SPEARBIT

Ribbon Finance Security Review

Auditors

Christoph Michel, Lead Security Researcher

Optimum, Lead Security Researcher

Jay Jonah8, Security Researcher

M4rio.eth, Security Researcher

Danyal Ellahi, Junior Security Researcher

Grmpyninja, Junior Security Researcher

Report prepared by: Pablo Misirov, Danyal Ellahi and Grmpyninja

October 7, 2022

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Sigma5 is a high-performance decentralized options exchange which runs on Arbitrum, a Layer 2 blockchain system. Sigma Five operates a centralized orderbook but trades and option settlements happen on-chain. In the future Sigma Five may settle on a more scalable platform such as a ZK-rollup once those technologies have matured.

Traders are able to deposit USDC and start trading options on the platform. Characteristics of Sigma5:

- Central limit orderbook trading system.
- Portfolio margin enables users to gain more capital efficiency.
- Leverage is enabled for short option positions.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Sigma5 according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)

- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 10 days in total, Ribbon Finance engaged with Spearbit to review Sigma5. In this period of time a total of 37 issues were found.

The important conclusion of this security assessment is that only a small part of the platform was audited and it is difficult to evaluate the security posture of the system as a whole. The main concern which emerged right after the initial analysis is that most of the business logic is implemented in the backend which was not in the scope of this assessment. Aforementioned backend is a single centralization point and acts as an intermediary between users and the smart contracts, thus holds enough privileges to cause even a loss of funds if ever compromised. It is strongly recommended to perform a risk assessment related to privileged roles handling, a full security review of the cloud infrastructure and the backend security assessment. The latter should especially include end-to-end data flow evaluation to ensure the component holding keeper's role cannot be abused by an attacker.

Summary

Project Name	Ribbon Finance
Repository	exchange-contracts
Commit	99d3c0ac4ca2f2dc50c67...
Type of Project	Options Vault, DeFi
Audit Timeline	Aug 15th - Aug 26th
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	1
Medium Risk	8
Low Risk	6
Gas Optimizations	3
Informational	19
Total Issues	37

5 Findings

5.1 High Risk

5.1.1 Can trade on already settled accounts

Severity: *High Risk*

Context: Exchange.sol#L98

Description: Trading options is still possible through `Exchange.trade` as long as the instrument is not fully `settled`. An already settled account can still sell options that expired in the money. If the buyer has not been settled, they will be credited the payout but the settled seller won't be settled and debited again. This leads to more tokens being credited than debited.

Example:

- Assume an option expires in the money. The option writers have to pay the payout to the option buyers.
- Account A is settled.
- Account A sells options to their second account B.
- When B is settled they are credited a profit.
- Account A was already settled and is never debited the payout they should pay.

In the best case, the backend catches this imbalance and tries to liquidate account A. (If the insurance fund is also already settled this further complicates matters.) In the worst case, the off-chain margin requirements do not (fully) take positions of already settled accounts into account and funds can be stolen.

> The Ribbon team stated that their off-chain matching engine stops matching trades as soon as the settlement is initiated. We still recommend enforcing this on the smart contracts due to potential bugs in the backend or transaction order issues.

Recommendation: Disable trading once settlement has *started*, instead of after settlement is finished.

Ribbon: Fixed at the contract level: commit 02e621e

Spearbit: The mentioned changes add a `settled` check to `liquidate` but don't change anything in `trade`. Note that a `settled` check only disables it *after all* users have been settled. Our recommendation is to disable trading once settlement *started* (the first user was settled) which will likely require a new state variable in `Instruments`. As it is, the issue is not fixed.

Ribbon: Makes sense. I've replaced this with a `settling` variable that gets set to true as soon as the settlement begins (and `settled` only gets set to true post-settlement). The exchange contract now checks whether settlement has started on each of the core functions (trade, liquidate, forceTrade). commit d249a3e

Spearbit: Acknowledged.

5.2 Medium Risk

5.2.1 Plaintext private key (`signingKey`) stored in local storage

Severity: *Medium Risk*

Context: Video presentation of `signingKey` registration.

Description: The web application presented by the client revealed that to enhance user experience each user can generate additional private keys inside the web app (client-side), which same as the main wallet's can sign orders without the necessity of confirming each transaction in e.g. Metamask's pop-up. Such private key is generated in the browser and registered by the backend which uses `registerSigningKey` function.

The issue however emerges in case a user decides to use `remember me` functionality, which simply stores the private key in browser's local storage in plaintext.

In case of a direct access to the device, so either attacker approaching the laptop or device theft and disk cloning it is possible to extract such private key and potentially use it to sign orders on behalf of another user. Additionally other cases like browser 0-day and malware also have a very easy task as they can simply dump the local storage at any time to extract such sensitive data.

Generally it is not recommended to store any sensitive data in a persistent storage in plaintext form (e.g. nobody should store bank account's password in a non-encrypted file on a disk). However, in many web applications `remember me` feature by storing e.g. an authenticated bearer token or session's token do exactly the same thing. Then still in such cases second layer of authentication is usually implemented, so for instance a second factor, SMS or a similar method, thus it can generally be seen also as a bad PR and bad security practices.

> As the web application was not in the scope, it is unknown if other authentication mechanisms are implemented to prevent submitting orders on user's behalf only if the signature is correct and it should be tested in a separate web application and backend security evaluation.

Recommendation: As a first line of defence consider using client-side encryption to at least not store sensitive data in local storage in plain text and later perform a security assessment of the backend and front-end to review the architecture thoroughly, understand potential threats and to decide on the solution which will be the best trade-off between security and usability in this case.

Ribbon: Acknowledged, we will likely use shorter expiry timestamps to avoid potential issues with client-side private keys.

Spearbit: Acknowledged.

5.2.2 Contract susceptible to options expiry price manipulation

Severity: *Medium Risk*

Context: Instruments.sol#L155

Description: The smart contract allows setting arbitrary prices as the option's expiry price because it does not utilize any oracle integration nor other decentralized method to determine the correct price of the underlying option's asset.

It was stated by the team, that the team will set that price based on their custom backend price oracle implementation which will gather prices from various places.

As the backend was not in the scope of this assessment it is impossible to determine and test how robust that custom in-house mechanism is and whether the system is or is not susceptible to oracle price manipulation.

From the disclosed compromises running own and custom price, oracle implementation is known to be a difficult problem, therefore it should be addressed with care.

Currently, the smart contract allows invoking the following function which sets the price used in payout calculations.

```
function setExpiryPrice(
    address _asset,
    uint256 _expiry,
    uint256 _expiryPrice
) external onlyKeeper {
    require(!isDisputePeriodOver(_asset, _expiry), "DISPUTE_PERIOD_OVER");
    Options.Price memory previousPrice = expiryPrice[_asset][_expiry];
    expiryPrice[_asset][_expiry] = Options.Price(
        _expiryPrice,
        previousPrice.timestamp == 0 ? block.timestamp : previousPrice.timestamp
    );
}

function isDisputePeriodOver(address _asset, uint256 _expiry) public view returns (bool) {
    Options.Price memory price = expiryPrice[_asset][_expiry];

    if (price.timestamp == 0) {
        return false;
    }

    return block.timestamp > (price.timestamp + disputePeriod[_asset]);
}
```

After the dispute is over (which by default is set to +1 tick after the price is set) it is possible to `settleOptions` and calculate payouts based on this price.

In case of a keeper compromise or oracle manipulation event, this price can be set to an incorrect value leading to incorrectly calculated payouts.

Recommendation: Perform a full security assessment of the custom price oracle implementation including multiple scenarios involving price manipulations.

Consider using well-known decentralized oracles as price feeds which can be loaded directly on-chain to mitigate the risk of a compromised keeper who sets an arbitrary price and starts `settleOption` with incorrect parameters.

Ribbon: Since we will be on our own rollup, we will be using a TWAP price the backend sets when calling the set expiry price function. (The dispute period will allow us to react to incorrect prices before settlement occurs).

Spearbit: Acknowledged.

5.2.3 Keeper privileges centralization

Severity: *Medium Risk*

Context: Keeper functions.

Description: The system has four actors (roles): `owner`, `authority`, `keeper` and regular blockchain users.

In short, three of these roles can be described as follows:

- Regular users are allowed to deposit funds where all other functionalities can be executed by three other actors in the system.
- `owner` has control over the most crucial parts like permissions granting, key parameters and upgrades and was stated by the team to be a multi-signature wallet, thus is known to be a crucial asset which can be protected.
- `authority` is a third role, which from the source code is simply a role allowing cross-communication between Exchange, Instruments and Accounts smart contracts.

Finally the most interesting is the `keeper` role which executes most typical and daily operations and this role is assigned to a backend component hosted in the client's infrastructure. Among other less relevant privileges, it does the following:

- Finalize trades between orders selected by the backend (orders matching engine).
- Finalize options by invoking `settleOption`.
- Set an arbitrary price used during options settlement - `setExpiryPrice`.
- Execute liquidations for any account, without any restrictions and proof on-chain, based purely on the backend's logic - `liquidate`.
- Execute force trades which creates an arbitrary trade for anyone - `forceTrade`.

It was discovered that in the smart contract there aren't any mechanisms limiting or containing a compromised `keeper` scenario. A compromised `keeper` can arbitrarily change instrument balances by invoking arbitrary orders for any user or through `liquidation` and effectively drain funds (collateral) or influence options settlement parameters and payouts.

The same situation may happen if the web app/backend were to be susceptible to common bugs from the Web2 world, which can result in potentially malicious flows where the keeper invokes functions in the wrong order or with crafted parameters.

Because the keeper is hosted and accessible to the team, a great amount of trust and effort has to be put into it in order to make it secure and robust. This component (role) concentrates on many privileges and is a very important central part of the ecosystem.

It is debatable whether such centralization is desirable as a lot of trust is put on a centralized web service which can invoke actions without many restrictions.

Recommendation: Due to the fact the web infrastructure can be compromised, consider a keeper equally important as the owner of the contract and make sure to perform a thorough security audit of the backend.

Make sure to evaluate a compromised keeper scenario, its impact, defensive mechanisms in the backend to detect it and procedures for recovery in case of a compromise.

If possible split responsibilities and limit to the minimum operations performed by a `keeper` to e.g. regular trades and less impactful actions and add other roles like `liquidator` responsible for liquidations. Splitting roles and assigning them to different components make it more difficult for the attacker to compromise the whole system in case of compromising a single key (wallet).

For the `setExpiryPrice` which is currently also assigned to keepers, consider changing access control to owner only or review the finding regarding price oracle manipulation for a more complex and robust solution.

Ribbon: Added a new keeper type called `authorizedKeeper`: commit e233a18.

Previously we had the `authority` and `keeper` roles.

`authorizedKeeper` is for functions where the keeper does more than simply relay signatures signed by the users (for example liquidations and force trades)

So the roles are now:

- `authority`: Contract authorized to change state
- `authorizedKeeper`: Keeper authorized to do more complex tasks than simply relay signatures
- `keeper`: Keeper authorized to simply relay signatures from users

Spearbit: Acknowledged. Note that the `onlyAuthorizedKeeper` modifier and role itself is not being used anywhere.

5.2.4 Potential out of funds denial of service condition - keeper

Severity: *Medium Risk*

Context: Architecture

Description: Due to the architectural decision most of the regular user actions are tunnelled through the backend and are eventually executed by the backend component which has a `keeper` role. That wallet must have enough funds to execute all user actions on-chain.

In case the backend does not implement throttling or any other means to detect and block potential abuses, a malicious user can send multiple requests to the web application, which will be forwarded and executed by the keeper.

One such easy-to-use function is `registerSigningKey`. The attacker can send thousands of HTTP requests to register a `signingKey` which results in thousands of registrations potentially exhausting the keeper's balance and leading to its denial of service.

`registerSigningKey` is just a single example, but any other function which originates from a user and eventually is executed by the keeper can be used in this scenario.

Recommendation: Implement throttling and low funds detection to protect against out-of-funds denial of service attacks.

Ribbon: Makes sense, the backend currently only relays the registerSigningKey message once a user has a trade matched, this minimizes the amount of times we send txs on-chain.

Spearbit: Acknowledged. Note that the fix is insufficient as other actions can also potentially be used to perform the very same attack scenario. It is important to evaluate all backend API calls (or sequences of API calls) which can be initiated by regular users of the app within the context of this attack scenario.

5.2.5 Digital signatures might be replayed within the same chain

Severity: *Medium Risk*

Context: Signing.sol#L12-L14

Description: The domain separator which is used throughout the system does not include the address of the contract where the already processed signatures are stored (used mainly to block replay attacks within the same contract). This may lead to cross-contract replay attacks within the same chain.

Recommendation: The domain separator should include also the underlying contract address.

EIP712 allows including `address verifyingContract` in the `deriveDomainSeparator`. The signature of `DOMAIN_TYPEHASH` then also needs to be adjusted to include `address verifyingContract`.

Ribbon: Will consider this, initially we did not add `verifyingContract` in case we ever needed to make an emergency migration to a new completely new contract (non-upgrade) and still be able to reuse signatures.

Spearbit: Acknowledged.

5.2.6 `setExpiryPrice` can be used to set expiry price before the actual option's expiry price is determined

Severity: *Medium Risk*

Context: Instruments.sol#L155-L166 - `setExpiryPrice`, Instruments.sol#L278-L286 - `isDisputePeriodOver`

Description: The function in the `Instruments` contract used to set the expiry price for the underlying option's asset can be invoked at any time before the dispute time is over. By default, if there is no price set it is basically any moment and after the price was set it is till `price.timestamp + disputePeriod[_asset]` timestamp. As in `setExpiryPrice` there is no verification whether `require(_expiry < block.timestamp)`, it effectively means that the expiry price can be set before the asset's actual price is determined in the real world.

The following check in `setExpiryPrice` is always true for the first price that is set.

```
require(!isDisputePeriodOver(_asset, _expiry), "DISPUTE_PERIOD_OVER");
```

Recommendation: Consider adding a require statement similar to `settleOption` which allows invoking `setExpiryPrice` only after option's expiration time, not before.

```
require(_expiry <= block.timestamp, "NOT_EXPIRED");
```

Ribbon: Recommendation implemented in #PR 24.

Spearbit: Fixed.

5.2.7 Missing `signingKey` expiry timestamp verification during order's validation

Severity: *Medium Risk*

Context: Exchange.sol#L452-L470, Accounts.sol#L607

Description: The smart contract allows users to register a `signingKey` via `registerSigningKey` which can be later used to sign orders without interactions with Metamask which improves the user experience while trading in the web application. As `signingKey` can be used to create orders on user's behalf it has an expiration time, so after that time the orders signed by this key are no longer valid. The smart contract however does not check that expiration time and any registered `signingKey` is valid until it is not explicitly revoked. The order verification is implemented in `_validateOrder` function which checks if a `signingKey` is registered just by checking its existence and without expiry date validation in `hasSigningKey`.

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

In case the `signingKey` is stolen it might be possible for the attacker to create arbitrary orders for the victim leading to loss of funds. Even if the backend checks the expiration time, in a more complex scenario when a keeper is compromised, the attacker can also leverage such missing expiration check to manually submit orders on the victim's behalf bypassing backend verification.

> The development team stated the backend validates the expiration time of a `signingKey` used to sign an order. However, it was also said that currently the expiration time is set to max int, meaning each front-end generated `signingKey` is valid forever. It was also stated that the main reason why there is no expiration check in the smart contract is due to a potential race condition, however as it is expected the orders are created and fulfilled within a short time span, a reasonable longer expiration time can be chosen to reduce the risk of such race condition and improve the security of the system in case of a leaked `signingKey` by reducing the attack window.

Recommendation: Decide on a reasonable expiration (longer than average order's fulfilment/cancelation) time for each generated `signingKey` and enforce the expiration check in the smart contract order's signature validation.

Additionally prevent the possibility to register an already expired `signingKey`.

Ribbon: Expiry timestamp verification not implemented in the smart contract to avoid race conditions when submitting orders. The verification would be in the backend.

Spearbit: Acknowledged.

5.2.8 Insurance fund needs to be settled last

Severity: *Medium Risk*

Context: Instruments.sol#L229

Description: When an account does not have a sufficient account balance to be settled, the settlement reverts and needs to be liquidated. This liquidation process usually moves the account's instrument position to the insurance fund which is then debited instead once settled.

Note that accounts that have already been settled once cannot be settled again. To make sure that the insurance fund is correctly debited and does not end up with a remaining balance it must always be settled last.

It can happen that the insurance fund is already settled but further liquidations follow (as liquidatios can be performed even after expiry). The insurance fund will end up with a larger balance than it should have. This creates an accounting error for the insurance fund and it might take on more debt than it can actually cover.

> A similar issue applies to `forceTrade` and already settled accounts. Consider disabling force trades once settlement started.
>
> The Ribbon team stated that they follow a process of checking balances before settling an option. We still recommend enforcing this on the smart contracts due to potential bugs in the backend or race conditions.

Recommendation: Always settle the insurance fund last.

Ribbon: Fixed, insurance fund gets settled at the end: commit 0e5370b

Spearbit: Acknowledged.

5.3 Low Risk

5.3.1 `forceTrade` for-loop does not stop early

Severity: *Low Risk*

Context: Exchange.sol#L285

Description: Within the for-loop of the force trade, an `if` statement is defined to exit the loop if the `takerAmount` is matched. This is implemented to save gas as the iteration can stop early. Currently, the `forceTrade` does not check the `takerAmount`, it checks the `amount` which will make the iteration run until finished. The Spearbit team mentions that on `trade`, this implementation is done correctly.

Recommendation: Consider replacing the `amount` from the if statement with `takerAmount`.

Ribbon: Recommendation implemented in #PR 24.

Spearbit: Fixed.

5.3.2 `_validateOrder` does not verify for an approved instrument

Severity: *Low Risk*

Context: Exchange.sol#L452-L470

Description: The `_validateOrder` function is used within the trade functionality to check if an order is valid, meaning that if it has a valid signing key by the `trader`. Nevertheless, it is missing a critical check to see if an instrument is valid to be traded. This will let anyone submit trades for an instrument that does not exist.

Because funds are not compromised, the Spearbit team decided on a low severity as we cannot correctly check the backend code to see what happens with trades for instruments that do not exist.

Recommendation: Add an existence check for the instrument within the `_validateOrder` function.

Ribbon: Instrument existence validation implemented in debit/credit functions. #PR 24 solves this issue.

Spearbit: Acknowledged.

5.3.3 Setting `disputePeriod` of any asset can be griefed by `owner`.

Severity: *Low Risk*

Context: Instruments.sol#L67-L71

Description: The `disputePeriod` of any asset can be set to a ridiculously high value such as `type(uint256).max`, which will cause the `settleOption()` functions to be uncallable due to this check: `require(isDisputePeriodOver(option.asset, option.expiry), "DISPUTE_PERIOD_NOT_OVER");`.

```
function isDisputePeriodOver(address _asset, uint256 _expiry) public view returns (bool) {
    ...
    // @audit would always return false if disputePeriod is set to a high value.
    return block.timestamp > (price.timestamp + disputePeriod[_asset]);
}
```

Recommendation: Consider adding to the function `setDisputePeriod` an upper limit to which the `disputePeriod` can be set to, e.g., 1 hour.

```
function setDisputePeriod(address asset, uint256 _disputePeriod) external onlyOwner {
+   require(_disputePeriod <= 1 hours);
    disputePeriod[asset] = _disputePeriod;
    ...
}
```

Ribbon: Acknowledged.

Spearbit: Acknowledged, changes have not been introduced.

5.3.4 Digital signatures never expire

Severity: *Low Risk*

Context: Exchange.sol#L452-L470, Orders.sol#L9-L20

Description: Digital signatures do not have an expiration date attached which means they can be submitted at any time regardless of major changes in market conditions. It is a best practice to add the desired expiration date to the sign message to limit the risk of executing old, unwanted orders.

```
struct Order {
    address maker;
    bool isBuy;
    uint256 limitPrice;
    uint256 amount;
    // Instrument ID
    uint256 instrument;
    // Salt for each order to differentiate them
    uint256 salt;
    // Order signature
    bytes signature;
}
```

Such old order executions can happen in various different cases. For instance, it can happen in case of a bug in the backend which can for instance (e.g., during a backup restore) lose some information about cancelled orders. Alternatively, in case an attacker gains access to any resource where those orders are stored which potentially can be e.g., logs, database, some analytical services etc. It might also be possible to simply use those orders, regardless of their status and try to post them on-chain to execute transactions.

Introducing a reasonable expiration time in `Order` itself reduces such risk by limiting the time window when the order can be actually used on-chain.

Recommendation: Consider adding an expiration date to signed messages that are in use in the system and make sure to validate that expired messages are not allowed.

It is a business decision to determine for how long the orders should be valid and it should be decided taking into account the risk involved with long expiration date vs user experience. It can be also a good idea to implement this as a choice for users creating orders where the default expiration time could be limited, but a user can always extend it or turn it off completely.

Ribbon: Acknowledged, order expiration will be handled on the backend level rather than at the contract level.

5.3.5 Missing instrument existence check in `creditInstrument` and `debitInstrument`

Severity: *Low Risk*

Context:

- Accounts.sol#L295-L308 - debitInstrument
- Accounts.sol#L273-L286 - creditInstrument
- Accounts.sol#L186 - add instrument and set flat

Description: The `Accounts` smart contract sets a flag when an instrument is added but it does not verify whether the instrument exists in `debitInstrument` and `creditInstrument` functions. This is verified only while adding/removing an instrument.

Recommendation: Add verification if instrument exists to `debitInstrument/creditInstrument` similar to collateral check.

```
if (!instruments[instrumentId]) revert InvalidInstrument();
```

Ribbon: Recommendation implemented in #PR 24.

Spearbit: Fixed.

5.3.6 Hardcoded decimals=6 used in the codebase

Severity: *Low Risk*

Context: Instruments.sol#L270, Orders.sol#L22

Description: The `totalOptionPayout` function calls `getOptionPayout` with a hardcoded collateral decimals of 6.

```
option.getOptionPayout(openInterest[instrumentId], expiryPrice, 6);
```

Same goes for:

```
uint256 internal constant PRICE_UNITS = 10**6;
```

This currently matches the quote token's 6 decimals but it should be dynamically set to the actual quote token decimals in case the quote token changes.

Recommendation: Change the hardcoded `6` to the `quote` token's decimals.

Ribbon: Fixed for `Instruments.totalOptionPayout`, but hardcoded value kept for `Orders.PRICE_UNITS`.

Spearbit: Fixed.

5.4 Gas Optimization

5.4.1 Verify `amount` is greater than 0 to avoid unnecessary `withdraw()` calls

Severity: *Gas Optimization*

Context: WithdrawProxy.sol#L28-L44

Description: `amount` should be checked to avoid unnecessary `withdraw()`.

Recommendation: Consider implementing the code snippet below in `withdraw()`.

```
require(amount != 0);
```

Ribbon: `WithdrawProxy.sol` removed in #PR 24.

Spearbit: Fixed.

5.4.2 Add address check to prevent unnecessary `transferInstrument()` calls

Severity: *Gas Optimization*

Context: Instruments.sol#L84-L89

Description: `transferInstrument()` allows the address parameters `from` and `to` to be the same address. This will result in an unnecessary execution resulting in no change and a waste of gas.

Recommendation: Consider filtering self-transfers on the backend.

Ribbon: Self-trading remains allowed in smart contract and the solution will be implemented as a backend check. No smart contract code changes implemented.

Spearbit: Acknowledged.

5.4.3 `getAccount` returns quote balance twice

Severity: *Gas Optimization*

Context: Accounts.sol#L581

Description: The `Account.getAccount` function returns the quote balance twice. Once as `quoteBalance`, and once as `balances[0]` as `quote` is pushed to the `_collaterals` array.

Recommendation: Consider removing the explicit `quoteBalance`.

Ribbon: Redundant variable removed in #PR 24.

Spearbit: Fixed.

5.5 Informational

5.5.1 Incorrect revert error in `liquidation`

Severity: *Informational*

Context: Exchange.sol#L222, Exchange.sol#L231

Description: Error messages in `liquidate` function should be swapped as in the first case the insurance fund is debited and in case of an error revert reason indicates it is `ACCOUNT_DEBIT` error, when it should be `INSURANCE_-DEBIT`. The other way around is in the second case.

```
function liquidate(...) {
...
      try _accounts.debit(insuranceFund, quote, quoteAmount) {} catch {
          revert InvalidLiquidation(account, instrument, instrumentAmount, RevertReason.ACCOUNT_DEBIT);
      }
...
      try _accounts.debit(account, quote, quoteAmount) {} catch {
          revert InvalidLiquidation(account, instrument, instrumentAmount,
↪   RevertReason.INSURANCE_DEBIT);
      }
...
```

Recommendation: Swap revert reasons in `liquidate` function to indicate the true reason of the error.

Ribbon: Error messages swapped to correctly reflect the result. Implementation in #PR 24.

Spearbit: Fixed.

5.5.2 Missing inline documentation

Severity: *Informational*

Context: Options.sol#L7-L16, Accounts.sol#L389-L395.

Description:

- `Option` struct is missing documentation for `isPut`, `strike` and `expiry`.

```
/// @dev Option data
/// @param asset The underlying asset for the option
/// @param isPut
struct Option {
    address asset;
    bool isPut;
    uint256 strike;
    uint256 expiry;
}
```

- `registerSigningKey` function is missing documentation for `signingKey` argument. The `signingKey` is an address associated with the private key generated in the front-end which is going to be registered as a valid key to sign user's orders.

```
/**
 * @notice Registers a signing key to be used for trading for an account
 * @param expiry is the Unix timestamp for when the key will cease to function
 * @param signingKeySig is the signature generated by the signing key
 * @param accountSig is the signature generated by the account for registering the new key
 */
function registerSigningKey(
    address signingKey,
    uint256 expiry,
    bytes memory signingKeySig,
    bytes memory accountSig
) external nonReentrant {
```

Recommendation: Add missing inline documentation.

Ribbon: Documentation fixed. Changes in #PR 24.

Spearbit: Fixed.

5.5.3 Using while loops/memory arrays to remove items from dynamic storage arrays may run out of gas

Severity: *Informational*

Context: Accounts.sol#L153, Accounts.sol#L199, Accounts.sol#L456

Description: While loops are used in the codebase to remove elements from storage arrays. These arrays may grow with time, which may potentially lead to a denial of service when trying to remove elements since the entire array is being copied from storage to memory during the process.

Recommendation: Consider using gas-efficient data structures like EnumerableSet.sol, where adding/removing/checking for inclusion is executed in a constant time.

Ribbon: Acknowledged.

5.5.4 Constructor's parameters `_chainId` can replaced with the onchain value

Severity: *Informational*

Context: Accounts.sol#L48

Description: The `Exchange`, `Instruments` and `Accounts` contracts are using the chainId for signature encoding. Currently this chainId is set via a constructor parameter instead of retrieving via `block.chainid`.

Recommendation: Consider removing the parameter as a whole and use `block.chainid` when constructing the signatures.

Ribbon: We would prefer to use chainId 1 (mainnet), rather than the chainId of the rollup the contract is running on.

Spearbit: This could lead to cross-chain replayability issues. We don't recommend ever deploying this contract on mainnet.

Ribbon: Makes sense, no plans on deploying to other chains. (Main reason for keeping chainId 1 is we would like users to have their metamask connected to mainnet while using the app rather than the chain of the rollup the contract is running on).

5.5.5 The `Account` and `Exchange` contracts rely on custom errors pattern, `Instruments` does not adhere to this pattern.

Severity: *Informational*

Context: Instruments.sol#L141

Description: The `Account` and `Exchange` contracts heavily rely on Customer Errors . The Ribbon team also mentioned that they rely on these Custom Errors in the backend as well. The Instruments contract does not adhere to this pattern. It is usually recommended to keep the same standard across the contracts not only for better readability but also because the `require` pattern mindset is different from the `revert with a Custom Error` pattern and can introduce unwanted bugs.

Recommendation: Move the `Instruments` contract to `revert with Custom Errors` pattern.

Ribbon: Missing custom errors implemented in #PR 24.

Spearbit: Fixed.

5.5.6 `_matchOrders` should be changed to a view function

Severity: *Informational*

Context: Exchange.sol#L361

Description: The `_matchOrders` function does not modify the contract's state which can be defined as a view function. View functions also introduce a some extra reentrancy protection.

Recommendation: Modify the `_matchOrders` function type to a view.

Ribbon: Recommendation implemented in #PR 24.

Spearbit: Fixed.

5.5.7 No check for `instrument` existence

Severity: *Informational*

Context: Accounts.sol#L310

Description: Currently the `resetInstrumentPosition` function does not check if the `instrument` exists before changing its state in the `accounts` mapping.

Recommendation: Ensure that the `instrument` exists before changing its state. The function should revert if the `instrument` does not exist.

Ribbon: Recommendation implemented in #PR 24.

Spearbit: Fixed.

5.5.8 Misleading `removeInstruments` function naming

Severity: *Informational*

Context: Accounts.sol#L195

Description: The `removeInstruments` function name is misleading since the current name implies that multiple instruments will be removed when in fact only one is being removed.

Recommendation: Consider renaming the function to `removeInstrument` instead of `removeInstruments` since the function only removes a single instrument id instead of multiple.

Ribbon: Functions renamed in #PR 24.

Spearbit: Fixed.

5.5.9 No check for a sufficient user `balance`

Severity: *Informational*

Context: Accounts.sol#L256

Description: Currently the `amount` argument is subtracted from `balance` before checking if the user has enough funds. This can result in a failure with a generic underflow message if the user's balance is too low.

Recommendation: Consider checking if the user `balance` is greater than or equal to the `amount` being passed in before performing any subtraction.

Ribbon: Generic overflow exception is considered to be enough. No changes are required.

Spearbit: Acknowledged.

5.5.10 Fees not implemented

Severity: *Informational*

Context: Exchange.sol

Description: The whitepaper mentions trading fees, delivery fees and liquidation fees but none of these are implemented in the smart contract.

Besides creating earnings for the protocol, introducing fees would also help with the mitigation of DOS attack vectors on the keepers.

Recommendation: Implement fees.

Ribbon: Acknowledged.

5.5.11 Typo in inline docs

Severity: *Informational*

Context: Options.sol#L34

Description: Typo in inline docs.

Recommendation: Change `is` to `are`.

```diff
-/// @notice Calculate the collateral the holders of an option is entitled to post-expiry
+/// @notice Calculate the collateral the holders of an option are entitled to post-expiry
```

Ribbon: Typo fixed in #PR 24.

Spearbit: Fixed.

5.5.12 Missing checks for `address(0)` when assigning values to address state variables.

Severity: *Informational*

Context: WithdrawProxy.sol#L22-L25, Base.sol#L63-L68, Exchange.sol#L51-L61, Instruments.sol#L41-L45

Description: Setters of address type parameters should include a zero-address check otherwise contract functionality may become inaccessible.

Recommendation: Add `address(0)` checks.

Ribbon: Acknowledged.

5.5.13 Unused Parameter in `WithdrawProxy.sol`

Severity: *Informational*

Context: WithdrawProxy.sol#L29

Description: The function `withdraw` has a parameter without a name and only the type `address`, making it useless.

Recommendation: Remove `address` parameter.

```
function withdraw(
-   address,
    uint256 amount,
    bytes calldata data
) external {
```

Ribbon: `WithdrawProxy.sol` removed in #PR 24.

Spearbit: Fixed.

5.5.14 `initialize()` function may be frontrun

Severity: *Informational*

Context: Accounts.sol#L74, Exchange.sol#L43, Executor.sol#L24, Instruments.sol#L17

Description: Even though the deploy script calls `initialize` at some point, it is not atomic (through something like a factory which deploys + initializes in a single transaction) and it currently does not check if the deployer was frontrun.

Recommendation: Consider calling the `initialize` function in an atomic manner and check that the deployer wasn't frontrun.

Ribbon: Deployment handled in a foundry script deploys and initializes contracts atomically.

Spearbit: Acknowledged.

5.5.15 One-step critical address change.

Severity: *Informational*

Context: Base.sol#L59-L67

Description: Setting the owner is a one-step transaction. This situation enables the scenario of contract functionality becoming inaccessible or making it so a malicious address that was accidentally set as `owner` could compromise the system.

Recommendation: Consider making the change of `owner` a two-step process where the first transaction registers a new address to be the owner, and the second transaction (from the new address) claims the ownership.

Ribbon: Acknowledged.

5.5.16 Missing emit event for `resetInstrumentPosition`

Severity: *Informational*

Context: Accounts.sol#L310-L312

Description: The `resetInstrumentPosition` function is invoked during the `settleOption`, and resets the position for a given account. It might be useful in off-chain monitoring tools to monitor such event for a given user.

```
    function resetInstrumentPosition(address account, uint256 instrument) external onlyAuthority
↪   nonReentrant {
        accounts[account].positions[instrument] = 0;
    }
```

Recommendation: Consider emitting events in important functions changing the internal state in case it does not significantly increase the gas usage.

Ribbon: Emit event added in #PR 24.

Spearbit: Fixed.

5.5.17 Wrong inline docs in `Instruments.sol`

Severity: *Informational*

Context: Instruments.sol#L168-259

Description: The `totalOptionPayout`, `_settleOption`, `settleOption` functions' inline docs are for `addOption`.

Recommendation: Write correct inline docs for these functions.

Ribbon: Fixed in commit d13b4c0.

Spearbit: Fixed.

5.5.18 Protocol does not support non-standard tokens

Severity: *Informational*

Context: Accounts.sol#L526

Description: Some ERC20 tokens make modifications to their ERC20's `transfer` or `balanceOf` functions. One type are deflationary tokens that charge a fee on every `transfer()` or `transferFrom()` while others are rebasing tokens that increase in value over time like Aave's aTokens (`balanceOf` changes over time).

Adding these tokens as collateral and depositing them leads to issues in the user balances and `totalBalance` accounting.

Recommendation: Be aware of this limitation and don't add non-standard tokens as collateral.

Ribbon: Acknowledged.

Spearbit: Acknowledged. No changes introduced.

5.5.19 Removing collateral tokens can lead to stuck funds

Severity: *Informational*

Context: Accounts.sol#L148

Description: Removing collateral tokens can lead to stuck funds as `_withdraw` requires the collateral token to be registered.

Recommendation: Ensure that all users withdrew their collateral tokens before removing a token as collateral.

Ribbon: Acknowledged.

Spearbit: Acknowledged. No changes introduced.