Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Aevo Exchange

Veridise Inc.

October 18, 2023

► **Prepared For:**

Ribbon finance
https://www.ribbon.finance

► **Prepared By:**

Alberto Gonzalez
Ajinkya Rajput

► **Contact Us:** contact@veridise.com

► **Version History:**

Oct. 3, 2023 Initial Draft

Contents

1 Executive Summary

From Sept. 18, 2023 to Sept. 28, 2023, Ribbon finance engaged Veridise to review the security of their Aevo Exchange. Aevo is a high-performance, order-book based decentralized exchange that allows trading in perpetuals, options and spot trades. Veridise conducted the assessment over 16 person-days, with 2 engineers reviewing code over 8 days for commit `58bcd6c`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Code assessment. The Aevo Exchange developers provided the source code of the Aevo Exchange contracts for review. To facilitate the Veridise auditors' understanding of the code, the Aevo Exchange developers also provided additional documentation summarizing the high-level function of each contract. Since the protocol design has both on-chain and off-chain components, the developers also provided documentation for the interaction between the off-chain and the contracts. The source code also contained some documentation in the form of READMEs and documentation comments on functions and storage variables.

The source code contained a test suite, which the Veridise auditors noted, that each testcase tests single functionality

Summary of issues detected. The audit uncovered 15 issues, 3 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, V-RIB-VUL-001erroneously reports a successful spot trade to the off-chain component which might have failed on-chain. V-RIB-VUL-002finds that the settlement fee is charged twice to the user in block trades. V-RIB-VUL-003finds that options can be traded after they have expired and their expiry price has been set allowing an attacker to make profitable trades. The Veridise auditors also identified several medium-severity issues, including V-RIB-VUL-005in which an attacker may self-destruct the contract if deployment missed calling `initialize()` on implementation contract. V-RIB-VUL-007notes that the block trade method does not verify the signature of the legs provided in the block. as well as a number of minor issues. The Aevo Exchange developers fixed 0 issues as of now.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the Aevo Exchange. The test suite in the current version tests all functionalities in isolation. We strongly recommend testing longer sequences of user interactions. We also emphasize on the corner cases. A detailed design document of the promise between the on-chain and off-chain components is strongly recommended.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise,

arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Aevo Exchange	`58bcd6c` - `58bcd6c`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Sept. 18 - Sept. 28, 2023	Manual & Tools	2	16 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	3	3
Medium-Severity Issues	4	4
Low-Severity Issues	2	2
Warning-Severity Issues	5	5
Informational-Severity Issues	1	1
TOTAL	15	15

Table 2.4: Category Breakdown.

Name	Number
Logic Error	6
Data Validation	4
Maintainability	3
Usability Issue	2

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Aevo Exchange's smart contracts. In our audit, we sought to answer the following questions:

- Are the EIP712 signatures implemented correctly?
- Is the settlement logic implemented correctly?
- Does an attacker have any extra information that allows him to always make a profitable trade?
- Is the interaction between on-chain and off-chain modules correct?
- Can an attacker steal funds?
- Is the access control model implemented correctly?
- Are the block trades implemented correctly?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard, as well as the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.
- *Fuzzing/Property-based Testing.* We also leverage fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, we formalize the desired behavior of the protocol as [V] specifications and then use our fuzzing framework OrCa to determine if a violation of the specification can be found.

Scope. The scope of this audit is limited to the `src/` folder of the source code provided by the Aevo Exchange developers, which contains the smart contract implementation of the Aevo Exchange. The following folders were excluded from the scope of this audit.

- src/migrations/*
- script/*
- mocks/*
- swap/*

During the audit, the Veridise auditors referred to the excluded files but assumed that they had been implemented correctly.

Methodology. Veridise auditors reviewed the reports of previous audits for Aevo Exchange (Sigma Five), inspected the provided tests, and read the Aevo Exchange documentation. They

then began a manual audit of the code assisted by both static analyzers and automated testing. During the audit, the Veridise auditors regularly met with the Aevo Exchange developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-RIB-VUL-001	Return value from _settleSpot() is unchecked	High	Fixed
V-RIB-VUL-002	Settlement fee charged twice in blockTrades	High	Intended Behavior
V-RIB-VUL-003	Options can be traded after setting expiry price	High	Fixed
V-RIB-VUL-004	Fee charged multiple times for unfulfilled orders	Medium	Intended Behavior
V-RIB-VUL-005	Implementation contracts need to be explicitly . . .	Medium	Fixed
V-RIB-VUL-006	Inconsistency in handling of deposit and withdraw	Medium	Intended Behavior
V-RIB-VUL-007	Blocktrade signature does include the legs array.	Medium	Intended Behavior
V-RIB-VUL-008	Signing keys expiration unchecked	Low	Intended Behavior
V-RIB-VUL-009	Blocktrade cannot request spot trades	Low	Fixed
V-RIB-VUL-010	Possible debit from a non existent account	Warning	Fixed
V-RIB-VUL-011	Use of two step ownership transfer	Warning	Intended Behavior
V-RIB-VUL-012	Domain typehash does not include contract addres	Warning	Intended Behavior
V-RIB-VUL-013	Wrong decimals	Warning	Fixed
V-RIB-VUL-014	Base.sol does not have storage gaps	Warning	Won't Fix
V-RIB-VUL-015	Inconsistent comment	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-RIB-VUL-001: Return value from _settleSpot() is unchecked

Severity	High	**Commit**	58bcd6c
Type	Logic Error	**Status**	Fixed
File(s)	`Exchange.sol`		
Location(s)	_transfer()		
Confirmed Fix At	a3f7363		

The on-chain off-chain design requires the on chain contracts follow a pattern where a external call returns a `RevertReason` enum in place where the contract is supposed to revert. The value `RevertReason.None` is returned when the external call runs successfully. An example of this pattern.

```
if (vars.takerFill == 0) {
    return (vars.takerHash, RevertReason.ALREADY_FILLED);
}
```

Snippet 4.1: `_trade()` in Exchange.sol

The function `_settleSpot()` which executes the spot trades implements this pattern. This function

1. Debits `quote` from `from` and credits it to `to`
2. Debits `base` from `to` and credits it to `from`

If any of these `credit()` and `debit()` calls fail, it returns `Accounts.RevertReason`. This revert reason is propagated back to it caller, i.e. `_transfer()`.

The returned revert reason from `_settleSpot()` is not checked in `_transfer()` and control flow reaches where `RevertReason.NONE` is returned to its caller `_trade()` representing the trade as successfully executed while it errored in `_settleSpot()`

Impact This implementation of `_transfer()` will communicate to the off-chain part of the protocol that the Spot trade executed successfully while the trade may have failed in `_settleFunction()`

Recommendation Check the `RevertReason` returned by `_settleSpot()` and return the revert reason if it is not `RevertReason.None`

Developer Response Developers confirmed this bug.

```
function _settleSpot(
        AccountsInterface _accounts,
        InstrumentsInterface _instruments,
        uint256 instrument,
        address from,
        address to,
        uint256 amount,
        uint256 collateralAmount
    ) internal returns (RevertReason) {
        (address quote, address base) = _instruments.spots(instrument);

        // Debit quote collateral from the taker
        (, AccountsInterface.RevertReason accountsReason) = _accounts.debit(from,
    quote, collateralAmount);
        if (accountsReason != AccountsInterface.RevertReason.NONE) {
            return _accountsRevertReason(accountsReason);
        }
        // Credit quote collateral to the maker
        (, accountsReason) = _accounts.credit(to, quote, collateralAmount);
        if (accountsReason != AccountsInterface.RevertReason.NONE) {
            return _accountsRevertReason(accountsReason);
        }

        // Credit collateral to the taker
        (, accountsReason) = _accounts.credit(from, base, amount);
        if (accountsReason != AccountsInterface.RevertReason.NONE) {
            return _accountsRevertReason(accountsReason);
        }
        // Debit collateral from the maker
        (, accountsReason) = _accounts.debit(to, base, amount);
        if (accountsReason != AccountsInterface.RevertReason.NONE) {
            return _accountsRevertReason(accountsReason);
        }

        // The settlement succeeded
        return RevertReason.NONE;
    }
```

Snippet 4.2: `_settleSpot()` in Exchange.sol

```
function _transfer(TradeVars memory vars, Orders.Order memory taker, Orders.Order
    memory maker)
    internal
    returns (RevertReason reason, int256 takerPnL, int256 makerPnL)
{
   ...
        if (vars.instrumentType == InstrumentsInterface.InstrumentType.Spot) {
            _settleSpot(
                vars.accounts,
                vars.instruments,
                taker.instrument,
                maker.maker,
                taker.maker,
                vars.amount,
                vars.collateralAmount
            );
        }
        ....
    // The transfer succeeded
    return (RevertReason.NONE, takerPnL, makerPnL);
}
```

Snippet 4.3: `_transfer()` in Exchange.sol

4.1.2 V-RIB-VUL-002: Settlement fee charged twice in blockTrades

Severity	High	**Commit**	58bcd6c
Type	Logic Error	**Status**	Intended Behavior
File(s)	`Exchange.sol`		
Location(s)	blockTrade()		
Confirmed Fix At			

`blockTrade()` function executes multiple taker orders in one external call. The taker orders are encapsulated in the `Quotes.Quote`.

```
    // Settle the taker quote fee
    RevertReason reason = _settleFee(vars, vars.takerHash, taker.account, taker.fee);
    if (reason != RevertReason.NONE) {
        revert InvalidTakerQuote(vars.takerHash, reason);
    }
```

Snippet 4.4: Snippets from `blockTrade()` in Exchange.sol

The block trade function first performs sanity checks and the calls the `settleFee()` for taker.

```
  for (uint256 i; i < makers.length; ++i) {
    // Verify the maker quote signature
    vars.makerHash = makers[i].hash(domainSeparator);
    if (!_verifyQuote(vars.accounts, makers[i], vars.makerHash)) {
        revert InvalidMakerQuote(vars.takerHash, vars.makerHash, RevertReason.
    INVALID_SIGNATURE);
    }

    // Settle the maker quote fee
    reason = _settleFee(vars, vars.makerHash, makers[i].account, makers[i].fee);
    if (reason != RevertReason.NONE) {
        revert InvalidMakerQuote(vars.takerHash, vars.makerHash, reason);
    }
```

Snippet 4.5: Snippets from `blockTrade()` in Exchange.sol

Then, for each `maker` Quote, sanity checks are performed and `_settleFee()` function is called for maker quote.

```
// Execute the trade for the leg
    (bytes32 order, RevertReason reason) = _trade(vars, takerOrder, makerOrders[i],
    makerFills[i], makerPnLs[i]);
    if (reason != RevertReason.NONE) {
        if (order == vars.takerHash) {
            revert InvalidTakerOrder(vars.takerHash, reason);
        } else {
            revert InvalidMakerOrder(vars.takerHash, order, reason);
        }
    }
```

Snippet 4.6: Snippets from `blockTrade()` in Exchange.sol

The quotes are then unpacked into taker and maker order arrays. And these orders are passed to `_trade()` function. As described in V-RIB-VUL-004 the `_trade()` function charges maker and taker fees again.

Impact For each order the settlement fee will be charged twice for each participant

Recommendation Maintain the amount of fees charged for order in struct `Order`. Check if the fee is already charged already in `_settleFee()`.

Developer Response The developers responded "We will leave this closed as Intended Behavior for now. There is difficulty here because the user's fee tier could be different for maker and taker, and each user may have a different fee tier."

4.1.3 V-RIB-VUL-003: Options can be traded after setting expiry price

Severity	High	**Commit**	58bcd6c
Type	Data Validation	**Status**	Fixed
File(s)	`Exchange.sol`		
Location(s)	trade()		
Confirmed Fix At	`https://github.com/ribbon-finance/exchange-contracts/pull/86/`		

Options contracts are settled after setting the expiry price of the asset in the option by calling `setExpiryPrice()`. This function performs sanity checks and sets the value of mapping `expiryPrice` (shown below) for given asset

```
/// @notice The expiry price for an expiring asset
mapping(address => mapping(uint256 => Price)) public expiryPrice;
```

Snippet 4.7: Snippet from InstrumentsStorage.sol

Options settlement then happens by calling `settleOption()` in Instrument.sol. Once the settlement of options contract begins, `settling[instrumentId]` is set to true.

```
if (!settling[vars.instrument]) {
    settling[vars.instrument] = true;
}
```

Snippet 4.8: Snippet from `settleOption()` in Instruments.sol

Trading of options in Exchange.sol is stopped once the option starts settling. This is implemented as a sanity check in `trade()`.

```
// Revert if the instrument has begun settling
if (vars.instruments.settling(taker.instrument)) {
    revert InvalidTakerOrder(vars.takerHash, RevertReason.NOT_TRADABLE);
}
```

Snippet 4.9: Snippet from `trade()` in Exchange.sol

The option settling is not guaranteed to happen immediately after the expiration and this function has to be called by `keeper` which can be any time after post expiration and any time after the expiry price is set by calling `setExpiryPrice()`

Impact Since,

- Mapping that stores the options, Instruments. `options`
- Mapping `settling`
- Mapping `expirtyPrice`

Are all public, an attacker can always trade on the winning side of option

1. That has expired
2. That has expiry price set up for its asset
3. That has not yet started settling

Recommendation Check that the option is not expired before executing the options trade.

Developer Response Awaiting developers response

4.1.4 V-RIB-VUL-004: Fee charged multiple times for unfulfilled orders

Severity	Medium	**Commit**	58bcd6c
Type	Logic Error	**Status**	Intended Behavior
File(s)	`Exchange.sol`		
Location(s)	_trade()		
Confirmed Fix At			

The protocol executes the orders by settling a taker order against list of maker orders on the same instrument that satisfy the price constraints on both taker and maker orders. The protocol loops over the available maker orders till the requested number of positions in taker order is not fulfilled by maker orders. The unfulfilled number of positions in taker order is maintained in variable `takerFill` and the fulfilled number of positions in maker orders is tracked in `makerFills[]`. The following snippet shows the relevant code from `_trade()` function.

Relevant snippets from `_trade()` in Exchange.sol

```
function _trade(
        TradeVars memory vars,
        Orders.Order memory taker,
        Orders.Order[] memory makers,
        uint256[] memory makerFills,
        int256[] memory makerPnLs
    ) internal returns (bytes32, RevertReason) {
            RevertReason reason = _settleFee(vars, vars.takerHash, taker.
    maker, taker.fee);
            if (reason != RevertReason.NONE) {
                return (vars.takerHash, reason);
            }
```

```
  // Transfer the maker fee
  reason = _settleFee(vars, vars.makerHash, makers[i].maker, makers[i].fee)
    ;
  if (reason != RevertReason.NONE) {
      return (vars.makerHash, reason);
  }
```

The `_trade()` function first calls `settleFee()` to charge the fee from taker. It then loops over available makers. For each maker the protocol performs a few sanity checks and then checks if the maker order can settle the taker order by calling `_matchOrders()`. If the order matches, the trade is executed in `_transfer()` function and then fee is charged from maker by calling `_settleFee()` for maker.

The protocol charges a fee per order from the taker and maker. The fee settlement happens while the trade is being executed in the `_trade()` function.

Impact In case

- If the taker order is not completely filled or,

- If a maker order is settled against multiple takers

the taker order can will be settled by making multiple calls to `trade()` and the makers and takers will be charged fees multiple times for the same order.

Recommendation Maintain the amount of fees charged for order in struct `Order`. Check if the fee is already charged already in `_settleFee()`.

Developer Response Developers confirmed this issue

4.1.5 V-RIB-VUL-005: Implementation contracts need to be explicitly initializedx

Severity	Medium	**Commit**	58bcd6c
Type	Maintainability	**Status**	Fixed
File(s)	`Exchange.sol`, `Accounts.sol`, `Executor.sol`, `Instruments.sol`		
Location(s)	initialize()		
Confirmed Fix At			

The contracts use the `UUPS` upgradeable pattern, which is riskier than the `Transparent` pattern because the implementations carry the logic for the upgrade and not the proxies. Hence, it is important to initialize the implementations as well.

An attacker can call `initialize()` on the implementation address with the owner account set to attackers address. This will escalate the attackers privilege to owner.

Impact Taking control of the implementation can have different consequences depending on the logic it has. Some of them may not be critical, but others may contain `delegatecalls` to arbitrary contracts which will allow to self-destruct the implementation. For example take a look at the following snippet from `Accounts.sol`:

```
function _fallback() internal {
    address implementation = logic;

    assembly {
        // Copy msg.data. We take full control of memory in this inline assembly
        // block because it will not return to Solidity code. We overwrite the
        // Solidity scratch pad at memory position 0.
        calldatacopy(0, 0, calldatasize())

        // Call the implementation.
        // out and outsize are 0 because we don't know the size yet.
        let result := delegatecall(gas(), implementation, 0, calldatasize(), 0, 0)

        // Copy the returned data.
        returndatacopy(0, 0, returndatasize())

        switch result
        // delegatecall returns 0 on error.
        case 0 { revert(0, returndatasize()) }
        default { return(0, returndatasize()) }
    }
}
```

Snippet 4.10: Function `_fallback()` from the `Accounts.sol` contract. The logic makes a `delegatecall` to the `logic` contract.

Currently, because the `logic` variable is declared as immutable, the attacker cannot change its value even if it has the ownership of the implementation.

The biggest bounty pay out was for an uninitialized implementation.

Reference

Recommendation Make sure to initialize the implementation contracts at deployment and in every upgrade of the contracts.

Developer Response Developers responded that this is handled in deploy scripts.

4.1.6 V-RIB-VUL-006: Inconsistency in handling of deposit and withdraw

Severity	Medium	**Commit**	58bcd6c
Type	Logic Error	**Status**	Intended Behavior
File(s)	`Accounts.sol`		
Location(s)	withdraw() withdrawInsuranceFund()		
Confirmed Fix At			

The token transfer and token balance is handled differently between the `deposit` and `withdraw` functions. Looking at the `deposit` function we can observe the following:

```
function deposit(address account, address collateral, uint256 amount) external
    nonReentrant returns (int256) {
    _addAccount(account);

    int256 balance = _deposit(account, collateral, amount);

    ERC20(collateral).safeTransferFrom(msg.sender, address(this), amount);

    return balance;
}
```

Snippet 4.11: Function `deposit` in the `Accounts.sol` contract.

The above logic credits the user via `_deposit` and then transfer the `collateral` tokens from the user to the contract. The `_deposit` logic is the following:

The above logic shows that the internal balances are tracked on what `getLegacyToken(collateral)` returns. So far, we know that:

- The tokens transferred to the contract are `collateral` tokens
- The internal balances are tracked as `getLegacyToken(collateral)`.

So, we should expect the same for withdrawals. The tokens transferred to the user should be `collateral` tokens but the internal balances should be updated using `getLegacyToken(collateral)`.

Looking at `withdraw` we can see that the parameter `collateral` is used unchanged in the token transfer. Hence, we can expect for this parameter to not be the legacy token and that the internal balances will be updated using `getLegacyToken(collateral)`. But if we follow the execution up to `_completeWithdraw` we will see that the same `collateral` address is used to update the internal balances.

Same inconsistency is found in the `withdrawInsuranceFund` function.

Impact Withdrawals may get DoSed naturally as users deposit migrated tokens and other users withdraw legacy tokens.

Recommendation Make withdrawals consistent with deposits.


```
function _deposit(address account, address collateral, uint256 amount) internal
    returns (int256) {
        if (account == address(0)) {
            revert ZeroAddress();
        }

        if (amount == 0) {
            revert ZeroAmount();
        }

        collateral = getLegacyToken(collateral);

        if (!collaterals[collateral]) {
            revert InvalidCollateral();
        }

        int256 balance = accounts[account].balance[collateral] + int256(amount);
        accounts[account].balance[collateral] = balance;

        totalBalance[collateral] += int256(amount);

        emit Deposit(account, collateral, amount, balance);

        return balance;
}
```

Snippet 4.12: Function `_deposit` in the `Accounts.sol` contract.

Developer Response Developers informed us that keepers pass only legacy tokens for withdraw and withdraw request for migrated tokens will always revert

4.1.7 V-RIB-VUL-007: Blocktrade signature does include the legs array.

Severity	Medium	**Commit**	58bcd6c
Type	Data Validation	**Status**	Intended Behavior
File(s)	`Quotes.sol`		
Location(s)	hash()		
Confirmed Fix At			

Keepers submit block trades on behalf of takers and makers. Block trades are represented in a structure called `Quote`:

```
struct Quote {
  bytes32 blockId;
  address account;
  bool isBuy;
  uint256 amount;
  uint256 salt;
  uint256 timestamp;
  bytes signature;
  Leg[] legs;
  int256 fee;
}
```

Snippet 4.13: Struct `Quote` from the `Quotes.sol` contract.

During the `blockTrade` function, the logic validates that the given `Quotes` were signed by the taker or the maker. The message signed is the hash of `domainSeparator` and the `Quote`:

```
function hash(Quote memory quote, bytes32 domainSeparator) internal pure returns (
    bytes32) {
   return Signing.deriveDigest(
          domainSeparator,
          keccak256(
             abi.encode(
               QUOTE_TYPEHASH,
               quote.blockId,
               quote.account,
               quote.isBuy,
               quote.amount,
               quote.salt,
               quote.timestamp
             )
          )
    );
}
```

Snippet 4.14: Function `hash` from the `Quotes.sol` contract. This function is used in the `blockTrade()` function from the `Exchange` contract.

The `legs` array in the `Quote` structure is not part of the signed message.

Impact Keepers can execute any legs on behalf of takers and makers without authorization.

Recommendation Include the `legs` array into the message to be hashed and signed.

Developer Response Developers informed us that they handle the validation of this part in their off-chain system

4.1.8 V-RIB-VUL-008: Signing keys expiration unchecked

Severity	Low	**Commit**	58bcd6c
Type	Data Validation	**Status**	Intended Behavior
File(s)	`Accounts.sol`		
Location(s)	hasSigningKey()		
Confirmed Fix At			

The protocol allows a user to store a set of signing keys and specify when those keys should expire as a timestamp. However, when a signing key is used, it is never checked to determine if it has expired.

```
1 function hasSigningKey(address account, address signingKey) external view returns (
      bool) {
2     return accounts[account].hasSigningKey[signingKey];
3 }
```

Snippet 4.15: Function `hasSigningKey` from the `Accounts.sol` contract.

The logic does not verify that the `signingKey` has not expired.

Impact An attacker can attempt to compromise any key that the user has used (including those that have expired) to sign trades.

Recommendation Check if the key has expired before using it

Developer Response Developers informed us that they check the signing key expiry off-chain before creating the request. They implemented it this way to handle a corner case when the match happens on the backend and expires on-chain.

4.1.9 V-RIB-VUL-009: Blocktrade cannot request spot trades

Severity	Low	**Commit**	58bcd6c
Type	Usability Issue	**Status**	Fixed
File(s)	`Exchange.sol`		
Location(s)	blockTrade()		
Confirmed Fix At			

The `Quote` orders include an array of `legs` which packs all the individual trades. To get all the trade orders the code use the `unpack` function:

```
(Orders.Order[] memory takerOrders, Orders.Order[][] memory makerOrders) = Quotes.
    unpack(taker, makers);
```

Snippet 4.16: Taker and makers quote orders are unpack into trade orders. Snippet from the `blockTrade` function of the `Exchange` contract.

The `quote` orders include a parameter `amount`. Later, the `unpack` function uses this value to compute the amounts for each leg trade:

```
takerOrder.amount = (leg.ratio * taker.amount) / Orders.PRICE_UNITS;

...

makerOrder.amount = (leg.ratio * quote.amount) / Orders.PRICE_UNITS;
```

Snippet 4.17: Trade orders amount are computed using the `Quote.amount` parameter and the `leg.ratio`. Code snippet from the `unpack` function of the `Quotes.sol` file.

The issue is that `Order.amount` is different for options/perpetuals and spot. However, they are computed using the same `Quote.amount`.

Impact If `Quote.amount` is passed with decimals of options or perpetuals, the leg proportion for spot trades will be computed using these decimals instead of the `base` decimals.

If `Quoute.amount` is passed with the decimals of spot's `base`, the leg proportion for options/perpetuals will be computed using this decimals instead of `POSITION_UNITS`.

Recommendation Validate on-chain that spot cannot be mixed with options or perpetuals in the same block.

Developer Response The offchain system will make sure only option and perpetual in same block

4.1.10 V-RIB-VUL-010: Possible debit from a non existent account

Severity	Warning	**Commit**	58bcd6c
Type	Logic Error	**Status**	Fixed
File(s)	`Accounts.sol`		
Location(s)	debit()		
Confirmed Fix At	a6b1c35		

The protocol defines a privileged operations viz. `Authority` operations that allow modifying the state of accounts. One such function is `debit()` which removes collateral from an account.

```
function debit(address account, address collateral, uint256 amount)
        external
        onlyAuthority
        nonReentrant
        returns (int256, RevertReason)
{
    if (!collaterals[collateral]) {
        return (0, RevertReason.INVALID_COLLATERAL);
    }

    _addAccount(account);

    int256 balance = accounts[account].balance[collateral] - int256(amount);
    accounts[account].balance[collateral] = balance;

    totalBalance[collateral] -= int256(amount);

    emit Debit(account, collateral, amount, balance);

    return (balance, RevertReason.NONE);
}
```

Snippet 4.18: `debit()` in Accounts.sol

This function first checks if the collateral is valid and the adds the account by calling `_addAccount()`. It then updates the balance by subtracting `amount` from `accounts[account].balance[collateral]`. The protocol allows the balances to be negative by design.

Impact It is possible for `debit()` to run successfully but the debited account may not exists.

Recommendation Revert or return with `RevertReason` if account does not exists.

Developer Response Developers said, "We can change this, we did this for safety"

4.1.11 V-RIB-VUL-011: Use of two step ownership transfer

Severity	Warning	**Commit**	58bcd6c
Type	Maintainability	**Status**	Intended Behavior
File(s)	`src/base/Base.sol`		
Location(s)	setOwner()		
Confirmed Fix At			

The current implementation of `_setOwner` is a 1-step ownership transfer:

```
function _setOwner(address newOwner) internal {
    if (newOwner == address(0)) {
        revert ZeroAddress();
    }

    owner = newOwner;

    emit OwnerUpdated(newOwner);
}
```

Snippet 4.19: Function `_setOwner` from the `Base.sol` contract.

Impact Any mistake made while transferring the ownership of the contract is not fixable.

Recommendation Use a two-step ownership transfer pattern.

Ref: `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable2Step.sol`

Developer Response Developers said, "We can do that"

4.1.12 V-RIB-VUL-012: Domain typehash does not include contract address

Severity	Warning	**Commit**	58bcd6c
Type	Data Validation	**Status**	Intended Behavior
File(s)	`Signing.sol`		
Location(s)	DOMAIN_TYPEHASH		
Confirmed Fix At			

Current implementation of the computation of the `DOMAIN_TYPEHASH` from the `Signing.sol` library does not include the address of the contract.

Impact Signatures can be used in other contracts with the same `name`, `version` and `[chaind.id](http://chaind.id)` even if they have a different address.

Recommendation Either include the contract address in the domain typehash or communicate clearly to the signers that their signatures for orders might be used in a new deployment.

Developer Response The addresses are omitted by design.

4.1.13 V-RIB-VUL-013: Wrong decimals

Severity	Warning	**Commit**	58bcd6c
Type	Logic Error	**Status**	Fixed
File(s)	`Orders.sol`		
Location(s)	getQuoteAmount()		
Confirmed Fix At	1282dd5		

Function `getQuoteAmount()` is used to get the amount of quote collateral for a given amount and given `limitPrice`.

```
1 function getQuoteAmount(uint256 amount, uint256 limitPrice) internal pure returns (
      uint256) {
2     return (amount * limitPrice) / PRICE_UNITS;
3 }
```

Snippet 4.20: `getQuoteAmount()` in Orders.sol

The `amount` has decimals of `POSITION_UNITS`, `limitPrice` is in decimals of `PRICE_UNITS`. The expected return value is decimals of `PRICE_UNITS`. Therefore denominator in the function above should be `POSITION_UNITS`.

Impact No impact in current version because the numerical value of `PRICE_UNITS` and `POSITION_UNITS` is same. Since the protocol uses upgradable contracts this might cause inconsistencies in future versions of protocol.

Recommendation Use correct constant symbols.

Developer Response Developers acknowledged this issue

4.1.14 V-RIB-VUL-014: Base.sol does not have storage gaps

Severity	Warning	**Commit**	58bcd6c
Type	Maintainability	**Status**	Won't Fix
File(s)	`Base.sol`		
Location(s)	N/A		
Confirmed Fix At			

Storage gaps are a convention for reserving storage slots in a base contract, allowing future versions of that contract to use up those slots without affecting the storage layout of child contracts.

The current implementation of `Base.sol` lacks storage gaps.

Impact Developers wont be able to add new storage variables to the `Base.sol` contract.

Recommendation Add storage gaps to `Base` or document very clearly in the code and in the internal documentation of the project that no more state variables can be added to the `Base` contract.

Ref: `https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable#storage-gaps`

Developer Response Developers acknowledged this issue

4.1.15 V-RIB-VUL-015: Inconsistent comment

Severity	Info	**Commit**	58bcd6c
Type	Usability Issue	**Status**	Fixed
File(s)	See Description		
Location(s)	N/A		
Confirmed Fix At	1dafe52		

The comments are inconsistent with the logic of functions in following example

```
/// @notice Transfer the insurance fund balance to a recipient
/// @dev Can only be called by the owner
/// @param collateral The collateral address
/// @param from The address to send the funds from
/// @param to The address to send the funds to
/// @param amount The amount to withdraw
/// @return fromBalance The new balance of the insurance fund account
/// @return toBalance The new balance of the recipient account
function _transferBetweenAccounts(address collateral, address from, address to,
    uint256 amount)
    internal
    returns (int256 fromBalance, int256 toBalance)
{
```

Snippet 4.21: `_transferBetweenAccounts()` in Accounts.sol

```
/// @notice Transfer the insurance fund balance to a recipient
/// @dev Can only be called by the owner
/// @param collateral The collateral address
/// @param from The address to send the funds from
/// @param to The address to send the funds to
/// @param amount The amount to withdraw
/// @return fromBalance The new balance of the insurance fund account
/// @return toBalance The new balance of the recipient account
function recoverFunds(address collateral, address from, address to, uint256 amount)
    external
    nonReentrant
    onlyOwner
    returns (int256 fromBalance, int256 toBalance)
{
```

Snippet 4.22: `recoverFunds()` in Accounts.sol

Impact N/A

Recommendation Fix comments

Developer Response Developers acknowledged this issue