SMART CONTRACT AUDIT REPORT

for

Sofa Protocol

Prepared By: Xiaomi Huang

PeckShield

April 28, 2024

Document Properties

Client	Sofa
Title	Smart Contract Audit Report
Target	Sofa
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 28, 2024	Xuxian Jiang	Final Release
1.0-rc1	April 17, 2024	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Sofa protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the audited protocol can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Sofa

Sofa provides a safe, open, fair and agile standard in developing a decentralized clearing solution to convert all counterparty settlement risks to EVM-based blockchain vaults. Unlike existing protocols,Sofa will record all vital instrument information directly on-chain and on the smart contract, allowing for the creation of transferrable Position Tokens as authentic, rehypothecated collateral claims. These claims will be recognized on supported exchanges and platforms as eligible collateral, unlocking a significant ecosystem liquidity boost. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Sofa

Item	Description
Name	Sofa
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 28, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/sofa-org/sofa-protocol.git (61503fc)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/sofa-org/sofa-protocol.git (c5bfd2c)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Sofa` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■ ■
Low	2	■ ■
Informational	1	■
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Sofa Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Constructor/Initialization Logic in Current Vaults	Coding Practices	Resolved
PVE-002	Medium	Possible Maker Signature Replay in DNTVault	Coding Practices	Resolved
PVE-003	Informational	Suggested Adherence of Checks-Effects-Interactions in AAVEDNTVault	Time And State	Resolved
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-005	Medium	Possible Costly Vault Share From Improper Initialization	Coding Practices	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Constructor/Initialization Logic in Current Vaults

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

To facilitate possible future upgrade, each `Sofa` vault is instantiated as a proxy with an actual logic contract in the backend. While examining the related contract construction and initialization logic, we notice current construction can be improved.

In the following, we shows its initialization routine. We notice its constructor does not have any payload. With that, it can be improved by adding the following statement, i.e., `_disableInitializers();`. Note this statement is called in the logic contract where the initializer is locked. Therefore any user will not able to call the `initialize()` function in the state of the logic contract and perform any malicious activity. Note that the proxy contract state will still be able to call this function since the constructor does not effect the state of the proxy contract.

```
contract SmartTrendVault is Initializable, ContextUpgradeable, ERC1155Upgradeable,
    ReentrancyGuardUpgradeable {

    ...

    function initialize(
        string memory name_,
        string memory symbol_,
        IPermit2 permit_,
        ISmartTrendStrategy strategy_,
        address weth_,
        address collateral_,
        address feeCollector_,
```

```
        ISpotOracle oracle_
    ) initializer external {
        name = name_;
        symbol = symbol_;

        WETH = IWETH(weth_);
        PERMIT2 = permit_;
        STRATEGY = strategy_;

        COLLATERAL = IERC20Metadata(collateral_);
        ORACLE = oracle_;

        DOMAIN_SEPARATOR = keccak256(
            abi.encode(
                EIP712DOMAIN_TYPEHASH,
                keccak256("Vault"),
                keccak256("1.0"),
                block.chainid,
                address(this)
            )
        );
        feeCollector = feeCollector_;
    }
```

Listing 3.1: `SmartTrendVault::initialize()`

Moreover, the above `initialize()` routine can be improved by also initializing the inherited contracts by calling `__Context_init()`, `__ERC1155_init(uri)`, and `__ReentrancyGuard_init()`.

Recommendation Improve the above-mentioned constructor routines in all existing upgradeable vaults, including `DNTVault`, `AAVEDNTVault`, `AAVESmartTrendVault`, `SmartTrendVault`, `LeverageDNTVault`, and `LeverageSmartTrendVault`.

Status This issue has been fixed in the following commit: `a21e25f`.

3.2 Possible Maker Signature Replay in DNTVault

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

`Sofa` supports a number of built-in option-related vaults. While reviewing the option-opening logic in current vaults, we notice the use of signature-based validation. And our analysis shows current signature-based validation can be improved with the addition of maker-side nonce.

To elaborate, we use the `DNTVault` as an example and show below the related `_mint()` routine. The routine allows for the agreement between maker and minter to be officially achieved. With that, there is a need to validate both minter and maker. Since the minter is the calling user, we only need to validate the maker with the maker-provided `makerSignature` (line 172). However, while examining the signature validation, we notice the message to sign does not include the `nonce` information, which indicates the maker signature may be replayed. Note this issue affects all current vaults.

```
147     function _mint(uint256 totalCollateral, MintParams memory params, address referral)
            internal {
148         require(block.timestamp < params.deadline, "Vault: deadline");
149         require(block.timestamp < params.expiry, "Vault: expired");
150         // require expiry must be 8:00 UTC
151         require(params.expiry % 86400 == 28800, "Vault: invalid expiry");
152         require(params.anchorPrices[0] < params.anchorPrices[1], "Vault: invalid strike
                prices");
153         require(params.makerBalanceThreshold <= COLLATERAL.balanceOf(params.maker), "
                Vault: invalid balance threshold");
154         require(referral != _msgSender(), "Vault: invalid referral");
155
156         {
157         // verify maker's signature
158         bytes32 digest =
159             keccak256(abi.encodePacked(
160                 "\x19\x01",
161                 DOMAIN_SEPARATOR,
162                 keccak256(abi.encode(MINT_TYPEHASH,
163                                      _msgSender(),
164                                      totalCollateral,
165                                      params.expiry,
166                                      keccak256(abi.encodePacked(params.anchorPrices)),
167                                      params.makerCollateral,
168                                      params.makerBalanceThreshold,
169                                      params.deadline,
```

```
                                              address(this)))
        ));
        (uint8 v, bytes32 r, bytes32 s) = params.makerSignature.decodeSignature();
        require(params.maker == ecrecover(digest, v, r, s), "Vault: invalid maker signature");

        // transfer makercollateral
        COLLATERAL.safeTransferFrom(params.maker, address(this), params.makerCollateral);
        }
        // mint product
        // startDate = ((expiry-28800)/86400+1)*86400+28800
        uint256 term = (params.expiry - (((block.timestamp - 28800) / 86400 + 1) * 86400 + 28800)) / 86400;
        require(term > 0, "Vault: invalid term");
        {
        uint256 productId = getProductId(term, params.expiry, params.anchorPrices, uint256(0));
        uint256 makerProductId = getProductId(term, params.expiry, params.anchorPrices, uint256(1));
        _mint(_msgSender(), productId, totalCollateral, "");
        _mint(params.maker, makerProductId, totalCollateral, "");
        }
        emit Minted(_msgSender(), params.maker, referral, totalCollateral, term, params.expiry, params.anchorPrices, params.makerCollateral);
    }
```

Listing 3.2: `DNTVault::_mint()`

Recommendation Revise the above routine to add the `nonce` information to prevent maker signature from being replayed.

Status This issue has been resolved as it is part of the design to use `makerBalanceThreshold` to defeat possible replays.

3.3 Suggested Adherence of Checks-Effects-Interactions in AAVEDNTVault

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `AAVEDNTVault`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [13] exploit, and the `Uniswap/Lendf.Me` hack [12].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `AAVEDNTVault` as an example, the `_burn()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. For example, the interaction with the external contract (line 262) start before effecting the update on internal states (lines 265 and 272), hence violating the principle.

```
240     function _burn(uint256 term, uint256 expiry, uint256[2] memory anchorPrices, uint256
             collateralAtRiskPercentage, uint256 isMaker) internal nonReentrant returns (
            uint256 payoff) {
241         (uint256 latestTerm, bool _isBurnable) = isBurnable(term, expiry, anchorPrices);
242         require(_isBurnable, "Vault: not burnable");
243 
244         // check if settled
245         uint256 latestExpiry = (block.timestamp - 28800) / 86400 * 86400 + 28800;
246         require(ORACLE.settlePrices(latestExpiry, 1) > 0, "Vault: not settled");
247 
248         uint256 productId = getProductId(term, expiry, anchorPrices,
                collateralAtRiskPercentage, isMaker);
249         uint256 amount = balanceOf(_msgSender(), productId);
250         require(amount > 0, "Vault: zero amount");
251 
252         // calculate payoff by strategy
253         uint256 payoffShare;
254         uint256 fee;
255         if (isMaker == 1) {
```

```
            (payoffShare, fee) = getMakerPayoff(latestTerm, latestExpiry, anchorPrices,
                collateralAtRiskPercentage, amount);
        } else {
            (payoffShare, fee) = getMinterPayoff(latestTerm, latestExpiry, anchorPrices,
                 collateralAtRiskPercentage, amount);
        }

        // burn product
        _burn(_msgSender(), productId, amount);

        // check self balance of collateral and transfer payoff
        if (payoffShare > 0) {
            totalFee += fee;
            payoff = payoffShare * ATOKEN.balanceOf(address(this)) / totalSupply;
            totalSupply -= payoffShare;
            emit Burned(_msgSender(), productId, amount, payoff);
        } else {
            emit Burned(_msgSender(), productId, amount, 0);
        }
    }
```

Listing 3.3: `AAVEDNTVault::_burn()`

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` principle. Fortunately, current routines are already protected with the `nonReentrant`. modifier that blocks possible `re-entrancy`.

Status This issue has been fixed in the following commit: `a21e25f`.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FeeCollector`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.4: USDT Token Contract

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transfer()` as well, i.e., `safeTransfer()`.

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
```

```
    }
```

Listing 3.5: `SafeERC20::safeApprove()`

In current implementation, if we examine the `FeeCollector::approve()` routine that is designed to approve routers for the spending. To accommodate the specific idiosyncrasy, there is a need to use `safeApprove()`, instead of `approve()` (line 31).

```
29     function approve(IERC20 token, address router) external {
30         require(router == routerV2  router == routerV3, "Collector: invalid router");
31         require(token.approve(router, type(uint256).max), "Collector: approve failed");
32     }
```

Listing 3.6: `FeeCollector::approve()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`.

Status This issue has been fixed in the following commit: `a21e25f`.

3.5 Possible Costly Vault Share From Improper Initialization

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Time and State [5]
- CWE subcategory: CWE-362 [2]

Description

As mentioned earlier, `Sofa` supports a number of built-in option-related vaults. In the process of examining certain `Aave`-related vaults, we notice the share calculation for minting users and the share calculation may lead to an issue that unnecessarily makes the share extremely expensive (and brings hurdles or even causes loss for later minting users).

To elaborate, we show below the `_mint()` routine from `AAVEDNTVault`. The issue occurs when the `Vault` is being initialized under the assumption that the current `vault` is empty.

```
    function _mint(uint256 totalCollateral, MintParams memory params, address referral)
        internal {
        require(block.timestamp < params.deadline, "Vault: deadline");
        require(block.timestamp < params.expiry, "Vault: expired");
        // require expiry must be 8:00 UTC
        require(params.expiry % 86400 == 28800, "Vault: invalid expiry");
        require(params.anchorPrices[0] < params.anchorPrices[1], "Vault: invalid strike
            prices");
```

```
169         require(params.makerBalanceThreshold <= COLLATERAL.balanceOf(params.maker), "Vault: invalid balance threshold");
170         require(referral != _msgSender(), "Vault: invalid referral");
171 
172 
173         {
174         // verify maker's signature
175         bytes32 digest =
176             keccak256(abi.encodePacked(
177                 "\x19\x01",
178                 DOMAIN_SEPARATOR,
179                 keccak256(abi.encode(MINT_TYPEHASH,
180                                     _msgSender(),
181                                     totalCollateral,
182                                     params.expiry,
183                                     keccak256(abi.encodePacked(params.anchorPrices)),
184                                     params.collateralAtRisk,
185                                     params.makerCollateral,
186                                     params.makerBalanceThreshold,
187                                     params.deadline,
188                                     address(this)))
189         ));
190         (uint8 v, bytes32 r, bytes32 s) = params.makerSignature.decodeSignature();
191         require(params.maker == ecrecover(digest, v, r, s), "Vault: invalid maker signature");
192 
193         // transfer makercollateral
194         COLLATERAL.safeTransferFrom(params.maker, address(this), params.makerCollateral);
195         }
196         // calculate atoken shares
197         uint256 term;
198         uint256 collateralAtRiskPercentage;
199         {
200         uint256 aTokenShare;
201         POOL.supply(address(COLLATERAL), totalCollateral, address(this), REFERRAL_CODE);
202         uint256 aTokenBalance = ATOKEN.balanceOf(address(this));
203         if (totalSupply > 0) {
204             aTokenShare = totalCollateral * totalSupply / (aTokenBalance - totalCollateral);
205         } else {
206             aTokenShare = totalCollateral;
207         }
208         totalSupply += aTokenShare;
209 
210         ...
211     }
```

Listing 3.7: `AAVEDNTVault::_mint()`

Specifically, when the `vault` is being initialized, the `shares` value directly takes the value of `totalCollateral` (line 205), which is manipulatable by the malicious actor. As this is the first time to deposit, the `totalSupply` equals the given input amount. With that, the actor can further donate a huge amount to the `vault` (via the `onBehalfOf` support in `Aave`) with the goal of making the `aTokenShare` extremely expensive (line 203).

An extremely expensive `vault` can be very inconvenient to use. Furthermore, it can lead to precision issue in truncating the computed `aTokenShare` for deposited assets (line 203). If truncated to be zero, the deposited assets are essentially considered dust and kept by the contract without returning back to the user.

Recommendation Revise current execution logic of `_mint()` to defensively calculate the share amount when the `vault` is being initialized. An alternative solution is to ensure guarded launch that safeguards the first deposit to avoid being manipulated.

Status This issue has been resolved.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Sofa` protocol, which provides a safe, open, fair and agile standard in developing a decentralized clearing solution to convert all counterparty settlement risks to `EVM`-based blockchain vaults. Unlike existing protocols, the protocol will record all vital instrument information directly on-chain and on the smart contract, allowing for the creation of transferrable `Position Tokens` as authentic, rehypothecated collateral claims. These claims will be recognized on supported exchanges and platforms as eligible collateral, unlocking a significant ecosystem liquidity boost. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[9] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[11] PeckShield. PeckShield Inc. https://www.peckshield.com.

[12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[13] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.