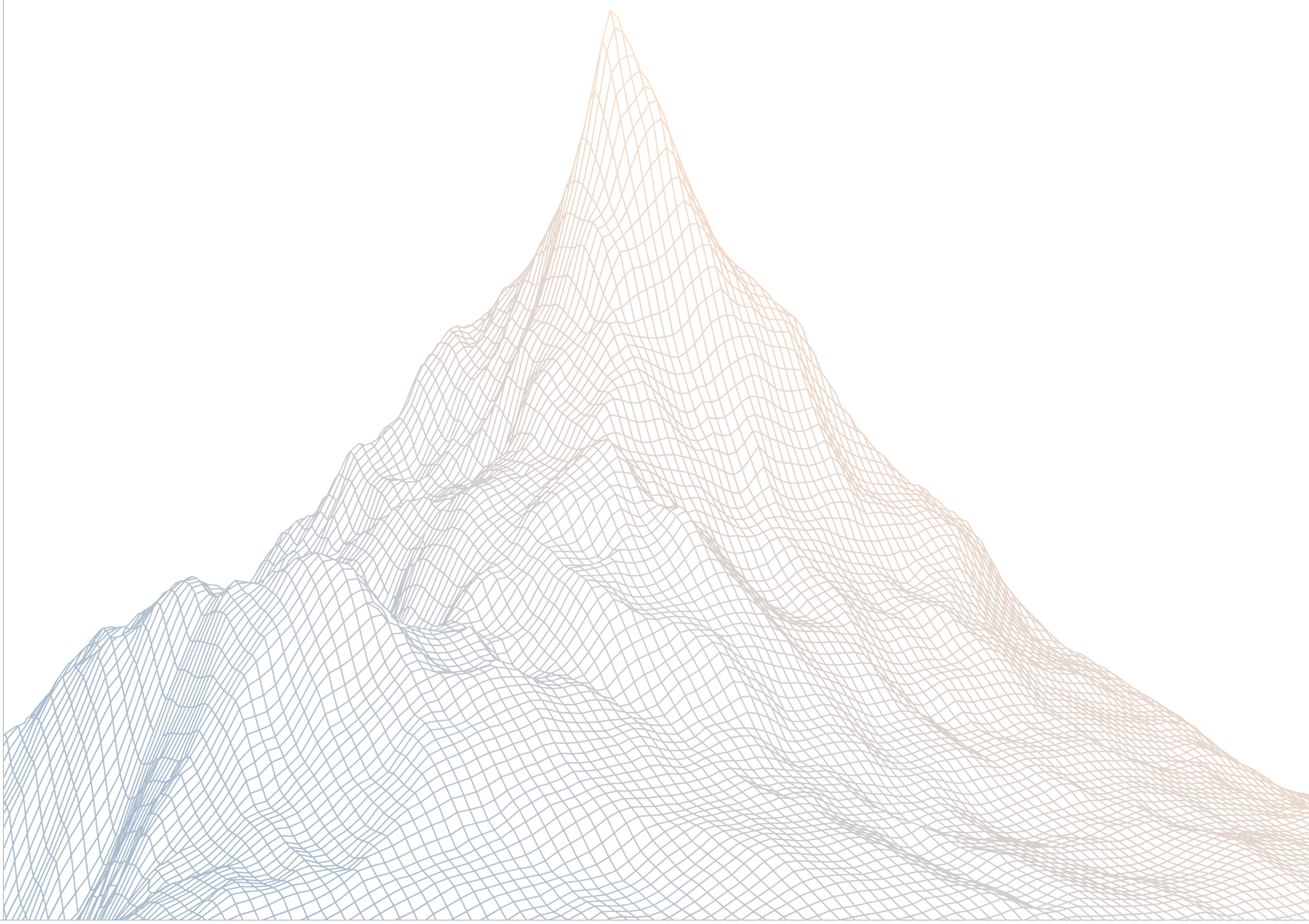
Vultisig

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: March 21th to March 23th, 2025

AUDITED BY: Ev_om
Spicymeatball

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 Vultisig

Vultisig is creating and maintaining an open source cryptocurrency wallet that takes the novel approach of providing a seedless, multi-factor, multi-chain wallet infrastructure based on MPC (Multi party computation) technology.

Distributed across all major platforms like iOS, Android, Windows and Linux users get a wallet with the enhanced security that comes from a multi-sig signing approach without the inconvenience of traditional multi-sig operations.

2.2 Scope

The engagement involved a review of the following targets:

Target	vultisig-contract
Repository	https://github.com/vultisig/vultisig-contract
Commit Hash	`6c299dd44235505c4081fa6780271cf55f873108`
Files	`Stake.sol` `StakeSweeper.sol`

2.3 Audit Timeline

March 21, 2025	Audit start
March 23, 2025	Audit end
March 24, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	2
Medium Risk	2
Low Risk	3
Informational	3
Total Issues	**10**

3

Findings Summary

ID	Description	Status
H-1	Users lose pending rewards when depositing additional tokens	Resolved
H-2	Sweep and reinvest functions are vulnerable to sandwich attacks	Acknowledged
M-1	Absence of minimum staking duration enables risk-free reward theft	Resolved
M-2	Ineffective deadline protection in _swapTokens()	Acknowledged
L-1	Unnecessary reward balance check in _claimRewards() can silently reduce user rewards	Resolved
L-2	Reward is not distributed before updating distribution parameters	Resolved
L-3	Lack of reentrancy protection in onApprovalReceived() function	Resolved
I-1	Redundant variable assignment in catch block	Resolved
I-2	Unnecessary conditional checks in reward calculation logic	Resolved
I-3	Sweeper allowance is not reset after the swap	Resolved

4 Findings

4.1 High Risk

A total of 2 high risk findings were identified.

[H-1] Users lose pending rewards when depositing additional tokens

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- Stake.sol

Description:

The `Stake.sol` contract implements a staking mechanism where users can deposit VULT tokens and earn USDC rewards proportionally to their stake. The contract uses a reward debt accounting system adapted from Sushiswap's Masterchef, which tracks user rewards through a combination of global `accRewardPerShare` and user-specific `rewardDebt` values.

The `_deposit()` function handles token deposits for users. When a user makes an additional deposit after already having staked tokens, the contract incorrectly recalculates their `rewardDebt` without first accounting for their pending rewards:

```
File: Stake.sol
    function _deposit(address _depositor, address _user, uint256 _amount) internal {
        // Update reward variables
        updateRewards();

        // Get user info
        UserInfo storage user = userInfo[_user];

        // Transfer tokens from the depositor to this contract
        stakingToken.safeTransferFrom(_depositor, address(this), _amount);

        // Update user staking amount
        user.amount += _amount;
```

```
186:         totalStaked += _amount;
187:
188:         // Update user reward debt
189:         user.rewardDebt = (user.amount * accRewardPerShare)
    / REWARD_DECAY_FACTOR_SCALING;
190:
191:         emit Deposited(_user, _amount);
192:     }
```

The problem is that when `user.rewardDebt` is recalculated on line 189, it uses the new increased `user.amount` (which includes the fresh deposit) multiplied by `accRewardPerShare`, and this completely overwrites the previous reward debt. This effectively zeroes out any pending rewards the user had accumulated before making the additional deposit.

Recommendations:

The contract should save the pending rewards before updating the reward debt and then subtract this value from the new reward debt calculation. This ensures users don't lose their accumulated rewards when making additional deposits.

Here's the correct implementation:

```
function _deposit(address _depositor, address _user, uint256 _amount)
    internal {
    // Update reward variables
    updateRewards();

    // Get user info
    UserInfo storage user = userInfo[_user];

    // Calculate pending rewards before updating user amounts
    uint256 pending = 0;
    if (user.amount > 0) {
        pending = (user.amount * accRewardPerShare)
    / REWARD_DECAY_FACTOR_SCALING - user.rewardDebt;
    }

    // Transfer tokens from the depositor to this contract
    stakingToken.safeTransferFrom(_depositor, address(this), _amount);

    // Update user staking amount
    user.amount += _amount;
    totalStaked += _amount;

    // Update user reward debt, preserving pending rewards
```

```
    user.rewardDebt = (user.amount * accRewardPerShare)
    / REWARD_DECAY_FACTOR_SCALING - pending;

    emit Deposited(_user, _amount);
}
```

This can be simplified to a simple, mathematically equivalent fix on L189:

```
user.rewardDebt += (_amount * accRewardPerShare)
    / REWARD_DECAY_FACTOR_SCALING;
```

This approach ensures that users' accumulated rewards are preserved when they make additional deposits.

Vultisig: Resolved with [PR-31]

Zenith: Verified

[H-2] Sweep and reinvest functions are vulnerable to sandwich attacks

SEVERITY: High	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: High

Target

- Stake.sol
- StakeSweeper.sol

Description:

The `_swapTokens()` function within `StakeSweeper` calculates expected swap output and applies slippage protection as follows:

```
File: StakeSweeper.sol
        try IUniswapRouter(defaultRouter).getAmountsOut(_amountIn, path) returns (uint256[] memory output) {
            amountsOut = output;
            if (amountsOut.length > 1) {
                expectedOut = amountsOut[amountsOut.length - 1];
            }
        } catch {}

        uint256 amountOutMin = expectedOut > 0 ? (expectedOut * minOutPercentage) / 100 : 1;

        // Execute swap
        uint256[] memory amounts = IUniswapRouter(defaultRouter).swapExactTokensForTokens(
            _amountIn, amountOutMin, path, _recipient, block.timestamp + 1 hours
        );
```

The issue is that `getAmountsOut()` computes expected output amounts directly from the current pool reserves, which can be manipulated through frontrunning. This makes the slippage protection (`minOutPercentage`) completely ineffective against sandwich attacks.

In a sandwich attack scenario:

1. An attacker detects a pending `sweep()` or `reinvest()` transaction
2. The attacker manipulates the pool price by executing a trade right before the user's transaction
3. The contract calculates the `expectedOut` and `amountOutMin` based on the manipulated pool state
4. The swap executes with artificially high slippage
5. The attacker executes another trade to profit from the price movement

This vulnerability directly impacts:

- The `sweep()` function in `Stake.sol`, which is intended to swap any non-staking tokens into reward tokens
- The `reinvest()` function, which swaps reward tokens back into staking tokens for users

The issue is exacerbated by the lack of access control on the `sweep()` function, making it callable by anyone who can trigger these vulnerable swaps.

Recommendations:

To address this vulnerability, implement the following changes:

1. Modify both `sweep()` and `reinvest()` functions to accept a `minAmountOut` parameter calculated off-chain
2. Update the `StakeSweeper` contract to accept and enforce these minimum output amounts
3. Restrict access to the `sweep()` function to privileged roles (e.g., owner or specified manager)

By implementing these changes, the contract would require off-chain price calculation to set proper slippage limits, making it resistant to sandwich attacks.

Vultisig: Acknowledged.

Zenith: As per private communication, the client intends to move away from using an AMM in which slippage protection is required.

4.2 Medium Risk

A total of 2 medium risk findings were identified.

[M-1] Absence of minimum staking duration enables risk-free reward theft

`SEVERITY`: Medium	`IMPACT`: Medium
`STATUS`: Resolved	`LIKELIHOOD`: Medium

Target

- Stake.sol

Description:

The reward distribution mechanism in `Stake.sol` operates as follows:

1. The `_updateRewards()` function is called manually or before any deposit, withdrawal, or claim operation
2. If `minRewardUpdateDelay` has passed since the last update or it is 0, the `_updateRewards()` updates the `accRewardPerShare` value. This also accounts for any additional rewards deposited into the contract since the last update
3. A portion of rewards (determined by `rewardDecayFactor`) is immediately available to current stakers

If `minRewardUpdateDelay` is 0, an attacker can exploit this mechanism through a sandwich attack:

1. Monitor mempool for incoming reward transfers
2. Flash loan a large amount of VULT tokens (significantly higher than existing stake)
3. Deposit these tokens to become the majority stakeholder
4. Either:
   - Call `sweep()` to process token swaps that add rewards
   - Front-run an incoming reward transfer

5. Immediately withdraw tokens and return the flash loan

This attack becomes profitable whenever the extractable rewards exceed the gas costs and potential flashloan fees. Furthermore, the attack can also be used to sandwich not only incoming rewards but also the passage of `minRewardUpdateDelay` since the last update if it is larger than 0, but then requires a multi-block sandwich attack and hence does not allow for flashloans.

In the worst case, if an attacker can flashloan an amount significantly larger than the total stake and `rewardDecayFactor` is set to 1, they could extract nearly all incoming rewards from legitimate long-term stakers.

Despite this vulnerability being explicitly acknowledged in the design choices, several factors point nevertheless to a heightened risk:

1. The large default `minRewardUpdateDelay` (1 day)
2. The small default `rewardDecayFactor` (10)
3. The contract explicitly accepts a `rewardDecayFactor` of 1, which would release 100% of rewards at once
4. Dedicated logic exists to handle the case where `rewardDecayFactor` equals 1

Recommendations:

The contract should implement a minimum staking duration to prevent sandwich attacks. This can be done by tracking deposit timestamps and enforcing a minimum lock period before withdrawals.

To only protect against flashloan attacks but not against multi-block value extraction attacks, which may be deemed an acceptable risk provided the reward amounts are always relatively small, it is enough to disallow a `minRewardUpdateDelay == 0` or to change the logic in L#111 and L#114 to:

```
bool timeDelayMet = (block.timestamp > lastRewardUpdateTime
    + minRewardUpdateDelay);
if (currentRewardBalance > lastRewardBalance && timeDelayMet) {
```

This will prevent `accRewardPerShare` from updating more than once per block.

Additionally, it is recommended to set a large enough `rewardDecayFactor` and small enough `minRewardUpdateDelay` to prevent this kind of attack.

Vultisig: Resolved with PR-29

Zenith: Verified. The risk of flashloan attack was mitigated as per our recommendation.

[M-2] Ineffective deadline protection in `_swapTokens()`

SEVERITY: Medium	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Medium

Target

- StakeSweeper.sol

Description:

The `StakeSweeper` contract is used by the `Stake` contract to sweep tokens and convert them to either reward tokens or staking tokens through interactions with the Uniswap router. The `_swapTokens()` function that executes the swaps sets the deadline parameter when calling the Uniswap router as follows:

```
File: StakeSweeper.sol
114:         uint256[] memory amounts
    = IUniswapRouter(defaultRouter).swapExactTokensForTokens(
115:             _amountIn, amountOutMin, path, _recipient, block.timestamp
    + 1 hours
116:         );
```

The fundamental problem is that `block.timestamp` represents the timestamp of the block in which the transaction is eventually executed, not when the transaction is submitted. When the Uniswap router processes this transaction, it compares the deadline against the same `block.timestamp` value, making the deadline check completely ineffective.

This means:

1. The deadline check in the Uniswap router becomes meaningless since `block.timestamp + 1 hour` will always be greater than `block.timestamp` at execution time.
2. A transaction could remain pending in the mempool for days or weeks, and still be valid when eventually mined.
3. Users have no protection against transactions being executed at unfavorable times after market conditions have changed.

This vulnerability affects both the `sweep()` and `reinvest()` functions in the `Stake` contract.

> [!NOTE] Slippage protection, if correctly implemented, is calculated based on the expected output at the time the transaction is submitted, not when it is eventually executed. If prices move significantly during the waiting period, this protection becomes less effective - this is the reason the `deadline` parameter exists.

Recommendations:

The deadline parameter should be set by the caller at the time of transaction submission, not dynamically calculated at execution time:

1. Modify the `_swapTokens()` function to accept and use a deadline parameter
2. Update the related functions in the `Stake` contract to pass a user-defined deadline to the `StakeSweeper` contract.

Alternatively, simply use `block.timestamp` as a deadline if the risk of adverse price movements on stale transactions is deemed acceptable.

Vultisig: Acknowledged

4.3 Low Risk

A total of 3 low risk findings were identified.

[L-1] Unnecessary reward balance check in `_claimRewards()` can silently reduce user rewards

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- Stake.sol

Description:

In the `_claimRewards()` function, there's a check that compares the user's pending rewards against the current contract balance of reward tokens:

```
File: Stake.sol
532:         uint256 currentRewardBalance
    = rewardToken.balanceOf(address(this));
533:         uint256 rewardAmount = pending > currentRewardBalance ?
    currentRewardBalance : pending;
```

This check is problematic for two reasons:

1. It's an unnecessary check that shouldn't occur in normal contract operations. The contract should always have sufficient reward tokens to cover all pending rewards, as the rewards are calculated based on tokens that have already been accounted for in `accRewardPerShare`.

2. If this condition is triggered (pending > currentRewardBalance), it silently reduces the user's reward amount to match the available balance instead of reverting the transaction. This means users would lose out on rewards they're entitled to receive without any indication that something went wrong.

Under normal circumstances, this check should never be triggered because the contract's accounting of rewards should ensure that the sum of all pending rewards doesn't exceed the contract's balance. If this check does get triggered, it indicates a fundamental accounting error in the contract that should be investigated rather than silently handled.

Recommendations:

Remove the balance check and simply assign the pending amount to the reward amount. If there's truly an issue with insufficient balance, it's better for the transaction to fail transparently so the issue can be identified and addressed.

Vultisig: Resolved with PR-28

Zenith: Verified

[L-2] Reward is not distributed before updating distribution parameters

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- Stake.sol#L405-L419

Description:

The staking contract owner can update the following parameters:

- `rewardDecayFactor`
- `minRewardUpdateDelay`

Both parameters determine how rewards are distributed to depositors. However, the contract does not distribute the current reward before updating these parameters.

For example:

- If there are 10,000 USDC in rewards and the current decay factor is 10, 1,000 USDC should be distributed during this stage.
- If the decay factor is then changed to 20 before distribution, only 500 USDC will be distributed instead, effectively reducing the expected reward.

Recommendations:

To prevent this issue, the contract should call `updateRewards` before modifying these parameters. This ensures that changes only affect future reward distributions and do not retroactively alter expected payouts.

Vultisig: Resolved with PR-32

Zenith: Verified

[L-3] Lack of reentrancy protection in `onApprovalReceived()` function

`SEVERITY`: Low	`IMPACT`: Low
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- Stake.sol

Description:

The `Stake` contract implements the `IERC1363Spender` interface to handle token approvals and deposits in a single transaction through the `onApprovalReceived()` function. This function can be called by invoking `approveAndCall()` on the staking token contract.

While most state-changing functions in the contract are protected against reentrancy attacks via the `nonReentrant` modifier, the `onApprovalReceived()` function lacks this protection. This function has effectively the same functionality as the `deposit()` function, which is protected by the `nonReentrant` modifier through its call to `depositForUser()`.

While this inconsistency in reentrancy protection doesn't appear to be directly exploitable in the current implementation, it represents a potential security risk if the contract is modified in the future.

Recommendations:

Add the `nonReentrant` modifier to the `onApprovalReceived()` function to maintain consistency with other state-changing functions in the contract.

Vultisig: Resolved with PR-27

Zenith: Verified

4.4 Informational

A total of 3 informational findings were identified.

[I-1] Redundant variable assignment in catch block

`SEVERITY`: Informational	`IMPACT`: Informational
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- Stake.sol

Description:

In the `migrate()` function of the Stake contract, there is an unnecessary assignment in the catch block that sets `migrationSuccess = false`. This is redundant because the variable is already initialised to `false` before the try-catch block and is only changed to `true` if the try block succeeds. This issue doesn't affect the contract's functionality but represents inefficient code and a minor gas waste.

Recommendations:

Remove the redundant assignment in the catch block, as the variable will already have the correct value if an exception occurs.

Vultisig: Resolved with PR-25

Zenith: Verified

[I-2] Unnecessary conditional checks in reward calculation logic

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- Stake.sol

Description:

There are several redundant conditional checks in the reward calculation logic that create unnecessary code complexity without adding any functional value:

1. In the `_updateRewards()` function, there's a condition checking if `minRewardUpdateDelay == 0`. This condition will never be reached because in that case, `timeDelayMet` will always be true.

```
if (currentRewardBalance > lastRewardBalance && (timeDelayMet ||
    minRewardUpdateDelay == 0)) {
```

2. Similarly, in the reward calculation logic, a ternary operator is used that has no use and could be simplified to `totalNewRewards / rewardDecayFactor`:

```
uint256 releasedRewards = rewardDecayFactor == 1 ? totalNewRewards :
    totalNewRewards / rewardDecayFactor;
```

3. The `pendingRewards()` function contains similar logical inefficiencies as well as inconsistencies with the `_updateRewards()` function, creating potential confusion during code maintenance and future updates.

While these issues don't have security implications, they reduce code clarity and efficiency.

Recommendations:

Simplify the conditional statements to remove redundancy and improve code clarity.

Vultisig: Resolved with [PR-26]

Zenith: Verified

[I-3] Sweeper allowance is not reset after the swap

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- StakeSweeper.sol#L118

Description:

The following line does not correctly reset the token allowance to the router if it wasn't fully consumed: `IERC20(_tokenIn).safeIncreaseAllowance(defaultRouter, 0)`

Recommendations:

If the goal is to reset the allowance to zero, consider using `forceApprove` from the same library instead.

Vultisig: Resolved with PR-30

Zenith: Verified