate this bug:

```
// timeStreamClient_test.go
func TestCandlesQueryAudit(t *testing.T) {
        query := CandlesQuery("PDEX-1' or 1='1", "5m", "2022-04-19T05:25:56.042Z", "2022-04-19T05:25:56.042Z")
        fmt.Println(query)
}
```

```
$ go test |grep '1='
    SELECT * FROM ""."" WHERE market='PDEX-1' or 1='1' AND time BETWEEN from_iso8601_timestamp('2022-04-19T05:25:56.042
```

Impact

A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands.

Recommendation

Follow the next best practices to avoid SQL injections:

- Utilize prepared statements https://go.dev/doc/database/prepared-statements
- Validate input value against allowed pattern https://medium.com/@apzuk3/input-validation-in-golang-bc24cdec1835
- Use domain specific types (like number or datetime) instead of strings

Fix the issue in the candlestick and ticker lambdas, and in the state-change timestream client.

Improper use of BLS encryption library routines

Improper use of BLS encryption library routines.

ID	PDXO-114
Scope	Cryptography, Security

Vulnerability Type	CWE-325: Missing Cryptographic Step
Severity	**MEDIUM**
Status	Fixed 65c62d9

Description

The Polkadex code which uses BLS routines from the blst library does not follow the library author recommendations and BLS-signatures IETF recommendations.

Upon acceptance of the finding the customer fully removed BLS primitives.

The lack of affinity check for public keys

The blst library Github page requires for implementor to verify affinity of the public key coordinates in the G1 group of pairing.

> The essential point to note is that it's the caller's responsibility to ensure that public keys are group-checked with blst_p1_affine_in_g1. This is because it's a relatively expensive operation and it's naturally assumed that the application would cache the check's outcome.

We did not find the calls to the blst_p1_affine_in_g1 function in the Polkadex codebase.

Absent proof of possession for public keys

The IETF recommendations on BLS cryptography mandate that public keys should be accompanied by proof of possession.

> All public keys used by Verify, AggregateVerify, and FastAggregateVerify MUST be accompanied by a proof of possession, and the result of evaluating PopVerify on each public key and its proof MUST be VALID.

While the code uses FastAggregateVerify we did not notice the proof of possession checks.

Insufficient checks against split zero attacks

There is a documented attack when several aggregated signature participants may collude to create a "0" signature which has multiple negative implications on the protocol. We did not identify any routines to address this attack.

The lack of automation to identify invalid signature creator

There is a documented attack when a malicious participant can make the signature invalid, and it is a non-trivial task to identify him, as signature aggregation hides the individual contribution to the signature. We did not find any automated routine to find the malicious node which breaks the signature aggregation.

Incompletness of BLST library

The `blst` library, which is utilized in the project, exhibits some potential incompleteness indicated by the presence of at least 8 `TODO`s within its codebase. These `TODO`s serve as reminders or placeholders for unfinished tasks or areas that require further implementation. While the presence of `TODO`s doesn't necessarily mean the library is unusable, it does warrant attention to ensure all intended functionality is properly implemented.

Impact

Improperly implementing cryptography measures in these specific cases can potentially result in denial of service attacks. However, it is important to acknowledge that there is also a likelihood of other information security breaches occurring.

Recommendation

Follow the guidelines from the BLST library, IETF recommendations and benchmark the implementation against known attacks. Particularly we recommend:

- Implement checks that each public key is in affine coordinates.
- Implement proof of possession checks for public keys.
- Implement unit tests for malicious participant who breaks the BLS aggregate signature and procedure to identify him.
- Investigate the risk of splitting zero attacks with the protocol designer. At least there is a possible approach to address this and similar threats - check whether all possible linear combinations of the public keys are equal to 0 (though it is infeasible if the number of users is large). See more here.

Engine main and proxy account withdrawal confusion

It is possible to execute certain "engine" service withdrawal functions for main account from the arbitrary (not linked) proxy account.

ID	PDXO-104
Scope	Engine
Vulnerability Type	CWE-285 Improper Authorization
Severity	LOW
Status	Fixed c47b2bb

Description

It is possible to execute certain "engine" service withdrawal functions for the main account from the arbitrary (not linked) proxy account. The withdraw function in engine.rs does not verify if the proxy account is linked to the main account in the WithdrawalRequest. The signature can be signed by any proxy account. The WithdrawalRequest::verify function is defined in the Polkadex/primitives/orderbook/src/types.rs and verifies the signature against the proxy public key.

The customer has acknowledged the issue and added a validation routine which checks that main and proxy accounts are linked. This mitigates the issue.

```
//engine.rs
    pub async fn withdraw(
        &mut self,
        request: WithdrawalRequest,
        changes: &mut StateChanges,
    ) -> Result<(), Error> {
        // Check if we should process more withdrawals
        if self.pending_withdrawals >= MAX_WITHDRAWALS_PER_SNAPSHOT {
            return Err(Error::WithdrawalLimitReached);
        }
        // Verify signature
        if !request.verify() {
            return Err(Error::SignatureCheckFailed);
        }
............
............
```

We have created a simple unit test proof of concept to demonstrate this bug:

```
//tests/withdaw.rs
#[tokio::test]
async fn withdraw_bob_balance() {
    let mut state_change = StateChanges::default();
    let (pair, _seed) = Pair::generate();
    let (pairBob, _seedBob) = Pair::generate();
```

```
    let alice_account = AccountId::from(pair.public());
    let bob_account = AccountId::from(pairBob.public());
    let (mut engine, _channel) = initialize_engine_and_channel().await;
    engine.add_balances(
        alice_account.clone(),
        get_trading_pair().base,
        Decimal::new(10000, 0),
    );
    engine.add_balances(
        bob_account.clone(),
        get_trading_pair().base,
        Decimal::new(10000, 0),
    );
    let payload = WithdrawPayloadCallByUser {
        asset_id: get_trading_pair().base,
        amount: "100".to_string(),
        timestamp: Default::default(),
    };
    let signature: Signature = pair.sign(&payload.encode()).into();
    let withdrawal_request = WithdrawalRequest {
        signature,
        payload,
        main: bob_account.clone(),
        proxy: alice_account.clone(),
    };
    assert!(engine
        .withdraw(withdrawal_request, &mut state_change)
        .await
        .is_ok());
    assert_balances(
        engine.get_balances(),
        bob_account.clone(),
        get_trading_pair().base,
        (
            &Decimal::from_str("9900").unwrap(),
            &Decimal::from_str("0").unwrap(),
        ),
    );
}
```

Put the above code in the v2/engine/serc/tests/withdraw.rs and launch the test. It will report that WithdrawalRequest signed by the Alice proxy account successfuly changes the main account balance for Bob.

```
cargo +nightly test
......
test tests::withdraw::withdraw_alice_balance ... ok
test tests::withdraw::withdraw_bob_balance ... ok
.....
```

Impact

This issue has Low severity because in the Polkadex orderbook client a check exists for WithdrwalMessage to ensure that proxy correlates with the main account. However, the lack of control in this layer violates the principle of multi-layer defense and, in the future, may lead to severe issues. Unsynchronization between the engine and chain balances may lead to bugs that are tricky to fix.

Recommendation

Verify that the proxy and main account are both linked with each other. Implement MultiSig or Threshold signature for withdrawal requests to ensure that requesting party has both proxy and main accounts in its possession.

Ambiguous return value in the authorization routine

The authorization lambda routine, that validates the submitted token, returns the ambiguous value (Access::ReadOnly, false) which prohibits access, but not gives the ReadOnly access.

ID	PDXO-101
Scope	Code quality

Description

The authorization lambda routine that validates the submitted token returns the ambiguous value (Access::ReadOnly, false), which prohibits access but not gives the ReadOnly access. The code of the validation routine in the lambda/authorizer/src/validation.rs

```
pub(crate) async fn validate_token(
    db_ref: &Client,
    token: String,
) -> anyhow::Result<(Access, bool)> {
    let admin_identifier = std::env::var("LAMBDA_ADMIN_SECRET")?;
    let read_only = std::env::var("READ_ONLY")?;
    return if token == admin_identifier {
        Ok((Access::Full, true))
    } else if token == read_only {
        Ok((Access::ReadOnly, true))
    } else if !check_is_db_inconsistent(db_ref).await?
        && check_proxy_registration_in_db(db_ref, token).await?
    {
        Ok((Access::ValidUser, true))
    } else {
        Ok((Access::ReadOnly, false))
    };
}
```

The access decision is made by the boolean part of the returned tuple, which goes into authorization JSON in the lambda/authorizer/src/main.rs

```
async fn function_handler(
.........
.........
    let (access, is_authorized) = validation::validate_token(db_client, token).await?;
    // Get the fields that are denied for this auth level.
    let denied_fields = validation::get_denied_fields_for_auth(access);
    Ok(AppSyncAuthorizerResponse {
        is_authorized,
        resolver_context: None,
        denied_field: Some(denied_fields),
    })
}
```

Impact

This issue is informational and serves as a reminder to address this ambiguous authorization variable in future updates, as it can mislead developers and become a source of authorization errors.

Recommendation

We recommend to use (Access::Denied, false) explicitly instead of (Access::ReadOnly, false). More details about the AWS AppSync authorization can be found here:

- https://docs.aws.amazon.com/appsync/latest/devguide/security-authz.html

Cancel order function signature verification relies only on the order id

Cancel order function verifies signature only against order id.

ID	PDXO-107
Scope	Code quality

Description

Cancel order function verifies the signature only against the order id. We can expect that potentially signed order ids can be used accidentally in other use cases, which may lead to cancel that orders by 3rd parties. The signature check is conducted only once in the v2/lambda/user-actions/src/actions.rs

```
pub async fn cancel_order(
    db: &DBClient,
    sns_ref: &SNSClient,
    id: OrderId,
    _main: AccountId,
    proxy: AccountId,
    market: String,
    signature: Signature,
) -> anyhow::Result<()> {
    // Verify signature
    if !verify_payload(&id, &proxy, signature) {
        return Err(anyhow::Error::msg("Signature verification failed"));
    }
```

Further, the check that orderId is linked with a proxy account is conducted in the dynamodb.rs:

```
pub async fn get_cancel_order_data(
    db: &Client,
    id: OrderId,
    proxy: AccountId,
    market: TradingPair,
) -> anyhow::Result<(Decimal, i64, OrderSide)> {
    let table_name = std::env::var("USERS_TABLE_NAME")?;
    let proxy_address = account_in_ss88_format(proxy.clone());
    println!("cancel order hash_key={proxy_address}");
    println!("cancel order range_key=order-{market}-{id:?}");
    let item = db
        .get_item()
        .table_name(table_name)
        .key("hash_key", AttributeValue::S(proxy_address))
        .key(
            "range_key",
            AttributeValue::S(format!("order-{market}-{id:?}")),
        )
        .send()
        .await?;
```

Note that the engine does not verify the canceled order signature at all.

Impact

This issue is informational and serves as a reminder to address the oversimplified payload for order cancellation in the future. Otherwise, if the orderId is being signed in any other use case, this can lead to the possibility of canceling that order for the attacker.

Recommendation

Implement a payload for order cancellation, which clearly specifies the cancellation requirement.

Cloudformation template is not synced with the code

The cloudformation template for DynamoDB is not synced with the code.

ID	PDXO-103
Scope	Code quality

Description

The cloudformation template for DynamoDB is not synced with the code. The field "cc" is present in the query in the lambda/chain-notifier/src/dynamodb_client.rs

```
impl DynamoDBClient {
.............
            .expression_attribute_values(":tt", AttributeValue::S(event.txn_type.to_string()))
.............
            .send()
            .await?;
        Ok(())
    }
```

But the "tt" attribute is absent in the cloud/cloudformation.yaml:

```
    AttributeDefinitions:
      - AttributeName: "hash_key"
        AttributeType: "S"
      - AttributeName: "range_key"
        AttributeType: "S"
      - AttributeName: "t"
        AttributeType: "N"
      - AttributeName: "st"
        AttributeType: "S"
      - AttributeName: "sid"
        AttributeType: "N"
      - AttributeName: "m"
        AttributeType: "S"
```

Impact

This issue is informational and serves as a reminder to address this unsynchronization in the future, as it may lead to application errors.

Recommendation

Implement "tt" attribute for dynamodb in the cloudformation.yaml.

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

Confusing mathematical operations

System supports some confusing mathematical operations with orders.

ID	PDXO-106
Scope	Code quality

Description

The system supports some confusing mathematical operations with orders. They include negative prices and quantities. We have created a simple test that proves this possibility.

```
Starting to process order Order { stid: 0, client_order_id: 0x0101010101010101010101010101010101010101010101010101010101010101
checking if match can happen
Starting to match orders
quote_order_qty: 6000, avg_filled_price: 0, filled_quantity: 0
quote_order_qty: 6000, avg_filled_price: 0, filled_quantity: 0
self.filled_quantity: -60.000
temp: 6000.000
Quotient: -100
self.filled_quantity: -60.000
temp: 6000.000
Quotient: -100
Maker amount: -60.000, taker amount: 6000.000
quote_order_qty: 6000, avg_filled_price: -100, filled_quantity: -60.000
```

The unit test code should be added to the v2/engine/src/tests/orderbook.rs

```rust
#[tokio::test]
pub async fn test_orderbook_limit_buy_market_audit() {
    let mut state_change = StateChanges::default();
    let trading_config = get_trading_pair_config();
    let (mut engine, _channel) = initialize_engine_and_channel().await;
    engine.open_market_config(trading_config.clone());
    engine.add_balances(
        get_alice_accounts().0,
        get_trading_pair().base,
        Decimal::new(10000, 0),
    );
    engine.add_balances(
        get_alice_accounts().0,
        get_trading_pair().quote,
        Decimal::new(10000, 0),
    );
    engine.add_balances(
        get_bob_accounts().0,
        get_trading_pair().base,
        Decimal::new(10000, 0),
    );
    engine.add_balances(
        get_bob_accounts().0,
        get_trading_pair().quote,
        Decimal::new(10000, 0),
    );
    let mut ask_order_1 =
        get_limit_order_for_alice(OrderSide::Ask, Decimal::new(1, 0), Decimal::new(-100, 0));
    assert!(engine
        .process_order(&mut ask_order_1, &mut state_change)
        .await
        .is_ok());
}
```

The code by default uses Bankers rounding to the nearest even, which differs from the usually expected rounding operations. We hope that all traders know special rounding rules in financial math. The code that rounds is in the Polkadex/primitives/orderbook/src/types.rs:

```
#[cfg(feature = "std")]
impl TryFrom<OrderDetails> for Order {
        type Error = anyhow::Error;
        fn try_from(details: OrderDetails) -> Result<Self, anyhow::Error> {
                let payload = details.payload;
                if let Ok(qty) = payload.qty.parse::<f64>() {
                        if let Ok(price) = payload.price.parse::<f64>() {
                                return if let Some(qty) = Decimal::from_f64(qty) {
                                        if let Some(price) = Decimal::from_f64(price) {
                                                if let Ok(quote_order_qty) = payload.quote_order_quantity.parse::<f64>(
                                                        if let Some(quote_order_qty) = Decimal::from_f64(quote_order_qty
                                                                if let Ok(trading_pair) = payload.pair.try_into() {
                                                                        Ok(Self {
                                                                                stid: 0,
                                                                                client_order_id: payload.client_order_id
                                                                                avg_filled_price: Decimal::zero(),
                                                                                fee: Decimal::zero(),
                                                                                filled_quantity: Decimal::zero(),
                                                                                id: H256::random(),
                                                                                status: OrderStatus::OPEN,
                                                                                user: payload.user,
                                                                                main_account: payload.main_account,
                                                                                pair: trading_pair,
                                                                                side: payload.side,
                                                                                order_type: payload.order_type,
                                                                                qty: qty.round_dp(8),
                                                                                price: price.round_dp(8),
                                                                                quote_order_qty: quote_order_qty.round_
                                                                                timestamp: payload.timestamp,
                                                                                overall_unreserved_volume: Decimal::zero
                                                                                signature: details.signature,
```

In addition, the same code for some reason uses string parsing into f64 and then into Decimal, while Decimal supports parsing from strings directly.

Another example of non-usual rounding is in the Engine component which refers to the Polkadex/primitives/orderbook/src/types.rs file:

```
pub fn rounding_off(a: Decimal) -> Decimal {
        // if we want to operate with a precision of 8 decimal places,
        // all calculations should be done with latest 9 decimal places
        a.round_dp_with_strategy(9, RoundingStrategy::ToZero)
}
```

The RoundingStrategy::ToZero always rounds to zero (e.g. 6.8->6), which is not what general user will expect.

Impact

This issue is informational and serves as a reminder to implement expected by general public math operations for orders or properly document them and periodically remind users about them.

Recommendation

Implement the following actions:

- Prohibit negative prices and quantities;
- Use direct converstion from string to Decimal instead of intermediate f64;
- Implement usual style rounding instead of Banker's style:
- https://docs.rs/rust_decimal/latest/rust_decimal/enum.RoundingStrategy.html#variant.BankersRounding

Dependency on the order of validators

The malicious validator can attempt to collect more signatures on the UnprocessedSnapshots item, as the Orderbook SnapshotSummary bitflags field respects only the order of validator, but not its public key.

ID	PDXO-116
Scope	Code quality

Description

The OCEX::submit_snapshot function adds the signature on a newly submitted snapshot if the validator has a different index in the [OCEX]::Authorities storage vector. In any case, when the Authorities vector gets for the same validator a new place (index) it is treated as a completely new validator and can add the signature again.

Simple exploitation path:

- Say we have some validator set (V1, V2, V3), and V1 is cheating. He is watching the OCEX::Authorities and OCEX::NextAuthorities. If he observes that his position is different in the Authorities and NextAuthorities e.g. (V1, V2, V3) => (V4, V1, V2) he conducts the following steps:

1. Just before the new session event he submits the snapshot with his bit index ([1,0,0]).
2. After change of Athorities he submits the same snapshot again with his new bit index ([0,1,0]).

In such a way he is able to collect two signatures on the same snapshot and pass the check of 2/3 validators.

We wrote a simple unit test to demonstrate that:

```
//Polkadex/pallets/ocex/src/tests.rs
........
fn test_submit_snapshot_two() {
        let _account_id = create_account_id();
        let mut t = new_test_ext();
        println!("In test");
        t.execute_with(|| {
                let (mut snapshot, _public) = get_dummy_snapshot(1);
                snapshot.withdrawals[0].fees = Decimal::from_f64(1.0).unwrap();
                let mut withdrawal_map = BTreeMap::new();
                for withdrawal in &snapshot.withdrawals {
                        match withdrawal_map.get_mut(&withdrawal.main_account) {
                                None => {
                                        withdrawal_map
                                                .insert(withdrawal.main_account.clone(), vec![withdrawal.clone()]);
                                },
                                Some(list) => {
                                        list.push(withdrawal.clone());
                                },
                        }
                }
                snapshot2 = snapshot.clone();
                snapshot2.bitflags = vec![0,1,0];
                snapshot.bitflags = vec![1,0,0];
                assert_ok!(OCEX::submit_snapshot(RuntimeOrigin::none(), snapshot.clone()));
                assert_ok!(OCEX::submit_snapshot(RuntimeOrigin::none(), snapshot2.clone()));
```

We also instrumented OCEX code with sum debug routines to monitor its internal state and we can see that the snapshot with the same hash was added two times and got processed.

```
last_snaphost_serial_number 0
```

```
Summary hash 0x1111…3263
last_snaphost_serial_number 0
Summary hash 0x1111…3263
Auth_index 248
total_validators 3
len 2
```

Impact

This is only an informational finding which reminds to properly document system behavior when the order of Orderbook authorities changes and align the code with documented functional requirements. We were not able to get sufficient assurance about exloitability of this bug due to the lack of unit test coverage for change of Orderbook authorities, as well as the lack of integrated tests for the submit_snapshot functionality. The unit test in the code does not call the ValidateUnsigned hook and does not invoke signature validation in the unit testing.

Recommendation

1. Properly document how the system should behave if Orderbook book authorities get changed (especially if their order is changed in the Authorities vector). Align the code with documentation.
2. Implement unit and integrational testing to cover snapshot processing before and after the change of Authorities.

Inconsistent configuration for AppSync endpoint authorization between the lambda and cognito user pool authorizers.

Authorization lambda implements support for the register_user mutation, while the graphql schema relies on the cognito user pool for authorization of this mutation.

ID	PDXO-100
Scope	Code quality

Description

The AppSync graphql schema in the cloud/schema.graphql specifies that regiter_user mutation is secured by the @aws_cognito_user_pools authorization:

```
type Mutation {
    register_user(input: UserActionInput!): String
    @aws_cognito_user_pools
    place_order(input: UserActionInput!): String
    @aws_lambda
    cancel_order(input: UserActionInput!): String
    @aws_lambda
    withdraw(input: UserActionInput!): String
    @aws_lambda
    publish(name: String!, data: String!): Channel
    @aws_lambda
}
```

While the authorizer lambda implements the register_user in lambda/authorizer/src/validation.rs:

```
pub fn get_denied_fields_for_auth(access: Access) -> Box<[String]> {
    match access {
        Access::ReadOnly => Box::new([
            String::from("Mutation.place_order"),
            String::from("Mutation.cancel_order"),
            String::from("Mutation.withdraw"),
```

```
            String::from("Mutation.publish"),
            String::from("Mutation.register_user"),
        ]),
        Access::ValidUser => Box::new([String::from("Mutation.publish")]),
        Access::Full => Box::new([]),
    }
}
```

Impact

This issue is informational and serves as a reminder to address this duplication of functionality in future updates, as it can mislead developers and become a source of authorization errors.

Recommendation

We recommend to properly document authorization decision archirecture, create functional requirements, align the code with that requirement and implement unit testing of the authorization mechanism. More details about the AWS AppSync authorization can be found here:

- https://docs.aws.amazon.com/appsync/latest/devguide/security-authz.html

Insufficient documentation and unit testing for the peer role control

The Orderbook client gossip logic relies on the underlying substrate code to check if the connected peer is FullNode or Validator, without properly documenting and testing this security control.

ID	PDXO-115
Scope	Code quality

Description

The Orderbook client gossip logic relies on the underlying substrate code to check if the connected peer is FullNode or Validator, without properly documenting and testing this security control. Particularly the ObservedRole object has the following comment in the sc_network page: "Role that the peer sent to us during the handshake, with the addition of what our local node knows about that peer". We cannot trust the role that peer sent to us during the handshake without validating it in the list of signatures.

```
//Polkadex/clients/orderbook/src/gossip.rs
........
impl<B> Validator<B> for GossipValidator<B>
where
        B: Block,
{
        fn new_peer(&self, _context: &mut dyn ValidatorContext<B>, who: &PeerId, role: ObservedRole) {
                info!(target:"orderbook","New peer connected: {:?}, role: {:?}",who,role);
                match role {
                        ObservedRole::Authority => {
                                self.validators.write().insert(*who);
                        },
                        ObservedRole::Full => {
                                self.fullnodes.write().insert(*who);
                        },
                        _ => {},
                };
        }
```

The Substrate documentation specifies that ObservedRole is a merged view of the local role information and reported information from the peer. The way how the peer-reported information should be properly validated is absent.

```
//https://paritytech.github.io/substrate/master/src/sc_network_common/role.rs.html#28
use codec::{self, Encode, EncodeLike, Input, Output};
/// Role that the peer sent to us during the handshake, with the addition of what our local node
/// knows about that peer.
///
/// > **Note**: This enum is different from the `Role` enum. The `Role` enum indicates what a
/// >                   node says about itself, while `ObservedRole` is a `Role` merged with the
/// >                   information known locally about that node.
#[derive(Debug, Clone)]
pub enum ObservedRole {
    /// Full node.
    Full,
    /// Light node.
    Light,
    /// Third-party authority.
    Authority,
}
```

Impact

This is only an informational finding to remind in the future properly document and cover with unit testing both use-case and abuse-case scenarios for verifying the peer role set. The Substrate implementation of the ObservedRole mechanism may change in the future and without unit testing the development team has a high probability to miss this important change. Improper verification of the peer roles can lead to the blockchain compromise.

Recommendation

1. Properly document the mechanism of role verification for remote peers.
2. Implement unit tests to cover malicious node behavior which tries to present itself as a validator.

Null pointer dereference

Httpwrapper service does not properly handles the lack of a filename header and crashes.

ID	PDXO-111
Scope	Code quality

Description

The Httpwrapper service code does not properly handles the lack of a filename header and crashes. Here is the invocation and the stack trace:

```
curl   -H "Content-Type: application/json" -d '{"productId": 123456, "quantity": 100}'  http://localhost:3333/upload
2023/05/11 23:33:34 http: panic serving 127.0.0.1:51304: runtime error: invalid memory address or nil pointer dereferenc
goroutine 116 [running]:
net/http.(*conn).serve.func1()
        /usr/lib/go-1.19/src/net/http/server.go:1850 +0xbf
panic({0x91e400, 0xf5a050})
        /usr/lib/go-1.19/src/runtime/panic.go:890 +0x262
main.HandleFileUpload({0xc3bfd8, 0xc0005d8000}, 0xc000436600)
        /home/polkadex/orderbook-1f7dc61-develop/v2/httpwrapper/httpHandlers.go:55 +0x54d
net/http.HandlerFunc.ServeHTTP(0xc0005d8000?, {0xc3bfd8?, 0xc0005d8000?}, 0x9b708c?)
        /usr/lib/go-1.19/src/net/http/server.go:2109 +0x2f
net/http.(*ServeMux).ServeHTTP(0x0?, {0xc3bfd8, 0xc0005d8000}, 0xc000436600)
        /usr/lib/go-1.19/src/net/http/server.go:2487 +0x149
net/http.serverHandler.ServeHTTP({0xc3afe0?}, {0xc3bfd8, 0xc0005d8000}, 0xc000436600)
        /usr/lib/go-1.19/src/net/http/server.go:2947 +0x30c
net/http.(*conn).serve(0xc0000a4320, {0xc3c580, 0xc000320210})
```

```
        /usr/lib/go-1.19/src/net/http/server.go:1991 +0x607
created by net/http.(*Server).Serve
        /usr/lib/go-1.19/src/net/http/server.go:3102 +0x4db
```

Impact

This issue is informational and serves as a reminder to properly handle null values. Service crashes drastically affect its performance and provide a bad image about the application quality, thus reducing general trust in it.

Recommendation

Check variables for null values before passing them for further processing.

Place order does not verify the link between the proxy and main accounts

Place order does not verify the link between the proxy and main accounts

ID	PDXO-108
Scope	Code quality

Description

The place order function does not verify the link between the main and proxy accounts. This opens the possibility of putting orders without a main account. The code v2/lambda/user-actions/src/actions.rs verifies that the request was submitted by someone who can just create a signature. E.g. the attacker may generate some account keys, and put the OrderPayload::user field value to his public key.

```
pub async fn place_order(
    db: &DBClient,
    sns_ref: &SNSClient,
    payload: OrderPayload,
    signature: Signature,
) -> anyhow::Result<()> {
    // check if order has expired. order timestamp is in ms
    is_order_expired(payload.timestamp)?;
    // Check signature
    if !verify_payload(&payload, &payload.user, signature.clone()) {
        return Err(anyhow::Error::msg("Signature verification failed"));
    }
    let config = batch_read_and_verify_order(db, &payload).await?;
```

Further, the check is conducted against the main account in the dynamodb.rs:

```
pub async fn batch_read_and_verify_order(
    db: &Client,
    payload: &OrderPayload,
) -> anyhow::Result<TradingPairConfig> {
    let table_name = std::env::var("USERS_TABLE_NAME")?;
    let pair = TradingPair::try_from(payload.pair.clone())?;
    let main_address = account_in_ss88_format(payload.main_account.clone());
    let proxy_address = account_in_ss88_format(payload.user.clone());
    let (asset_id, required_amount) = match payload.side {
        OrderSide::Ask => {
            // Sellers are selling base assets
            (pair.base, Decimal::from_str(&payload.qty)?)
        }
        OrderSide::Bid => {
            // Bidders are buying using quote assets
```

```
            match payload.order_type {
                // price*qty for bid limit order
                OrderType::LIMIT => (
                    pair.quote,
                    Decimal::from_str(&payload.qty)?
                        .saturating_mul(Decimal::from_str(&payload.price)?),
                ),
                // quote_order_qty for bid market order
                OrderType::MARKET => (
                    pair.quote,
                    Decimal::from_str(&payload.quote_order_quantity)?,
                ),
            }
        }
    };
    // Batch read the required data from Dynamodb
    println!("checking from user hash key proxy-{proxy_address}");
```

It can be seen that the batch_read_and_verify_order function tells that is going to check from user hash key proxy-{proxy_address}, but this check is not performed.

```
    let config_query = db
        .get_item()
        .table_name(table_name.clone())
        .key(
            "hash_key".to_string(),
            AttributeValue::S("market".to_string()),
        )
        .key("range_key".to_string(), AttributeValue::S(pair.to_string()))
        .send();
    let balance_query = db
        .get_item()
        .table_name(table_name.clone())
        .key(
            "hash_key".to_string(),
            AttributeValue::S(main_address.clone()),
        )
        .key(
            "range_key".to_string(),
            AttributeValue::S(format!("balance-{asset_id}")),
        )
        .send();
    let (config_response, balance_response) = tokio::try_join!(config_query, balance_query)?;
    // Get this trading pair if this is active
    println!("config response from db {:?}", config_response.item());
    println!("balance response from db {:?}", balance_response.item());
    //check if trading pair is present
    let config_str = config_response
        .item()
        .ok_or(anyhow::Error::msg("trading pair not fetched"))?
        .get("config")
        .ok_or(anyhow::Error::msg("cannot find config"))?
        .as_s()
        .map_err(|_| anyhow::Error::msg("cannot parse config as string"))?;
    let config_in_db: TradingPairConfig = serde_json::from_str(config_str)?;
    assert!(config_in_db.operational_status);
    // Check if user has enough free balance
    let free_val = balance_response
        .item()
        .ok_or(anyhow::Error::msg("balance item not found"))?
        .get("f")
        .ok_or(anyhow::Error::msg("free balance not found"))?
        .as_n()
        .map_err(|_| anyhow::Error::msg("cannot parse balance as number"))?;
    let free = Decimal::from_str(free_val)?;
    if required_amount > free {
        return Err(anyhow::Error::msg(format!(
            "Not enough balance: free: {free:?}, required: {required_amount:?}"
        )));
```

```
    }
    Ok(config_in_db)
}
```

Impact

This issue is informational and serves as a reminder to explicitly check the link between the proxy and main addresses. As there are no unit tests for the user-actions lambda we were unable to fully reproduce this weakness in the allocated timeframe.

Recommendation

Implement explicit checks for the link between the main and proxy accounts in every component of the system.

Strings usage instead of objects

The code in the candlestick lambda dynamically creates the string from the SQL query result and then parses it into an array. While the best way is to create the array or specific structure without an aid of intermediate strings.

ID	PDXO-110
Scope	Code quality

Description

The code in the candlestick/ticker lambdas dynamically creates the string from the SQL query result and then parses it into an array. While the best way is to create the array or specific structure without the aid of intermediate strings. We see that the processRowType function constructs a string with elements in it.

```
func processRowType(data []*timestreamquery.Datum, metadata []*timestreamquery.ColumnInfo) string {
        value := ""
        for j := 0; j < len(data); j++ {
                if metadata[j].Type.ScalarType != nil {
                        // process simple data types
                        value += processScalarType(data[j])
                } else if metadata[j].Type.TimeSeriesMeasureValueColumnInfo != nil {
                        // fmt.Println("Timeseries measure value column info")
                        // fmt.Println(metadata[j].Type.TimeSeriesMeasureValueColumnInfo.Type)
                        datapointList := data[j].TimeSeriesValue
                        value += "["
                        value += processTimeSeriesType(datapointList, metadata[j].Type.TimeSeriesMeasureValueColumnInfo
                        value += "]"
                } else if metadata[j].Type.ArrayColumnInfo != nil {
                        columnInfo := metadata[j].Type.ArrayColumnInfo
                        // fmt.Println("Array column info")
                        // fmt.Println(columnInfo)
                        datumList := data[j].ArrayValue
                        value += "["
                        value += processArrayType(datumList, columnInfo)
                        value += "]"
                } else if metadata[j].Type.RowColumnInfo != nil {
                        columnInfo := metadata[j].Type.RowColumnInfo
                        datumList := data[j].RowValue.Data
                        value += "["
                        value += processRowType(datumList, columnInfo)
                        value += "]"
                } else {
                        log.Fatal("Bad column type")
                }
                // comma seperated column values
                if j != len(data)-1 {
```

```
                        value += ", "
                }
        }
        return value
}
.............
.............
func processCandleQueryResult(s string) CandleStick {
        s = strings.TrimPrefix(s, "[")
        s = strings.TrimSuffix(s, "]")
        candle := strings.Split(s, ",")
        log.Infof("candle: %v", candle)
        //note: this order is coming from how the query returns the data
        return CandleStick{
                Time:        candle[0],
                Volume:      candle[1],
                VolumeQuote: candle[2],
                High:        candle[3],
                Low:         candle[4],
                Open:        candle[5],
                Close:       candle[6],
        }
}
```

This coding pattern is duplicated across both the ticker and candlestick lambdas.

Impact

This issue is informational and serves as a reminder to use structures and objects to pass complex data types instead of strings. This provides more maintainable code, type validation and prevents tricky errors when the separator symbol may appear in the values and break the string parsing logic. Duplication of this code across several lambdas increases costs of further application code maintenance.

Recommendation

Use objects and structs to pass complex data types instead of strings. The code should not duplicate itself in the different lambda functions but should be packaged into a separate library.

The lack of email validation in the register user action.

The "register user" action does not validate its email address.

ID	PDXO-105
Scope	Code quality

Description

There is no email validation in the v2/lambda/user-actions/src/actions.rs against any kind of regular expression.

```
pub async fn register_user(
    db: &DBClient,
    email: String,
    main: AccountId,
    _signature: Signature,
) -> anyhow::Result<()> {
    let main_address =
        main.to_ss58check_with_version(Ss58AddressFormat::from(POLKADEX_MAINNET_SS58));
    // We create hex_u8 array, Same is done while signing in Polkadot.js
    let _payload = hex::decode(hex::encode(email.clone()))
        .map_err(|e| anyhow::Error::msg(format!("Unable to create hex uint: {e:?}")));
```

```
    // TODO: Verify signature is still not working
    // if !verify_payload(&payload?, &main, signature) {
    //     println!("Signature verification failed");
    //     return Err(anyhow::Error::msg("Signature verification failed"));
    // }
    let table_name = std::env::var("USERS_TABLE_NAME")?;
    db.update_item()
        .table_name(table_name)
        .key("hash_key", AttributeValue::S(email.clone()))
        .key("range_key", AttributeValue::S(email.clone()))
        .update_expression("add #accounts :account")
        .expression_attribute_names("#accounts", "accounts")
        .expression_attribute_values(":account", AttributeValue::Ss(vec![main_address]))
        .send()
        .await?;
    Ok(())
}
```

Impact

This issue is informational and serves as a reminder to implement proper email and other input validation filters in the future. The lack of input validation could lead to different errors, including dangerous ones like injections, scripting and request forgery.

Recommendation

Implement validation routines for email and other user supplied inputs.

The lack of limits to user registration

There are no limits to the number of addresses that user can register.

ID	PDXO-102
Scope	Code quality

Description

The user registration code does not limit the number of addresses the user can link to the account. The code which is in charge of registration is in cloud/graphql:

```
..........
..........
type MainAddressConnection @aws_lambda
@aws_cognito_user_pools {
    hash_key: String
    range_key: String
    accounts: [String]
}
..........
..........
type Mutation {
    register_user(input: UserActionInput!): String
    @aws_cognito_user_pools
..........
..........
```

As well as in the lambda/user-actions/src/actions.rs:

```
    db: &DBClient,
    email: String,
```

```
    main: AccountId,
    _signature: Signature,
) -> anyhow::Result<()> {
    let main_address =
        main.to_ss58check_with_version(Ss58AddressFormat::from(POLKADEX_MAINNET_SS58));
    // We create hex_u8 array, Same is done while signing in Polkadot.js
    let _payload = hex::decode(hex::encode(email.clone()))
        .map_err(|e| anyhow::Error::msg(format!("Unable to create hex uint: {e:?}")));
    // TODO: Verify signature is still not working
    // if !verify_payload(&payload?, &main, signature) {
    //     println!("Signature verification failed");
    //     return Err(anyhow::Error::msg("Signature verification failed"));
    // }
    let table_name = std::env::var("USERS_TABLE_NAME")?;
    db.update_item()
        .table_name(table_name)
        .key("hash_key", AttributeValue::S(email.clone()))
        .key("range_key", AttributeValue::S(email.clone()))
        .update_expression("add #accounts :account")
        .expression_attribute_names("#accounts", "accounts")
        .expression_attribute_values(":account", AttributeValue::Ss(vec![main_address]))
        .send()
        .await?;
    Ok(())
}
```

No limit is being applied to the number of accounts. In addition, the comment says that the signature of this request is not verified.

Impact

This issue is informational and serves as a reminder to address the lack of limits and signature verification in the future. The development team informed the auditing team that they execute the monitoring approach instead of hard limits, and only apply restrictions in the code when a realistic threat is observed.

Recommendation

We recommend to implement basic resource constraints to achieve proactive safeguards against potential abuses.

Undocumented behavior for set_snaphost_nonce

A very powerful function in the OCEX pallet - set_snapshot, which changes the snapshot id, lacks proper documentation and unit testing of consequences.

ID	PDXO-113
Scope	Code quality

Description

A very powerfull function in the OCEX pallet - set_snapshot, which changes the snapshot id, lacks proper documentation and unit testing of consequences.

```
//Polkadex/pallets/ocex/src/lib.rs
........
               /// Sets snapshot id as current. Callable by governance only
                ///
                /// # Parameters
                /// * `origin` - signed member of T::GovernanceOrigin
                /// * `new_snapshot_id` - u64 id of new *current* snapshot
                #[pallet::call_index(8)]
```

```
#[pallet::weight(<T as Config>::WeightInfo::set_snapshot())]
pub fn set_snapshot(origin: OriginFor<T>, new_snapshot_id: u64) -> DispatchResult {
        T::GovernanceOrigin::ensure_origin(origin)?;
        <SnapshotNonce<T>>::put(new_snapshot_id);
        Ok(())
}
```

Impact

Multiple places in the code assume sequential incrementation of the snapshot_id. And reverting the system state may require multiple additional manual corrections of the cloud state (dynamodb/appsync), chain data, and off-chain storage. The volatility of time required for these corrections decreases the benefits of having this mechanism in the source code base. This is only an informational finding and serves as a reminder for properly documenting and testing this functionality.

Recommendation

Any recovery control to operate properly should not rely solely on one technological aspect; therefore it is required to:

- Implement unit testing to cover that cloud and on-chain components of the system operate properly after resetting the snapshot.
- Document the consequences of changing the snapshot.
- Document and test disaster recovery scenarios when snapshot change is required.

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.)