



# Orion – Liquidity Aggregator

Smart Contract Security  
Assessment

Prepared by: Halborn

Date of Engagement: August 18th, 2023 – October 13th, 2023

Visit: [Halborn.com](https://Halborn.com)

DOCUMENT REVISION HISTORY	5
CONTACTS	6
1 EXECUTIVE OVERVIEW	7
1.1 INTRODUCTION	8
1.2 ASSESSMENT SUMMARY	8
1.3 TEST APPROACH & METHODOLOGY	9
2 RISK METHODOLOGY	10
2.1 EXPLOITABILITY	11
2.2 IMPACT	12
2.3 SEVERITY COEFFICIENT	14
2.4 SCOPE	16
3 ASSESSMENT SUMMARY & FINDINGS OVERVIEW	17
4 FINDINGS & TECH DETAILS	18
4.1 (HAL-01) INCORRECT PRICES ARE FETCHED DURING LIQUIDATION - CRITICAL(9.4)	20
Description	20
Code Location	20
BVSS	20
Recommendation	21
Remediation Plan	21
4.2 (HAL-02) USERS WITH NO ORN STAKED CAN NOT BE LIQUIDATED - HIGH(8.8)	22
Description	22
BVSS	25
Recommendation	26

Remediation Plan	26
4.3 (HAL-03) SELF-SIGNED ORDERS CAN BE REUSED IN OTHER CHAINS - HIGH(8.8)	27
Description	27
BVSS	28
Recommendation	28
Remediation Plan	28
4.4 (HAL-04) SWAPS COULD SET INACCURATE FILLORDERS - MEDIUM(5.6)	29
Description	29
Code location	29
BVSS	30
Recommendation	30
Remediation Plan	31
4.5 (HAL-05) MISSING STALENESS CHECKS IN THE CHAIN-LINK.LATESTROUNDDATA() CALLS - LOW(2.5)	32
Description	32
Code location	32
BVSS	32
Recommendation	32
Remediation Plan	33
4.6 (HAL-06) LACK OF DISABLEINITIALIZERS CALL TO PREVENT UNINITIALIZED CONTRACTS - LOW(2.5)	34
Description	34
BVSS	34
Recommendation	35
Remediation Plan	35
4.7 (HAL-07) HARDCODED DOMAIN SEPARATOR - LOW(2.5)	36

Description	36
BVSS	36
Recommendation	36
Remediation Plan	36
4.8 (HAL-08) HARDCODED CHAINID - LOW(2.5)	37
Description	37
BVSS	37
Recommendation	37
Remediation Plan	37
4.9 (HAL-09) SECRET VALUES CAN BE RETRIEVED FROM MEMPOOL - INFORMATIONAL(0.0)	38
Description	38
Code location	38
BVSS	39
Recommendation	39
Remediation Plan	39
4.10 (HAL-10) LACK OF A DOUBLE-STEP TRANSFEROWNERSHIP PATTERN - INFORMATIONAL(0.0)	40
Description	40
Code Location	40
BVSS	40
Recommendation	41
Remediation Plan	42
4.11 (HAL-11) USE CUSTOM ERRORS INSTEAD OF REVERT STRINGS TO SAVE GAS - INFORMATIONAL(0.0)	43
Description	43

	BVSS	43
	Recommendation	43
	Remediation Plan	43
5	AUTOMATED TESTING	44
5.1	STATIC ANALYSIS REPORT	45
	Description	45
	Results	46

## DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	10/09/2023
0.2	Document Updates	10/13/2023
0.3	Draft Review	10/13/2023
0.4	Draft Review	10/13/2023
1.0	Remediation Plan	11/03/2023
1.1	Remediation Plan Updates	11/06/2023
1.2	Remediation Plan Review	11/06/2023
1.3	Remediation Plan Review	11/06/2023

## CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	<a href="mailto:Rob.Behnke@halborn.com">Rob.Behnke@halborn.com</a>
Steven Walbroehl	Halborn	<a href="mailto:Steven.Walbroehl@halborn.com">Steven.Walbroehl@halborn.com</a>
Gabi Urrutia	Halborn	<a href="mailto:Gabi.Urrutia@halborn.com">Gabi.Urrutia@halborn.com</a>
Gokberk Gulgun	Halborn	<a href="mailto:Gokberk.Gulgun@halborn.com">Gokberk.Gulgun@halborn.com</a>



# EXECUTIVE OVERVIEW





## 1.1 INTRODUCTION

Orion engaged Halborn to conduct a security assessment on their smart contracts beginning on August 18th, 2023 and ending on October 13th, 2023. The security assessment was scoped to the smart contracts provided to the Halborn team.

## 1.2 ASSESSMENT SUMMARY

The team at Halborn was provided two weeks for the engagement and assigned a full-time security engineer to verify 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 security risks that were addressed and accepted by the Orion team.

## 1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can 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](#))
- 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. ([Brownie](#), [Anvil](#), [Foundry](#))

## 2. 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.

## 2.1 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)	1
	Specific (AO:S)	0.2
Attack Cost (AC)	Low (AC:L)	1
	Medium (AC:M)	0.67
	High (AC:H)	0.33
Attack Complexity (AX)	Low (AX:L)	1
	Medium (AX:M)	0.67
	High (AX:H)	0.33

Exploitability  $E$  is calculated using the following formula:

$$E = \prod m_e$$

## 2.2 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)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact  $I$  is calculated using the following formula:

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

## 2.3 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.

Coefficient ( $C$ )	Coefficient Value	Numerical Value
Reversibility ( $r$ )	None (R:N)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope ( $s$ )	Changed (S:C)	1.25
	Unchanged (S:U)	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



## 2.4 SCOPE

### IN-SCOPE CODE & COMMITS:

- Repository: `orion-exchange-private`
  - Commit ID: `45c67b084b957ca9203e397402885aeee1fb09f7`
  - Smart contracts **in scope**:
    - All smart contracts under `/contracts` folder **except** the ones located under `/contracts/dexes`.

### OUT-OF-SCOPE:

- Third-party libraries and dependencies.
  - Economic attacks.
- 

### REMEDIATION COMMITS:

- Repository: `orion-exchange-private`
  - Commit IDs:
    - `af075b821a966f058bfc5ab1aa8ee70c67953d6d`
    - `b206e0f56153cd0625dcaebb264066fdbe4993a1`

### 3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	2	1	4	3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) INCORRECT PRICES ARE FETCHED DURING LIQUIDATION	Critical (9.4)	SOLVED - 11/06/2023
(HAL-02) USERS WITH NO ORN STAKED CAN NOT BE LIQUIDATED	High (8.8)	SOLVED - 11/03/2023
(HAL-03) SELF-SIGNED ORDERS CAN BE REUSED IN OTHER CHAINS	High (8.8)	SOLVED - 11/03/2023
(HAL-04) SWAPS COULD SET INACCURATE FILLORDERS	Medium (5.6)	RISK ACCEPTED
(HAL-05) MISSING STALENESS CHECKS IN THE CHAINLINK.LATESTROUNDDATA() CALLS	Low (2.5)	RISK ACCEPTED
(HAL-06) LACK OF DISABLEINITIALIZERS CALL TO PREVENT UNINITIALIZED CONTRACTS	Low (2.5)	RISK ACCEPTED
(HAL-07) HARDCODED DOMAIN SEPARATOR	Low (2.5)	RISK ACCEPTED
(HAL-08) HARDCODED CHAINID	Low (2.5)	RISK ACCEPTED
(HAL-09) SECRET VALUES CAN BE RETRIEVED FROM MEMPOOL	Informational (0.0)	ACKNOWLEDGED
(HAL-10) LACK OF A DOUBLE-STEP TRANSFEROWNERSHIP PATTERN	Informational (0.0)	ACKNOWLEDGED
(HAL-11) USE CUSTOM ERRORS INSTEAD OF REVERT STRINGS TO SAVE GAS	Informational (0.0)	ACKNOWLEDGED



# FINDINGS & TECH DETAILS



## 4.1 (HAL-01) INCORRECT PRICES ARE FETCHED DURING LIQUIDATION - CRITICAL(9.4)

### Description:

In the commit `af075b821a966f058bfc5ab1aa8ee70c67953d6d`, during the re-mediation plan, some lines of code were added to allow a liquidator to specify the collateral asset that will be used to cover the debt instead of only `ORN` as it was implemented before these changes. However, when `collateralAsset` is not the address of `ORN` token, the code tries to fetch the collateral's price but instead of it, the `getAssetPrice` function uses as argument the `redeemedAsset` address. Therefore, both tokens will always have the same price in the following operations.

### Code Location:

In the `MarginalFunctionality.partiallyLiquidate()` function:

Listing 1: `MarginalFunctionality.sol` (Lines 279,283)

```

279     (uint64 price, uint64 ts1) = getAssetPrice(redeemedAsset,
    ↳ constants._oracleAddress);
280     require(ts1 + constants.priceOverdue > block.timestamp, "E9");
    ↳ //Price is outdated
281
282     if (collateralAsset != constants._orionTokenAddress) { //
283         (uint64 collateralPrice, uint64 ts2) = getAssetPrice(
    ↳ redeemedAsset, constants._oracleAddress);
284         require(ts2 + constants.priceOverdue > block.timestamp, "
    ↳ E9"); //Price is outdated
285         price = (price * 1e8) / collateralPrice;
286     }

```

### BVSS:

AO:A/AC:L/AX:L/C:N/I:H/A:N/D:H/Y:N/R:N/S:U (9.4)

#### Recommendation:

It is recommended to use the `collateralAsset` argument instead of `redeemedAsset` to fetch prices of the collateral token.

#### Remediation Plan:

**SOLVED:** The `Orion team` solved the issue by fetching prices from the collateral token instead of the one to redeem in the following commit ID:

- [b206e0f56153cd0625dcaebb264066fdb4993a1](#).

## 4.2 (HAL-02) USERS WITH NO ORN STAKED CAN NOT BE LIQUIDATED - HIGH (8.8)

### Description:

In the `Exchange` contract, the function `partiallyLiquidate()` can be called by any user to cover some of overdue broker liabilities and get `ORN` in exchange:

Listing 2: `Exchange.sol` (Line 391)

```

382 /**
383  * @dev method to cover some of overdue broker liabilities and get
384  *   ↳ ORN in exchange
385  *   ↳ same as liquidation or margin call
386  * @param broker - broker which will be liquidated
387  * @param redeemedAsset - asset, liability of which will be
388  *   ↳ covered
389  * @param amount - amount of covered asset
390  */
391 function partiallyLiquidate(address broker, address redeemedAsset,
392   ↳ uint112 amount) public {
393     MarginalFunctionality.UsedConstants memory constants =
394     ↳ getConstants(broker);
395     MarginalFunctionality.partiallyLiquidate(
396         collateralAssets,
397         liabilities,
398         assetBalances,
399         assetRisks,
400         constants,
401         redeemedAsset,
402         amount
403     );
404 }

```

This function internally called `MarginalFunctionality.partiallyLiquidate()`:

Listing 3: MarginalFunctionality.sol (Lines 281-289)

```

244 /**
245  * @dev partially liquidate, that is cover some asset liability to
    ↳ get
246      ORN from misbehavior broker
247  */
248 function partiallyLiquidate(
249     address[] storage collateralAssets,
250     mapping(address => Liability[]) storage liabilities,
251     mapping(address => mapping(address => int192)) storage
    ↳ assetBalances,
252     mapping(address => uint8) storage assetRisks,
253     UsedConstants memory constants,
254     address redeemedAsset,
255     uint112 amount
256 ) public {
257     //Note: constants.user - is broker who will be liquidated
258     Position memory initialPosition = calcPosition(
259         collateralAssets,
260         liabilities,
261         assetBalances,
262         assetRisks,
263         constants
264     );
265     require(
266         initialPosition.state == PositionState.NEGATIVE ||
    ↳ initialPosition.state == PositionState.OVERDUE,
267         "E7"
268     );
269     address liquidator = msg.sender;
270     require(assetBalances[liquidator][redeemedAsset] >= int192(
    ↳ uint192(amount)), "E8");
271     require(assetBalances[constants.user][redeemedAsset] < 0, "E15
    ↳ ");
272     assetBalances[liquidator][redeemedAsset] -= int192(uint192(
    ↳ amount));
273     assetBalances[constants.user][redeemedAsset] += int192(uint192
    ↳ (amount));
274
275     if (assetBalances[constants.user][redeemedAsset] >= 0)
276         removeLiability(constants.user, redeemedAsset, liabilities
    ↳ );
277
278     (uint64 price, uint64 timestamp) = getAssetPrice(redeemedAsset

```



```

    ↪ , constants._oracleAddress);
279     require((timestamp + constants.priceOverdue) > block.timestamp
    ↪ , "E9"); //Price is outdated
280
281     reimburseLiquidator(
282         amount,
283         price,
284         liquidator,
285         assetBalances,
286         constants.liquidationPremium,
287         constants.user,
288         constants._orionTokenAddress
289     );
290
291     Position memory finalPosition = calcPosition(
292         collateralAssets,
293         liabilities,
294         assetBalances,
295         assetRisks,
296         constants
297     );
298     require(
299         uint(finalPosition.state) < 3 && //POSITIVE,NEGATIVE or
    ↪ OVERDUE
300         (finalPosition.weightedPosition > initialPosition.
    ↪ weightedPosition),
301         "E10"
302     ); //Incorrect state position after liquidation
303     if (finalPosition.state == PositionState.POSITIVE)
304         require(finalPosition.weightedPosition < 10e8, "Can not
    ↪ liquidate to very positive state");
305 }

```

The liquidator is rewarded with **ORN**. This functionality is implemented in the `reimburseLiquidator()` function:

Listing 4: `MarginalFunctionality.sol` (Lines 325-327)

```

307 /**
308  * @dev reimburse liquidator with ORN: first from stake, than from
309  *   ↳ broker balance
310  */
311 function reimburseLiquidator(
312     uint112 amount,
313     uint64 price,
314     address liquidator,
315     mapping(address => mapping(address => int192)) storage
316     ↳ assetBalances,
317     uint8 liquidationPremium,
318     address user,
319     address orionTokenAddress
320 ) internal {
321     int192 _orionAmount = int192((int256(uint256(amount)) * int256
322     ↳ (uint256(price))) / 1e8);
323     _orionAmount += uint8Percent(_orionAmount, liquidationPremium)
324     ↳ ; //Liquidation premium
325     // There is only 100m Orion tokens, fits i64
326     require(_orionAmount == int64(_orionAmount), "E11");
327     int192 onBalanceOrion = assetBalances[user][orionTokenAddress
328     ↳ ];
329
330     require(onBalanceOrion >= _orionAmount, "E10");
331     assetBalances[user][orionTokenAddress] -= _orionAmount;
332     assetBalances[liquidator][orionTokenAddress] += _orionAmount;
333 }

```

Although, if the user being liquidated does not have any **ORN** in his balances, his position can never be liquidated.

**BVSS:**

**A0:A/AC:L/AX:L/C:N/I:H/A:N/D:M/Y:N/R:N/S:U (8.8)**

### Recommendation:

It is recommended to update the `partiallyLiquidate()` implementation, so users cannot avoid liquidation in this scenario.

### Remediation Plan:

**SOLVED:** The `Orion team` solved the issue by letting users choose which collateral asset to use for his liquidation reward in the following commit IDs:

- `af075b821a966f058bfc5ab1aa8ee70c67953d6d`.
- `b206e0f56153cd0625dcaebb264066fdb4993a1`.

## 4.3 (HAL-03) SELF-SIGNED ORDERS CAN BE REUSED IN OTHER CHAINS - HIGH (8.8)

Description:

The buy/sell orders are implemented with the following struct:

Listing 5: LibValidator.sol (Line 35)

```

35 struct Order {
36     address senderAddress;
37     address matcherAddress;
38     address baseAsset;
39     address quoteAsset;
40     address matcherFeeAsset;
41     uint64 amount;
42     uint64 price;
43     uint64 matcherFee;
44     uint64 nonce;
45     uint64 expiration;
46     uint8 buySide; // buy or sell
47     bool isPersonalSign;
48     bytes signature;
49 }

```

These buy/sell orders can be self-signed if `isPersonalSign` is set to true. As such, the sender will have to provide a signature which is built as:

Listing 6: LibValidator.sol

```

133 function getEthSignedOrderHash(Order memory _order) public pure
    ↳ returns (bytes32) {
134     return
135         keccak256(
136             abi.encodePacked(
137                 "order",
138                 _order.senderAddress,
139                 _order.matcherAddress,
140                 _order.baseAsset,

```

```

141         _order.quoteAsset,
142         _order.matcherFeeAsset,
143         _order.amount,
144         _order.price,
145         _order.matcherFee,
146         _order.nonce,
147         _order.expiration,
148         _order.buySide
149     )
150     ).toEthSignedMessageHash();
151 }

```

`chain.id` and as the Orion protocol is supposed to be deployed in multiple chains, this signature could be used to perform a cross-chain signature replay attack. For example:

1. Alice creates a personal signed order to buy 1000 `DAI` on the `Ethereum` mainnet.
2. This signature is replayed by a malicious user to send an order on behalf of Alice in the `Arbitrum` blockchain.

#### BVSS:

**A0:A/AC:L/AX:L/C:N/I:H/A:N/D:M/Y:N/R:N/S:U (8.8)**

#### Recommendation:

It is recommended to include a domain separator with a `chain.id` to the `getEthSignedOrderHash()` function in order to prevent signature replay attacks.

#### Remediation Plan:

**SOLVED:** The `Orion` team solved the issue by removing self-signed orders in the following commit ID:

- [af075b821a966f058bfc5ab1aa8ee70c67953d6d](#).

## 4.4 (HAL-04) SWAPS COULD SET INACCURATE FILLORDERS – MEDIUM (5.6)

### Description:

Every time an order is modified or filled, the `filledAmounts` mapping must be modified according to the amount of assets that will be covered regarding an order. The `updateFilledAmount` function is responsible for updating this mapping to track the current amount of assets already filled in an existing order.

However, there is a case where this mapping is not being updated accordingly. Since a swap can return an undetermined amount of tokens, the `fillThroughPools` function updates the `filledAmounts` mapping right before the swap is performed, taking as reference for the amount of tokens to fill to the `desc.minReturnAmount` variable used as slippage. Therefore, the protocol won't be able to track the correct amount of tokens filled in case more tokens are returned after the swap's execution.

### Code location:

Listing 7: `contracts/ExchangeWithGenericSwap.sol` (Lines 50,55)

```
49 LibValidator.checkOrderSingleMatch(order, desc, filledAmount,
   ↳ block.timestamp);
50 updateFilledAmount(order, filledAmount);
51 // stack to deep
52 {
53     address senderAddress = order.senderAddress;
54     uint256 matcherFee = order.matcherFee;
55     (returnAmount, spentAmount, gasLeft) = _swap(senderAddress,
   ↳ executor, desc, permit, data, matcherFee);
56 }
57 LibExchange._updateBalance(order.matcherAddress, order.
   ↳ matcherFeeAsset, int(uint(order.matcherFee)), assetBalances,
   ↳ liabilities);
```

Listing 8: contracts/libs/LibValidator.sol (Line 180)

```

168         uint256 amountQuote = uint256(filledAmount) * order.price /
    ↳ 10**8;
169     if (order.buySide == 1) {
170         require(order.quoteAsset == address(desc.srcToken) && order.
    ↳ baseAsset == address(desc.dstToken), "E3As");
171         (amount_spend, amount_receive) = (amountQuote,
    ↳ filledAmount);
172     } else {
173         require(order.baseAsset == address(desc.srcToken) && order.
    ↳ quoteAsset == address(desc.dstToken), "E3As");
174         (amount_spend, amount_receive) = (filledAmount,
    ↳ amountQuote);
175     }
176
177     require(order.senderAddress == desc.dstReceiver, "
    ↳ IncorrectReceiver");
178     require(amount_spend == desc.amount, "IncorrectAmount");
179     require(order.matcherFeeAsset == address(desc.dstToken), "
    ↳ IncorrectFeeToken");
180     require(amount_receive >= desc.minReturnAmount, "
    ↳ IncorrectAmount");

```

BVSS:

A0:A/AC:L/AX:L/C:N/I:N/A:L/D:M/Y:N/R:N/S:U (5.6)

Recommendation:

It is recommended to update the `filledAmounts` mapping with the amount obtained from swapping assets. For example, the following LOC could be added right after the swap:

Listing 9

```

1 updateFilledAmount(order, returnAmount - filledAmount);

```

### Remediation Plan:

**RISK ACCEPTED:** The Orion team accepted the risk of this issue and stated that:

This situation occurs when a trader creates a buy order that is filled through decentralized exchanges (dexes). Some dexes may not be able to execute exact receive trades, leading to the transformation of the trader's buyOrder into a sellOrder. While this approach may result in a final filledAmount greater than the initial order.amount, we prioritize ensuring that the trade's price is always better or at least equal to what the user initially signed in their order. We understand that there may be a slight discrepancy in the stored filledAmount value due to this process, but we maintain the commitment to ensuring that the settlement price is not worse than what the user expected.



## 4.5 (HAL-05) MISSING STALENESS CHECKS IN THE CHAINLINK.LATESTROUNDDATA() CALLS - LOW (2.5)

### Description:

In the `PriceOracle` contract, the `latestRoundData()` function of Chainlink price feeds is called to retrieve the price of different assets.

Although, this call is performed without any kind of staleness checks. If there is a problem with Chainlink starting a new round and finding consensus on the new value for the oracle (e.g. Chainlink nodes abandoning the oracle, chain congestion, vulnerability/attacks on the Chainlink system) consumers of these contracts may continue using obsolete data.

### Code location:

`PriceOracle.sol`

- Line 176:

```
(, int _basePrice, , uint timestamp, )= AggregatorV3Interface(
baseAggregator).latestRoundData();
```

- Line 198:

```
(, int _aPrice, , uint aTimestamp, )= AggregatorV3Interface(
currentAggregator).latestRoundData();
```

### BVSS:

A0:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (2.5)

### Recommendation:

It is recommended to use Chainlink's `latestRoundData()` function with checks on the return data with proper revert messages if the price is

stale or the round is incomplete, for example:

Listing 10: latestRoundData example call (Line 2)

```
1 uint256 private constant GRACE_PERIOD_TIME = 7200; // 2 hours
2 (uint80 roundId, int256 price, uint256 startedAt, uint256
↳ updatedAt, uint80 answeredInRound) = AggregatorV3Interface(
↳ baseAggregator).latestRoundData();
3 require(answeredInRound >= roundID, "Stale price");
4 require(price > 0, "invalid price");
5 require(block.timestamp <= updatedAt + GRACE_PERIOD_TIME, "Stale
↳ price");
```

If in the case that we are dealing with Price Feeds in Arbitrum, the sequencer state should be validated, ensuring that is up before accepting any price as valid.

Remediation Plan:

**RISK ACCEPTED:** The Orion team accepted the risk of this issue.

## 4.6 (HAL-06) LACK OF DISABLEINITIALIZERS CALL TO PREVENT UNINITIALIZED CONTRACTS - LOW (2.5)

### Description:

The `Exchange` contract uses the `Initializable` module from OpenZeppelin. In order to prevent leaving the contract uninitialized [OpenZeppelin's documentation](#) recommends adding the `_disableInitializers` function in the constructor to automatically lock the contract when it is deployed:

#### Listing 11: DisableInitializers function

```

1 /**
2  * @dev Locks the contract, preventing any future reinitialization
3  * . This cannot be part of an initializer call.
4  * Calling this in the constructor of a contract will prevent that
5  * contract from being initialized or reinitialized
6  * to any version. It is recommended to use this to lock
7  * implementation contracts that are designed to be called
8  * through proxies.
9  */
10 function _disableInitializers() internal virtual {
11     require(!_initializing, "Initializable: contract is
12     initializing");
13     if (_initialized < type(uint8).max) {
14         _initialized = type(uint8).max;
15         emit Initialized(type(uint8).max);
16     }
17 }

```

### BVSS:

A0:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (2.5)

#### Recommendation:

Consider calling the `_disableInitializers` function in the `Exchange` contract's constructor:

#### Listing 12: Constructor preventing uninitialized contracts

```
1 /// @custom:oz-upgrades-unsafe-allow constructor
2 constructor() {
3     _disableInitializers();
4 }
```

#### Remediation Plan:

**RISK ACCEPTED:** The `Orion team` accepted the risk of this issue.

## 4.7 (HAL-07) HARDCODED DOMAIN SEPARATOR - LOW (2.5)

### Description:

The variable `DOMAIN_SEPARATOR` in the `LibValidator` library is defined as a constant and will not change after the contract deployment. However, if a hard fork happens, the `DOMAIN_SEPARATOR` would become invalid on one of the forked chains as the `block.chainid` would have changed.

#### Listing 13: LibValidator.sol

```
24 bytes32 public constant DOMAIN_SEPARATOR =
25     keccak256(
26         abi.encode(
27             EIP712_DOMAIN_TYPEHASH,
28             keccak256(bytes(DOMAIN_NAME)),
29             keccak256(bytes(DOMAIN_VERSION)),
30             CHAIN_ID,
31             DOMAIN_SALT
32         )
33     );
```

### BVSS:

A0:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (2.5)

### Recommendation:

Consider using the [implementation from OpenZeppelin](#), which recalculates the domain separator if the current `block.chainid` is not the cached chain ID.

### Remediation Plan:

**RISK ACCEPTED:** The `Orion team` accepted the risk of this issue.

## 4.8 (HAL-08) HARDCODED CHAINID - LOW (2.5)

### Description:

The variable `CHAIN_ID` in the `LibValidator` library is defined as a constant and hardcoded to 1.

#### Listing 14: LibValidator.sol

```
12 uint256 public constant CHAIN_ID = 1;
```

### BVSS:

A0:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (2.5)

### Recommendation:

It is recommended to use `chain.id` to set the `CHAIN_ID` state variable. This can be done in the constructor during the contract deployment.

### Remediation Plan:

**RISK ACCEPTED:** The `Orion team` accepted the risk of this issue.

## 4.9 (HAL-09) SECRET VALUES CAN BE RETRIEVED FROM MEMPOOL – INFORMATIONAL (0.0)

### Description:

The `ExchangeWithAtomic` smart contract allows users to perform operations over different orders, locking them as `atomic`. These `atomic` orders require a hash associated to a secret that only the user who locks the order knows. This secret should be provided when the user wants to perform any of the available `atomic` operations.

However, since the secret is revealed when a function is executed over the locked order, during a period of time the secret remains readable for every user who is watching the `mempool` which receives the associated transaction. Therefore, these `atomic` operations do not give any kind of defense against front-run attacks, for example.

### Code location:

Listing 15: `contracts/libs/LibAtomic.sol` (Lines 82,90,91)

```

80 function doRedeemAtomic(
81     LibAtomic.RedeemOrder calldata order,
82     bytes calldata secret,
83     mapping(bytes32 => bool) storage secrets,
84     mapping(address => mapping(address => int192)) storage
    ↳ assetBalances,
85     mapping(address => MarginalFunctionality.Liability[]) storage
    ↳ liabilities
86 ) public {
87     require(!secrets[order.secretHash], "E17R");
88     require(getEthSignedAtomicOrderHash(order).recover(order.
    ↳ signature) == order.sender, "E2");
89     require(order.expiration / 1000 >= block.timestamp, "E4A");
90     require(order.secretHash == keccak256(secret), "E17");
91     secrets[order.secretHash] = true;

```

Listing 16: contracts/libs/LibAtomic.sol (Lines 100,108,109)

```

98 function doClaimAtomic(
99     address receiver,
100     bytes calldata secret,
101     bytes calldata matcherSignature,
102     address allowedMatcher,
103     mapping(bytes32 => LockInfo) storage atomicSwaps,
104     mapping(address => mapping(address => int192)) storage
    ↳ assetBalances,
105     mapping(address => MarginalFunctionality.Liability[]) storage
    ↳ liabilities
106 ) public returns (LockInfo storage swap) {
107     bytes32 secretHash = keccak256(secret);
108     bytes32 coHash = getEthSignedClaimOrderHash(ClaimOrder(
    ↳ receiver, secretHash));
109     require(coHash.recover(matcherSignature) == allowedMatcher, "
    ↳ E2");
110

```

BVSS:

A0:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:F/S:C (0.0)

Recommendation:

It is not recommended to use this kind of mechanism to prevent of front-run attacks since, as it was explained, secret values can be retrieved from `mempools`.

Remediation Plan:

ACKNOWLEDGED: The `Orion team` acknowledged the risk of this issue.



## 4.10 (HAL-10) LACK OF A DOUBLE-STEP TRANSFER OWNERSHIP PATTERN – INFORMATIONAL (0.0)

### Description:

The current ownership transfer process for all the contracts inheriting from `Ownable` or `OwnableUpgradeable` involves the current owner calling the `transferOwnership()` function:

#### Listing 17: Ownable.sol

```

97 function transferOwnership(address newOwner) public virtual
   ↳ onlyOwner {
98     require(newOwner != address(0), "Ownable: new owner is the
   ↳ zero address");
99     _setOwner(newOwner);
100 }

```

If the nominated EOA account is not a valid account, it is entirely possible that the owner may accidentally transfer ownership to an uncontrolled account, losing the access to all functions with the `onlyOwner` modifier.

### Code Location:

Contracts using the `Ownable/OwnableUpgradeable` library

- `PriceOracle`

### BVSS:

A0:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:F/S:C (0.0)

**Recommendation:**

It is recommended to implement a two-step process where the owner nominates an account and the nominated account needs to call an `acceptOwnership()` function for the transfer of the ownership to fully succeed. This ensures the nominated EOA account is a valid and active account. This can be easily achieved by using OpenZeppelin's `Ownable2Step` contract instead of `Ownable`:

**Listing 18: Ownable2Step.sol (Lines 52-56)**

```

1 // SPDX-License-Identifier: MIT
2 // OpenZeppelin Contracts (last updated v4.8.0) (access/
↳ Ownable2Step.sol)
3
4 pragma solidity ^0.8.0;
5
6 import "../Ownable.sol";
7
8 /**
9  * @dev Contract module which provides access control mechanism,
↳ where
10 * there is an account (an owner) that can be granted exclusive
↳ access to
11 * specific functions.
12 *
13 * By default, the owner account will be the one that deploys the
↳ contract. This
14 * can later be changed with {transferOwnership} and {
↳ acceptOwnership}.
15 *
16 * This module is used through inheritance. It will make available
↳ all functions
17 * from parent (Ownable).
18 */
19 abstract contract Ownable2Step is Ownable {
20     address private _pendingOwner;
21
22     event OwnershipTransferStarted(address indexed previousOwner,
↳ address indexed newOwner);
23
24     /**
25      * @dev Returns the address of the pending owner.
26      */

```

```

27     function pendingOwner() public view virtual returns (address)
    ↳ {
28         return _pendingOwner;
29     }
30
31     /**
32      * @dev Starts the ownership transfer of the contract to a new
    ↳ account. Replaces the pending transfer if there is one.
33      * Can only be called by the current owner.
34      */
35     function transferOwnership(address newOwner) public virtual
    ↳ override onlyOwner {
36         _pendingOwner = newOwner;
37         emit OwnershipTransferStarted(owner(), newOwner);
38     }
39
40     /**
41      * @dev Transfers ownership of the contract to a new account
    ↳ (`newOwner`) and deletes any pending owner.
42      * Internal function without access restriction.
43      */
44     function _transferOwnership(address newOwner) internal virtual
    ↳ override {
45         delete _pendingOwner;
46         super._transferOwnership(newOwner);
47     }
48
49     /**
50      * @dev The new owner accepts the ownership transfer.
51      */
52     function acceptOwnership() external {
53         address sender = _msgSender();
54         require(pendingOwner() == sender, "Ownable2Step: caller is
    ↳ not the new owner");
55         _transferOwnership(sender);
56     }
57 }

```

#### Remediation Plan:

**ACKNOWLEDGED:** The Orion team acknowledged the risk of this issue.

## 4.11 (HAL-11) USE CUSTOM ERRORS INSTEAD OF REVERT STRINGS TO SAVE GAS - INFORMATIONAL (0.0)

### Description:

Failed operations in several contracts are reverted with an accompanying message selected from a set of hard-coded strings.

In the [EVM](#), emitting a hard-coded string in an error message costs ~50 more gas than emitting a custom error. Additionally, hard-coded strings increase the gas required to deploy the contract.

### BVSS:

A0:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

### Recommendation:

Custom errors are available from Solidity version [0.8.4](#) up. Consider replacing all revert strings with custom errors. Usage of custom errors should look like this:

#### Listing 19

```
1 error CustomError();
2
3 // ...
4
5 if (condition)
6     revert CustomError();
```

### Remediation Plan:

ACKNOWLEDGED: The [Orion team](#) acknowledged the risk of this issue.



# AUTOMATED TESTING



## 5.1 STATIC ANALYSIS REPORT

### Description:

**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.

## Results:

```

LibExchange.tryDeposit(address,uint256,address) (contracts/libs/LibExchange.sol#90-103) uses arbitrary from in transferFrom: SafeERC20.safeTransferFrom(IERC20(asset),user,address(this),amountInBase) (contracts/libs/LibExchange.sol#90)
SafeTransferHelper.safeTransferFrom(address,address,address,uint256) (contracts/Libs/SafeTransferHelper.sol#78-90) uses arbitrary from in transferFrom: SafeERC20.safeTransferFrom(IERC20(token),from,to,value) (contracts/Libs/SafeTransferHelper.sol#87)
Reference: https://github.com/crytic/sliether/wiki/Detector-Documentation#arbitrary-from-in-transferFrom

UniswapV3Pool is re-used:
- UniswapV3Pool (contracts/extensions/UniswapV3Callbacks.sol#10-14)
- UniswapV3Pool (contracts/interfaces/UniswapV3Pool.sol#45-55)
Errors is re-used:
- Errors (contracts/helpers/Errors.sol#6-10)
- Errors (contracts/utills/Errors.sol#6-10)
ReentrancyGuard is re-used:
- ReentrancyGuard (contracts/utills/ReentrancyGuard.sol#4-33)
- ReentrancyGuard (node_modules/@openzeppelin/contracts/security/ReentrancyGuard.sol#22-77)
Address is re-used:
- Address (contracts/tokens/OWN.sol#161-171)
- Address (node_modules/@openzeppelin/contracts/utills/Address.sol#9-244)
SafeERC20 is re-used:
- SafeERC20 (contracts/tokens/OWN.sol#173-205)
- SafeERC20 (node_modules/@openzeppelin/contracts/token/ERC20/utills/SafeERC20.sol#19-143)
IERC20 is re-used:
- IERC20 (contracts/tokens/OWN.sol#8-77)
- IERC20 (node_modules/@openzeppelin/contracts/token/ERC20/IERC20.sol#9-78)
Reference: https://github.com/crytic/sliether/wiki/Detector-Documentation#same-reused

CurveExtension.curveSwapStableAmountIn(address,IERC20,int128,int128,address,uint256) (contracts/extensions/CurveExtension.sol#17-34) ignores return value by assetOut.transfer(to,exchangedAmount) (contracts/extensions/CurveExtension.sol#32)
CurveExtension.curveSwapStableAmountOut(address,address,IERC20,IERC20,int128,int128,uint256,uint256,address) (contracts/extensions/CurveExtension.sol#36-73) ignores return value by assetOut.transfer(to,amountOut) (contracts/extensions/CurveExtension.sol#64)
CurveExtension.curveSwapStableAmountIn(address,IERC20,int128,int128,uint256,uint256,address) (contracts/extensions/CurveExtension.sol#75-88) ignores return value by assetOut.transfer(to,exchangedAmount) (contracts/extensions/CurveExtension.sol#87)
Reference: https://github.com/crytic/sliether/wiki/Detector-Documentation#unchecked-transfer

CurveExtension.get_D(uint256[],uint256) (contracts/extensions/CurveExtension.sol#90-114) performs a multiplication on the result of a division:
- D = (A * P + B) / (x * D) (contracts/extensions/CurveExtension.sol#102)
- D = (A * P + B) / (x * D) (contracts/extensions/CurveExtension.sol#105)
CurveExtension.get_y(int128,int128,uint256,uint256) (contracts/extensions/CurveExtension.sol#116-154) performs a multiplication on the result of a division:
- c = (c * D) / (x * M * COIN) (contracts/extensions/CurveExtension.sol#137)
- c = (c * D) / (x * M * COIN) (contracts/extensions/CurveExtension.sol#139)
MarginalFunctionality.uintPercent(int192,uint8) (contracts/libs/MarginalFunctionality.sol#48-53) performs a multiplication on the result of a division:
- c = int92(a / d) * B (contracts/libs/MarginalFunctionality.sol#53)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utills/math/Math.sol#45-134) performs a multiplication on the result of a division:
- denominator = denominator / twos (node_modules/@openzeppelin/contracts/utills/math/Math.sol#101)
- inverse = (3 - denominator) * 2 (node_modules/@openzeppelin/contracts/utills/math/Math.sol#110)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utills/math/Math.sol#45-134) performs a multiplication on the result of a division:
- denominator = denominator / twos (node_modules/@openzeppelin/contracts/utills/math/Math.sol#101)
- inverse = 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utills/math/Math.sol#120)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utills/math/Math.sol#45-134) performs a multiplication on the result of a division:
- denominator = denominator / twos (node_modules/@openzeppelin/contracts/utills/math/Math.sol#101)
- inverse = 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utills/math/Math.sol#121)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utills/math/Math.sol#45-134) performs a multiplication on the result of a division:
- denominator = denominator / twos (node_modules/@openzeppelin/contracts/utills/math/Math.sol#101)
- inverse = 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utills/math/Math.sol#122)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utills/math/Math.sol#45-134) performs a multiplication on the result of a division:
- denominator = denominator / twos (node_modules/@openzeppelin/contracts/utills/math/Math.sol#101)
- inverse = 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utills/math/Math.sol#123)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utills/math/Math.sol#45-134) performs a multiplication on the result of a division:
- denominator = denominator / twos (node_modules/@openzeppelin/contracts/utills/math/Math.sol#101)
- inverse = 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utills/math/Math.sol#124)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utills/math/Math.sol#45-134) performs a multiplication on the result of a division:
- denominator = denominator / twos (node_modules/@openzeppelin/contracts/utills/math/Math.sol#101)
- inverse = 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utills/math/Math.sol#125)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utills/math/Math.sol#45-134) performs a multiplication on the result of a division:
- prod0 = prod0 / twos (node_modules/@openzeppelin/contracts/utills/math/Math.sol#104)
- result = prod0 * inverse (node_modules/@openzeppelin/contracts/utills/math/Math.sol#131)
Reference: https://github.com/crytic/sliether/wiki/Detector-Documentation#divide-before-multiply

Contract locking ether found:
- Contract EthReceiver (contracts/utills/EthReceiver.sol#46-10) has payable functions:
- EthReceiver.receive() (contracts/utills/EthReceiver.sol#9-11)
But does not have a function to withdraw the ether
Reference: https://github.com/crytic/sliether/wiki/Detector-Documentation#contracts-that-lock-ether

CurveExtension.get_y(int128,int128,uint256,uint256) (contracts/extensions/CurveExtension.sol#143) is a local variable never initialized
Exchange.updateSettledSkits(address[],uint8[]) i (contracts/Exchange.sol#117) is a local variable never initialized
SafeTransferHelper.safePermit(IERC20,bytes).success (contracts/Libs/SafeTransferHelper.sol#118) is a local variable never initialized
CurveExtension.get_D(uint256[],uint256) j (contracts/extensions/CurveExtension.sol#101) is a local variable never initialized
CurveExtension.get_D(uint256[],uint256) i (contracts/extensions/CurveExtension.sol#93) is a local variable never initialized
Exchange.getBalances(address[],address) i (contracts/Exchange.sol#203) is a local variable never initialized
LibExchange.updateOrderBalance(bLibValidator,Order,uint12,int12,uint8,Mapping(address => Mapping(address => int192)),Mapping(address => MarginalFunctionality.Liability[])) swap (contracts/Libs/LibExchange.sol#172) is a local variable never initialized
CurveExtension.get_y(int128,int128,uint256,uint256) j (contracts/extensions/CurveExtension.sol#132) is a local variable never initialized
MarginalFunctionality.updateLiability(address,address,Mapping(address => MarginalFunctionality.Liability[])) uint12,int12,i (contracts/Libs/MarginalFunctionality.sol#229) is a local variable never initialized
Exchange.fillOrders(LibValidator,Order,LibValidator,Order,uint4,uint12) data (contracts/Exchange.sol#248) is a local variable never initialized
CurveExtension.get_D(uint256[],uint256) j (contracts/extensions/CurveExtension.sol#99) is a local variable never initialized
Reference: https://github.com/crytic/sliether/wiki/Detector-Documentation#uninitialized-local-variables

ExchangeWithGenericSwap.fillThroughPools(uint12,LibValidator,Order,IExecutionExecutor,LibValidator.SwapDescription,bytes,bytes) (contracts/ExchangeWithGenericSwap.sol#41-58) ignores return value by LibExchange.updateBalance(order,matcherAddress,order.matcherFeeAsset,int256(uint256(order.matcherFee)),assetBalances,liabilities) (contracts/ExchangeWithGenericSwap.sol#57)
LibAtomic.doloAtomic(LibAtomic.LockOrder,Mapping(bytes32 => LibAtomic.LockInfo),Mapping(address => Mapping(address => int192)),Mapping(address => MarginalFunctionality.Liability[])) (contracts/Libs/LibAtomic.sol#44-78) ignores return value by LibExchange.updateBalance(msg.sender,swap.asset,-1 * remaining,assetBalances,liabilities) (contracts/Libs/LibAtomic.sol#60)
LibAtomic.dofundAtomic(LibAtomic.RedemOrder,bytes,Mapping(bytes32 => bool),Mapping(address => Mapping(address => int192)),Mapping(address => MarginalFunctionality.Liability[])) (contracts/Libs/LibAtomic.sol#80-96) ignores return value by LibExchange.updateBalance(order.receiver,order.asset,int256(uint256(order.amount)),assetBalances,liabilities) (contracts/Libs/LibAtomic.sol#95)
LibAtomic.doloAtomic(address,bytes,bytes,address,Mapping(bytes32 => LibAtomic.LockInfo),Mapping(address => int192)),Mapping(address => MarginalFunctionality.Liability[])) (contracts/Libs/LibAtomic.sol#98-118) ignores return value by LibExchange.updateBalance(receiver,swap.asset,int256(uint256(swap.amount)),assetBalances,liabilities) (contracts/Libs/LibAtomic.sol#117)
LibAtomic.dofundAtomic(bytes32,Mapping(bytes32 => LibAtomic.LockInfo),Mapping(address => Mapping(address => int192)),Mapping(address => MarginalFunctionality.Liability[])) (contracts/Libs/LibAtomic.sol#120-133) ignores return value by LibExchange.updateBalance(swap.sender,swap.asset,int256(uint256(swap.amount)),assetBalances,liabilities) (contracts/Libs/LibAtomic.sol#132)
Reference: https://github.com/crytic/sliether/wiki/Detector-Documentation#unused-return

```

- Arbitrary from in transferFrom issue does not pose any risk since the contract controls this value.
- Flagged ignored/unused return value does not pose any risk.
- No major issues found by Slither.



THANK YOU FOR CHOOSING

// HALBORN

