3A DAO Smart Vault

Audit report prepared by 0xMilenov

1 About 0xMilenov

Independent Security Researcher

As a software engineer based in Bulgaria, 0xMilenov is a passionate expert in blockchain security, with a keen focus on **Web3, smart contracts, and vulnerability detection.**

Collaborating with renowned figures, he's contributed significantly to Web3 security projects and developed impactful tools.

0xMilenov's blend of technical expertise, community building, and proactive security research establish him as a dynamic and influential figure in the blockchain security landscape.

Feel free to reach out on X or 0xMilenov

2 Protocol Summary

3A DAO Smart Vault

This audit centered on the **Smart Vault** contract, which manages various functionalities including creation, collateralization, borrowing, liquidation of vaults, and execution of whitelisted methods.

Usage flow:

```
1. Vault creation and ownership
2. Collateral management
3. Debt and borrowing
4. Execution of whitelisted functions
```

Checked possibile vulnerable places

```
- `Access control checks` (onlyFactory, onlyVaultOwner)
- `Collateral and debt management`
- `Rewards and borrow rates`
```

Specifics

```
- `executeBatch and execute Functions`: Verified the execution control based on a whitelist, ensuring only approved
↪   functions are executed with proper parameter validations.

- `_claimPendingRewards`: Assessed the rewards claiming mechanism, focusing on error handling and fee deductions.
```

During the audit of the Smart Vault contract, two vulnerabilities were identified in the auxiliary contracts affecting auction liquidations and borrow rate calculations

1. Incorrect Auction Price Setting in AuctionManager.sol:

   The auction logic mistakenly allows the lowestHF parameter to be set >= 100%, which can lead to auction prices being set higher than the collateral's worth, causing potential losses for bidders.

   Recommendation: Set lowestHF below 100% and ensure input validation restricts values to less than 1e18 (100%).

2. Inaccurate Borrow Rate Calculation in VaultBorrowRate.sol:

   The getBorrowRate() function mishandles the normalization of collateral values when different tokens have varying decimal places. This results in incorrect borrow rate calculations.

   Recommendation: Adjust the normalization process to divide by 1e18 instead of 10 ** _priceFeed.decimals(_collateralAddress) to ensure accuracy across different token types.

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

3 Risk Classification

	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

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

3.2 Likelihood

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

3.3 Action required for severity levels

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

4 Executive Summary

Overview

Project	3A DAO Smart Vault
Repository	3a-borrowing-protocol-v2
Commit	...
Date	April 2024

Issues Found

Severity	Count
Critical Risk	0

High Risk	0
Medium Risk	2
Low Risk	0
Informational	0
Gas Optimizations	0
Total Issues	**2**

Summary of Findings

Title	**Status**
[M-1] Faulty calculation of borrow rates results	Fixed
[M-2] Auction Manager creates limitations for bidders	Fixed

5 Findings

5.1 Medium Risk

[M-1] Faulty calculation of borrow rates results

Context VaultBorrowRate.sol

Description The function `getBorrowRate()` is responsible for computing the weighted borrow rate. However, it contains a defect in calculating the collateral value when dealing with multiple types of collateral tokens.

This issue is particularly pronounced when the tokens vary in decimal places, such as USDC which has 6 decimals, and WETH which has 18 decimals.

While the normalization process converts the collateral amount to 18 decimal, **the following calculation fails to properly account for tokens with lesser decimals, leading to inaccurate borrow rate calculations** because it uses `10 ** _priceFeed.decimals(_collateralAddress)` in the denominator instead of `1e18`.

```
function getBorrowRate(
    address _vaultAddress
) external view returns (uint256) {

        uint256 _normalizedCollateralAmount = _collateralAmount *
            (10 ** (18 - _priceFeed.decimals(_collateralAddress)));
        uint256 _collateralValue = (_normalizedCollateralAmount * _price) /
            (10 ** _priceFeed.decimals(_collateralAddress)); -----> should divide by 1e18
        uint256 _weightedFee = (_collateralValue * _borrowRate) / 1e18;

}
```

Recommendation

```
function getBorrowRate(
    address _vaultAddress
) external view returns (uint256) {

        uint256 _normalizedCollateralAmount = _collateralAmount *
            (10 ** (18 - _priceFeed.decimals(_collateralAddress)));
+       uint256 _collateralValue = (_normalizedCollateralAmount * _price) / 1e18;
        uint256 _weightedFee = (_collateralValue * _borrowRate) / 1e18;

}
```

[M-2] Auction Manager creates limitations for bidders

Context: AuctionManager.sol

Description When starting a new liquidation auction, the maximum debt available for auction is established to match the stable coin debt of the vault:

```
uint256 _highestDebtToAuction = _debtToAuction;
```

The variable `lowestHF`, which reflects a percentage of the collateral value, is initially set at `1.05e18` (105%). The setter function, `setLowestHealthFactor()`, for this variable only verifies that the value is greater than zero.

```
uint256 public lowestHF = 1.05 ether; // 105%
...
...
...
function setLowestHealthFactor(uint256 _lowestHF) external onlyOwner {
    require(_lowestHF > 0, 'lowest-hf-is-0');
    lowestHF = _lowestHF;
}
```

Allowing lowestHF to be >= 1e18 (>= 100%), coupled with an if-statement designed to prevent underflow by reversing price bounds, and the fact that the collateral is already valued higher than the debt, **leads to setting the auction price higher than the collateral's worth. This scenario results in bidders experiencing losses rather than gains.**

```
if (_highestDebtToAuction < _lowestDebtToAuction) {
    uint256 _debtToAuctionTmp = _lowestDebtToAuction;
    _lowestDebtToAuction = _highestDebtToAuction;
    _highestDebtToAuction = _debtToAuctionTmp;   <-- this is bigger than the actual debt
}
```

In situations with minor price fluctuations and a stable market, when the health factor slightly falls below `1e18`, the auction starts off unprofitable but eventually becomes less expensive than the value of the collateral. Conversely, in cases of substantial price shifts and volatile markets, the lowest auction price stays above the collateral value, resulting in bad debt.

Recomendation `lowestHF` need to be under 1e18

```
+ uint256 public lowestHF = 0.95 ether; // 95%
...
...
...
function setLowestHealthFactor(uint256 _lowestHF) external onlyOwner {
+     require(_lowestHF > 0 && _lowestHF < 1e18, 'wrong-hf');
      lowestHF = _lowestHF;
}
```