Making Web3 Space Safe for Everyone

KALOS

Unstable Protocol Marketplace

Security Assessment

Published on : 27 May. 2024
Version v1.1

Security Report Published by KALOS

v1.1 27 May. 2024

Auditor : Andy Koo, Jinu Lee

Found issues

Severity of Issues	Findings	Resolved	Acknowledged	Comment
Critical	1	1	-	-
High	1	-	1	-
Medium	2	2	-	-
Low	5	3	-	2
Tips	7	5	-	2

TABLE OF CONTENTS

TABLE OF CONTENTS
ABOUT US
Executive Summary
OVERVIEW
- Protocol overview
- Scope
- Access Controls

FINDINGS
- 1. Miscalculation of the user's collateral ratio in _checkHealth function.
- 2. The vaults Storage Variable Not Properly Updating Its Entries
- 3. The Debtor's Reward Also Needs to Be Updated When Liquidation is Called
- 4. The nonReentrant modifier needs to be applied
- 5. Incorrect implementation of transfer in the nUSD.executeFlashloan function
- 6. Missing validation 1 - StakedEthVault
- 7. Missing validation 2 - emUSM
- 8. Missing validation 3 - StakedEthVault
- 9. Incorrect implementation in UnstableConfigurator
- 10. Shared rewardManager role across multiple vaults
- 11. Incoherent Basis Points notation.
- 12. Borrowed fee calculation doesn't retrospectively reflect borrowApr updates.
- 13. Front-run to cause a denial of service (DoS) when snUSD is deployed.
- 14. When the cooldown is adjusted, users are still required to adhere to the previously established duration.
- 15. The pause state is not accounted for when withdrawing.
- 16. isDepegged function only checks for marketRate lower than redemptionRate

DISCLAIMER
Appendix. A
- Severity Level
- Difficulty Level
- Vulnerability Category

ABOUT US

Making Web3 Space Safe for Everyone

Pioneering a safer Web3 space since 2018, KALOS proudly won 2nd place in the Paradigm CTF 2023. As a leader in the global blockchain industry, we unite the finest in Web3 security expertise.

Our team consists of top security researchers with expertise in blockchain/smart contracts and experience in bounty hunting. Specializing in the audit of mainnets, DeFi protocols, bridges, and the zkEVM protocol, KALOS has successfully safeguarded billions in crypto assets.

Supported by grants from the Ethereum Foundation and the Community Fund, we are dedicated to innovating and enhancing Web3 security, ensuring that our clients' digital assets are securely protected in the highly volatile and ever-evolving Web3 landscape.

Inquiries: audit@kalos.xyz
Website: https://kalos.xyz

Executive Summary

Purpose of this report

This report was prepared to audit the security of the Unstable Protocol smart contracts. KALOS conducted the audit focusing on whether the system is soundly implemented and designed as specified in the published materials, in addition to the safety and security of the Unstable Protocol. In detail, we have focused on the following

- Project availability issues like Denial of Service.
- Strict access control on storage variables to prevent unauthorized access.
- Function access control measures.
- Safeguards against the freezing and theft of stored assets.
- Yield calculation processes to prevent manipulation or asset theft.
- comprehensive error handling for unhandled exceptions, ensuring protocol robustness.

Codebase Submitted for the Audit

The code used in this Audit can be found on GitHub (https://github.com/LSDfi-cafe/unstable-protocol/tree/feature/auditors).

The last commit of the code used for this Audit is “b7d86906924e67eeaef9e0227df1333eb7ea2083”.

The last commit of the code patched for this Audit is “61577e2e2135a0ae44a8cced9808a0e6cf50cfa0”.

Audit Timeline

Date	Event
2024/05/07	Audit Initiation
2024/05/21	Delivery of v1.0 report.
2024/05/27	Delivery of v1.1 report.

Findings

KALOS found 1 Critical, 1 High, 2 medium, and 5 Low severity issues. There are 7 Tips issues explained that would improve the code's usability or efficiency upon modification.

Severity	Issue	Status
Critical	Miscalculation of the user's collateral ratio in _checkHealth function.	(Resolved - v1.1)
Low	The vaults Storage Variable Not Properly Updating Its Entries	(Resolved - v1.1)
Medium	The Debtor's Reward Also Needs to Be Updated When Liquidation is Called.	(Resolved - v1.1)
Tips	The nonReentrant modifier needs to be applied.	(Resolved - v1.1)
Low	Incorrect implementation of transfer in the nUSD.executeFlashloan function	(Resolved - v1.1)
Tips	Missing validation 1 - StakedEthVault	(Resolved - v1.1)
Tips	Missing validation 2 - emUSM	(Resolved - v1.1)
Tips	Missing validation 3 - StakedEthVault	(Resolved - v1.1)
Low	Incorrect implementation in UnstableConfigurator	(Resolved - v1.1)
Low	Shared rewardManager role across multiple vaults	(Commented - v1.1)
Tips	Incoherent Basis Points notation.	(Resolved - v1.1)
High	Borrowed fee calculation doesn't retrospectively reflect borrowApr updates	(Acknowledged - v1.1)
Tips	Front-run to cause a denial of service (DoS) when snUSD is deployed.	(Commented - v1.1)
Low	When the cooldown is adjusted, users are still required to adhere to the previously established duration.	(Commented - v1.1)
Medium	The pause state is not accounted for when withdrawing.	(Resolved - v1.1)
Tips	isDepegged function only checks for marketRate lower than redemptionRate	(Commented - v1.1)

OVERVIEW

Protocol overview

• **Staked Eth Vault**

The StakedEthVault.sol contract is a vault contract where non-rebasing LST/LRT tokens can be deposited. The contract accrues a reward proportional to the user's deposit amount. The deposited assets can be used as collateral for minting the nUSD token, which is pegged to 1 USD. The user can borrow the nUSD up to the collateral ratio set on the UnstableConfigurator.sol contract by the contract owner/admin. There is a liquidation and rigid redemption function which liquidates or redeems the user's debts. This contract utilizes the Oracle contract to calculate the collateral token's price.

• **Oracles**

Each token's oracle contract retrieves the price of each non-rebasing LST/LRT to the ETH. The contract retrieves the market ratio (if possible) and the redemption rate of the underlying token.

Contract	Oracle Decimal	Oracle Interval	Deviation threshold
ApxETHOracle.sol	8	86400	1%
CsETHOracle.sol	N/A	N/A	N/A
PufETHOracle.sol	8	86400	1%
RsETHOracle.sol	18	86400	0.5%
UnshETHOracle.sol	N/A	N/A	N/A
WeETHOracle.sol	18	86400	0.5%
WstETHOracle.sol*	18	86400	0.5%

* There is no direct WstETH/ETH Oracle Exists and stETH/ETH oracle is used with its redemption rate to the WstETH.

• **nUSD / snUSD**

The nUSD token is minted by collateralizing the LST/LRT tokens to the vault contract. These minted nUSD tokens can be staked to the snUSD contract. The transferred rewards nUSD tokens are accumulated to the snUSD contract, and the user can redeem the rewards and staked tokens after the cooltime has passed.

• **USM / esUSM**

The USM token is a governance token of Unstable Protocol. The USM token can be converted to escrowed USM. The from and to address of a transfer of the esUSM is restricted only to white-listed addresses. The esUSM can be converted back to USM through a vesting process. There is no reward emission feature in the esUSM contract.

• **Communal Farm**

The CommunalFarm.sol contract is a fork of Frax's multi-token rewards contract. Users can stake their ERC20 tokens to earn rewards in multiple tokens. The contract supports multiple reward tokens with different rates that managers can update. Various entities can manage the reward rates for their specific tokens.

Notice

1. There are certain functions that allow the contract owners to mint or transfer tokens. (snUSDBase.rescueTokens onlyConfiguratorOrAdmins, nUSD.mint onlyVault)
2. The contract owner of the UnstableConfigurator contract is capable of setting asset-related values, such as oracles and fee rates, which can impact users' assets utilizing the StakedEthVault contract. Notably, the borrowApr value can be updated instantly(max apr: 1000%); therefore, users should be cautious of any changes.
3. The vulnerability or other incident effects on the collateral tokens can directly affect the value of the nUSD token. The maximum damage can be the maximum nUSD issuance per vault set in the configurator.
4. The collateral token of the StakedEthVault contract is non-based LST/LRT. The usage of rebasing collateral tokens on the contract is out of the scope.

Scope

```
unstable/
├── vaults
│   └── base
│       └── StakedEthVault.sol
├── token
│   ├── snUSDBase.sol
│   ├── snUSD.sol
│   ├── nUSDDepot.sol
│   ├── nUSD.sol
│   ├── esUSM.sol
│   └── USM.sol
├── oracles
│   ├── WstETHOracle.sol
│   ├── WeETHOracle.sol
│   ├── UnshETHOracle.sol
│   ├── RsETHOracle.sol
│   ├── PufETHOracle.sol
│   ├── PendlePTOracle.sol
│   └── ApxETHOracle.sol
├── interfaces
│   ├── IsnUSD.sol
│   ├── InUSDDefinitions.sol
│   ├── InUSD.sol
│   ├── IesUsmToken.sol
│   ├── IZkOracle.sol
│   ├── IVault.sol
│   ├── IUsmToken.sol
│   ├── IERC20.sol
│   └── IConfigurator.sol
├── farms
│   └── CommunalFarm.sol
└── configuration
    └── UnstableConfigurator.sol
```

Access Controls

- Unstable Configurator
    - Owner : Updates protocol-wide used addresses, including treasury, reward manager, oracles, nUSD, and vaults, as well as fee-related configurations, collateral ratios, and market supplies. Have privileges to withdraw and distribute fee tokens accumulated on the contract.
    - Admin : Sets the pause state of the vault's nUSD mint/burn call. Distributes fee tokens accumulated on the contract.

- Staked ETH Vault
    - Reward Manager : Adds reward tokens and sets the reward duration of each reward token.

- Communal Farm
    - Owner : Updates reward-related configurations like duration, multiplier, pause state, reward tokens, and their managers.
    - Token Manager : Sets reward rates for each reward token.

- esUSM
    - Owner : Sets the whitelisted addresses to which the esUSM tokens can be transferred and updates the redeem ratio and duration configurations.

Each privileged account has permissions that can change the crucial part of the system. It is highly recommended to maintain the private key as securely as possible and strictly monitor the system state changes.

FINDINGS

1. Miscalculation of the user's collateral ratio in _checkHealth function.

ID: UP-01
Type: Logic Error
File: contracts/unstable/vaults/base/StakedEthVault.sol

Severity: Critical
Difficulty: Low

Issue

The vaultSafeCollateralRatio of the configurator is using BPS (10000) notation. However, the user's collateral ratio is based on 10,000 * 1e18. Therefore, the health check always returns true.

```
  function _checkHealth(address user) internal view returns(bool) {
        uint256 price = getAssetPrice();
        if (((depositedAsset[user] * price * 10_000) / getBorrowedOf(user)) / 1e18 <
configurator.getSafeCollateralRatio(address(this))) {
            return false;
        } else {
            return true;
        }
    }
```

[https://github.com/LSDfi-cafe/unstable-protocol/blob/7159949c93b45ad86b4e79fb01472960799dab13/contracts/unstable/vaults/base/StakedEthVault.sol#L285C1-L292C6]

Recommendation

The calculation ((depositedAsset[user] * price * 10_000) / getBorrowedOf(user)) needs to be divided by 1e18.

Fix Comment

[e577ec3] Applied the division by 1e18.

2. The vaults Storage Variable Not Properly Updating Its Entries

ID: UP-02	Severity: Low
Type: Logic Error	Difficulty: Low

File: contracts/unstable/configuration/UnstableConfigurator.sol

Issue

The vault storage variable in UnstableConfigurator.sol is an array of addresses, but it is currently pushing 0x0 when a vault is enabled.

```
    function enableVault(address vault) external onlyOwner {
        require(!vaultEnabled[vault], "Vault already enabled");
        address collateral = IVault(vault).collateralAsset();
        require(
            zkOracleAddress[collateral] != address(0),
            "Set zkOracle before activating vault"
        );
        require(IVault(vault).getAssetPrice() > 0, "Price oracle not working");

        vaultEnabled[vault] = true; //enable vaults
        vaults.push(); //add to vaults list

        //if collateral is unique
        if (collateralEnabled[collateral] == false) {
            collaterals.push(collateral); //add to collateral list
            collateralEnabled[collateral] = true; //add to lookup
        }
        emit VaultEnabled(vault);
    }
```

[https://github.com/LSDfi-cafe/unstable-protocol/blob/7159949c93b45ad86b4e79fb01472960799dab13/contracts/unstable/configuration/UnstableConfigurator.sol#L232-L247]

Recommendation

The address needs to be pushed properly: vaults.push(vault).

Fix Comment

[e577ec3] Fixed the push logic to update address value.

3. The Debtor’s Reward Also Needs to Be Updated When Liquidation is Called

ID: UP-03	Severity: Medium
Type: Logic Error	Difficulty: Low

File: contracts/unstable/vaults/base/StakedEthVault.sol

Issue

The liquidation function decreases the debtor's depositedAsset value. Since the reward is not updated before the liquidation process, the debtor would lose the reward accrued before the liquidation call.

```
    function liquidation(address provider, address debtor, uint256 assetAmount) external
updateReward(provider) nonReentrant virtual {
```

[https://github.com/LSDfi-cafe/unstable-protocol/blob/7159949c93b45ad86b4e79fb01472960799dab13/contracts/unstable/vaults/base/StakedEthVault.sol#L138]

Recommendation

Update the debtor's reward information before executing the liquidation logic.

Fix Comment

[e577ec3] The debtor’s reward information is updated before the liquidation logic.

4. The nonReentrant modifier needs to be applied

ID: UP-04 Severity: Tips
Type: Reentrancy Difficulty: N/A
File: contracts/unstable/vaults/base/StakedEthVault.sol

Issue

Since several collateral tokens use the proxy pattern, it is safe to use the nonReentrant modifier on functions that interact with these tokens. When the collateral's logic is modified to include unexpected features, there is a risk of malicious reentrant calls. Currently, the only function with this modifier is the getReward function.

Recommendation

We recommend applying the nonReentrant modifier to the following functions: redemption, liquidation, withdraw, depositAssetToMint, burn, and mint.

Fix Comment

[9b92729] Nonreentrant modifiers are applied to the functions mentioned.

5. Incorrect implementation of transfer in the nUSD.executeFlashloan function

ID: UP-05
Type: Logic Error
File: contracts/unstable/token/nUSD.sol

Severity: Low
Difficulty: N/A

Issue

If calling the transfer function inside the executeFlashloan function, the code _transfer(from: msg.sender, to: msg.sender, amount: amount) will work. A flash loan is a type of loan where a user borrows assets with no upfront collateral and returns the borrowed assets within the same blockchain transaction. However, this function works the user to use their own assets and pay a fee.

```
    function executeFlashloan(uint256 amount, bytes calldata data) external {
        transfer(msg.sender, amount);
        IFlashBorrower(msg.sender).onFlashLoan(amount, data);
        bool success = transferFrom(msg.sender, address(this), amount);
        require(success, "Transfer Failed");
        uint256 burnShare = getFee(amount);
        _burn(msg.sender, burnShare);
        emit Flashloaned(msg.sender, amount, burnShare, block.timestamp);
    }
```

[https://github.com/LSDfi-cafe/unstable-protocol/blob/7159949c93b45ad86b4e79fb01472960799dab13/contracts/unstable/token/nUSD.sol#L64-L72]

Recommendation

The patch can be applied as shown in the example below:

```
    function executeFlashloan(uint256 amount, bytes calldata data) external {
        _mint(msg.sender, amount);
        IFlashBorrower(msg.sender).onFlashLoan(amount, data);
        uint256 burnShare = getFee(amount);
        _burn(msg.sender, amount + burnShare);
        emit Flashloaned(msg.sender, amount, burnShare, block.timestamp);
    }
```

[example]

Fix Comment

[28c60a5] Modified the flash loan logic to mint the loan amount and transfer the fees to the configurator contract. The maximum amount that can be minted is capped to the total supply of nUSD.

6. Missing validation 1 - StakedEthVault

ID: UP-06 Severity: Tips
Type: Input Validation Difficulty: N/A
File: contracts/unstable/vaults/base/StakedEthVault.sol

Issue

StakedEthVault.addReward function is missing a validation to check if the _rewardsDuration is greater than 0.

```
    function addReward(address _rewardsToken, uint256 _rewardsDuration) public onlyRewardManager {
        require(_rewardsToken != address(collateralAsset) && _rewardsToken != address(nUSD), "Reward 
cannot be collateral asset or nUSD");
        require(rewardData[_rewardsToken].rewardsDuration == 0, "Reward already exists");
        rewardTokens.push(_rewardsToken);
        rewardData[_rewardsToken].rewardsDuration = _rewardsDuration;
    }
```

[https://github.com/LSDfi-cafe/unstable-protocol/blob/7159949c93b45ad86b4e79fb01472960799dab13/contracts/unstable/vaults/base/StakedEthVault.sol#L494-L499]

Fix Comment

[7ac5112] The _rewardsDuration value is validated to be greater than 0.

7. Missing validation 2 - emUSM

ID: UP-07
Type: Input Validation
File: contracts/unstable/token/esUSM.sol

Severity: Tips
Difficulty: N/A

Issue

emUSM.redeem function is missing a validation to check if the duration is greater than maxRedeemDuration.

```
    function redeem(uint256 esUSMAmount, uint256 duration) external nonReentrant {
        require(esUSMAmount > 0, "redeem: esUSMAmount cannot be null");
        require(duration >= minRedeemDuration, "redeem: duration too low");

        _transfer(msg.sender, address(this), esUSMAmount);
        esUSMBalance storage balance = esUsmBalances[msg.sender];

        // get corresponding USM amount
        uint256 usmAmount = getUsmByVestingDuration(esUSMAmount, duration);
        emit Redeem(msg.sender, esUSMAmount, usmAmount, duration);

        // if redeeming is not immediate, go through vesting process
        if(duration > 0) {
            // add to SBT total
            balance.redeemingAmount = balance.redeemingAmount.add(esUSMAmount);

            // add redeeming entry
            userRedeems[msg.sender].push(RedeemInfo(usmAmount, esUSMAmount,
_currentBlockTimestamp().add(duration)));
        } else {
            // immediately redeem for USM
            _finalizeRedeem(msg.sender, esUSMAmount, usmAmount);
        }
    }
```

[https://github.com/LSDfi-cafe/unstable-protocol/blob/7159949c93b45ad86b4e79fb01472960799dab13/contracts/unstable/token/esUSM.sol#L205-L227]

Recommendation

patch-a: `if (duration > maxRedeemDuration) duration = maxRedeemDuration;`

patch-b: `require(duration <= maxRedeemDuration, "redeem: duration too high");`

Fix Comment

[7ac5112] The duration value is validated to be less than maxRedeemDuration.

8. Missing validation 3 - StakedEthVault

ID: UP-08
Severity: Tips
Type: Input Validation
Difficulty: N/A
File: contracts/unstable/vaults/base/StakedEthVault.sol

Issue

StakedEthVault.notifyRewardAmount function is missing a validation to check _rewardsToken is enabled. The code(`rewardData[_rewardsToken].rewardRate = reward / rewardData[_rewardsToken].rewardsDuration;`) causes a zero division error, so it's not cause problems, but it is advisable to add validation to ensure _rewardsToken is enabled.

```
    function notifyRewardAmount(address _rewardsToken, uint256 reward) external onlyRewardManager
updateReward(address(0)) {
        IERC20(_rewardsToken).safeTransferFrom(msg.sender, address(this), reward);
        if (block.timestamp >= rewardData[_rewardsToken].periodFinish) {
            rewardData[_rewardsToken].rewardRate = reward / rewardData[_rewardsToken].rewardsDuration;
        } else {
            uint256 remaining = rewardData[_rewardsToken].periodFinish - block.timestamp;
            uint256 leftover = remaining * rewardData[_rewardsToken].rewardRate;
            rewardData[_rewardsToken].rewardRate = (reward + leftover) /
rewardData[_rewardsToken].rewardsDuration;
        }
        rewardData[_rewardsToken].lastUpdateTime = block.timestamp;
        rewardData[_rewardsToken].periodFinish = block.timestamp +
rewardData[_rewardsToken].rewardsDuration;
        emit RewardAdded(reward);
    }
```

[https://github.com/LSDfi-cafe/unstable-protocol/blob/7159949c93b45ad86b4e79fb01472960799dab13/contracts/unstable/vaults/base/StakedEthVault.sol#L501-L513]

Fix Comment

[89abeb1] The validation logic of whether the rewardsDuration is 0 is applied to the function.

9. Incorrect implementation in UnstableConfigurator

ID: UP-09 Severity: Low
Type: Logic Error Difficulty: N/A
File: contracts/unstable/configuration/UnstableConfigurator.sol

Issue

1. Unreachable condition

```
    function getRedemptionFee(address vault, uint256 collateralRatio) external view returns(uint256
providerFee, uint256 protocolFee) {
        require(vaultEnabled[vault], "Vault not enabled");
        require(collateralRatio >= 100_00, "Cannot redeem when collateral ratio is below 100%");
        RedemptionConfig memory config = getRedemptionConfig(vault);
        require(config.enabled, "Redemption not enabled");
        require(collateralRatio <= config.maxCollateralRatio, "Cannot redeem collateral ratio above
max"); // <- (0)
        ...
        else if(collateralRatio > config.maxCollateralRatio) { // <- (1)
            totalFee = config.baseFee * config.maxMultiplier / 100_00;
            providerFee = totalFee * providerShare / 100_00;
        }
```

[https://github.com/LSDfi-cafe/unstable-protocol/blob/7159949c93b45ad86b4e79fb01472960799dab13/contracts/unstable/configuration/UnstableConfigurator.sol#L470-L496]

(1) condition is unreachable because the require statement (0) already ensures that collateralRatio is within the maximum limit.

2. Hardcoded provider share

Although providerShare is defined within RedemptionConfig, the function uses a hardcoded value instead.

```
    struct RedemptionConfig {
        bool enabled; //whether a vault can redeem
        uint16 baseFee; // base fee for redemption
        uint16 maxMultiplier; // fee for redemption
        uint16 maxCollateralRatio; // collateral ratio for max fee multiplier
        uint16 providerShare; // share of redemption fee that goes to provider
    }
    ...
    function getRedemptionFee(address vault, uint256 collateralRatio) external view returns(uint256
providerFee, uint256 protocolFee) {
        ...
        uint16 providerShare = 80_00; //80% of redemption fee goes to provider
```

[https://github.com/LSDfi-cafe/unstable-protocol/blob/7159949c93b45ad86b4e79fb01472960799dab13/contracts/unstable/configuration/UnstableConfigurator.sol#L470-L496]

3. Misconfiguration leading to underflow
   The maxMultiplier is intended to be 3% (3_00) but is set to 3, which is inconsistent and leads to an underflow.

```
    constructor(address _etherOracle) {
        flashloanFee = 500; //500 bps = 5%
        defaultSafeCollateralRatio = 150_00; //15000 bps = 150%
        defaultBadCollateralRatioDistance = 15_00; //1500 bps = 15%
        defaultOriginationFee = OriginationFeeConfig(0, 1000); //0% at 0% utilization, 10% at 100%
utilization
        defaultKeeperReward = 500; //500 bps = 5% reward
        defaultDepegThreshold = 300; //300 bps = 3% depeg
        defaultRedemptionConfig = RedemptionConfig(true, 100, 3, 200_00, 75_00); //100 bp fee, 3x max
multiplier, 200% cr for max multiplier, 75% fee to provider
```

[https://github.com/LSDfi-cafe/unstable-protocol/blob/main/contracts/unstable/configuration/UnstableConfigurator.sol#L117-L124]

And the getRedemptionFee function calculation must be adjusted:

```
// origin
totalFee = config.baseFee * (100_00 + (config.maxMultiplier-100_00) *
(collateralRatio-safeCollateralRatio) / (config.maxCollateralRatio -safeCollateralRatio)) / 100_00;

// patch example
totalFee = config.baseFee * (100_00 + (config.maxMultiplier) * (collateralRatio - safeCollateralRatio)
/ (config.maxCollateralRatio - safeCollateralRatio)) / 100_00;
```

[https://github.com/LSDfi-cafe/unstable-protocol/blob/7159949c93b45ad86b4e79fb01472960799dab13/contracts/unstable/configuration/UnstableConfigurator.sol#L499]

Fix Comment

[e8da6d4]

1. Removed unreachable logic.
2. The hardcoded providerShare value is replaced with the config.providerShare.
3. The misconfigured defaultRedemptionConfig value is corrected.

10. Shared rewardManager role across multiple vaults

ID: UP-10 Severity: Low
Type: Access & Privilege Control Difficulty: High
File: contracts/unstable/vaults/base/StakedEthVault.sol

Issue

The RewardManager contract, managed by the UnstableConfigurator contract, is designed to handle rewards and has authority over withdrawing ERC20 tokens from multiple StakedEthVault contracts. This shared authority introduces several potential issues, particularly if the RewardManager authority is extended to third parties.

```
    modifier onlyRewardManager() {
        require(configurator.isRewardManager(msg.sender), "Not reward manager");
        _;
    }
```

[https://github.com/LSDfi-cafe/unstable-protocol/blob/7159949c93b45ad86b4e79fb01472960799dab13/contracts/unstable/vaults/base/StakedEthVault.sol#L544-L547]

```
    mapping(address => bool) public isRewardManager;          //user that can manage rewards
    ...
    function setRewardManager(address _rewardManager) external onlyOwner {
        require(_rewardManager != address(0), "Reward manager cannot be the zero address");
        if (!isRewardManager[_rewardManager]) {
            isRewardManager[_rewardManager] = true;
            emit RewardManagerSet(_rewardManager);
        }
    }
```

[https://github.com/LSDfi-cafe/unstable-protocol/blob/7159949c93b45ad86b4e79fb01472960799dab13/contracts/unstable/configuration/UnstableConfigurator.sol#L156-L162]

Recommendation

Independent RewardManager role for each StakedEthVault to ensure isolated and secure reward management.

Comment by core contributors

This is their intentional implementation. The rewards manager role is a low-risk role, and a patch will not be applied to avoid over-complication.

11. Incoherent Basis Points notation.

ID: UP-11
Type: Off-standard
Severity: Tips
Difficulty: N/A
File: contracts/unstable/vaults/base/StakedEthVault.sol
contracts/unstable/configuration/UnstableConfigurator.sol

Issue

The large number notation, some numbers are grouped into two digits (e.g., 10_00) instead of the conventional three-digit grouping (e.g., 1,000 or 100_000). This can cause confusion and reduce the readability of the data presented.

Additionally, using a Basis Points (BPS) variable name for 100_000 can improve readability.

```
    function setRedemptionFee(
        address vault,
        bool enabled,
        uint16 baseFee,
        uint16 maxMultiplier,
        uint16 maxCollateralRatio,
        uint16 providerShare
    ) external onlyOwner {
        require(baseFee <= 10_00, "Max Redemption Fee is 10%");
        require(maxMultiplier <= 100_000, "Max multiplier is 10x");
        require(
            maxCollateralRatio >= 150_00 && maxCollateralRatio <= 300_00,
            "Cr for max multiplier must be 150%-300%"
        );
        require(
            providerShare >= 50_00 && providerShare <= 100_00,
            "Provider share must be 50%-100%"
        );
```

[https://github.com/LSDfi-cafe/unstable-protocol/blob/7159949c93b45ad86b4e79fb01472960799dab13/contracts/unstable/configuration/UnstableConfigurator.sol#L351]

Recommendation

Ensure consistency in number notation and consider using BPS notation.

Fix Comment

[7ac5112] The number notation is updated to be coherent with each other.

12. Borrowed fee calculation doesn't retrospectively reflect borrowApr updates.

ID: UP-12
Severity: High
Type: Logic Error
Difficulty: Medium
File: contracts/unstable/vaults/base/StakedEthVault.sol

Issue

When calculating the borrowed fee in StakedEthVault, the formula used is "seconds since the user's last fee update * borrowApr". If the borrowApr changes before the updateFee function is called, the fee calculation does not retroactively include the seconds that have passed since the last fee update. Instead, the entire fee calculation is updated with the new borrowApr value.

```
    function _newFee(address user) internal view returns (uint256) {
        uint256 secondsInYear = 86_400 * 365;
        uint256 secondsSinceLastFee = block.timestamp - feeUpdatedAt[user];
        return borrowed[user] * configurator.borrowApr(address(this)) * secondsSinceLastFee /
secondsInYear / 10_000;
    }
```

[https://github.com/LSDfi-cafe/unstable-protocol/blob/7159949c93b45ad86b4e79fb01472960799dab13/contracts/unstable/vaults/base/StakedEthVault.sol#L301-L305]

This issue leads to an inaccurate fee calculation, as it fails to account for the time passed with the previous borrowApr, potentially resulting in unexpected fees for users.

Recommendation

```
contract UnstableConfigurator is Ownable {
    // ...
    function setBorrowApr(address vault, uint16 newApr) external onlyOwner {
        require(newApr <= 1000_00, "Borrow APR cannot exceed 1000%");
        StakedEthVault(vault).updateFeeByConfigurator();
        borrowApr[vault] = newApr;
        emit BorrowAprChanged(vault, newApr);
    }
}

contract StakedEthVault is ReentrancyGuard {
    // ...
    uint256 lastFeeUpdateAt;
    uint256 accumulatedFees;

    uint256 constant secondsInYear = 86_400 * 365;

    constructor(address _collateral, address _configurator) {
        // ...
        lastFeeUpdateAt = block.timestamp;
    }
    function updateFeeByConfigurator() public {
        _updateFee(address(0));
    }

    function _updateFee(address user) internal {
        if (block.timestamp > lastFeeUpdateAt) {
            uint256 secondsSinceLastFee = block.timestamp - lastFeeUpdateAt;
            accumulatedFees += configurator.borrowApr(address(this)) * secondsSinceLastFee;
            lastFeeUpdateAt = block.timestamp;
        }
        if (accumulatedFees > feeAccumulatedUser[user]) {
            feeStored[user] += _newFee(user);
            feeAccumulatedUser[user] = accumulatedFees;
        }
    }

    function _newFee(address user, uint256 _accumulatedFees) internal view returns (uint256) {
        uint256 userFee = _accumulatedFees - feeAccumulatedUser[user];
        return borrowed[user] * userFee / secondsInYear / 10_000;
    }

    function _newFee(address user) internal view returns (uint256) {
        return _newFee(user, accumulatedFees);
    }

    function getBorrowedOf(address user) public view returns (uint256) {
        uint256 secondsSinceLastFee = block.timestamp - lastFeeUpdateAt;
        uint256 _accumulatedFees = accumulatedFees + configurator.borrowApr(address(this)) *
secondsSinceLastFee;
        return borrowed[user] + feeStored[user] + _newFee(user, _accumulatedFees);
    }
```

[example]

Comment by core contributors

They acknowledge the issue and have plans to address it with a future patch.

13. Front-run to cause a denial of service (DoS) when snUSD is deployed.

ID: UP-13
Type: Logic Error
File: contracts/unstable/token/snUSDBase.sol

Severity: Tips
Difficulty: Low

Issue

The _checkMinShares() function ensures that after user deposits and withdrawals, the contract maintains a minimum of 0 or at least 1e18 shares.

```
/// @notice Minimum non-zero shares amount to prevent donation attack
uint256 private constant MIN_SHARES = 1 ether;

function _checkMinShares() internal view {
  uint256 _totalSupply = totalSupply();
  require(_totalSupply >= MIN_SHARES || _totalSupply == 0, "Min shares violation");
}
```

[https://github.com/LSDfi-cafe/unstable-protocol/blob/7159949c93b45ad86b4e79fb01472960799dab13/contracts/unstable/token/snUSDBase.sol#L133]

However, this solution creates a new vulnerability. Assume a scenario where a malicious user donates nUSD to a newly created snUSD pool with a totalSupply of 0. If the malicious user donates 1 nUSD (1e18) token to the pool, the calculation for issuing MIN_SHARES (1e18 shares) would require a significant amount of nUSD due to the way the shares are calculated.

```
assets.mulDiv(totalSupply() + 10 ** _decimalsOffset(), totalAssets() + 1, rounding) > MIN_SHARES
  -> assets * (10 ** _decimalsOffset()) / (totalAssets() + 1) > MIN_SHARES
  -> assets * (10 ** 0) / (1e18(donated assets) + 1) > 1e18
  result: require(assets > 1e18 * (1e18 + 1));
```

```
    function _convertToShares(uint256 assets, Math.Rounding rounding) internal view virtual returns
(uint256) {
        return assets.mulDiv(totalSupply() + 10 ** _decimalsOffset(), totalAssets() + 1, rounding);
    }
```

[ERC4626.sol]

Recommendation

When Deploying the snUSD contract, immediately spend 1 nUSD (1e18) to mint 1 snUSD

Comment by core contributors

They are aware of the issues associated with a new erc4626 vault and agree to initialize it with some nUSD before opening up the protocol/contracts to the public. If this issue occurred during the first deployment, they plan to re-deploy.

14. When the cooldown is adjusted, users are still required to adhere to the previously established duration.

ID: UP-14
Type: Logic Error
File: contracts/unstable/token/snUSD.sol

Severity: Low
Difficulty: Low

Issue

The snUSD contract enforces cooldown periods before users can unstake their funds if the cooldown is on. If the cooldown is turned off by the admin, users can withdraw their funds immediately.

However, users who initiated a withdrawal under the cooldown period cannot withdraw immediately, even if the cooldown is turned off because their funds are still locked.

This creates an unfair situation, as they must wait unnecessarily while others can withdraw instantly.

```
function unstake(address receiver) external {
  UserCooldown storage userCooldown = cooldowns[msg.sender];
  uint256 assets = userCooldown.underlyingAmount;

  require(block.timestamp >= userCooldown.cooldownEnd, "Cooldown not finished");

  userCooldown.cooldownEnd = 0;
  userCooldown.underlyingAmount = 0;

  depot.withdraw(receiver, assets);
}

function cooldownAssets(uint256 assets, address owner) external ensureCooldownOn returns (uint256) {
  require(assets <= maxWithdraw(owner), "Excessive withdraw amount");

  uint256 shares = previewWithdraw(assets);

  cooldowns[owner].cooldownEnd = uint104(block.timestamp) + cooldownDuration;
  cooldowns[owner].underlyingAmount += assets;

  _withdraw(_msgSender(), address(depot), owner, assets, shares);

  return shares;
}

function cooldownShares(uint256 shares, address owner) external ensureCooldownOn returns (uint256) {
  require(shares <= maxRedeem(owner), "Excessive redeem amount");

  uint256 assets = previewRedeem(shares);
```

```
    cooldowns[owner].cooldownEnd = uint104(block.timestamp) + cooldownDuration;
    cooldowns[owner].underlyingAmount += assets;

    _withdraw(_msgSender(), address(depot), owner, assets, shares);

    return assets;
  }
```

[https://github.com/LSDfi-cafe/unstable-protocol/blob/7159949c93b45ad86b4e79fb01472960799dab13/contracts/unstable/token/snUSD.sol#L66-L108]

Recommendation

When the cooldown is turned off, all users should have immediate access to their funds, including those with previous withdrawal requests under the cooldown.

Comment by core contributors

In any case, if a user chooses to initiate a cooldown given the information they had at the time, it's not really "unfair" if the cooldown is later disabled, as they are unstaked based on the information and logical configuration at that time.

15. The pause state is not accounted for when withdrawing.

ID: UP-15
Severity: Medium
Type: Access & Privilege Control
Difficulty: Low
File: contracts/unstable/farms/CommunalFarm.sol

Issue

The withdrawLockedMultiple and withdrawLockedAll functions do not consider the withdrawalsPaused state variable, so users can withdraw even when the admin pauses the withdrawal process.

```
    function withdrawLocked(bytes32 kek_id) nonReentrant public {
        require(withdrawalsPaused == false, "Withdrawals paused");
        _withdrawLocked(msg.sender, msg.sender, kek_id, true);
    }

    function withdrawLockedMultiple(bytes32[] memory kek_ids) nonReentrant public {
        _getReward(msg.sender, msg.sender);
        for (uint256 i = 0; i < kek_ids.length; i++){
            _withdrawLocked(msg.sender, msg.sender, kek_ids[i], false); //don't collect rewards each
iteration
        }
    }

    function withdrawLockedAll(address user) nonReentrant public {
        _getReward(msg.sender, msg.sender);
        LockedStake[] memory locks = lockedStakes[user];
        for(uint256 i = 0; i < locks.length; i++) {
            if(locks[i].liquidity > 0 && block.timestamp >= locks[i].ending_timestamp){
                _withdrawLocked(msg.sender, msg.sender, locks[i].kek_id, false);
            }
        }
```

[https://github.com/LSDfi-cafe/unstable-protocol/blob/7159949c93b45ad86b4e79fb01472960799dab13/contracts/unstable/farms/CommunalFarm.sol#L408-L428]

Recommendation

The withdrawalsPaused state variable should be checked before the withdrawal logic execution.

Fix Comment

[21d1c38] Pause state checking logic is added to the withdrawal functions.

16. isDepegged function only checks for marketRate lower than redemptionRate

ID: UP-16 Severity: Tips
Type: Input Validation Difficulty: N/A
File: contracts/unstable/vaults/base/StakedEthVault.sol

Issue

The isDepegged function is designed to validate if a token is in a depegged state by checking if the market rate is lower than the redemption rate. However, the function currently only checks for scenarios where the market rate is less than the redemption rate.

```
    function isDepegged() public view returns (bool) {
        uint256 marketRate = getMarketRate();
        if(marketRate == 0) {
            return false; //if market rate oracle is zero, presume market rate is not configured and
take no action
        }
        uint256 redemptionRate = getRedemptionRate();
        uint256 minMarketRate = redemptionRate * (10_000 -
configurator.getDepegThreshold(address(this))) / 10_000;
        return marketRate < minMarketRate;
    }
```

[https://github.com/LSDfi-cafe/unstable-protocol/blob/7159949c93b45ad86b4e79fb01472960799dab13/contracts/unstable/vaults/base/StakedEthVault.sol#L352-L360]

Recommendation

Modify the isDepegged function to both scenarios where the market rate is significantly lower or higher than the redemption rate.

Comment by core contributors

This is intentional and not an issue. The rate actually used for internal calculations is the minimum of the marketRate and the redemptionRate - that is, in cases when the marketRate is significantly higher than the redemptionRate the price actually used is capped at the redemptionRate. Therefore, this scenario doesn't pose an issue.

DISCLAIMER

This assessment does not offer any warranties regarding the discovery of all potential issues within its scope; in essence, the evaluation results do not ensure the absence of subsequent issues. KALOS also cannot guarantee the performance of any code added to the project after the version reviewed during our assessment.
KALOS provides recommended solutions for each finding, offering guidance on how an issue may be addressed. It is important to note that while these recommendations convey ideas for resolution, they may not constitute tested or functional code. We encourage partners to view these recommendations as a starting point for discussion, with KALOS available to offer additional guidance and advice as required.Furthermore, the contents of this assessment report are intended solely for informational purposes and should not be construed as legal, tax, investment, or financial advice. KALOS neither solicits nor endorses projects based on the information contained herein.

Appendix. A

Severity Level

CRITICAL	Must be addressed as a vulnerability that has the potential to seize or freeze substantial sums of money.
HIGH	Has to be fixed since it has the potential to deny users compensation or momentarily freeze assets.
MEDIUM	Vulnerabilities that could halt services, such as DoS and Out-of-Gas, need to be addressed.
LOW	Issues that do not comply with standards or return incorrect values
TIPS	Tips that makes the code more usable or efficient when modified

Difficulty Level

	Low	**Medium**	**High**
Privilege	anyone	Miner/Block Proposer	Admin/Owner
Capital needed	Small or none	Gas fee or volatile as price change	More than exploited amount
Probability	100%	Depend on environment	Hard as mining difficulty

Vulnerability Category

Arithmetic	• Integer under/overflow vulnerability • floating point and rounding accuracy
Access & Privilege Control	• Manager functions for emergency handle • Crucial function and data access • Count of calling important task, contract state change, intentional task delay
Denial of Service	• Unexpected revert handling • Gas limit excess due to unpredictable implementation
Miner Manipulation	• Dependency on the block number or timestamp. • Frontrunning
Reentrancy	•Proper use of Check-Effect-Interact pattern. •Prevention of state change after external call • Error handling and logging.
Low-level Call	• Code injection using delegatecall • Inappropriate use of assembly code
Off-standard	• Deviate from standards that can be an obstacle of interoperability.
Input Validation	• Lack of validation on inputs.
Logic Error/Bug	• Unintended execution leads to error.
Documentation	•Coherency between the documented spec and implementation
Visibility	• Variable and function visibility setting
Incorrect Interface	• Contract interface is properly implemented on code.

End of Document