Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Ribbon Finance: Sigma Five Exchange

Veridise Inc.
June 27, 2022

► **Prepared For:**

Ken Chan | Ribbon Finance
`https://www.ribbon.finance/`

► **Prepared By:**

Jon Stephens
Kostas Ferles
Muhammad Khattak
James Dong
Yanju Chen

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

► **Version History:**

August 4, 2022	V1
July 1, 2022	Draft

Contents

1 Executive Summary

From June 27 to July 10, Ribbon Finance engaged Veridise to review the security of their Sigma Five Exchange. The review covered the on-chain contracts that govern the exchange. Veridise conducted the assessment over 6 person-weeks, with 3 engineers reviewing code over 2 weeks from commit `8323100` to `8323100` of the `ribbon-finance/exchange-contracts` repository. The auditing strategy involved tool-assisted analysis of the source code performed by Veridise engineers. The tools that were used in the audit included a combination of static analyzers and bounded model checkers.

Summary of issues detected. The audit uncovered 18 issues, 7 of which are assessed to be of high or critical severity by the Veridise auditors. These bugs can lead to a variety of undesired behaviors, including an inability to settle options (V-RIB-VUL-001), a double-spending attack that allows users to be credited multiple times for the same purchase (V-RIB-VUL-002), a denial-of-service attack that would prevent options from being settled (V-RIB-VUL-004) and a possibility for funds to be locked in a contract (V-RIB-VUL-007). In addition to these high severity bugs, the Veridise auditors also identified several moderate-severity issues including a lack of timestamp expiration checks (V-RIB-VUL-008,V-RIB-VUL-009,V-RIB-VUL-011).

Code assessment. The Sigma Five Exchange implements an order-book based centralized exchange with components that exist both on- and off-chain. On the blockchain, Sigma Five Exchange settles options, matches trades and stores user account information. The contracts provide very little functionality to un-authorized users, only allowing them to deposit collateral in their account. Rather, users must interact with the off-chain interface which then makes requests to the on-chain contracts. As such Sigma Five Exchange makes extensive use of EIP-712 so that on-chain contracts can verify user consent. In addition, the Sigma Five Exchange provides proxies that are intended to interact with bridges between the Ethereum and Arbitrum networks.

Ribbon Finance provided the source code for the Sigma Five Exchange contracts for review. A foundry-based test-suite accompanied the source-code with tests written by the developers. These tests encompass registering signing keys, depositing collateral, withdrawing collateral, creating orders, executing both put and call trades, and calculating option payouts. In addition, the client provided a whitepaper describing the goals of the exchange and documentation describing the intended behavior for the contracts.

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
Sigma Five Exchange	`8323100`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
June 27 - July 10, 2022	Manual & Tools	3	6 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	4	4
High-Severity Issues	3	3
Medium-Severity Issues	4	4
Low-Severity Issues	3	3
Warning-Severity Issues	3	3
Informational-Severity Issues	1	1
TOTAL	18	18

Table 2.4: Category Breakdown.

Name	Number
Logic Error	4
Validation Error	4
Denial of Service	2
Locked Funds	1
Double Spending	1
Uninitialized Variable	1
Replay Attack	1
Unused Parameter	1
Access Control	1
Standard Library	1
Gas Optimization	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of the on-chain portion of the Sigma Five Exchange. In our audit, we sought to answer the following questions:

- Is it possible to perform a trade without a user's consent?
- Can user funds be lost or locked?
- Is it possible for a user to manipulate the price of an asset?
- Does the contract correctly hash structured data as defined by EIP-712?
- Can signed trades be replayed by a malicious user?
- Can one malicious user prevent trades from occurring?

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 evaluate how the code behaves given unexpected inputs. To do this, we created several unit tests using the Foundry testing framework, and then we applied the property-based testing capabilities of Foundry to fuzz potentially vulnerable methods.

Scope. This audit is restricted to the on-chain behaviors of the Sigma Five Exchange. As such, Veridise engineers first reviewed the provided whitepaper to understand the desired behavior of the protocol as a whole. They then expected the provided tests and contract documentation to understand the desired behavior of the provided contracts as well as how the developers intend to interact with them. Afterward, the Sigma Five Exchange contracts were assessed for bugs and security issues.

In terms of the audit, the key components include the following:

- The order-matching and trading logic.
- Option tracking and settlement
- User signature validation
- User account management which includes registering/revoking signatures and depositing/withdrawing funds

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:

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:

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

Vulnerability Report 4

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	Options Never Settled	Critical	Fixed
V-RIB-VUL-002	Double Credit/Debit	Critical	Fixed
V-RIB-VUL-003	Option Asset Never Credited/Debited	Critical	Intended Behavior
V-RIB-VUL-004	No Asset Ownership Check In Trade	Critical	Acknowledged
V-RIB-VUL-005	Option Position Never Updated	High	Fixed
V-RIB-VUL-006	Trade DoS	High	Acknowledged
V-RIB-VUL-007	Locked Funds in DepositProxy	High	Fixed
V-RIB-VUL-008	No Check for Signing Key Expiration	Medium	Acknowledged
V-RIB-VUL-009	No Check For Option Expiration	Medium	Fixed
V-RIB-VUL-010	No Negative Position Check in Withdraw	Medium	Acknowledged
V-RIB-VUL-011	Option Expiration During Trade	Medium	Fixed
V-RIB-VUL-012	Improper Data Hashing of Dynamic Type	Low	Fixed
V-RIB-VUL-013	Possible Duplication of Valid Orders	Low	Acknowledged
V-RIB-VUL-014	settled [instrumented] Might Never Be True	Low	Fixed
V-RIB-VUL-015	Unused Parameter in settleOption	Warning	Fixed
V-RIB-VUL-016	Owners Should Not Be Keepers	Warning	Acknowledged
V-RIB-VUL-017	Use OpenZeppelin ECDSA	Warning	Acknowledged
V-RIB-VUL-018	setOwner Can be Made External	Info	Fixed

4.1 Detailed Description of Bugs

In this section, we describe each uncovered vulnerability in more detail.

4.1.1 V-RIB-VUL-001: Options Never Settled

Severity	Critical	**Commit**	8323100
Type	Uninitialized Variable	**Status**	Fixed
Files	src/Instruments.sol, src/storage/InstrumentsStorage.sol		
Functions	settleOption, totalOptionPayout		

Description The `InstrumentsStorage` contract defines the variables `openInterest` and `expiryPrice` but these variables are never set. As a result, an instrument's `openInterest` and `expiryPrice` will always be set to 0. Having these values set to 0 will cause `totalOptionPayout` to always return 0 (since both the amount and expiryPrice is 0). This in turn causes `settleOption` to always return without crediting or debiting user accounts since the total payout is 0.

```
    function settleOption(uint256 instrumentId, uint256) external onlyKeeper {
        ...
        uint256 totalPayout = totalOptionPayout(instrumentId);
        if (totalPayout == 0) {
            return;
        }
        ...
    }
```

Snippet 4.1: Locations where `openInterest` and `expiryPrice` affect the behavior of `settleOption`

Recommendation Add functionality to adjust `openInterest` and `expiryPrice` appropriately.

4.1.2 V-RIB-VUL-002: Double Credit/Debit

Severity	Critical	**Commit**	8323100
Type	Double Spending	**Status**	Fixed
Files	src/Instruments.sol		
Functions	_addAccount/settleOption		

Description When an account is removed using `_addAccount`, it is currently only removed from `instrumentAccounts`. This means that the corresponding account still exists in `_instrumentAccounts` . By making multiple trades, a single account can therefore be added to `_instrumentAccounts` multiple times. If this were to occur, when the option is settled the user will be credited or debited (based on their current position) as many times as they appear in `_instrumentAccounts`. An attacker can use such a technique to obtain ownership over additional assets and possibly drain the exchange's pool of assets.

```
    function _addAccount(
        uint256 instrument,
        address account,
        int256 position
    ) internal {
        if (position == 0) {
            instrumentAccounts[instrument][account] = false;
        } else if (!instrumentAccounts[instrument][account]) {
            _instrumentAccounts[instrument].push(account);
            instrumentAccounts[instrument][account] = true;
        }
    }
```

Snippet 4.2: Function that adds/removes accounts participating in an instrument

```
    function settleOption(uint256 instrumentId, uint256) external onlyKeeper {
        for (uint256 i; i < usersLength; i++) {
            address user = _instrumentAccounts[instrumentId][i];
            int256 userBalance = AccountsInterface(accounts).getInstrumentPosition(
                user, instrumentId);
            if (userBalance > 0) {
                uint256 payout = (uint256(userBalance) * totalPayout)
                    / openInterest[instrumentId];
                AccountsInterface(accounts).credit(user, quote, payout);
            } else {
                uint256 payout = (uint256(-userBalance) * totalPayout)
                    / openInterest[instrumentId];
                AccountsInterface(accounts).debit(user, quote, payout);
            }
        }
    }

```

Snippet 4.3: Location where `settleOption` pays out the option

Recommendation Remove users from `_instrumentAccount` in addition to `instrumentAccount` in `_AddAccount`.

4.1.3 V-RIB-VUL-003: Option Asset Never Credited/Debited

Severity	Critical	**Commit**	8323100
Type	Logic Error	**Status**	Intended Behavior
Files	src/Instruments.sol		
Functions	settleOption		

Description Both when an option is traded and when it is settled, users are debited/credited the Account contract's `quote` token. As a result, the asset specified by an option is never transferred between users. We believe based on the documentation that the intended behavior is that the `settleOption` function resolve user positions by crediting/debiting the `option.asset` rather than `quote`.

```
    function settleOption_(uint256 instrumentId, uint256) external onlyKeeper {
        ...
        address quote = AccountsInterface(accounts).quote();
        for (uint256 i; i < usersLength; i++) {
            address user = _instrumentAccounts-[instrumentId][i];
            int256 userBalance = AccountsInterface(accounts).getInstrumentPosition(
                user, instrumentId);
            if (userBalance > 0) {
                uint256 payout = (uint256(userBalance) * totalPayout)
                    / openInterest[instrumentId];
                AccountsInterface(accounts).credit(user, quote, payout);
            } else {
                uint256 payout = (uint256(-userBalance) * totalPayout)
                    / openInterest[instrumentId];
                AccountsInterface(accounts).debit(user, quote, payout);
            }
        }
    }
```

Snippet 4.4: Location where settleOption pays out the option

Recommendation In `settleOption`, credit/debit the `option.asset` rather than the `quote`.

Developer Response The intention here is to credit/debit the quote asset. The intention is to purchase and settle options with respect to the quote currency.

4.1.4 V-RIB-VUL-004: No Asset Ownership Check In Trade

Severity	Critical	**Commit**	8323100
Type	Denial of Service	**Status**	Acknowledged
Files	src/exchange.sol		
Functions	trade		

Description When a trade occurs, there is no check that an account can provide the corresponding assets when the option is settled. A user can therefore be credited for selling assets that they may never own. Furthermore, when the option is eventually settled, if a user does not own a sufficient number of assets the `debit` call will revert since `accounts[account].balance [collateral] - amount` will cause an underflow. A malicious user can therefore prevent the option from ever settling while maintaining their ownership of the assets used to purchase the option.

```
    function settleOption(uint256 instrumentId, uint256) external onlyKeeper {
        ...
        for (uint256 i; i < usersLength; i++) {
            address user = _instrumentAccounts[instrumentId][i];
            int256 userBalance = AccountsInterface(accounts).getInstrumentPosition(
                user, instrumentId);
            if (userBalance > 0) {
                uint256 payout = (uint256(userBalance) * totalPayout)
                    / openInterest[instrumentId];
                AccountsInterface(accounts).credit(user, quote, payout);
            } else {
                uint256 payout = (uint256(-userBalance) * totalPayout)
                    / openInterest[instrumentId];
                AccountsInterface(accounts).debit(user, quote, payout);
            }
        }
    }
```

Snippet 4.5: Function settleOption

```
    function debit(
        address account,
        address collateral,
        uint256 amount
    ) external onlyAuthority nonReentrant returns (uint256) {
        if (!collaterals[collateral]) revert InvalidCollateral();
        uint256 balance = accounts[account].balance[collateral] - amount;
        accounts[account].balance[collateral] = balance;
        totalBalance[collateral] -= amount;
        emit Debit(account, collateral, amount);
        return balance;
    }
```

Snippet 4.6: Function debit

Recommendation Add a check in trade that will determine if a user owns the corresponding asset they are trading.

Developer Response The developers are adding a solution where the insurance fund would prevent this from occurring.

4.1.5 V-RIB-VUL-005: Option Position Never Updated

Severity	High	**Commit**	8323100
Type	Logic Error	**Status**	Fixed
Files	src/Instruments.sol		
Functions	settleOption		

Description In `settleOption`, a user's position is not updated after their account is debited/-credited based on their position. As a result, the instrument position stored by the contract might not reflect their true position. This could potentially impact liquidations/withdraws, since it may appear that an account has a negative position on an option even though it has already been settled. It could also cause individuals to be debited/credited multiple times if `settleOption` was invoked multiple times.

```
    function settleOption(uint256 instrumentId, uint256) external onlyKeeper {
        ...
        for (uint256 i; i < usersLength; i++) {
            address user = _instrumentAccounts[instrumentId][i];
            int256 userBalance = AccountsInterface(accounts).getInstrumentPosition(
                user, instrumentId);
            if (userBalance > 0) {
                uint256 payout = (uint256(userBalance) * totalPayout)
                    / openInterest[instrumentId];
                AccountsInterface(accounts).credit(user, quote, payout);
            } else {
                uint256 payout = (uint256(-userBalance) * totalPayout)
                    / openInterest[instrumentId];
                AccountsInterface(accounts).debit(user, quote, payout);
            }
        }
    }
```

Snippet 4.7: Location where `settleOption` pays out the option without updating position

Recommendation Update the user's position in `settleOption`.

4.1.6 V-RIB-VUL-006: Trade DoS

Severity	High	**Commit**	8323100
Type	Denial of Service	**Status**	Acknowledged
Files	src/Exchange.sol		
Functions	trade		

Description Method `trade` always reverts if any of the provided resting bids invalid. If there is no validation of orders in the front-end, then malicious actors can craft invalid resting bids that would render function `trade` unusable.

Below are all the checks that can revert the transaction (FileName:LineNum):

- Exchange.sol: 99 (invalid aggressive bid).
- Exchange.sol: 104 (invalid aggressive bid, insufficient amount).
- Exchange.sol: 127 (incompatible orders).
- Exchange.sol: 131 (incompatible instruments).
- Exchange.sol: 182 (invalid aggressive bid, not sure this is ever feasible).
- Exchange.sol: 205 (improperly signed order).
- Exchange.sol: 212 (invalid limit price).
- Exchange.sol; 217 (invalid order amount).
- Exchange.sol: 266 (unsuccessful debit taker-> maker).
- Exchange.sol: 275 (unsuccessful debit maker-> taker).

Recommendation Consider ignoring invalid resting bids or add validation in the front-end (if not being done already).

Developer Response The developers are performing validations in the off-chain component to prevent this issue.

4.1.7 V-RIB-VUL-007: Locked Funds In DepositProxy

Severity	High	**Commit**	8323100
Type	Locked Funds	**Status**	Fixed
Files	src/bridge/DepositProxy.sol		
Functions	processDeposit		

Description The `DepositProxy` contract allows users to deposit collateral in the contract and then forward that collateral to the `Accounts` contract. While it appears that the `DepositProxy` contract can be instantiated to forward any collateral in the constructor, it will only forward the collateral specified by `Accounts.quote`. If a user attempts to forward any collateral that is not `Accounts.quote`, it will be permanently locked inside the contract.

```
    constructor(address _collateral, address _accounts)
            Auth(msg.sender, Authority(address(0))) {
        collateral = ERC20(_collateral);
        accounts = _accounts;
    }
```

Snippet 4.8: The constructor for `DepositProxy` that allows a user to specify the collateral

```
    function processDeposit(address account, uint256 amount) external {
        require(msg.sender == keeper, "NOT_KEEPER");
        collateral.safeApprove(accounts, amount);
        Accounts(accounts).deposit(account, Accounts(accounts).quote(), amount);
        emit DepositProcessed(account, amount);
    }
```

Snippet 4.9: The collateral forwarding function that will only forward `Account.quote`

Recommendation Either forward the specified collateral to the `Accounts` contract or don't allow the user to specify the collateral. In addition, a withdraw function should be added that allows the owner to withdraw any currency that is owned by the contract.

4.1.8 V-RIB-VUL-008: No Check For Signing Key Expiration

Severity	Medium	**Commit**	8323100
Type	Validation Error	**Status**	Acknowledged
Files	Accounts.sol		
Functions	N/A		

Description 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. This can increase a user's attack surface since an attacker can attempt to compromise any key that the user has used (including those that have expired) to sign trades.

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

Snippet 4.10: The function that checks a signing key without determining if it has expired

Recommendation Check if the key has expired before using it.

4.1.9 V-RIB-VUL-009: No Check For Option Expiration

Severity	Medium	**Commit**	8323100
Type	Validation Error	**Status**	Fixed
Files	Options.sol, Instruments.sol		
Functions	N/A		

Description The Option struct contains an expiration timestamp which is never checked. When calculating `totalOptionPayout`, the protocol merely checks if the timestamp has been set, but does not determine if the option has expired. As a result, the protocol allows the option to be settled at any time rather than at the specified time.

```
    function totalOptionPayout(uint256 instrumentId) public view returns (uint256) {
        require(instruments[instrumentId] == InstrumentType.Option, "NOT_OPTION");
        Options.Option memory option = options[instrumentId];
        if (option.expiry == 0) {
            return 0;
        }
        uint256 expiryPrice = expiryPrice[options[instrumentId].asset][instrumentId];
        return option.getOptionPayout(openInterest[instrumentId], expiryPrice, 6);
    }
```

Snippet 4.11: Function totalOptionPayout

Recommendation Enforce the expiration timestamp of an option.

4.1.10 V-RIB-VUL-010: No Negative Position Check In Withdraw

Severity	Medium	**Commit**	8323100
Type	Validation Error	**Status**	Acknowledged
Files	src/Accounts.sol		
Functions	withdraw		

Description There is a period of time between when an option is traded and when it is settled where the option asset is still owned by the seller. During this time, it is possible for the account to withdraw the sold assets from the protocol completely. If this occurs, when the option is settled and a user does not own a sufficient number of assets the `debit` call will revert since `accounts[account].balance[collateral] - amount` will cause an underflow. A malicious user can therefore prevent the option from ever settling while maintaining their ownership of the assets used to purchase the option.

```
    function _withdraw(
        address collateral,
        uint256 amount,
        uint256 salt,
        bytes memory data,
        bytes memory signature
    ) internal returns (address, uint256) {
        bytes32 withdrawHash = Signing.deriveDigest(DOMAIN_SEPARATOR,
            keccak256(abi.encode(WITHDRAW_TYPEHASH, collateral, amount, salt,
    keccak256(data)))
        );
        if (!keepers[msg.sender])
            revert BadWithdraw(withdrawHash, RevertReason.NOT_KEEPER);
        if (!collaterals[collateral])
            revert BadWithdraw(withdrawHash, RevertReason.INVALID_COLLATERAL);
        if (amount == 0)
            revert BadWithdraw(withdrawHash, RevertReason.INVALID_AMOUNT);
        if (hashes[withdrawHash])
            revert BadWithdraw(withdrawHash, RevertReason.INVALID_HASH);
        address account = Signing.recover(withdrawHash, signature);
        if (account == address(0))
            revert BadWithdraw(withdrawHash, RevertReason.INVALID_ACCOUNT);
        uint256 balance = accounts[account].balance[collateral];
        if (balance < amount)
            revert BadWithdraw(withdrawHash, RevertReason.INVALID_BALANCE);

        ...
    }
```

Snippet 4.12: The checks performed when a user attempts to perform a withdraw

Recommendation Disallow withdraws if a user has a negative position in the corresponding asset.

Developer response This will be addressed in the off-chain back-end system. It will validate withdrawals pre-settlement since it also does that for open orders.

4.1.11 V-RIB-VUL-011: Option Expiration During Trade

Severity	Medium	**Commit**	0ac8d33
Type	Validation Error	**Status**	Fixed
Files	Exchange.sol		
Functions	trade		

Description Following issue V-RIB-VUL-009, we think that there should be an option expiry check in the `trade` function of Exchange.sol. Without such a check, it is possible for a user to perform a trade after the option has expired if it is partially filled. This could cause a user's funds to become locked if an option is settled only once or if it has been marked as settled. Alternatively it can allow a user to instantly gain funds by looking for partially-filled expired options that would yield instant profits (if not settled).

```
    function _validateOrder(
        AccountsInterface _accounts,
        Orders.Order calldata order,
        bytes32 orderDigest
    ) internal view returns (bool) {
        if (order.signature.length == 0) {
            return false;
        }
        address signer = Signing.recover(orderDigest, order.signature);
        if (signer == address(0)) {
            return false;
        } else if (signer == order.maker) {
            return true;
        } else if (_accounts.hasSigningKey(order.maker, signer)) {
            return true;
        } else {
            return false;
        }
    }
```

Snippet 4.13: The code that validates a trade

Recommendation Check if an option has passed its expiration in `trade`.

4.1.12 V-RIB-VUL-012: Improper Data Hashing Of Dynamic Type

Severity	Low	**Commit**	8323100
Type	Logic Error	**Status**	Fixed
Files	Executor.sol		
Functions	deriveCallDigest		

Description According to the EIP-712 standard, dynamic types must be encoded using `keccak256` . Function `deriveCallDigest` in `Executor.sol` passes `call.data` of type `bytes` to `abi.encode` without hashing it first.

```
    function deriveCallDigest(Call calldata call) internal view returns (bytes32) {
        return keccak256(abi.encode(CALL_TYPEHASH, call.index, call.fail,
            call.target, call.data));
    }
```

Snippet 4.14: The code in `deriveCallDigest` that doesn't adhere to the EIP-712 standard

Recommendation Consider changing the last argument of `abi.encode` to `keccak256(call.data)`.

4.1.13 V-RIB-VUL-013 Possible Duplication Of Valid Orders

Severity	Low	**Commit**	8323100
Type	Replay Attack	**Status**	Acknowledged
Files	Exchange.sol		
Functions	trade		

Description Because trades can be partially filled, a malicious user can duplicate valid orders (the `fills` mapping in Exchange is indexed by the hash of the original order). If the front-end does not (or is unable to) detect such scenarios, this can lead to situations where the exchange's order book is filled with duplicate orders which in turn can lead to potential price manipulation attempts.

Recommendation Ensure that there are no duplicate orders in the order books.

Developer response The off-chain component will ensure there is no duplication.

4.1.14 V-RIB-VUL-014: settled[instrumented] Might Never Be Set To True

Severity	Low	Commit	0ac8d33
Type	Logic Error	Status	Fixed
Files	Options.sol		
Functions	_settleOption		

Description When determining if an option has been settled, the protocol uses a counter to determine if all of the accounts have been debited/credited appropriately. It does so by incrementing the counter every time a account's position is settled. They then consider the option to be settled once the counter becomes equal to the length of `instrumentAccountsArray`. However, the counter is not incremented if an account has a position of zero. Since it is possible for accounts in `instrumentAccountsArray` to have a zero position, it can be the case that an option will never be marked as settled.

```
    function _settleOption(
        uint256 instrumentId,
        uint256 start,
        uint256 end
    ) internal {
        ...

        for (uint256 i = start; i < end; i++) {
            address user = instrumentAccountsArray[instrumentId][i];
            int256 instrumentBalance = AccountsInterface(accounts)
                .getInstrumentPosition(user, instrumentId);
            if (instrumentBalance == 0) {
                continue;
            }

            settledInstrumentAccounts[instrumentId] += 1;

            ...
        }

        if (settledInstrumentAccounts[instrumentId] == usersLength) {
            settled[instrumentId] = true;
        }
    }
```

Snippet 4.15: The code in `_settleOption` that adjusts and checks the counter

Recommendation To achieve the intended behavior, perhaps a map that keeps track of the settled accounts per instrument could be used.

4.1.15 V-RIB-VUL-015: Unused Parameter in settleOption

Severity	Warning	**Commit**	8323100
Type	Unused Parameter	**Status**	Fixed
Files	src/Instruments.sol		
Functions	settleOption		

Description The function `settleOption` defines two parameters but the second parameter has no name, only a type. It therefore cannot be used by the function.

```
    function settleOption(uint256 instrumentId, uint256) external onlyKeeper {
        ...
    }
```

Snippet 4.16: The unused parameter in `settleOption`

Recommendation Remove the unused parameter.

4.1.16 V-RIB-VUL-016: Owners should Not Be Keepers

Severity	Warning	**Commit**	8323100
Type	Access Control	**Status**	Acknowledged
Files	Executor.sol		
Functions	execute		

Description Because `Executor.sol` allows keepers to call arbitrary functions, we would advise to disallow for an owner to also be a keeper. If an attacker can control the input to function `execute` for a keeper that is also an owner, they can potentially call functions with an `onlyOwner` modifier (e.g., `updateKeeper` , `updateOwner`).

```
function execute(Call[] calldata calls) external nonReentrant
        returns (Result[] memory results) {
    if (!keepers[msg.sender]) revert NotKeeper();

    uint256 length = calls.length;
    results = new Result[](length);
    for (uint256 i; i < length; ++i) {
        bytes32 callHash = deriveCallDigest(calls[i]);
        ExecutedCall memory _executed = executed[callHash];
        if (_executed.executed) {
            results[i].called = true;
            results[i].success = _executed.success;
            continue;
        }

        (bool success, bytes memory result) = calls[i].target.call(calls[i].data)
;
        if (!success && !calls[i].fail) {
            if (result.length == 0) revert UnknownRevert(i);
            assembly {
                revert(add(32, result), mload(result))
            }
        }

        executed[callHash] = ExecutedCall(true, success);
        results[i].data = result;
        results[i].success = success;
        emit Executed(callHash, calls[i].index, success);
    }

    return results;
}
```

Snippet 4.17: The `execute` function that allows keepers to make arbitrary calls

Developer response The developer will be using a multisig for owner, keeper will be an EOA.

4.1.17 V-RIB-VUL-017: Use OpenZeppelin ECDSA

Severity	Warning	**Commit**	8323100
Type	Standard Library	**Status**	Acknowledged
Files	src/libraries/Signing.sol		
Functions	N/A		

Description Currently the developers are using an implementation of EIP712 taken from ProjectOpenSea. We would recommend that the developers instead use OpenZeppelin's ECDSA implementation. This implementation has been rigorously tested and has some additional checks that are missing from the Signing implementation.

Recommendation Use the OpenZeppelin ECDSA to verify signers and hash structured data.

Developer response The developers are comfortable with using their current implementation, but have added checks from OpenZeppelin's implementation that were missing.

4.1.18 V-RIB-VUL-018: setOwner can Be Made External

Severity	Informational	**Commit**	8323100
Type	Gas Optimization	**Status**	Fixed
Files	src/Base/Base.sol		
Functions	setOwner		

Description The `setOwner` function is currently marked as public but since it is never called from within the contract, it can be made external to reduce gas costs.

```
    function setOwner(address newOwner) public onlyOwner {
        _setOwner(newOwner);
    }
```

Snippet 4.18: The public `setOwner` function

Recommendation Make `setOwner` external.