code4rena

SOFA.org

Smart Contract Security Assessment

Version 1.0

Audit dates: May 15 — May 27, 2024

Audited by: elhajin
peakbolt
carrotsmuggler

Contents

1. Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 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

2. Executive Summary

2.1 About SOFA.org

SOFA.org is a decentralized, non-profit organization focused on advancing the DeFi ecosystem. Our modus operandi is to promote the highest DeFi standards, support high-quality projects, and promote adoption of blockchain technologies across mainstream finance.

2.2 Scope

Repository	sofa-org/sofa-protocol/
Commit Hash	2260263d88c83791acd07ed07ce4da898c24dcd6

2.3 Audit Timeline

DATE	EVENT
May 15, 2024	Audit start
May 27, 2024	Audit end
Nov 04, 2024	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	3
Medium Risk	6
Low Risk	2
Informational	1
Total Issues	12

3. Findings Summary

ID	DESCRIPTION	STATUS
H-1	Signatures from makers can be re-used due to malleability	Resolved

H-2	Incorrect Expiry Used in `getHlPrices` Function When Burning A Product `ID` Allow Double Withdrawal Exploit in DNT Vaults	Resolved
H-3	Aave Vaults are vulnerable to share inflation attacks	Resolved
M-1	Chainlink oracle can record stale prices	Acknowledged
M-2	Potential Underflow in `getMinterPayoff` Function Causing Minter Asset Loss and Stuck Funds	Resolved
M-3	Failure to set `settlePrices[]` will prevent redemption of product	Resolved
M-4	Ineffective swap deadline for `swapRCH()`	Resolved
M-5	Use of ETH `transfer()` could cause burning of ETH products to fail	Resolved
M-6	Aave Vaults lack support for claiming of reward tokens	Acknowledged
L-1	Incorrect Assumptions Could Lead to Stuck `ETH` in the Contract	Acknowledged
L-2	Lack of Enforcement on `spreadAPR` Exceeding `borrowAPR` in `LeverageSmartTrendVault` Contract	Acknowledged
I-1	Consider adding `merkleRoot != 0` check	Resolved

4. Findings

4.1 High Risk

A total of 3 high risk findings were identified.

[H-1] Signatures from makers can be re-used due to malleability

Severity: High	Status: Resolved

Context :

- AAVEDNTVault.sol#L195-L197
- AAVESmartTrendVault.sol#L194-L196
- DNTVault.sol#L175-L177
- LeverageDNTVault.sol#L190-L192
- LeverageSmartTrendVault.sol#L188-L190
- SmartTrendVault.sol#L173-L175

Description: Maker signatures used are malleable. The contract uses `ecrecover` to recover the signer of he signatures, and then stores the hash of `v,r,s` to denote a used signature.

The issue is that if `(v,r,s)` is a valid signature, so is `(v,r, -s mod n)`. This is a well known feature of the elliptic curve cryptography. The hash of this manipulated signature is different from the original one, so it allows the same signature to be used twice.

More details about the signature malleability can be found in the following post: https://medium.com/draftkings-engineering/signature-malleability-7a804429b14a

This vulnerability allows maker signatures to be used twice. So makers can be signed up to be exposed to positions twice the size of the position they were anticipating.

Recommendation :

Either use the latest openzeppelin ECDSA library, or implement a nonce system for maker signatures to prevent re-use. Openzeppelin ECDSA library already makes sure that the passed `s` value is only in the lower range.

SOFA.org: Fixed in commit #e4d8. It is still considered acceptable regardless of the fix, as the market makers allow users to re-use the signature twice.

Zenith: Verified to have adopted OpenZeppelin ECDSA library for signature verification to prevent signature re-use.

[H-2] Incorrect Expiry Used in `getHlPrices` Function When Burning A Product `ID` Allow Double Withdrawal Exploit in DNT Vaults

Severity: High   Status: Resolved

Context :

- DNTVault
- AAVEDNT
- leverageDNT

Description:

- The `getHlPrices` function may include prices after the intended **expiry** time when the `burn()` function is called in `DNTVault` contract after the **expiry** date (more than 1 day atleast). This can lead to incorrect settlement calculations and allow an attacker to double withdraw his bet.
- the expiry passed to `getHlPrices()` might be later than the product's actual **expiry**, that's because we always pass `latestExpiry` , which is basically current time rounded to `getMakerPayoff` and `getMinterPayoff` functions :

```
      // some code ...
   >>   uint256 latestExpiry = (block.timestamp - 28800) / 86400 * 86400
+ 28800;

      if (isMaker == 1) {
   >>     payoff = getMakerPayoff(latestTerm, latestExpiry, anchorPrices,
amount);
      } else {

   >>     (payoff, settlementFee) = getMinterPayoff(latestTerm,
latestExpiry, anchorPrices, amount);

      }
```

- Note that the *expiry* passed to `getHlPrices()` might be later than the product's actual *expiry*, causing the function to iterate over and include prices that should not be considered and exclude some that should be included. This results in settling the payoff incorrectly, as prices beyond the product's expiry can skew the high/low price determination.

```
  >>   payoff = STRATEGY.getMakerPayoff(anchorPrices,
ORACLE.getHlPrices(term, expiry), amount);
```

- since the minter and the maker call this function separately, they might use different `latestExpiry` values if they burn their tokens at different times. This discrepancy can lead to inconsistent settlement values for the two parties.

Attack Example:

- An attacker could take advantage of that by opening a position with themselves and monitoring the prices after the expiry. When prices are favorable to the minter, the attacker burns the minter's token id.
- Conversely, when prices favor the maker, the attacker closes the maker's position. This allows the attacker to double withdraw their bet, effectively draining the protocol with a large position.

`Note` This issue is present in all **DNT Vaults**,

Recommendation:

- For `DNT` vaults, always ensure that the time used for settlement is equal to or less than the product's real **expiry**. Implement checks to prevent using a later expiry time :

```
function _burn(uint256 term, uint256 expiry, uint256[2] memory
anchorPrices, uint256 isMaker) internal nonReentrant returns (uint256
payoff) {
       uint256 productId = getProductId(term, expiry, anchorPrices,
isMaker);

       (uint256 latestTerm, bool _isBurnable) = isBurnable(term, expiry,
anchorPrices);
       require(_isBurnable, "Vault: not burnable");

       // check if settled
-         uint256 latestExpiry = (block.timestamp - 28800) / 86400 * 86400
+ 28800;
+         uint256 current = (block.timestamp - 28800) / 86400 * 86400 +
28800;
+         uint256 latestExpiry = current > expiry ? expiry : current

       require(ORACLE.settlePrices(latestExpiry, 1) > 0, "Vault: not
settled");
       // more code ...
   }
```

```
function _burnBatch(Product[] calldata products) internal nonReentrant
returns (uint256 totalPayoff) {
          //some code ..
       for (uint256 i = 0; i < products.length; i++) {
           // check if settled
-            uint256 latestExpiry = (block.timestamp - 28800) / 86400 *
86400 + 28800;
+            Product memory product = products[i];
+            uint256 current = (block.timestamp - 28800) / 86400 * 86400
+ 28800;
+            uint256 latestExpiry = current > product.expiry ?
product.expiry  : current
+            require(ORACLE.settlePrices(latestExpiry, 1) > 0, "Vault:
not settled");

-            Product memory product = products[i];
         }
           // more code ....
       }
```

SOFA.org: Fixed in commit #a83b

Zenith: Verified - The DNT vaults now ensure that settlement calculation will not include prices after product expiry.

[H-3] Aave Vaults are vulnerable to share inflation attacks

Severity: High	**Status:** Resolved

Context:

- AAVESmartTrendVault.sol#L226-L227
- AAVEDNTVault.sol#L228-L229

Description: In the Aave vaults, `SHARE_MULTIPLIER` is used to prevent share inflation attack by increasing share precision. This is to prevent attacks that cause an user's share to be rounded down to zero or a small number, thus receiving incorrect share.

However, the `_mint()` actually reduces the share precision again with a division by `SHARE_MULTIPLIER`, causing user to be minted incorrect shares amount when there is an share inflation attack.

```
        _mint(_msgSender(), productId, aTokenShare / SHARE_MULTIPLIER,
"");
        _mint(params.maker, makerProductId, aTokenShare /
SHARE_MULTIPLIER, "");
```

Recommendation: Remove the division by `SHARE_MULTIPLIER` so that the minted share amount are in `SHARE_MULTIPLIER` precision. Note that this will also require changes to other share computation (e.g. in burn(), harvest()) that had assumed collateral precision for shares amount. And `decimals()` should reflect the increased precision as well.

SOFA.org: Fixed in commit #7aab

Zenith: Verified, the share precision is now correctly increased based on `SHARE_MULTIPLIER` during mint and other share computations.

4.2 Medium Risk

A total of 6 medium risk findings were identified.

[M-1] Chainlink oracle can record stale prices

Severity: Medium	Status: Acknowledged

Context :

- SpotOracle.sol#L20-L40

Description : The `SpotOracle` uses chainlink price feeds to record spot prices of assets. This is then used to close valut positions of smart vaults later down the line. The price is read off the `latestRoundData` function.

```
(
            /* uint80 roundID */,
            int price,
            /*uint startedAt*/,
            /*uint timeStamp*/,
            /*uint80 answeredInRound*/
        ) = PRICEFEED.latestRoundData();
        require(price > 0, "Oracle: invalid price");
```

This is then stored in the oracle.

```
settlePrices[expiry] = uint256(getLatestPrice());
```

The issue is that chainlink recommends certain checks before using their price feed. The most important one is that of staleness. Prices on chainlink contracts can be outdated due to issues with the oracle network, or high network congestion. To prevent using outdated/stale prices, the recommended fix is to have a hardcoded time threshold and to check the time since the `timeStamp` of the answer against this threshold. If found to be higher, the price can be categorized as stale.

Using stale prices can lead to a number of issues. Minter or maker position can be settled at incorrect prices and can be denied payouts.

Recommendation: We recommend adding a `STALENESS_THRESHOLD` value in the contract. Generally, this can be set to 1.1-2x the heartbeat value of the pricefeed. Then, record the timestamp of the data and check the time against this threshold.

```
require((block.timestamp - timeStamp) < STALENESS_THRESHOLD);
```

SOFA.org: Acknowledged. Our rule is to fetch the Chainlink price every day at 8 AM UTC for settlement. It doesn't matter much if the Chainlink price is not updated exactly on time. We cannot fetch the price again a few minutes later just because the 8 AM price was not updated within the STALENESS_THRESHOLD. Doing so might result in fetching a price updated after 8 AM, which would still be problematic.

[M-2] Potential Underflow in `getMinterPayoff` Function Causing Minter Asset Loss and Stuck Funds

Severity: Medium	Status: Resolved

Context:

- AAVEDNTVault.sol#L205
- AAVESmartTrendVault.sol#L204
- LeverageDNTVault.sol#L212-L213
- LeverageSmartTrendVault.sol#L206-L207

Description:

- The `collateralAtRiskPercentage` can exceed `1e18` in certain cases, leading to potential underflow issues in the `getMinterPayoff` function. This occurs when `params.collateralAtRisk` is greater than `(totalCollateral - tradingFee)`, which is possible since it can be equal to `totalCollateral` (or slightly less).

```
require(params.collateralAtRisk <= totalCollateral, "Vault: invalid
collateral");
```

- The current check allows `params.collateralAtRisk` to be equal to `totalCollateral`.
- When calculating `collateralAtRiskPercentage`, subtracting the `tradingFee` from the denominator can make `collateralAtRiskPercentage` exceed `1e18`.

```
 uint256 collateralAtRiskPercentage = params.collateralAtRisk * 1e18 /
(totalCollateral - tradingFee);
```

- In this case if the minter wins, they cannot withdraw their assets due to an underflow when they try to burn thier product ID, since `amount * 1e18` will always be less than `amount * collateralAtRiskPercentage` here :

```
 payoffShare = payoffShareWithFee - fee + (amount * 1e18 - amount *
collateralAtRiskPercentage) / 1e18;
```

- This lead to a loss of assets for the minter and stuck funds in the contract

`Note` This issue is present in all vault mentioned in Context section.

Recommendation:

- Implement the check to prevent `params.collateralAtRisk` from exceeding `(totalCollateral - tradingFee)`.

SOFA.org: Fixed in commit #6fd7

Zenith: Verified.

[M-3] Failure to set `settlePrices[]` will prevent redemption of product

Severity: Medium	Status: Resolved

Context:

- SpotOracle.sol#L20-L26
- HlOracle.sol#L47-L74

Description: Both `SpotOracle` and `HlOracle` are responsible for recording the `settlePrices[]` via `settle()`.

```
    function settle() public {
        uint256 expiry = block.timestamp - block.timestamp % 86400 +
28800;
        require(settlePrices[expiry] == 0, "Oracle: already settled");
        settlePrices[expiry] = uint256(getLatestPrice());

        emit Settled(expiry, settlePrices[expiry]);
    }
```

However, if `settle()` is not executed on the day of settlement, the `settlePrices[expiry]` will remains zero, causing all expiring product to be un-burnable/un-redeemable for SmartTrend vaults as they require a non-zero settle price. And in case of DNT vaults, the issue will cause an incorrect payout as maker will get 100% of it.

This issue could occur for L2 like Arbitrum, where the sequencer downtime will prevent any transaction to be included in the block. The impact is high though probability is low as it requires sequencer to be down for >= 24hrs.

Recommendation: It is recommended to set a `latestExpiryUpdated` variable to track the last day it get updated. Upon updating the next expiry prices, verify if any days were skipped. For any missed days (expiry), assign prices based on the average of the `latestExpiryUpdated` and the current prices.

Customer: Fixed in commit #f9109f

Zenith: Verified

[M-4] Ineffective swap deadline for `swapRCH()`

Severity: Medium	Status: Resolved

Context:

- FeeCollector.sol#L40-L77

Description: `swapRCH()` uses `block.timestamp`, which renders the swap deadline ineffective. That is because the value of `block.timestamp` is only determined during execution of the tx, causing the deadline check to always pass.

```
    function swapRCH(
        address token,
        uint256 minPrice,
        address[] calldata path
    ) external onlyOwner {
        // last element of path should be rch
        require(path.length <= 4, "Collector: path too long");
        require(path[path.length - 1] == rch, "Collector: invalid path");

        uint256 amountIn = IERC20(token).balanceOf(address(this));
        IUniswapV2Router(routerV2).swapExactTokensForTokens(
            amountIn,
            amountIn * minPrice / 1e18,
            path,
            address(this),
            block.timestamp + 10 minutes
        );
    }
```

Recommendation: This can be fixed by passing in the swap deadline as a parameter in `swapRCH()` for both routerV2 and routerV3 to ensure an absolute deadline.

SOFA.org: Fixed in commit #d874

Zenith: Verified

[M-5] Use of ETH `transfer()` could cause burning of ETH products to fail

Severity: Medium	Status: Resolved

Context:

- AAVEDNTVault.sol#L247
- AAVEDNTVault.sol#L300
- AAVESmartTrendVault.sol#L244
- AAVESmartTrendVault.sol#L294
- DNTVault.sol#L320
- DNTVault.sol#L368
- LeverageDNTVault.sol#L235
- LeverageDNTVault.sol#L280
- LeverageSmartTrendVault.sol#L230
- LeverageSmartTrendVault.sol#L270
- SmartTrendVault.sol#L310
- SmartTrendVault.sol#L354

Description: The vault uses `transfer()` for sending ETH to the caller after determining the payout amount.

```
                payable(_msgSender()).transfer(payoff);
```

However, use of `transfer()` has the risk that it will not work if the gas cost increases/decrease. The assumption that gas cost stays fixed no longer holds true since Istanbul hard fork. see https://swcregistry.io/docs/SWC-134/. When that happens, it will prevent the caller from receiving ETH payout.

In the past `transfer()` was recommended to prevent re-entrancy attacks (i.e. via receive/fallback) as it limits forwarded gas to 2300. Now the security recommendation to prevent re-entrancy is to ensure Check-Effects-Interactions Pattern or use a Reentrancy guard.

Recommendation: Recommend to change to the following:

```
        (bool success, ) = _msgSender().call{value: payoff, gas: 100_000}
("");
        require(success, "Failed to send ETH");
```

SOFA.org: Fixed in commit #da49

Zenith: Verified.

[M-6] Aave Vaults lack support for claiming of reward tokens

Severity: Medium	Status: Acknowledged

Context:

- AAVEDNTVault.sol#L235-L249
- AAVESmartTrendVault.sol#L233-L246

Description: The Aave vaults are designed to supply the collaterals into Aave Pools to receive interest-bearing aTokens. When redeeming the product on expiry using `burn()/ethBurn()`, the users will be able to receive additional returns from the interests earned in Aave Pools.

However, the Aave Vaults do not support claiming of reward tokens, which are emitted to holders of certain aTokens. For example, aTokens like aUSDC, has 1% APY ARB reward on Arbitrum (https://app.aave.com/markets/?marketName=proto_arbitrum_v3). That means these reward tokens will be lost as they cannot be claimed.

```
    function burn(uint256 term, uint256 expiry, uint256[2] calldata
anchorPrices, uint256 collateralAtRiskPercentage, uint256 isMaker)
external {
        uint256 payoff = _burn(term, expiry, anchorPrices,
collateralAtRiskPercentage, isMaker);
        if (payoff > 0) {
            require(POOL.withdraw(address(COLLATERAL), payoff,
_msgSender()) > 0, "Vault: withdraw failed");
        }
    }

    function ethBurn(uint256 term, uint256 expiry, uint256[2] calldata
anchorPrices, uint256 collateralAtRiskPercentage, uint256 isMaker)
external onlyETHVault {
        uint256 payoff = _burn(term, expiry, anchorPrices,
collateralAtRiskPercentage, isMaker);
        if (payoff > 0) {
            require(POOL.withdraw(address(COLLATERAL), payoff,
address(this)) > 0, "Vault: withdraw failed");
            WETH.withdraw(payoff);
            payable(_msgSender()).transfer(payoff);
        }
    }
```

Recommendation: For applicable aTokens, implement the reward claim mechanism so that users can get their share amount of the reward before withdrawing from the Aave pool. Refer to https://docs.aave.com/developers/whats-new/multiple-rewards-and-claim.

SOFA.org: Acknowledged.

1. Supporting reward claiming and distribution will significantly increase the complexity of the contract.
2. We currently have no plans to support Arbitrum USDC, and there are no rewards for other tokens at this time.

4.3 Low Risk

A total of 2 low risk findings were identified.

[L-1] Incorrect Assumptions Could Lead to Stuck `ETH` in the Contract

Severity: Low	Status: Acknowledged

Context:

- DNTVault.sol#L72-L73

Description:

- The comment above the `receive` function incorrectly assumes that depositing `ETH` will help with withdrawals if the balance is insufficient. In reality, `ETH` should be wrapped to `WETH` since the vault settles with `WETH`. Additionally, there is no direct way to withdraw `ETH` from the contract, making deposited `ETH` inaccessible unless upgraded.
- The withdrawal process relies on `WETH` availability. If there isn't enough `WETH`, the transaction will revert, even if `ETH` is deposited to cover the needed amount, because the contract tries to withdraw the amount of `WETH` without checking its availability.

```
if (payoff > 0) {
    require(POOL.withdraw(address(COLLATERAL), payoff, address(this)) >
0, "Vault: withdraw failed");
>>  WETH.withdraw(payoff);
    payable(_msgSender()).transfer(payoff);
}
```

`Note` This issue is present in all **DNT Vaults**, making it a critical vulnerability that needs to be addressed promptly.

Recommendation:

- I recommend to Implement checks to ensure `WETH` availability before attempting withdrawals. If there is not enough WETH for withdrawals, use the deposited ETH for that purpose if any is available.

```
if (payoff > 0) {
    require(POOL.withdraw(address(COLLATERAL), payoff, address(this)) >
0, "Vault: withdraw failed");
+     uint wethBalance = WETH.balanceOf(address(this)) ;
+     uint toWithdraw = wethBalance > payoff ? payoff : wethBalance
-    WETH.withdraw(payoff);
+   WETH.withdraw( toWithdraw);
-    payable(_msgSender()).transfer(payoff);
+   require(address(this).balance >= payoff);
+  (bool success, ) = _msgSender().call{value: payoff, gas: 100_000}("");
+   require(success, "Failed to send ETH");
}
```

SOFA.org: Acknowledged. We should make sure users can withdraw the correct payoff. We can transfer some WETH into the vault when it is not enough to withdraw.

[L-2] Lack of Enforcement on `spreadAPR` Exceeding `borrowAPR` in `LeverageSmartTrendVault` Contract

Severity: Low	Status: Acknowledged

Context :

- leverageSmartTrendVault

Description:

- in LeverageSmartTrendVault contract If `spreadAPR` exceeds `borrowAPR`, the mint function will always revert due to an underflow here :

```
   uint256 borrowFee = minterCollateral * LEVERAGE_RATIO * borrowAPR *
(params.expiry - block.timestamp) / SECONDS_IN_YEAR / 1e18;
   uint256 spreadFee = minterCollateral * LEVERAGE_RATIO * spreadAPR *
(params.expiry - block.timestamp) / SECONDS_IN_YEAR / 1e18;
>> require(borrowFee - spreadFee >= params.collateralAtRisk -
params.makerCollateral, "Vault: invalid collateral at risk");
```

- This will cause a Dos for the contract, making minting a new product impossible.

Recommendation:

- Add a check to ensure `spreadAPR` is always less than or equal to `borrowAPR` in both the `initialize` function and the `updateSpreadAPR` function:

```
+  require(spreadAPR_ <= borrowAPR, "Vault: spreadAPR must be less or
equal then borrowAPR");
```

SOFA.org: Acknowledged. We will make sure that borrowAPR is always greater than spreadAPR when they are initialized.

4.4 Informational

A total of 1 informational findings were identified.

[I-1] Consider adding `merkleRoot != 0` check

Severity: Informational	Status: Resolved

Context:

- MerkleAirdrop.sol#L42-L53

Description: `claim()` fails to verify that `merkleRoot != 0`, which could result in an inaccurate revert error.

```
    function claim(uint256 index, uint256 amount, bytes32[] calldata
merkleProof) external {
        require(!isClaimed(index), "MerkleAirdrop: Drop already
claimed.");

        bytes32 node = keccak256(abi.encodePacked(_msgSender(), amount));
        require(MerkleProof.verify(merkleProof, merkleRoots[index],
node), "MerkleAirdrop: Invalid proof.");

        _setClaimed(index);

        token.mint(_msgSender(), amount);

        emit Claimed(index, _msgSender(), amount);
    }
```

Recommendation: Recommend to add a check for `merkleRoot != 0`, same as what is done for `claimMultiple()`.

SOFA.org: Fixed in commit #2260

Zenith: Verified