Spectre Protocol Audit

Final Report

December 10, 2021

Team Omega

Summary

Spectre has asked Team Omega to audit the contracts that define the behavior of the Spectre Protocol.

We found 5 high severity issues - these are issues that can lead to a loss of funds, and that we believe are essential to fix.

We classified 3 issues as “medium” - these are issues we believe you should definitely address.

In addition, 11 issues were classified as “low”, and 13 issues were classified as “info” - we believe the code would improve if these issues were addressed as well.

Scope of the Audit

The audit concerns the solidity code committed in the github repository https://github.com/spectrexyz/protocol/ with commit hash 284fa9af72f000e98e2ba0229f609e4d07272573.
The following files were audited:

```
./token/sIERC20.sol
./token/sIERC721.sol
./token/sERC20.sol
./token/sERC721.sol
./broker/Broker.sol
./broker/IBroker.sol
./broker/libraries/Sales.sol
./broker/libraries/Proposals.sol
./utils/Splitter.sol
./utils/ISplitter.sol
./pool/FractionalizationBootstrappingPoolMiscData.sol
./pool/FractionalizationBootstrappingPoolFactory.sol
./pool/FractionalizationBootstrappingPool.sol
./pool/FractionalizationBootstrappingPoolUserDataHelpers.sol
./pool/interfaces/sIERC20.sol
./issuer/IIssuer.sol
./issuer/libraries/Issuances.sol
./issuer/libraries/Proposals.sol
./issuer/Issuer.sol
./issuer/interfaces/IFractionalizationBootstrappingPool.sol
./issuer/interfaces/IFractionalizationBootstrappingPoolFactory.sol
./issuer/interfaces/IBalancer.sol
./vault/IVault.sol
./vault/libraries/Cast.sol
./vault/libraries/ERC165Ids.sol
./vault/libraries/Spectres.sol
./vault/Vault.sol
./channeler/Channeler.sol
./channeler/IChanneler.sol
```

We did not audit the dependencies - including the Balancer code, that is also included in the repository. With respect to FractionalizationBootstrappingPool.sol, we limited our audit to the code in as far as it diverges from Balancer's WeightedPool2Tokens.sol

Resolution

The issues were subsequently addressed by Spectre in commit hash d23838d8ae1f26f7170bc2aea62c1977ce45423e. Team Omega reviewed these changes. Our observations are listed below each issue.

Methods Used

Code Review

We manually inspected the source code to identify potential security flaws.

The contracts were compiled, deployed, and tested in a test environment.

Automatic analysis

We have used several automated analysis tools, including Slither and Remix, to detect common potential vulnerabilities. No (true) high severity issues were identified with the automated processes. Some low severity issues, concerning mostly the solidity version setting and functions visibility, were found, and we have included them below in the appropriate parts of the report.

Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

Severity definitions

High	Vulnerabilities that can lead to loss of assets or data manipulations.
Medium	Vulnerabilities that are essential to fix, but that do not lead to assets loss or data manipulations
Low	Issues that do not represent direct exploit, such as poor implementations, deviations from best practice, high gas costs, etc
Info	Matters of opinion

Findings

General

G1. Assert copyright [low] [not resolved]

The code is distributed with the GPL license, but the license is meaningless without asserting the copyright on each source file. From the LICENSE.md file:

```
            How to Apply These Terms to Your New Programs

      [...] attach the following notices to the program.  It is safest
      to attach them to the start of each source file to most effectively
      state the exclusion of warranty; and each file should have at least
      the "copyright" line and a pointer to where the full notice is found.

        <one line to give the program's name and a brief idea of what it does.>
        Copyright (C) <year>  <name of author>

      [etc]
```

Severity: Low
Recommendation: Assert copyright in the way described in the license file.
Resolution: The Spectre team indicated that they will wait until they have a proper legal entity to claim the copyright

G2. A floating pragma is set instead of a fixed pragma [low] [resolved]

The Solidity pragma version is set as floating: pragma solidity ^0.8.0 instead of fixed. It is recommended to change this to a fixed one to ensure consistency of the bytecode.
Severity: Low
Recommendation: Remove the ^ symbol in the pragma definition to make the solidity version fixed.q
Resolution: This issue was resolved.

G3. Use exact versions for solidity dependencies In package.json [low] [resolved]

In package.json, smart contract dependencies are specified for a range of versions, not a specific one:

```
        "@balancer-labs/v2-pool-utils": "^2.0.1",
        "@balancer-labs/v2-pool-weighted": "^2.0.1",
        "@balancer-labs/v2-solidity-utils": "^2.0.0",
        "@balancer-labs/v2-vault": "^2.0.0",
        "@openzeppelin/contracts": "^4.0.0",
        "@openzeppelin/contracts-upgradeable": "^4.0.0"
```

Running npm install may upgrade some of these dependencies, and so developers may find themselves using different package versions.
Severity: Low. There is a risk that new bugs or incompatibilities will be silently introduced, and it may be a problem if, for example, compiled and deployed code needs to be verified.
Recommendation: Pin the packages to a precise version, so there is no ambiguity about the dependencies.
Resolution: This issue was resolved.

G4. Declare state variables to be public instead of defining custom getters [info] [not resolved]

In many places in the code we see the following pattern:

```
string private _foo;

function foo() public view returns (string) {
        return _foo;
}
```

Where the getter function contains no additional logic, this pattern can be replaced by a less verbose pattern: declare the state variable to be public, so the compiler creates a getter function automatically:

```
string public foo;
```

This would reduce the amount of code, and thereby lower the risk of errors while making the code more readable. Examples are sERC20.vault(), Broker.vault(), Broker.issuers(), etc.
Severity: Info
Recommendation: Define state variables that have public getters as public variables.
Resolution: This issue was not resolved.

G6. Change visibility from public to external where possible [low] [not resolved]

Some gas can be saved by setting the visibility of functions that are currently declared public, but are not called internally, to external.
For example Vault.tokenTypeOf, Vault.balanceOf, Vault.balanceOfBatch, etc.
Severity: Low
Recommendation: Declare public functions to be external, where possible.
Resolution: This issue was not resolved

G7. Prices are sticky-up if there is low liquidity [low] [not resolved]

Suppose the supply of tokens is very low, and the spot price P in the pool is rather high. There are no trades to be made, because no existing holders are willing to sell (the supply is low) and buyers are not willing to pay P to obtain tokens.
Suppose also there is a demand for tokens at a price P' that is lower than P, and that the original NFT owner is willing to sell tokens at price P'. In the current algorithm, there is no way for the owner to sell these tokens - as she has no way of issuing tokens at this lower price.

(The current set up allows some ways to increase the liquidity, but they seem work-arounds: for example, an account that has the the MINT_ROLE can minting new tokens for free and put them on the market (but see also issue C2), and the Issuer's "banker" can obtain new tokens at a large discount: she will pay the spot price P+fees, and immediately receive a large percentage back)
Severity: Low
Recommendation: Allow the original NFT holder (probably the "banker") to be able to always issue new tokens at a price of her choosing.
Resolution: This issue was discussed with the Spectre team, that decided not to address the issue.

G8. Permission management is complex [low] [not resolved]

The Spectre contracts permission management is complex, which makes it hard to judge the overall safety of the system, both for us in this audit as well as for future users of the system.

The complexity is derived from the fact that (a) there are several different patterns used to define permissions and (b) there is a high number of roles and permissions that need to be managed and assigned.

There are different patterns used to define permissions:

- Permissions that are set when a new NFT/sERC20 pair is created (the guardian of an issuance, the guardian of a sale, the admin role of the sERC20 tokens).
- Permissions that are held or are settable by accounts that have the DEFAULT_ADMIN_ROLE in the contracts that inherit from AccessControlUpgradeable.
- The Pool contract uses BasePoolAuthorization, which is Balancer's permission management contract, to manage access to certain specific functions.

These permissions are quite fine-grained, which leads to a large number of roles and permissions to keep track of. Some of those roles are assigned in the constructor of the contracts (such as Issuer.DEFAULT_ADMIN_ROLE, Vault.DEFAULT_ADMIN_ROLE, Broker.DEFAULT_ADMIN_ROLE), some when fractionalizing an NFT (the issuance's admin, the issuance's broker, the sale's admin, etc), while other roles can be assigned later by accounts that hold the relevant DEFAULT_ADMIN_ROLE (such as the Broker.ESCAPE_ROLE or the sERC20.MINT_ROLE)

Although this makes for a flexible and configurable system, there are two problems with this approach:

The complexity of the system makes it hard to understand and keep track of the different permissions, leaving more room for errors and mistakes. These risks do not only apply to the current code base, but apply throughout the whole life cycle of the contracts: mistakes can be made in the way permissions are defined in the code itself, in the way the code is subsequently deployed, or at a later stage, when roles are re-assigned.

Although trusting a specific account with powers to pause the system, or even change the contracts, is a common practice in the DeFi space, the pattern that is used here requires a much higher level of trust from users, as they must monitor a complex and evolving permission system with many roles and (presumably) as many agents.
Severity: Low - although complexity itself is a risk, we did not find any serious vulnerabilities, although we do mention a number of smaller issues below.
Recommendation: We recommend simplifying the role system - both limiting the number of roles, as well as the ways in which they are managed. Specifically, we recommend using one single account as the system's "super user" (instead of having a separate DEFAULT_ADMIN_ROLE for each contract) and limit the powers of that account to some simple emergency functions such as pause and upgrade.
We also recommend that you extend the test suite with more tests regarding roles and permissions.
Resolution: The Spectre team decided to not address this issue

G9. Npm audit gives high severity warnings (none in the Solidity dependencies) [info] [resolved]

Running `npm audit` shows there are several (63, out of which 36 classified as high) vulnerabilities in the npm dependencies. None are in the Solidity packages, but it would be a best practice to address those issues by upgrading packages for which a fix was released.
Severity: Info (no solidity packages affected)
Recommendation: Upgrade packages for which a fix was released.
Resolution: This issue was resolved to the extent that was possible.

G10. solcover is configured incorrectly causing coverage rating to drop almost 50% [info] [resolved]

Running `npm run test:coverage` returns coverage of only about 50%, since the .solcover.js is not properly configured to skip the balancer contracts, and since these are not tested, the total coverage drops from above 90% to just around 50%.
Severity: Info
Recommendation: Fix the .solcover.js configuration to skip the Balancer files. Generally, it is recommended to bring the coverage to 100% and ensure it is kept at this level with proper CI tools.
Resolution: This issue was resolved.

Broker.sol

B1. Buyout total price is based on supply, not cap, and owner will not receive her share [high] [resolved]

In _priceOfFor function on lines 416ff, the price that is paid in a buyout based on what it would cost to buy the outstanding supply of the sERC20 token, rather than the token cap, which represents the maximum amount of tokens that can be issued. This buyout sum can subsequently be claimed by the token holders, in proportion to the amount of tokens they hold.

This seems wrong: before the buyout has happened, the as of yet unminted tokens do represent real value (new tokens can be issued for a price comparable to the market price, which accrues to the original NFT issuer). When a buyout happens, the original NFT issuer will not get her share that corresponds to the tokens that are not issued yet.

Severity: High - the original NFT owner does not get her share of the buyout price
Recommendation: Base the buyout price on the cap, not on the total supply, and transfer the share proportional to the amount of un-minted tokens to the original NFT owner.
Resolution: Spectre has indicated that this behavior is intentional. After discussing the matter further, Spectre decided to add a configuration option with which a user can choose if the buyout price is based on the cap or on the outstanding supply of the token when registering a new sale.

B2. Buyout can be front-run [high] [not resolved]

If the multiplier is > 100%, a buyout will effectively pay a premium to all existing token holders. This makes it attractive to do certain front-running operations.

For example, a user is offering $1000 to buy out an NFT that has a supply of 100 shares worth 5$ each and a cap of 1000 tokens - effectively paying a premium of 100%. Before the transaction is mined, an attacker issues 50 new shares to herself (at, say, a price of $5.50 - the current price plus 10% of fees) - paying $275. She now holds ⅓ of the token supply, and so will be able to claim $333 from the buyout.
Severity: High - all existing token holders will effectively be robbed of their share of the buyout premium.
Recommendation: This issue does not seem to have a straightforward fix. A possible solution would be to have a certain recurring time period (an hour a day, a week or something similar) where trading and issuance are paused during which users can safely perform the buyouts.
Resolution: The Spectre team has acknowledged the issue, and decided not to address it due to the UX challenges that are involved with a resolution

B3. Buyout premium not available for liquidity providers due to arbitrage [medium] [resolved]

After a successful buyout, the remaining sERC20 token holders can redeem their tokens for the buyout price, which is (if the multiplier > 100%) higher than the spot price in the Balancer pool. This allows arbitrage traders to buy tokens from the pool at the spot price and redeem these for the buyout price. This will raise the pool price of the sERC20 token to the buyout price.
Severity: Medium. Liquidity providers that do not act quickly enough will miss out on the premium set by the buyout price. This presents an extra risk to liquidity providers: although LPs will not directly lose funds, they will be faced with impermanent loss.
Recommendation: Pause trading in the pool after a buyout happens. But this is just a partial fix, see the previous issue, B2.
Resolution: Pool trading now is paused after a buyout, which mitigates the issue. The related issue B2 is left unresolved.

B4. Sale state is not updated after escaping a token [medium] [resolved]

When calling _escape_ on line 297, the state of the sale is not updated, and so remains active despite the NFT already being detached from it.

Severity: Medium - users could lose funds buying out the NFT that has been escaped without them noticing it.
Severity: Medium
Recommendation: Set the sale state to Closed when the NFT is transferred out.
Resolution: This issue was resolved.

B5. Check if token is already registered [low] [resolved]

The register function does not check if a token is already registered, and so any account that holds the REGISTER_ROLE (now or in the future) can overwrite important data and steal the token.
Severity: Low
Recommendation: Checking if the sERC20 is already registered is cheap, and removes the risk of existing tokens to be overwritten.
Resolution: This issue was resolved.

B6. Irrelevant proposals are kept in storage forever [low] [resolved]

Rejected, expired, and withdrawn proposals are kept in storage for no apparent reason (also accepted proposals could be removed eventually, but there might be a reason to keep them around). It would save gas if these would be deleted from storage when the reject or withdraw proposal functions are called.
Severity: Low
Recommendation: Delete irrelevant proposals to save gas costs.
Resolution: This issue was resolved.

B7. Proposals logic is nearly identical between Broker and Issuer [info] [not resolved]

The Proposals logic on the Broker and Issuer contracts are nearly identical. It would be more efficient and clean to have that shared logic separated and then used by both to avoid repeating the code.
Severity: Info
Recommendation: Extract the common logic from the Broker and Issuer and use a shared code to inherit in both.
Resolution: This issue was not resolved.

B8. Reject and withdraw logic are nearly identical [info] [not resolved]

The Proposals withdraw and reject logic are nearly identical. It would be more efficient to unite them into a single function, or extract the common code to a separate function called by both.
Severity: Info
Recommendation: Merge the reject and withdraw function into a single one or move the common code to its own function.
Resolution: This issue was not resolved.

B9. Make reserve price settable [low] [resolved]

The reserve price currently is not changeable. What a reasonable value is here depends both on the issuer as well as what the market is willing to pay - and both of these may change over time. Specifically, this can become a problem if the reserve price is too high. The fact that the price is expressed in ETH, which is relatively volatile, makes this situation more likely to occur.
Similarly, it might also be useful to have the multiplier of the buyout price settable
Severity: Low - if the reserve price is too high, buyouts will effectively become unfeasible
Recommendation: Make the reserve price settable by the admin of the token.
Resolution: This issue was resolved.

Channeler.sol

C1. Channeler contract is DEFAULT_ADMIN_ROLE of the new sERC20 token [info] [resolved]

The fractionalize(..) function of the Channeler contract will clone the sERC20 token and assign the DEFAULT_ADMIN_ROLE to the address of the Channeler contract. This effectively disables role management logic on the sERC20 token, as the Channeler defines no functions to manage these roles (except granting the MINT_ROLE to the Issuer contract). Note specifically that both the PAUSE_ROLE and SNAPSHOT_ROLE that exist in the sERC20 contract become unusable, therefore, if this is intentional, it may be advisable to remove both these roles and disable their related functionalities.
Severity: We categorize this as info as we are not sure this is as intended.
Recommendation: If unintentional, assign admin role to the intended address.
Resolution: The Spectre team is aware of this issue and explained that the current behavior is as intended.

Issuer.sol

I1. Wrong comment at line 63 [info] [resolved]

At line 63 the comment says there is no check the allocation is lower than 100%, but this check does exist on line 97.
Severity: Info
Recommendation: Remove the comment.
Resolution: This issue was resolved.

I2. Make reserve price settable [low] [resolved]

The reserve price currently is not changeable. What a reasonable value is here depends both on the issuer as well as what the market is willing to pay - and both of these may change over time. Specifically, this can become a problem if the reserve price is too high. The fact that the price is expressed in ETH, which is very volatile, makes this situation more likely to occur.
See also issue B10.
Severity: Low - if the price is too high, issuing new tokens will effectively be paused
Recommendation: Make the reserve price settable by the admin of the token.
Resolution: This issue was resolved.

I3. Unnecessary check of reward > 0 [info] [resolved]

On line 503 the condition unnecessarily checks that either the reward or the value are greater than 0, but if the value is 0 the reward must also necessarily be 0, so it'll be enough to check that value is not 0.
Severity: Info
Recommendation: Remove the check of reward > 0.
Resolution: This issue was resolved.

I4. Issues B5, B6, B7, B8 apply for the Broker contract as well

Since the channeler contract has many similarities with the Broker contract, these issues from the Broker apply here in an identical manner.
Resolution: See relevant issues resolutions.

FractionalizationBootstrappingPool.sol

P1. Price of ERC20 in balancer pool is sticky-down [medium] [not resolved]

The interplay between the adjustment of the balancer pool weights and the logic of buying new coins from the pool makes the price “sticky-down”, meaning that the price of the token may react slowly - and may even go down - as a result of an increase in market demand.

For suppose the current spot price and the TWAP (which is the average pool price in the last 24 hours) are equal to P, and the market is willing to pay P + X (in the sense that _all_ buyers are willing to pay P+X or more). Buyers have the option to buy tokens from the pool at price P (plus slippage) or to issue new tokens at price TWAP (plus fees), where TWAP is the average pool price in the last 24 hours, which for this example we assume to be equal to P.

In the second case, if the user chooses to issue new tokens, this will adjust the weights of the pool and thereby lower the pool’s spot price. In the first case, buying from the pool will raise the pool’s spot price. The issuance price, which is based on the 24 hour TWAP, lags behind - and this puts a limit on the speed at which the price increases - because if the difference between spot price and twap becomes higher than the issuance fees, there is an arbitrage opportunity, where a trader can issue new tokens and swap them to the pool for profit. The selling will bring down the pool price - not just as a direct effect of selling new tokens to the pool, but also because increasing the total supply will adjust the weights of the pool and therefore puts further downward pressure on the price.

The net effect is that when demand is high, the price of tokens will be “sticky”: the price will react sluggishly, or may even go down.

Severity: Medium. The Spectre issuer misses out on some income, as she will be selling (the shares of) her NFT below market price, and runs the risk of having the NFT “bought out” at a price below market price.
Recommendation: While it is inevitable that issuing new tokens will put downward pressure on the market price, this effect is amplified by the fact that the new issuance adds additional downward pressure on the price, as it effectively lowers the price offered in the balancer pool by adjusting the pool weights. You should consider removing the interplay between supply and pool weights from the algorithm, or possibly change it into a time-based Dutch Action-style price discovery mechanism as is used in the original LBP price discovery mechanism design.
Resolution: This issue was not resolved. The Spectre team affirmed that this is the intended behaviour.

FractionalizationBootstrappingPoolFactory.sol

F1. Owner of balancer pool is set to address(0) [info] [resolved]

In the create function, which deploys a new FractionalizationBootstrappingPool, the owner is set to address(0). This means that the actions that are singled out in FractionalizationBootstrappingPool._isOwnerOnlyAction - these are the functions setSwapFeePercentage and setAssetManagerPoolConfig - will effectively be disabled, while other functions that have the authenticate modifier (such as setPause) will be callable by the authorizer of the Balancer Vault (i.e. by the Balancer Governance, but not by any address from the Spectre system).
Severity: Medium - this fact that this functionality is disabled does not seem to be intended, and there is some added systematic risk by allowing the Balancer governance to pause the pool.
Recommendation: Set the owner of the new balancer pool to a real address. And add tests for testing access to these functions.
Resolution: The Spectre team indicated that the functions setSwapFeePercentage and setAssetManagerPoolConfig are disabled by design. Furthermore, the team indicated that the fact that Balancer Governance can pause the pool has been implemented at the request of the Balancer team.

sERC20.sol

E1. onERC20Transferred is called before balances are updated [high] [resolved]

If a token is transferred by a direct call on the sERC20 contract, on line 216, in the hook _beforeTokenTransfer, Vault.onERC20Transferred is called. Subsequently, the onERC20Transferred function calls onERC1155Received on the receiver address - so this function is called before the balances of the sERC20 contract are updated.

This is problematic for two reasons.

1) This is not according to the ERC1155 specifications, which state that "An ERC1155-compliant smart contract MUST call this function on the token recipient contract, at the end of a \`safeTransferFrom\` after the balance has been updated".
2) The receiver is not a trusted contract, and may be malicious. Calling this function before the balances are updated opens up functions that use the sERC20.transfer, sERC20.mint and sERC20.burn functions to re-entrancy attacks.

   For example, an attacker can re-enter the Splitter.withdraw function and withdraw more tokens than expected from the Splitter contract. For this, she crafts an AttackerContract that implements onERC1155Received, which calls the Vault.withdraw function. Suppose the Splitter contract holds 1000 tokens, no withdrawals have been made yet, and the AttackerContract

> receives 10% of the splits (this last assumption makes the attack hard to implement). The attacker now calls the withdraw(sERC20, AttackerContract), which will do a sERC20.transfer call to the AttackerContract for 100 tokens. The Attacker contract's onERC1155Received function, which is called before the balances are updated, now calls the withdraw function again. As the balance of the Splitter function has not been updated yet, the calculation results in the AttackerContract having the right to withdraw 110 tokens - of which 100 have already been withdrawn. Winding up, the Splitter will now transfer first 10 and then 100 tokens to the AttackerContract.

Severity: High - even if the specific attack is unlikely to be implemented, there are systematic risks
Recommendation: An easy fix is to call Vault.onERC20Transferred in the _afterTokenTransfer hook instead of the _beforeTokenTransfer hook.
Resolution: This issue was resolved.

E2. An sERC20.transfer to a contract that correctly supports ERC1155Receiver may revert in some cases [high] [resolved]

The sERC20 implementation calls Vault.onERC20Transferred, which will in turn try to call beneficiary.onERC1155Received if the beneficiary is a contract that supports the ERC1155Receiver interface. This is line 426 of Vault.sol:

```
IERC1155Receiver(to).onERC1155Received(operator, from, id, amount, "")
```

The last argument of the onERC1155Received function is the calldata argument. A problem arises if the receiver contract does not accept an empty calldata argument - in this case, the transfer will be reverted.
Severity: High. It is unlikely but possible that a user will not be able to retrieve her funds from the contracts.
Recommendation: Do not revert if the onERC1155Received reverts or returns an unexpected value.
Resolution: This issue was resolved.

E3. sERC20 inherits from a open Zeppelin contract marked as "draft" [info] [resolved]

The sERC20 contract inherits from import "@openzeppelin/contracts-upgradeable/token/ERC20/extensions/draft-ERC20PermitUpgradeable.sol";
It is generally a bad practice to use code marked as "draft" in main production.
Severity: Info
Recommendation: Unless the Permit functionality from it is deemed strictly necessary, it might be advisable to wait until the contract is no longer a draft before including it.
Resolution: This issue was resolved, the use of draft-ERC20PermitUpgradeable.sol was removed.

Splitter.sol

S1. Declare visibility for _splits [info] [resolved]

On line 20 the visibility definition of _splits is missing, this was probably intended to be private, but we recommend, as per G4, to make it public.
Severity: Info
Recommendation: Declare _splits as a public variable.
Resolution: This issue was resolved, _splits has been declared a private variable.

S2. Remove requirement in register that the total allocation must be < 100% [info] [resolved]

On line 64, it is required that:

```
require(total < HUNDRED, "Splitter: total allocation must be inferior to 100%");
```

Where the total is the sum of the splits and the _protocolFee. The lines immediately following normalize these values, making this check pretty much meaningless.
Severity: Info - at most, this is a small UX irritation, as it is natural to provide input values here that sum to 100.
Recommendation: Remove the requirement.
Resolution: This issue was resolved.

Vault.sol

V1. Fractionalization can be front-run [high] [resolved]

The fractionalize on the vault can be called by anyone, and the ERC721 token must be approved beforehand. An attacker could front-run the fractionalize call to the vault after the ERC721 owner approved the Vault for transfer, and pass their own parameters to the fractionalize call - for example, setting himself as admin and broker and essentially giving himself power to steal the token from the vault.
Severity: High - users can lose their NFTs.
Recommendation: In the Vault contract limit the call to fractionalize to the owner of the NFT and the Channeler contract, and in the Channeler contract limit the call exclusively to the NFT owner.
Resolution: This issue was resolved.

V2. Make the broker argument part of the Vault settings [low] [not resolved]

The vault allows the caller of fractionalize to set the Broker to any address, although they should always be the main Broker contract, this is unnecessary and opens a door for mistakes and issues (see V1 above). This might also apply to the admin contract.
Severity: Low. But see V1 above.
Recommendation: Add a global variable for the Broker contract and use it in the fractionalize operation instead of accepting the broker from the user. Admin (which should normally be the Channeler) might be better being set in a similar central manner.
Resolution: This issue was not resolved. The Spectre team has chosen to leave this as is to prioritize customizability in this case.

V3. Unlock overloaded function can be removed [info] [resolved]

On line 346ff, the function unlock(sERC20,address,data) is defined, which is an overloaded version of unlock(address,address,data). Overloading functions in smart contracts is in general not recommended, and in this case does not seem to provide significant functionality.
Severity: Info
Recommendation: Remove one version from the contract.
Resolution: This issue was resolved.