SOFA.ORG

Sofa Protocol Contract Review

Version: 2.0

June, 2024

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the SOFA.org smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the SOFA.org smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the SOFA.org smart contracts.

Overview

Sofa protocol offers on-chain structured financial products to users at large. In this review, there were two main kinds of products being considered: DNT (Double No Touch) and Smart Trend.

DNT products have a fixed period during which a product's price must stay within a certain range. Smart Trend products reward on a sliding scale if a product's price is within a certain range at the moment of conclusion.

These products also come in variants that invest the collateral amounts in the AAVE protocol during the product's operation and vaults that aim to provide a simulation of leveraged use of the main two products.

Security Assessment Summary

Scope

The scope of this time-boxed review was strictly limited to files hosted on the sofa-protocol repository at commit 65bb4c3.

Retesting activities were performed on commit ef1e9b9.

Specifically, all files contained in the following directories were in the audit scope:

- `oracles/`
- `strategies/`
- `tokenomics/`
- `vaults/`

Note: third party libraries and dependencies, such as OpenZeppelin, were excluded from the scope of this assessment.

Approach

The manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team also utilised the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Coverage Limitations

Due to a time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 19 issues during this assessment. Categorised by their severity:

- Critical: 2 issues.
- High: 1 issue.
- Medium: 3 issues.
- Low: 7 issues.
- Informational: 6 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the SOFA.org smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
SFA-01	Oracle Expiry Calculations Can Resolve To Future Timestamps	Critical	Resolved
SFA-02	Donation Attacks On Empty AAVE Vaults Can Steal Deposits	Critical	Resolved
SFA-03	Incorrect Calculation Of `minterCollateral`	High	Resolved
SFA-04	Rounding Issue Causing Dust Amount Being Locked	Medium	Resolved
SFA-05	`COLLATERAL` Token Is Not Part of The `digest`	Medium	Closed
SFA-06	Fee Rates Can Change on Active Products	Medium	Closed
SFA-07	Replay Attack on APR Changes in the Leverage Vaults	Low	Closed
SFA-08	Stale Oracle Prices Are Processed As Recent	Low	Closed
SFA-09	Missed Price Data Is Filled In With Linear Derivations	Low	Closed
SFA-10	Any Contract In Any Transaction Initiated By Owner Can Spend RCH Before The Start Time	Low	Closed
SFA-11	Possible Overflow for High Decimal Tokens	Low	Closed
SFA-12	Initialisers Not Disabled on Implementation Contracts	Low	Closed
SFA-13	Collateral Can Be Locked For Extremely Long Periods	Low	Closed
SFA-14	Defunct Hardhat Import In Vaults	Informational	Resolved
SFA-15	Bitmap Is Unnecessary	Informational	Closed
SFA-16	Minter Or Maker May Contribute Zero Collateral	Informational	Closed
SFA-17	Burn Batch Allows Duplicates	Informational	Closed
SFA-18	Reentrancy In AAVE Vaults	Informational	Resolved
SFA-19	Miscellaneous General Comments	Informational	Closed

SFA-01	Oracle Expiry Calculations Can Resolve To Future Timestamps		
Asset	`oracles/HlOracle.sol & oracles/SpotOracle.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The calculation of the `expiry` timestamp in the oracles can resolve to future times. This has different consequences for each oracle contract.

This is the calculation in question:

HlOracle.sol

```
uint256 expiry = block.timestamp - block.timestamp % 86400 + 28800;
```

Consider a timestamp of 1,719,014,400:

```
expiry = 1719014400 - (1719014400 % 86400) + 28800
= 1719014400 - 0 + 28800
= 1719043200
```

The result of `expiry` will be 8 hours (28,800 seconds) in the future.

The calculation in question is found on line [**22**] of `SpotOracle.sol`.

For `SpotOracle.sol`, any call to settle prices before 08:00 will result in that time being recorded with a time in the future of 08:00. Even it the protocol has a cron job running at 8am each day for this oracle, a user can still simply call `settle()` 8 hours early to settle an early price.

This allows users to mint products that officially have an exposure period of 24 hours, but actually have an exposure of only 16 hours, as the expiry price will be set 8 hours in advance.

The calculation in question is also on lines [**18,37,49**] of `HlOracle.sol`.

`HlOracle` is set up to use Chainlink Automation, which would result in similar consequences as `SpotOracle`, except that it has the following protective modifier:

HlOracle.sol

```
modifier oracleAvailable() {
        // latestUpdateOracleTime >= 8:00 UTC
        uint256 latestUpdateOracleTime = AUTOMATED_FUNCTIONS_CONSUMER.lastUpkeepTimeStamp();
        require(latestUpdateOracleTime >= block.timestamp - block.timestamp % 86400 + 28800, "Oracle: not updated");
        _;
}
```

This modifier means that all calls to `checkUpkeep()` and `settle()` will revert for the first 8 hours of each day.

This in turn means that, if there is a period of down time, the `oracleAvailable` modifier could prolong this down time before `settle()` would be able to run again and fill in the missing data. As that oracle updates by calling `AUTOMATED_FUNCTIONS_CONSUMER.s_lastResponse()`, the derived prices that are updated to might also be different.

Recommendations

Modify the expiry calculation to avoid future times. Consider using:

```
uint256 expiry = (block.timestamp-28800) - ((block.timestamp-28800) % 86400) + 28800;
```

Resolution

In the case of `SpotOracle`, the following check was added in commit 1a47157:

SpotOracle.sol

```
require(block.timestamp >= expiry, "Oracle: not expired");
```

In the case of `HlOracle`, the modifier `oracleAvailable` will perform checks to ensure the last update time for oracle prices has occurred after expiry. While this does not prevent expiry from being in the future it requires the oracle to have updated the price in the future which should not occur.

SFA-02	Donation Attacks On Empty AAVE Vaults Can Steal Deposits		
Asset	`vaults/AAVEDNTVault.sol & vaults/AAVESmartTrendVault.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

It is possible for a malicious user to mint, transfer and burn products in the AAVE vaults to manipulate the value of `totalSupply` to 1. Once this has been done, the attacker can transfer a large value of ATokens to the vault directly. New deposits below this higher balance amount will then receive zero product tokens in return for their balance, or potentially have the value of their deposit reduced.

Consider the following scenario assuming an attacker as Alice, and a newly created `AAVEDNTVault` with USDC as the collateral token. Suppose that Alice also controls AliceMM, a market maker, and AliceT, an alternative address.

1. At 07:59:59, Alice mints a product with AliceMM as the market maker, for 10 USDC collateral.
   Alice can zero out the minting fee by minting a product without user collateral, ie. where `params.collateralAtRisk == params.makerCollateral == 10 USDC`. Alice also sets the anchor prices to `0.01` and `0.02`, far below the current oracle prices. This means that AliceMM will be the immediate "winner".
   In `_mint()`, the following values are set:
   - `totalSupply = 1e7 * 1e18 = 1e25`
   - `aTokenShare = 1e25`, which is the number of product tokens Alice receives.
2. Alice transfers 1 product token away from AliceMM to AliceT. Alice MM now has 999,999,999,999,999,999,999,999 product tokens.
3. At 08:00:01, Alice calls `burn()`, which burns her product tokens for no value. AliceMM also calls `burn()`. This account receives the maximum payoff, as calculated by `getMakerPayoff()`. This calculation will be:

   ```
   maxPayoff = amount * collateralAtRiskPercentage / 1e18
   = 999_999_999_999_999_999_999_999_999 * 1e18 / 1e18
   = 999_999_999_999_999_999_999_999
   ```

   On line [**349**], this value is subtracted from `totalSupply`, leaving a `totalSupply` of 1.
   This share of the AToken balance will be transferred to AliceMM, leaving the vault with an AToken balance of 1.
   This will burn all but one of the ATokens on the vault contract.
   The vault now has `totalSupply == 1` and an AToken balance of 1.
4. Alice now donates 1,000 USDC worth of AAVE ATokens direct to the vault.
   So, now, `totalSupply == 1` and the AToken balance is 1e9.
   AliceT controls these 1,000,000,001 ATokens with the single winning product token.
5. Any attempt to mint new products for less than 1e9 USDC now gives zero shares, effectively gifting the collateral to AliceT:

```
New depositor's ATokenShare = totalCollateral * totalSupply / (aTokenBalance - totalCollateral)
= totalCollateral * 1 / (1e9)
= 0 (if totalCollateral < 1e9)
```

Consider a deposit of 1,900 USDC collateral:

```
aTokenShare = 1.9e9 * 1 / (1e9) = 1
```

The new depositor receives 1 share. AliceT has 1 share. These users split the balance, which is now 2.9e9. So AliceT takes 0.45e9 of their ATokens.

This issue is less likely to occur in the smart trend vault, as the fastest expiring product for that vault is 1 day and any deposits during this period would prevent the attack.

Recommendations

The simplest countermeasure to donation attacks is to ensure that the vault is never empty. This can be achieved by minting a product in the `initialize()` function and only redeeming the tokens for this product after the vault is deactivated.

Resolution

The development team have stated their intention to mint and hold positions in the affected vaults before going live, and to check that such an attack has not taken place before putting a vault live.

SFA-03	Incorrect Calculation Of `minterCollateral`		
Asset	`vaults/LeverageDNTVault.sol & vaults/LeverageSmartTrendVault.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

Following discussions with the development team, it was established that the calculation of `minterCollateral` within the function `_mint()` is incorrect. This will in turn produce incorrect calculations of trading fees and of the permitted collateral ratios between the user and the market maker.

This is the original calculation:

LeverageDNTVault.sol

```
uint256 minterCollateral = (totalCollateral - params.makerCollateral) * APR_BASE / (APR_BASE + LEVERAGE_RATIO * APR_BASE +
    ↪  LEVERAGE_RATIO * borrowAPR * (params.expiry - block.timestamp) / SECONDS_IN_YEAR);
```

Each time that a product is minted, the `borrowFee` and `spreadFee` will be incorrectly calculated as these variables are derived from `minterCollateral`.

The correct calculation is derived from the following equation:

LeverageDNTVault.sol

```
// (totalCollateral - makerCollateral) = minterCollateral  + minterCollateral * LEVERAGE_RATIO * borrowAPR / SECONDS_IN_YEAR *
    ↪  (expiry - block.timestamp)
```

The left hand side of this equation is the amount of collateral provided by the user.

The right hand side is a calculation of a simulation of leverage. It consists of a nominal amount of collateral provided by the user (`minterCollateral`) added to the borrow interest on borrowing that amount again `LEVERAGE_RATIO` times for the active duration of the minted product.

From this equation, the correct calculation can be derived:

LeverageDNTVault.sol

```
uint256 minterCollateral = (totalCollateral - params.makerCollateral) * APR_BASE /
(APR_BASE  + LEVERAGE_RATIO * borrowAPR * (params.expiry - block.timestamp) / SECONDS_IN_YEAR);
```

Recommendations

After discussions with the development team, it was established that the calculation of the `minterCollateral` should be revised to:

LeverageDNTVault.sol

```
// (totalCollateral - makerCollateral) = minterCollateral  + minterCollateral * LEVERAGE_RATIO * borrowAPR / SECONDS_IN_YEAR *
    ↪  (expiry - block.timestamp)
uint256 minterCollateral = (totalCollateral - params.makerCollateral) * APR_BASE /
(APR_BASE  + LEVERAGE_RATIO * borrowAPR * (params.expiry - block.timestamp) / SECONDS_IN_YEAR);
```

Resolution

The development team have modified the calculation as described in commit ef1e9b9.

SFA-04	Rounding Issue Causing Dust Amount Being Locked		
Asset	`vaults/LeverageSmartTrendVault.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

A rounding error in calculating the minter payoff would result in collateral dust amount being locked forever in the vault.

The minter payoff has the following expression :

LeverageSmartTrendVault.sol

```
341 payoff = payoffWithFee - fee + (amount * 1e18 - amount * collateralAtRiskPercentage) / 1e18;
```

Reducing `amount * collateralAtRiskPercentage` from `amount * 1e18` and then dividing the result by `1e18` will round down the result by 1 wei. So, for every minter payoff, 1 wei will be locked in the vault.

Recommendations

Replace `(amount * 1e18 - amount * collateralAtRiskPercentage) /1e18` with `amount - (amount * collateralAtRiskPercentage) / 1e18`, similar to the function `LeverageDNTVault.getMinterPayoff()`.

Resolution

The recommendation has been implemented in commit ef1e9b9.

SFA-05	`COLLATERAL` Token Is Not Part of The `digest`		
Asset	`vaults/*`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `COLLATERAL` token is not part of the `digest` that the market maker signs. This means that if the vault is upgraded to a new implementation that allows the admin to change to another collateral token, a user can call `mint()` function using a signature intended for the old token, if it has not expired yet.

This scenario, whilst improbable, could be especially impactful if the new collateral token has a different value than the previous one. For example, Consider an MM signing a transaction for 1e9 USDC and then this being used to trade with 1e9 WBTC.

Recommendations

Include the `COLLATERAL` in the `digest`.

Resolution

The development team acknowledged the issue and stated that significant governance protections would prevent the `COLLATERAL` token of an existing vault from being modified.

SFA-06	Fee Rates Can Change on Active Products		
Asset	`vaults/*`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

When the parties mint a product, they are aware of the fee rate at that point. However, the fee that the minter pays on burning their tokens can be modified between the time when the product is minted and burnt. This could result in a user of the protocol paying significantly different fees from those they had agreed to on minting.

This occurs because the fee paid by the minter is calculated at burning, based on the fee rate at that point in time.

```
function getMinterPayoff(uint256 expiry, uint256[2] memory anchorPrices, uint256 amount) public view returns (uint256 payoff,
    ↪  uint256 fee) {
    uint256 payoffWithFee = STRATEGY.getMinterPayoff(anchorPrices, ORACLE.settlePrices(expiry), amount);
    fee = payoffWithFee * IFeeCollector(feeCollector).settlementFeeRate() / 1e18;
    payoff = payoffWithFee - fee;
}
```

Recommendations

Consider including the settlement fee in the product ID hash. This would lock settlement fee during the mint process.

Resolution

The development team have acknowledged the issue and stated an intention to reconsider the issue in future versions of the protocol.

Under current conditions the settlement fee may only be adjusted in the `FeeCollector` contract via an access controlled function. The access control is limited to the Sofa DAO which has governance and time lock mechanics to reduce the possibility of malicious updates to the fee.

SFA-07	Replay Attack on APR Changes in the Leverage Vaults		
Asset	`vaults/LeverageDNTVault.sol & vaults/LeverageSmartTrendVault.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

It is possible to execute a replay attack on the `mint()` function of the leverage vaults to exploit changes to the APR rates.

If a call to the `mint()` reverts due to the following require statement:

LeverageDNTVault.sol

```
207 require(borrowFee - spreadFee >= params.collateralAtRisk - params.makerCollateral, "Vault: invalid collateral at risk");
```

the Market Maker's signature can still be reused to call the `mint()` function again with the same arguments.

This is because the `digest` that the market maker signs doesn't include `borrowAPR` or `spreadAPR`. Consequently, if the first call to `mint()` fails due to the mentioned `require` statement, and as long as the expiration is not reached and the deadline has not passed, the user can call the `mint()` function again with the same arguments after the APR updates, even though the market maker has not signed to consent to minting a product with the new APRs.

Recommendations

Include both `borrowAPR` and `spreadAPR` in the `digest`.

Resolution

The development team observed that the risk to the market maker's signature being reused, is mitigated by the fact that the APR rates do not affect their exposure.

It is still possible for the user to sign a transaction which could be frontrun by a transaction changing the APR rates, and thus the user could be charged more fees that expected. This risk is managed by the protocol clearly announcing any changes to the APR rates well in advance.

SFA-08	Stale Oracle Prices Are Processed As Recent		
Asset	`oracles/SpotOracle.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The data returned by `PRICEFEED.latestRoundData()` contains information about the staleness of the oracle's price data. However, this is discarded and treated as always fresh.

As the oracle feed is supplying freshness data and the `SpotOracle` has the capability to deal with price delays, it makes sense to integrate this functionality and not allow products to be resolved on stale data.

Recommendations

Consider returning the updated time from `getLatestPrice()` and using this value to calculate the `expiry` value in `settle()` instead of `block.timestamp`. This will result in only updating the oracle's prices up to the freshness level of the price feed.

Resolution

The development team have acknowledged the issue and resolved to monitor for potential developments.

SFA-09	Missed Price Data Is Filled In With Linear Derivations		
Asset	`oracles/HlOracle.sol & oracles/SpotOracle.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

In the `settle()` functions of both oracle files, there is logic which, if a day or more of price data is missing, will fill in that price data with evenly spaced prices between known data points. For example, if the price on Monday is 120 and the price on Thursday is 150, then this code will record that the price on Tuesday was 130 and on Thursday was 140.

This is recorded regardless of what the actual prices on those days were. If the price on Tuesday was 200, this would have a significant impact on the payouts of some Smart Trend vaults, but could be far more impactful for Double No Touch vaults, which could reward significant amounts to the wrong party.

Chainlink oracles provide access to historical data, although it is easier processed off chain.

In the event of a network outage, a DOS attack, a technical error or any other circumstance which prevents the recording of price data, this behaviour could result in users of the protocol receiving the wrong payout amounts.

Recommendations

Given the likely requirement for an off-chain system to restore the correct data, and also the low probability of the oracle missing updates for multiple days, the development team may consider this issue one that can be managed at this stage.

Alternatively, consider removing the linear derived price code and adding a facility for a trusted user to update missing price days.

Resolution

The development team have acknowledged the issue and resolved to monitor for potential developments.

SFA-10	Any Contract In Any Transaction Initiated By Owner Can Spend RCH Before The Start Time		
Asset	`tokenomics/RCH.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Any contract that is interacted with by `owner` can call `RCH` and bypass the check on line [23]. This means that it is quite possible for third parties to interact with the RCH token before `tradingStartTime` .

The line in question uses `tx.origin` :

```
if (block.timestamp < tradingStartTime) {
    require(tx.origin == owner() || _msgSender() == owner(), "RCH: token transfer not allowed before trading starts");
}
```

The first part of the or test only requires that the transaction be initiated by `owner` . This means that any contract in any transaction by `owner` will pass this test.

This issue only applies when `owner` is a EOA. However, if `owner` will be a contract, testing `tx.origin` is redundant.

This issue is also mitigated by the requirement that the contract in question either own RCH tokens or have an allowance to spend RCH tokens. This may significantly limit the scope of the potential problem.

Recommendations

If `owner` is not an EOA, consider removing the first half of the test.

Alternatively, the development team may consider that this issue can be managed by carefully limiting the use of `owner` before `tradingStartTime` . If that is not considered sufficient protection, consider removing the first half of the test.

Resolution

The development team acknowledged the issue and stated that there will be significant precautions undertaken to limit the use of `owner` before the start time, and to monitor transactions to prevent the use of RCH tokens in this period.

SFA-11	Possible Overflow for High Decimal Tokens		
Asset	`vaults/AAVEDNTVault.sol & vaults/AAVESmartTrendVault.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

If the vault accepts as collateral a token with high decimals, there is a possible overflow that would deny a user from minting.

In fact, for high decimal tokens (e.g 24), the `totalSupply` would be of the order of 42 decimals because of the multiplication on line [**212**].

Hence, if the `totalSupply` is as high as `1e11` and so it is represented in solidity as `1e52`, line [**210**] could revert because `totalCollateral * totalSupply` could exceed `max(uint256)`.

Recommendations

Consider monitoring the decimals of collateral tokens and reducing `SHARE_MULTIPLIER` if necessary.

Resolution

The development team have acknowledged the issue and resolved to monitor for potential developments.

SFA-12	Initialisers Not Disabled on Implementation Contracts		
Asset	`vaults/*`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The setup of each vault deploys two contracts, a proxy contract and an implementation. The implementation contract does not disable the initialisers allowing it to be initialised by third parties.

The main danger of uninitialised contracts is for a third party to initialise the contract and make a delegate call which self-destructs, thereby deleting the proxy contract.

The threat from uninitialised implementation contracts in this case is limited as there are no further delegate calls. Nevertheless, this it may reflect poorly on the protocol to have implementation contracts initialised by a third party attempting to fraud users.

Recommendations

Add the following to the constructor code, or alternatively initialise all implementation contracts on deployment:

```
constructor() {
    _disableInitializers();
}
```

Resolution

The development team have acknowledged the issue and resolved to monitor for potential developments.

SFA-13	Collateral Can Be Locked For Extremely Long Periods		
Asset	`vaults/*`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

There is no limit to the expiry time of the minted products. Particularly in the case of smart trend vaults, this could result in collateral being accidentally locked for excessive periods.

Furthermore, there are a number of unbounded loops related to terms. Extensive term lengths may result in breaching block gas limits when iterating through terms in `HlOracle`. It would no longer be possible to close the position if this is the case.

Similarly, if missed days is too high then oracles may not be able to call `settle()` if it breaches block gas limits.

- `HlOracle.getHlPrices()`
- `HlOracle.settle()`
- `SpotOracle.settle()`

Recommendations

It is recommended to add a maximum allowed `term` for newly minted products.

Consider a fallback method in-case an oracle has a significant number of missed days and is unable to call `settle()`.

Resolution

The development team have acknowledged the issue and resolved to monitor for potential developments.

SFA-14	Defunct Hardhat Import In Vaults
Asset	`vaults/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The vaults contain the following line:

```
import "hardhat/console.sol";
```

This module is used in development, but not production.

Recommendations

Remove the Hardhat import.

Resolution

The Hardhat imports have been removed.

SFA-15	Bitmap Is Unnecessary
Asset	`tokenomics/MerkleAirdrop.sol & utils/SignatureBitMap.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

`MerkleAirdrop.sol` and `SignatureBitMap.sol` are using two `uint256` values in a mapping to store a series of negative/positive values in the range of a single `uint256`. The same information can be stored in a mapping of `uint256` to `bool` values, which is simpler and also slightly more gas efficient.

`MerkleAirdrop.sol` is doing this in the variable `claimedBitMap` which could be of type `mapping(address => mapping(uint256 => bool))` and simply key its entries by the parameter `index`.

`SignatureBitMap.sol` is out of scope, but is mentioned as it may be of potential interest.

Recommendations

Consider modifying the file to the suggested structure.

Resolution

The development team have acknowledged the issue and resolved to monitor for potential developments.

SFA-16	Minter Or Maker May Contribute Zero Collateral
Asset	`vaults/*`
Status	**Closed:** See Resolution
Rating	Informational

Description

Certain undesirable edge cases exists around providing zero collateral.

The following cases may arise:

- minter collateral may be zero;
- maker collateral may be zero;
- both minter and maker collateral may be zero resulting in zero total collateral.

For the case where minter collateral is zero then no fees will be generated by the protocol as fees are calculated as a percentage of minter collateral.

In the case where total collateral is zero then it will not be possible to burn the product as the balance will be zero preventing burning.

It is worth noting that the code will handle the cases where one party contributes the entire collateral. The development team may deem this a worthwhile use case.

Recommendations

Consider reverting in the cases where either or both parties provide zero collateral.

Resolution

The development team have acknowledged the issue and resolved to monitor for potential developments.

SFA-17	Burn Batch Allows Duplicates
Asset	`vaults/*`
Status	**Closed:** See Resolution
Rating	Informational

Description

Each vault allows duplicate products to appear in the array `products` in the function `burnBatch()`.

The following function is taken from `LeverageDNTVault.sol` however, similar code is present in all vaults.

There are no checks in `_burnBatch()` to ensure duplicate products are not sent:

LeverageDNTVault.sol

```
283 function _burnBatch(Product[] calldata products) internal nonReentrant returns (uint256 totalPayoff) {
    uint256[] memory productIds = new uint256[](products.length);
285 uint256[] memory amounts = new uint256[](products.length);
    uint256[] memory payoffs = new uint256[](products.length);
287 uint256 settlementFee;
    for (uint256 i = 0; i < products.length; i++) {
289     Product memory product = products[i];

291     (uint256 latestTerm, uint256 latestExpiry, bool _isBurnable) = isBurnable(product.term, product.expiry,
            ↪ product.anchorPrices);
        require(_isBurnable, "Vault: not burnable");
293
        // check if settled
295     require(ORACLE.settlePrices(latestExpiry, 1) > 0, "Vault: not settled");

297     uint256 productId = getProductId(product.term, product.expiry, product.anchorPrices, product.collateralAtRiskPercentage,
            ↪ product.isMaker);
        uint256 amount = balanceOf(_msgSender(), productId);
299     require(amount > 0, "Vault: zero amount");

301     // calculate payoff by strategy
        if (product.isMaker == 1) {
303         payoffs[i] = getMakerPayoff(latestTerm, latestExpiry, product.anchorPrices, product.collateralAtRiskPercentage,
                ↪ amount);
        } else {
305         uint256 fee;
            (payoffs[i], fee) = getMinterPayoff(latestTerm, latestExpiry, product.anchorPrices, product.collateralAtRiskPercentage,
                ↪ amount);
307         if (fee > 0) {
                settlementFee += fee;
309         }
        }
311     if (payoffs[i] > 0) {
            totalPayoff += payoffs[i];
313     }

315     productIds[i] = productId;
        amounts[i] = amount;
317 }
    if (settlementFee > 0) {
319     totalFee += settlementFee;
    }
321 // burn product
    _burnBatch(_msgSender(), productIds, amounts); // @audit send balance deducted here
323 emit BatchBurned(_msgSender(), productIds, amounts, payoffs);
}
```

The impact is rated as informational severity as the internal call to `ERC1155Upgradeable._burnBatch()` will perform balance checks. Since each product will burn the entire balance if a product is attempted to be burnt twice the duplicate occurrence will revert due to insufficient balance.

Recommendations

Consider ensuring that product IDs are unique to prevent a batch function from containing duplicates. This may be achieved cheaply by performing sorting off-chain and requiring each `productId` to be strictly greater than the last.

Resolution

The development team have acknowledged the issue and resolved to monitor for potential developments.

SFA-18	Reentrancy In AAVE Vaults
Asset	`AAVEDNTVault.sol & AAVESmartTrendVault.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

There is a reentrancy vector within the AAVE vaults when `supply()` is called.

The issue exists within the vault's `mint()` function, as the call to `POOL.supply()` will perform a transfer of the underlying token. If the underlying token is reenterable then it will be possible to reenter into `mint()`.

AAVEDNTVault.sol

```
206 uint256 aTokenShare;
    POOL.supply(address(COLLATERAL), totalCollateral, address(this), REFERRAL_CODE); // @audit potentially reenterable
208 uint256 aTokenBalance = ATOKEN.balanceOf(address(this));
    if (totalSupply > 0) {
210     aTokenShare = totalCollateral * totalSupply / (aTokenBalance - totalCollateral);
    } else {
212     aTokenShare = totalCollateral * SHARE_MULTIPLIER;
    }
214 totalSupply += aTokenShare;
```

The issue is raised as informational as AAVE currently does not support tokens which relinquish control of execution. Furthermore, the external transfer would occur before the aToken balance is modified, removing the benefit to an attacker of performing this attack.

Recommendations

It is recommended to add `nonRenterant` modifier to the `_mint()` function to prevent potential reentrancy.

Resolution

This issue has been fixed in commit ef1e9b9. The function `_mint()` of `AAVESmartTrendVault` and `AAVEDNTVault` now has the `nonRenterant` modifier.

SFA-19	Miscellaneous General Comments
Asset	All contracts
Status	**Closed:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Use Readable Format**
   Related Asset(s): tokenomics/RCH.sol
   Use `37_000_000` instead of `37000000` for the constant variable `MAX_SUPPLY`.
   Consider the suggested modification.
2. **Gas: Copy Variable Before Check**
   Related Asset(s): vaults/*
   Consider the following code:

   ```
   function harvest() external {
       require(totalFee > 0, "Vault: zero fee");
       uint256 fee = totalFee;
       totalFee = 0;
   ```

   It would save on one `SLOAD` to check `fee`, like so:

   ```
   function harvest() external {
       uint256 fee = totalFee;
       require(fee > 0, "Vault: zero fee");
       totalFee = 0;
   ```

   Consider the suggested modification.
3. **Unnecessary `if` Test**
   Related Asset(s): oracles/HlOracle.sol
   Consider the following test from lines [**91-93**]:

   ```
   if (settlePrices[expiry][1] > 0 && settlePrices[expiry][1] > hlPrices[1]) {
       hlPrices[1] = settlePrices[expiry][1];
   }
   ```

   As `hlPrices[1]` starts at zero and can only get bigger, the first condition in this test is unnecessary.
   Consider removing the first condition in the test.
4. **Modifiable State Variables Styled As Constants**
   Related Asset(s): vaults/*
   The following variables are storage variables:

   ```
   IWETH public WETH;
   IPermit2 public PERMIT2;
   IDNTStrategy public STRATEGY;
   IERC20Metadata public COLLATERAL;
   IHlOracle public ORACLE;
   ```

However, they are written in block capitals, which is usually used to signify a constant.

It is acknowledged that these variables are treated as constants but they are modifiable in storage, which could have future security implications.

Consider turning these variables into constants, or `immutable` variables that are set in the constructor. Alternatively, consider changing them to the standard "camel case".

5. **Different Functions With Identical Names**

   Related Asset(s): vaults/*

   The following functions are defined in the vault contracts, but also have related functions in Openzeppelin's `ERC1155Upgradeable.sol` which are also called, often within functions of the same name:

   - `_mint()`
   - `_mintBatch()`
   - `_burn()`
   - `_burnBatch()`

   It is acknowledged that the function signatures are different, as the parameters are not the same. However, the code would be clearer if these functions did not share identical names.

   Consider changing the function names in the vault, even minimally. For example `_vaultMint()`, `_productMint()`, `_sofaMint()` or even `_vMint()`.

6. **DNT Vault Anchor Prices Can Be Impossible To Avoid**

   Related Asset(s): vaults/AAVEDNTVault.sol & vaults/DNTVault.sol & vaults/LeverageDNTVault.sol

   The check for DNT vault anchor prices is:

   ```
   require(params.anchorPrices[0] < params.anchorPrices[1], "Vault: invalid strike prices");
   ```

   Note that, even if `params.anchorPrices[0]+1 == params.anchorPrices[1]`, it is still inevitable that the prices will be touched on minting.

   Regardless of the gap between the anchor prices, it is also possible to mint when the current price is touching one of them.

   This may be an acceptable liability, and one which the user is considered responsible for. Nevertheless, creating products that instantly expire has questionable legitimate utility but is a potentially useful tool when crafting attacks.

   Consider specifying a minimum gap between anchor prices for DNT vaults.
   Consider checking that the anchor prices are not already touched when minting products for DNT vaults.

7. **Missing Sanity Check**

   Related Asset(s): oracles/HlOracle.sol

   There is no on-chain check that in the `settle()` function that `currentPrices[0] < currentPrices[1]`.

   Consider adding this check to ensure that the prices are in ascending order.

8. **Missing Events on Configuration Variable Changes**

   Related Asset(s): vaults/LeverageSmartTrendVault.sol & vaults/LeverageDNTVault.sol

   Consider emitting events when the `updateBorrowAPR` and `updateSpreadAPR` functions successfully update their variables.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The development team's responses to the raised issues above are as follows.

1. **Use Readable Format**
   Related Asset(s): tokenomics/RCH.sol
   This change was implemented as can be seen in described in commit ef1e9b9:
2. **Gas: Copy Variable Before Check**
   This change was implemented as can be seen in described in commit ef1e9b9:
3. **Unnecessary `if` Test**
   This change was implemented as can be seen in described in commit ef1e9b9:
4. **Modifiable State Variables Styled As Constants**
   This change was implemented as can be seen in described in commit ef1e9b9:
5. **Different Functions With Identical Names**
   The development team have acknowledged the issue and resolved to monitor for potential developments.
6. **DNT Vault Anchor Prices Can Be Impossible To Avoid**
   The development team have acknowledged the issue and resolved to monitor for potential developments.
7. **Missing Sanity Check**
   This change was implemented as can be seen in described in commit ef1e9b9:
8. **Missing Events on Configuration Variable Changes**
   The development team have acknowledged the issue and resolved to monitor for potential developments.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `Forge` framework was used to perform these tests and the output is given below.

```
Ran 1 test for test/SpotOracleTest.t.sol:SpotOracleTest
[PASS] test_settle() (gas: 90310)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 7.49ms (118.31µs CPU time)

Ran 5 tests for test/HlOracleTest.t.sol:HlOracleTest
[PASS] test_checkUpkeep_case1() (gas: 39732)
[PASS] test_checkUpkeep_case2() (gas: 154236)
[PASS] test_checkUpkeep_case3() (gas: 38757)
[PASS] test_getHlPrices() (gas: 262042)
[PASS] test_performUpkeep() (gas: 151829)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 11.44ms (2.03ms CPU time)

Ran 7 tests for test/DNTVaultTest.t.sol:DNTVaultTest
[PASS] test_burnBatch() (gas: 487235)
[PASS] test_burn_case1() (gas: 387751)
[PASS] test_burn_case2() (gas: 383060)
[PASS] test_harvest() (gas: 233548)
[PASS] test_initialize() (gas: 40213)
[PASS] test_mint() (gas: 233631)
[PASS] test_mintBatch() (gas: 362709)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 14.00ms (26.12ms CPU time)

Ran 2 tests for test/MerkleAirdropTest.t.sol:MerkleAirdropTest
[PASS] test_claim() (gas: 600937)
[PASS] test_claimMultiple() (gas: 1114824)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 25.72ms (1.42ms CPU time)

Ran 12 tests for test/AAVESmartTrendVault.t.sol:AAVESmartTrendVaultTest
[PASS] test_burnBatch() (gas: 611924)
[PASS] test_burnBatchErrors() (gas: 617227)
[PASS] test_burnErrors() (gas: 462768)
[PASS] test_burnVanilla() (gas: 519429)
[PASS] test_ethBurn() (gas: 597607)
[PASS] test_ethBurnBatch() (gas: 677702)
[PASS] test_harvest() (gas: 379255)
[PASS] test_initialize() (gas: 105358)
[PASS] test_mint() (gas: 325014)
[PASS] test_mintErrors() (gas: 564481)
[PASS] test_mintEth() (gas: 342919)
[PASS] test_onlyETHVault() (gas: 20543)
Suite result: ok. 12 passed; 0 failed; 0 skipped; finished in 25.73ms (52.80ms CPU time)

Ran 5 tests for test/LeverageDNTVaultTest.t.sol:LeverageDNTVaultTest
[PASS] test_burnBatch() (gas: 493487)
[PASS] test_burn_case1() (gas: 396404)
[PASS] test_burn_case2() (gas: 395329)
[PASS] test_initialize() (gas: 48911)
[PASS] test_mint() (gas: 242295)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 23.58ms (10.42ms CPU time)

Ran 3 tests for test/RCHTest.t.sol:RCHTest
[PASS] test_mint() (gas: 83334)
[PASS] test_transfer_after_tradingTime() (gas: 116699)
[PASS] test_transfer_before_tradingTime() (gas: 118169)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 15.09ms (719.38µs CPU time)

Ran 7 tests for test/SmartTrendVaultTest.t.sol:SmartTrendVaultTest
[PASS] test_burnBatch() (gas: 419844)
[PASS] test_burn_case1() (gas: 315377)
[PASS] test_burn_case2() (gas: 317168)
[PASS] test_harvest() (gas: 232842)
[PASS] test_initialize() (gas: 40125)
```

```
[PASS] test_mint() (gas: 234065)
[PASS] test_mintBatch() (gas: 359754)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 12.73ms (16.82ms CPU time)

Ran 5 tests for test/LeverageSmartTrendVaultTest.t .sol:LeverageSmartTrendVaultTest
[PASS] test_burnBatch() (gas: 430628)
[PASS] test_burn_case1() (gas: 343338)
[PASS] test_burn_case2() (gas: 328872)
[PASS] test_initialize() (gas: 48955)
[PASS] test_mint() (gas: 240216)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 23.17ms (10.63ms CPU time)

Ran 14 tests for test/AAVEDNTVaultTest.t.sol:AAVEDNTVaultTest
[PASS] test_burn() (gas: 2058650)
[PASS] test_burnBatch() (gas: 2167595)
[PASS] test_burnBatchErrors() (gas: 2119134)
[PASS] test_burnErrors() (gas: 2078737)
[PASS] test_donationAttack_vuln() (gas: 2116492)
[PASS] test_ethBurn() (gas: 2142779)
[PASS] test_ethBurnBatch() (gas: 2254815)
[PASS] test_harvest() (gas: 380182)
[PASS] test_initialize() (gas: 105381)
[PASS] test_mint() (gas: 327086)
[PASS] test_mintErrors() (gas: 700581)
[PASS] test_mintEth() (gas: 344925)
[PASS] test_mint_overflow_vuln() (gas: 6015661)
[PASS] test_onlyETHVault() (gas: 20608)
Suite result: ok. 14 passed; 0 failed; 0 skipped; finished in 21.03ms (60.43ms CPU time)

Ran 10 test suites in 227.40ms (180.00ms CPU time): 61 tests passed, 0 failed, 0 skipped (61 total tests)
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

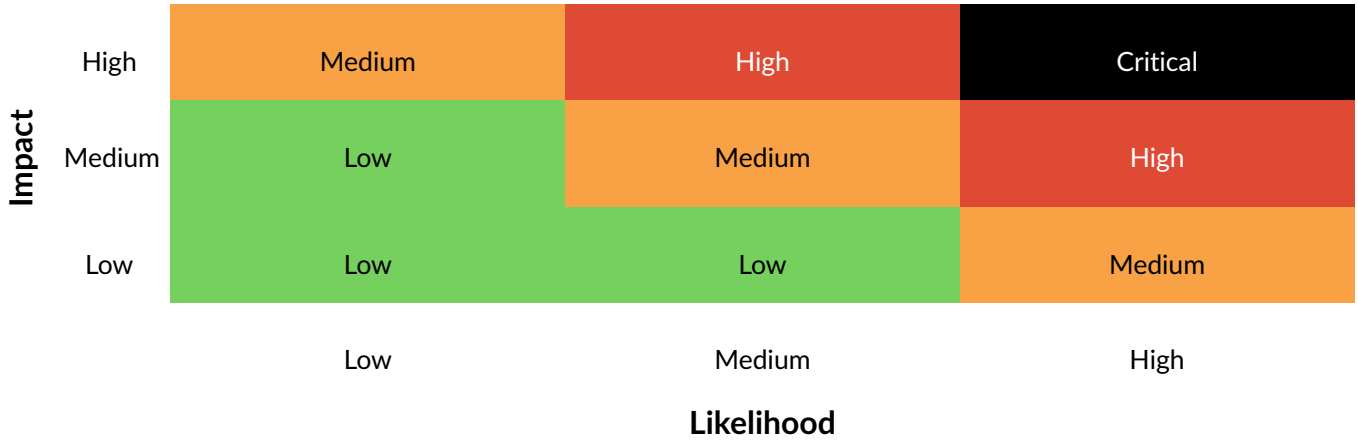

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].