



Pelz

COINSAFE SECURITY REPORT

Prepared For
COINSAFE

Prepared By
PELZ

OCTOBER 2025

Introduction

A time-boxed security review of the Coinsafe protocol was done by Pelz, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About Coinsafe

CoinSafe is a smart savings platform built using the EIP-2535 Diamond Standard, enabling modular and upgradeable architecture. It allows users to save supported tokens through flexible savings options, including Emergency Savings, Targeted Savings, and Automated Savings. The protocol incentivizes disciplined saving while providing customizable deposit strategies over fixed periods.

Severity Classification

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

Impact

- High - Leads to a significant loss of assets in the protocol or significantly harm a group of users
- Medium - Only a small amount of funds is lost or core contract functionality is broken or affected
- Low - Can lead to any kind of unexpected behaviour with no major impact

Likelihood

- High - Attack path is possible with reasonable assumptions that mimic on chain conditions and the cost of the attack is relatively low compared to the value lost or stolen

- Medium - Only a conditionally incentivised attack vector but still likely
- Low - Has too many or too unlikely assumptions

Actions Required For Severity Levels

- High - Must fix (before deployment, if not already deployed)
- Medium - Should fix
- Low - Could fix

Security Assessment Summary

review commit hash:

- [b900a672d2e5369018955fc622c13066fec45a0a](#)

The following number of issues were found, categorised by their severity:

Critical & High: 3 issues

Medium : 1 issue

Low : 3 issues

Findings Summary

ID	Title	Severity	Status
[H-01]	Use of ReentrancyGuard introduces storage collision in Diamond Standard	High	Resolved
[H-02]	Public <code>deactivateSafe</code> function allows anyone to reset another user's streaks and rewards	High	Resolved
[H-03]	Users can farm reward points by repeatedly topping up with minimal amounts	High	Resolved
[M-01]	Public <code>activateSafe</code> function allows anyone to activate another user's savings plan	Medium	Resolved
[L-01]	Off-by-one error allows token addition during plan unlock	Low	Resolved
[L-02]	Flashloan can bypass balance check in <code>depositToPool</code>	Low	Resolved
[L-03]	<code>topUpSafe</code> allows top-up when safe is already unlocked due to off-by-one error	Low	Resolved

[H-01] Use of ReentrancyGuard introduces storage collision in Diamond Standard

Severity

Impact: High

Likelihood: High

Description

The `AutomatedSavingsFacet` inherits from `ReentrancyGuard`, which introduces its own state variable `_status`. In a Diamond Standard architecture, facets are not allowed to define their own storage variables unless they use a shared storage pattern with a fixed slot.

Here, the `ReentrancyGuard` contract defines the following state:

```
uint256 private _status;
```

This creates a **storage conflict**, because the Diamond proxy and other facets may already be using the same storage slot, leading to unexpected behavior or data corruption.

For example, if storage slot 0 is used for a critical variable in the diamond, inheriting `ReentrancyGuard` will overwrite it during construction or execution, causing subtle and dangerous bugs.

Recommendations

- Remove inheritance of `ReentrancyGuard` in all facets.
- Use a **custom reentrancy guard** that stores its variable in a dedicated storage slot using a library with an inline assembly layout (e.g., `LibReentrancyGuardStorage` with `keccak256("your.project.reentrancy.guard")` as the slot).
- Review all functions currently using `nonReentrant` and migrate them to the new guard implementation.

[H-02] Public `deactivateSafe` function allows anyone to reset another user's streaks and rewards

Severity

Impact: High

Likelihood: High

Description

The `deactivateSafe` function is declared `public`, even though the comment specifies it should be `internal`:

```
/**  
 * @dev This internal function:  
 * ...
```

```
/*
function deactivateSafe(address _user) public {
```

Since there is **no access control**, any user can call this function to:

- Deactivate another user's automated savings plan
- Remove them from the active savings array
- Reset their **StreakData**, including:
 - `currentStreak`
 - `lastSuccessfulSave`
 - `longestStreak`
 - All `saveTimestamps`
- Set their multiplier to zero

This means a malicious actor can **repeatedly attack active users** to prevent them from maintaining savings streaks and earning rewards.

Recommendations

- Change the function visibility from `public` to `internal`.
- If it must be externally callable, add proper access control to ensure only the **user themselves** or a **trusted automation system** can call it.

[H-03] Users can farm reward points by repeatedly topping up with minimal amounts

Severity

Impact: High

Likelihood: High

Description

In the `topUpSafe` function, users receive reward points through:

```
CustomLib.awardPoints(msg.sender, diamond.stackersPointWeight);
```

This reward is granted **regardless of the `_amount` deposited**. This means users can repeatedly call `topUpSafe` with **tiny amounts** (e.g., 1 wei or 1 token unit), and still earn full points on each call.

Since there's no cooldown, threshold, or dynamic point scaling based on amount, this allows malicious actors to **farm points at near-zero cost** and gain an unfair advantage in the rewards system.

Example

A user can automate:

```
for (uint256 i = 0; i < 1000; i++) {
    topUpSafe(safeId, token, 1); // deposit 1 unit each time
}
```

They would receive `1000 * stackersPointWeight` points for essentially dust deposits.

Recommendations

- Implement a **minimum threshold** for `_amount` to qualify for points (e.g., `>= 1e18` or a reasonable token-specific value).
- Alternatively, make the awarded points **proportional to `_amount`** deposited:

```
CustomLib.awardPoints(msg.sender, _amount / 1e18 * baseWeight);
```

- Add a **cooldown** or **daily cap** on how many times points can be awarded per user.

[M-01] Public `activateSafe` function allows anyone to activate another user's savings plan

Severity

Impact: Medium

Likelihood: High

Description

The `activateSafe` function is marked `public`, but the comment clearly indicates it was meant to be an **internal** function:

```
/**
 * @dev This internal function:
 * ...
 */
function activateSafe(address _user) public {
```

Because it is public and lacks any access control, **any external account can call this function and activate the automated savings plan for any user**, even if the user did not initiate the activation.

This could result in:

- Users being added to the `automatedSavingsUsers` array without consent
- Plans marked as active

Recommendations

- Change the function visibility from `public` to `internal`.
- If external access is required under controlled conditions, wrap the logic inside a properly protected external function with appropriate checks (e.g., only the user or a privileged role can call it).

[L-01] Off-by-one error allows token addition during plan unlock

Severity

Impact: Low

Likelihood: Low

Description

In the `addTokenToAutomatedPlan` function, there's a logic check that attempts to block users from adding tokens after the plan has expired:

```
if (block.timestamp > plan.unlockTime) {
    revert LibEventsAndErrors.PlanExpired();
}
```

However, this condition still allows users to add tokens **at the exact moment of unlock**, i.e., when `block.timestamp == plan.unlockTime`. This violates the expected behavior where users should not be able to add tokens **once the plan unlocks**.

Example

Assume `plan.unlockTime = 1690000000`.

If `block.timestamp == 1690000000`, the condition passes, and the user can still add a token even though the plan should be considered unlocked.

This off-by-one logic could lead to unintended behavior, like users adding tokens to plans that are no longer active.

Recommendations

- Change the check to:

```
if (block.timestamp >= plan.unlockTime) {
    revert LibEventsAndErrors.PlanExpired();
}
```

[L-02] Flashloan can bypass balance check in `depositToPool`

Severity

Impact: Low

Likelihood: Medium

Description

In the `depositToPool` function, the following check is used to confirm the user has enough tokens:

```
if (IERC20(_token).balanceOf(msg.sender) < _amount) {
    revert LibEventsAndErrors.InsufficientFunds();
}
```

This check is **ineffective** because it can be **bypassed using a flash loan**. A user can momentarily inflate their balance within the same transaction, pass the check, and proceed to deposit — even though they don't truly own the funds.

Since the actual transfer is performed via `safeTransferFrom`, which enforces balance and allowance at the time of transfer, this check is redundant and potentially misleading.

Recommendations

- Remove this balance check entirely, as it gives a false sense of security.
- Rely on `safeTransferFrom` to enforce actual transferability.

[L-03] `topUpSafe` allows top-up when safe is already unlocked due to off-by-one error

Severity

Impact: Low

Likelihood: Medium

Description

In the `save()` function, the intention is to only allow top-ups **before the safe is unlocked**, using the condition:

```
if (likelySafe.unlockTime > block.timestamp) {
    topUpSafe(...);
} else {
    revert SafeIsUnlocked();
}
```

This clearly treats the **unlock time itself** as the point when no further top-ups should occur i.e., once `block.timestamp == unlockTime`, the safe is considered **unlocked**.

However, in the `topUpSafe()` function, the condition is written as:

```
if (likelySafe.unlockTime < block.timestamp) {
    revert SafeIsUnlocked();
}
```

This allows users to **top up at the exact unlock time**, which directly contradicts the logic in `save()`. As a result:

- `save()` will revert if `block.timestamp == unlockTime`
- But a direct call to `topUpSafe()` will still succeed at the same time

Recommendations

- Align the condition in `topUpSafe()` with `save()` by changing the check to:

```
if (block.timestamp >= likelySafe.unlockTime) {
    revert SafeIsUnlocked();
}
```

[I-01] Redundant zero address check on `msg.sender`

Description

The `depositToPool` function contains a redundant and impossible check:

```
if (msg.sender == address(0)) {
    revert LibEventsAndErrors.AddressZeroDetected();
}
```

In Solidity, `msg.sender` **can never be the zero address** in any legitimate transaction or call context. This line introduces unnecessary code and gas usage, and may indicate misunderstanding of the EVM behavior.

Recommendations

- Remove this check.
- Review the rest of the codebase for similar unnecessary `msg.sender == address(0)` checks and clean them up.

[I-02] Unnecessary allowance check before `safeTransferFrom`

Description

The following code checks the user's allowance manually:

```
uint256 allowance = IERC20(_token).allowance(msg.sender, address(this));  
if (allowance < _amount) {  
    revert LibEventsAndErrors.InsufficientAllowance();  
}
```

However, this is unnecessary because `safeTransferFrom` (via OpenZeppelin's SafeERC20) already performs this check and reverts if the allowance is insufficient. This makes the manual check redundant and wasteful.

Recommendations

- Remove the manual allowance check.
- Let `safeTransferFrom` handle the allowance verification during the actual token transfer. Confirmed, Pelz — this is a clear **inconsistency in ownership enforcement**. Mixing custom `onlyOwner` modifiers with `LibDiamond.enforceIsContractOwner()` in the same facet is not just stylistically messy — it risks future logic drift and bugs if the ownership logic ever changes.

[I-03] Inconsistent ownership enforcement in `OwnershipFacet.sol`

Description

The `OwnershipFacet.sol` contract uses two different approaches to enforce owner-only access:

1. Custom modifier:

```
modifier onlyOwner() {  
    require(msg.sender == LibDiamond.contractOwner(), "Unauthorized  
caller");  
    _;  
}
```

Used in:

```
function getContractBalance(...)  
function withdrawFunds(...)
```

2. Library-based check:

```
LibDiamond.enforceIsContractOwner();
```

Used in:

```
function addAcceptedToken(...)
```

While both methods are intended to restrict access to the contract owner, **maintaining two separate mechanisms introduces inconsistency** and potential risk — especially if the ownership logic inside `LibDiamond.contractOwner()` or `enforceIsContractOwner()` is ever updated in the future.

Recommendations

- **Standardize** ownership enforcement throughout the codebase.
- Remove the `onlyOwner` modifier and use `LibDiamond.enforceIsContractOwner()` consistently across all owner-restricted functions in this facet.