Guild Audits

MAY 30TH, 2025

COINSAFE INITIAL AUDIT REPORT

By Guild Audits

TABLE OF CONTENTS

DISCLAIMER

The Guild Audit team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

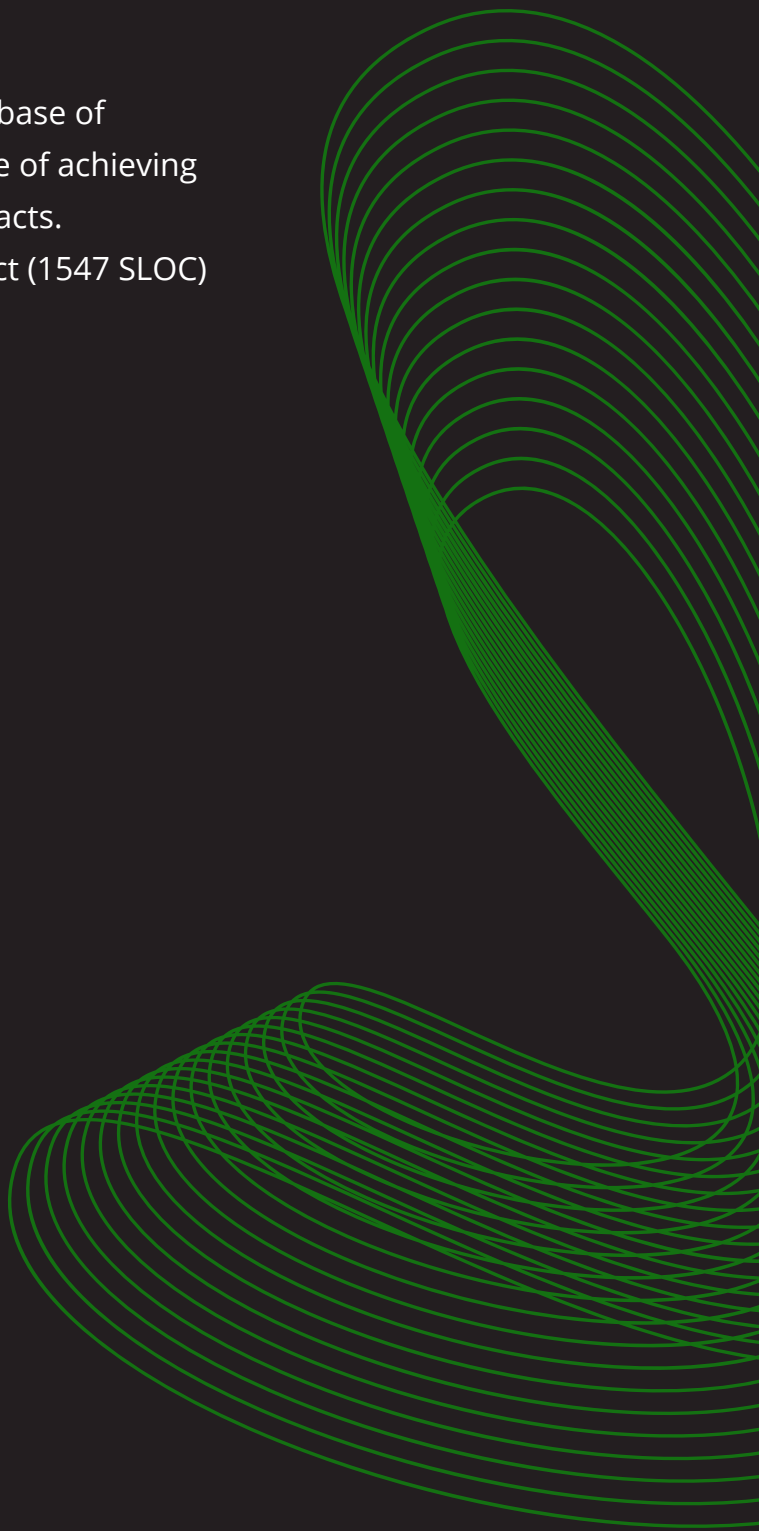
EXECUTIVE SUMMARY

This section will represent the summary of the whole audit.

PROJECT AUDIT SCOPE AND FINDINGS

The motive of this audit is to review the codebase of MulticallWithPermit contracts for the purpose of achieving secured, correctness and quality smart contracts.
Number of Contracts in Scope: All the contract (1547 SLOC)
Duration for Audit: 14 days

Vulnerability Summary:
Total issues: 8
Total High: 1
Total Medium: 4
Total Low: 3



FINDINGS

Index	Title	Severity	Status of Issues
01	[H-01]: Improper Use of LibDiamond.DiamondStorage for Application State Hinders Upgradeability and Modularity	High	Open
02	[M-01]: Unbounded Loop in AutomatedSavingsFacet::deactivateSafe() for User Removal Can Lead to High Gas Costs	Medium	Open
03	[M-02]: Ambiguous Streak Logic in AutomatedSavingsFacet::applyStreak() with Multiple Token Frequencies	Medium	Open
04	[M-03]: Unbounded Loop in AutomatedSavingsFacet::getAndExecuteAutomatedSavingsPlansDue() May Lead to Denial of Service	Medium	Open
05	[M-04]: Incorrect User Context in AutomatedSavingsFacet::deactivateSafe() Fails to Deactivate User Plans	Medium	Open
06	[I-01] - Redundant and Inefficient Storage Access in AutomatedSavingsFacet.sol	Informational	Open
07	[I-02]: Redundant Zero Value Check in AutomatedSavingsFacet::createAutomatedSavingsPlan()	Informational	Open
08	[I-03]: Incorrect User Context in AutomatedSavingsFacet::activateSafe()	Informational	Open

MODE OF AUDIT AND METHODOLOGIES

The mode of audit carried out in this audit process is as follows:
Manual Review: This is the first and principal step carried out to understand the business logic behind a project. At this point, it involves research, discussion and establishment of familiarity with contracts. Manual review is critical to understand the nitty-gritty of the contracts.
Automated Testing: This is the running of tests with audit tools to further cement discoveries from the manual review. After a manual review of a contract, the audit tools used are Slither, Echidna and others.
Functional Testing: Functional testing involves manually running unit, static, and dynamic testing. This is done to find out possible exploit scenarios that could be used to exploit the contracts.

The methodologies for establishing severity issues:

- High Level Severity Issues
- Medium Level Severity Issues
- Low Level Severity Issues
- Informational Level Severity Issues

TYPES OF SEVERITY

Every issue in this report has been assigned a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

These are critical issues that can lead to a significant loss of funds, compromise of the contract's integrity, or core function of the contract not working. Exploitation of such vulnerabilities could result in immediate and catastrophic consequences, such as:

- Complete depletion of funds.
- Permanent denial of service (DoS) to contract functionality.
- Unauthorized access or control over the contract

Medium Severity Issues

These issues pose a significant risk but require certain preconditions or complex setups to exploit. They may not result in immediate financial loss but can degrade contract functionality or pave the way for further exploitation. Exploitation of such vulnerabilities could result in partial denial of service for certain users,leakage of sensitive data or unintended contract behavior under specific circumstances.

Low Severity Issues

These are minor issues that have a negligible impact on the contract or its users. They may affect efficiency, readability, or clarity but do not compromise security or lead to financial loss. Impacts are minor degradation in performance,confusion for developers or users interacting with the contract and low risk of exploitation with limited consequences.

Informational

These are not vulnerabilities in the strict sense but observations or suggestions for improving the contract's code quality, readability, and adherence to best practices.

There is no direct impact on the contract’s functionality or security, it is aimed at improving code standards and developer understanding.

TYPES OF ISSUES

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the audit and have been successfully fixed.

Acknowledged

Vulnerabilities that have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

REPORT OF FINDINGS

HIGH SEVERITY ISSUE

[H-01]: Improper Use of LibDiamond.DiamondStorage for Application State Hinders Upgradeability and Modularity

- **Summary:**

The LibDiamond.DiamondStorage struct in LibDiamond.sol is currently used to store a wide range of application-specific state variables (e.g., totalSupply, userBalances, automatedSavingsPlans). This approach deviates from recommended Diamond Standard storage patterns, centralizing all application state into the core diamond library's storage slot. This can lead to significant upgradeability challenges, increased risk of storage collisions, and reduced modularity of facets.

- **Description**

The Diamond Standard, as defined in EIP-2535, provides LibDiamond primarily for managing facet cuts, ownership, and the core diamond proxy logic. Its DiamondStorage (typically located at DIAMOND_STORAGE_POSITION) is intended for variables essential to the diamond's operation, such as selectorToFacetAndPosition, facetFunctionSelectors, facetAddresses, and contractOwner.
The current implementation in c:\Users\DELL\Documents\AUDIT\Coinsafe_contract-diamond\contracts\libraries\LibDiamond.sol extends LibDiamond.DiamondStorage to include numerous custom application state variables:

Status: --

```
// In LibDiamond.DiamondStorage
// ... standard diamond storage variables ...
address contractOwner;
// ============================ CUSTOM VARIABLES ============================
//@audit-issue All these custom variables are not supposed to be stored in the libDiamond.sol storage...
uint256 totalSupply;
string name;
string symbol;
// ... and many other application-specific variables like:
// address[] automatedSavingsUsers;
// mapping(address => mapping(address => uint256)) userAvailableBalances;
// mapping(address => AutomatedSavingsPlan) automatedSavingsPlans;
// mapping(address => StreakData) userStreaks;
```

This design has several drawbacks:

- **Upgradeability Issues:** If you need to modify or add state variables for a specific piece of functionality (e.g., add a new field to AutomatedSavingsPlan), you would be modifying the global LibDiamond.DiamondStorage struct. This makes upgrades more complex and riskier. If the layout of this central struct changes, it can affect all facets, even those unrelated to the change. It also makes it harder to reason about storage layout during upgrades.

- **Storage Collisions:** While using a single struct under a specific slot (DIAMOND_STORAGE_POSITION) avoids direct slot collisions between facets, it centralizes the risk. Any error in managing this large, shared struct can have widespread consequences.

- **Reduced Modularity:** The Diamond Standard promotes modularity by allowing facets to be developed and upgraded independently. When application state is tightly coupled within LibDiamond.DiamondStorage, facets become less independent. A facet's state is not self-contained but part of a global monolith.

- **Deviation from Best Practices:**

Facet-Specific Diamond Storage: A recommended pattern is for each facet (or group of related facets) to define its own storage struct and use a unique storage slot (derived from a unique string hash, e.g., keccak256("com.mycompany.automatedsavings.storage")). This isolates the facet's state.

AppStorage Pattern: For state that needs to be shared across multiple facets, the AppStorage pattern is often preferred. This involves defining a dedicated storage struct (e.g., AppStorage) at a well-known storage slot (e.g., keccak256("com.mycompany.app.storage")). Facets can then import and use this AppStorage struct. This is cleaner than embedding all application state directly into LibDiamond.

By placing all custom variables within LibDiamond.DiamondStorage, the contract essentially uses LibDiamond not just for diamond mechanics but as a global state container, which is not its primary design purpose and negates many benefits of the diamond pattern.

Impact Explanation

Critical. This architectural choice severely impacts the long-term maintainability and upgradeability of the diamond.

- **Upgrade Complexity and Risk:** Future upgrades that require state changes will be more difficult and error-prone. Changing the LibDiamond.DiamondStorage struct requires careful consideration of all facets. A mistake could lead to data corruption or render parts of the contract unusable.

- **Hindered Facet Independence**: It becomes harder to add, remove, or replace facets without potentially affecting or needing to understand the entire global state structure.

The core promise of easier and safer upgrades with diamonds is significantly undermined by this storage approach.

Likelihood Explanation

The system is currently built this way. Any future development or upgrade involving state changes will directly encounter the complexities and risks introduced by this centralized storage pattern.

Proof of Concept

You can read more from the Diamond Storage and AppStorage patterns.

Recommendation

Refactor the storage architecture to align with Diamond Standard best practices for state management. Choose one or a combination of the following:

1. Facet-Specific Diamond Storage (Recommended for isolated state):

- For state variables primarily used by a single facet (or a very tightly coupled group of facets, like AutomatedSavingsFacet), create a separate storage struct within that facet's context.
- Define a unique storage slot for this struct using keccak256 of a unique string.
- Example for AutomatedSavingsFacet:

```
// In AutomatedSavingsFacet.sol (or a dedicated library for its storage)
library AutomatedSavingsStorageLib {
    bytes32 constant AUTOMATED_SAVINGS_STORAGE_POSITION = keccak256("io.coinsafe.storage.automatedsavings");

    struct AutomatedSavingsStorage {
        mapping(address => LibDiamond.AutomatedSavingsPlan) automatedSavingsPlans;
        address[] automatedSavingsUsers;
        mapping(address => bool) isInAutomatedSavingsArray;
        mapping(address => LibDiamond.StreakData) userStreaks;
        // ... other state specific to automated savings
    }

    function automatedSavingsStorage() internal pure returns (AutomatedSavingsStorage storage rss) {
        bytes32 position = AUTOMATED_SAVINGS_STORAGE_POSITION;
        assembly {
            rss.slot := position
        }
    }
}
```

AutomatedSavingsFacet would then call AutomatedSavingsStorageLib.automatedSavingsStorage() to access its dedicated state.

2. **AppStorage Pattern (Recommended for shared state):**

- If certain state variables truly need to be shared across multiple, distinct facets (e.g., userBalances, acceptedTokens), group them into a dedicated AppStorage struct.
- Place this AppStorage at its own unique storage slot.
- Facets that need to access this shared state can import and use the AppStorage library.
- Example:

```
// In a new file, e.g., AppStorage.sol
library AppStorageLib {
    bytes32 constant APP_STORAGE_POSITION = keccak256("io.coinsafe.storage.app");

    struct AppStorage {
        mapping(address => bool) acceptedTokens;
        address[] acceptedTokenAddresses;
        mapping(address => mapping(address => uint256)) userAvailableBalances;
        // ... other genuinely shared state
    }

    function appStorage() internal pure returns (AppStorage storage ds) {
        bytes32 position = APP_STORAGE_POSITION;
        assembly {
            ds.slot := position
        }
    }
}
```

3. **Cleanup LibDiamond.DiamondStorage:**

- Remove all application-specific state variables from LibDiamond.DiamondStorage. It should only contain variables essential for the EIP-2535 diamond mechanics (selectors, facet addresses, owner, etc.).

Transition Plan:

Migrating existing state will be a significant effort and must be planned carefully, potentially involving a contract upgrade that reads old state and writes it to the new storage locations.

By adopting these patterns, you will:

- Greatly improve the modularity and upgradeability of your diamond.
- Reduce the risk of unintended state interactions between facets.
- Make the codebase easier to understand and maintain.

Status: Open

MEDIUM SEVERITY ISSUE

[M-01]: Unbounded Loop in AutomatedSavingsFacet::deactivateSafe() for User Removal Can Lead to High Gas Costs

- **Summary:**

The deactivateSafe() function iterates over the diamond.automatedSavingsUsers array to find and remove a user. If this array is very large, the gas cost of this iteration can become significant, potentially making plan deactivation expensive or, in extreme cases, fail due to out-of-gas.

- **Description**

When a user's automated savings plan is deactivated via deactivateSafe(address _user), the function attempts to remove the _user from the diamond.automatedSavingsUsers array. This is done by iterating through the array to find the user's index:

```
// In deactivateSafe(address _user)
address[] storage usersInArray = diamond.automatedSavingsUsers;

for (uint256 i = 0; i < usersInArray.length; i++) {
    if (usersInArray[i] == _user) {
        // Replace with the last element and then remove the last element
        usersInArray[i] = usersInArray[usersInArray.length - 1];
        usersInArray.pop();
        diamond.isInAutomatedSavingsArray[_user] = false; // This is correctly set to false
        break;
    }
}
// diamond.isInAutomatedSavingsArray[_user] = false; // This line is redundant if the user is found and removed. If not found, it correctly sets it.
```

The "swap and pop" technique used for removal (usersInArray[i] = usersInArray[usersInArray.length - 1]; usersInArray.pop();) is gas-efficient for the removal itself (O(1) once the index is known). However, the for loop to find the index i is O(N) in the worst case, where N is the length of usersInArray.

If diamond.automatedSavingsUsers contains a large number of addresses, iterating through it to find the specific _user can consume a substantial amount of gas. While less critical than the unbounded loop in getAndExecuteAutomatedSavingsPlansDue() (as deactivateSafe is typically called for a single user and less frequently by a keeper for expired plans), it can still lead to high transaction costs for users or keepers.

impact Explanation

For a large number of users, deactivating a plan could become unexpectedly expensive due to the linear search. In extreme scenarios with a very large user base, it might approach gas limits, though this is less likely to cause a complete DoS of the function compared to the getAndExecuteAutomatedSavingsPlansDue issue, as it's usually a one-off operation per user deactivation. However, it degrades user experience with high gas fees and poses a risk if called within another function that has its own gas constraints (like getAndExecuteAutomatedSavingsPlansDue calling it for expired plans).

Recommendation

To avoid the O(N) search, maintain a separate mapping that stores the index of each user within the diamond.automatedSavingsUsers array.

1. Add a new mapping to **LibDiamond.DiamondStorage:**

```
// In LibDiamond.DiamondStorage// mapping(address => uint256)
userToArrayIndex;
```

2. When adding a user to **diamond.automatedSavingsUsers** in **activateSafe()** (or plan creation), store their index:

```
diamond.userToArrayIndex[_user] = diamond.automatedSavingsUsers.length - 1;
```

3. **In deactivateSafe():**

Retrieve the index directly:

```
uint256 indexToRemove = diamond.userToArrayIndex[_user];
```

- Perform the swap and pop:

```
address lastUser = usersInArray[usersInArray.length - 1];
usersInArray[indexToRemove] = lastUser;
diamond.userToArrayIndex[lastUser] = indexToRemove;
usersInArray.pop();
delete diamond.userToArrayIndex[_user];
```

This makes the finding and removal operation O(1).

Diff Suggestion for deactivateSafe and Related activateSafe (assuming userToArrayIndex is added to LibDiamond.DiamondStorage):

```
        if (diamond.isInAutomatedSavingsArray[_user]) {
            LibDiamond.AutomatedSavingsPlan storage plan = diamond
-               .automatedSavingsPlans[msg.sender];
+               .automatedSavingsPlans[_user]; // Assumed prior fix for user context
            LibDiamond.StreakData storage streak = diamond.userStreaks[_user];
            plan.isActive = false;

            address[] storage usersInArray = diamond.automatedSavingsUsers;
-
-           for (uint256 i = 0; i < usersInArray.length; i++) {
-               if (usersInArray[i] == _user) {
-                   // Replace with the last element and then remove the last element
-                   usersInArray[i] = usersInArray[usersInArray.length - 1];
-                   usersInArray.pop(); // Now this works
-                   diamond.isInAutomatedSavingsArray[_user] = false;
-                   break;
-               }
+           uint256 userIndex = diamond.userToArrayIndex[_user]; // Assuming userToArrayIndex exists
+
+           // Ensure user is actually in the array at the stored index before proceeding
+           if (userIndex < usersInArray.length && usersInArray[userIndex] == _user) {
+               address lastUser = usersInArray[usersInArray.length - 1];
+               usersInArray[userIndex] = lastUser;
+               diamond.userToArrayIndex[lastUser] = userIndex;
+               usersInArray.pop();
+               delete diamond.userToArrayIndex[_user];
            }
-
+           // Fallback in case user was not removed properly
            diamond.isInAutomatedSavingsArray[_user] = false;
-
+           // Fallback in case user was not removed properly
            diamond.isInAutomatedSavingsArray[_user] = false;
-
            streak.currentStreak = 0;
            streak.lastSuccessfulSave = 0;
            streak.longestStreak = 0;

        if (!diamond.isInAutomatedSavingsArray[_user]) {
            LibDiamond.AutomatedSavingsPlan storage plan = diamond
-               .automatedSavingsPlans[msg.sender];
+               .automatedSavingsPlans[_user]; // Assumed prior fix for user context
            plan.isActive = true;

-           diamond.automatedSavingsUsers.push(msg.sender);
-           diamond.isInAutomatedSavingsArray[msg.sender] = true;
+           diamond.automatedSavingsUsers.push(_user); // Assumed prior fix for user context
+           diamond.userToArrayIndex[_user] = diamond.automatedSavingsUsers.length - 1;
+           diamond.isInAutomatedSavingsArray[_user] = true;
        }
    }
```

Status: Open

MEDIUM SEVERITY ISSUE

[M-02]: Ambiguous Streak Logic in AutomatedSavingsFacet::applyStreak() with Multiple Token Frequencies

- **Summary:**

The applyStreak() function's calculation of maxStreaks based on the individual token's frequency, combined with a single currentStreak for the user and a daily increment cap, can lead to counter-intuitive streak progression. Specifically, a user's streak might not increment for a less frequent token if a more frequent token has already pushed currentStreak beyond the maxStreaks calculated for the less frequent one.

- **Description**

The applyStreak(address _user, uint256 _streakWindow) function manages user saving streaks. Key aspects of its logic are:

1. maxStreaks is calculated as plan.duration / _streakWindow, where _streakWindow is the frequency of the specific token whose saving is currently being processed.
2. streak.currentStreak is a single counter for the user, shared across all tokens in their plan.
3. The streak can only increment once per day: if (lastDay == currentDay && streak.lastSuccessfulSave != 0) return;
4. The streak increments if streak.currentStreak < maxStreaks.

Consider a scenario:

- A user has a 30-day plan.
- Token A: frequency = 1 day (_streakWindow = 1 day). maxStreaks for Token A context = 30 / 1 = 30.
- Token B: frequency = 7 days (_streakWindow = 7 days). maxStreaks for Token B context = 30 / 7 = 4.

If the user saves daily with Token A for 5 consecutive days (and these are on different calendar days):

- streak.currentStreak becomes 5.
- streak.lastSuccessfulSave is updated daily.

On Day 7, Token B's saving is due and successfully executed. applyStreak() is called with _user and _streakWindow = 7 days (Token B's frequency).
Inside applyStreak():

- maxStreaks is calculated as plan.duration / 7 days = 4.
- The condition to increment the streak is if (streak.currentStreak < maxStreaks), which becomes if (5 < 4). This is false.

As a result, streak.currentStreak does not increment, even though the user successfully maintained their saving schedule for Token B. The user's currentStreak (5) already exceeds the maxStreaks (4) as calculated from Token B's perspective.
This behavior arises because maxStreaks is dynamically calculated based on the currently processing token's frequency, while currentStreak is a global user streak. The daily increment cap further solidifies the currentStreak as a "daily user saving streak." The maxStreaks check, as implemented, can thus prematurely halt streak progression for tokens with longer frequencies if shorter frequency tokens have already built up the currentStreak.

impact Explanation

This logic can lead to user confusion and a perception that streaks are not being awarded fairly or consistently, especially for users who have multiple tokens with different saving frequencies in their plan. It might disincentivize users from adding tokens with longer frequencies if they observe that these tokens do not contribute to increasing their streak count once a certain threshold (defined by shorter frequency tokens) is met. This can affect user engagement with the streak feature.

Recommendation

Re-evaluate the intended behavior of maxStreaks in relation to the user-level currentStreak and the daily increment cap.
If the currentStreak is meant to be a "daily plan participation streak" (which the daily cap suggests), then maxStreaks should reflect the maximum number of daily streaks possible for the entire plan duration, not tied to individual token frequencies.

Status: Open

MEDIUM SEVERITY ISSUE

[M-03]: Unbounded Loop in AutomatedSavingsFacet::getAndExecuteAutomatedSavingsPlansDue() May Lead to Denial of Service

- **Summary:**

The getAndExecuteAutomatedSavingsPlansDue() function iterates over the entire diamond.automatedSavingsUsers array to process savings plans. If this array grows significantly, the transaction executing this function can exceed the block gas limit, leading to a Denial of Service (DoS) where no automated savings can be processed.

- **Description**

1. The getAndExecuteAutomatedSavingsPlansDue() function is responsible for processing all due automated savings plans. It achieves this by iterating through every user stored in the diamond.automatedSavingsUsers array:

```
// In getAndExecuteAutomatedSavingsPlansDue()
address[] memory usersInArray = diamond.automatedSavingsUsers;

for (uint256 i = 0; i < usersInArray.length; i++) { // @audit-issue this can cause out of DOS due to userslength. Call `getAndExecuteAutomatedSavingsPlansDue` in range
    address user = usersInArray[i];
    LibDiamond.AutomatedSavingsPlan storage plan = diamond
        .automatedSavingsPlans[user];

    // ... logic to check and execute plan ...
}
```

The core issue is that the for loop iterates from **i = 0** to **usersInArray.length - 1** in a single transaction. As the number of users with automated savings plans increases, the **usersInArray.length** also increases. Each iteration involves SLOADs (to read **plan** data, **tokenDetails, userAvailableBalances)** and potentially SSTOREs (if a saving is executed). If the array becomes sufficiently large, the cumulative gas cost of this loop can exceed the block gas limit.

When this happens, any attempt to call **getAndExecuteAutomatedSavingsPlansDue()** will fail (revert due to out-of-gas). This means that no automated savings plans can be processed for any user, effectively halting this core functionality of the contract.

Impact Explanation:

If the number of users with automated savings plans grows large, this function will become uncallable due to exceeding the block gas limit. This constitutes a Denial of Service for the automated savings execution feature, preventing all users' scheduled savings from being processed. This directly impacts the primary utility of the automated savings plans.

Likelihood Explanation

The likelihood increases as the platform gains users. Without a mechanism to limit the number of users processed per call, it's probable that a popular system will eventually hit gas limits for this function. This is a common scalability issue in smart contracts that process unbounded arrays.

Recommendation

To mitigate this, implement a pagination or batching mechanism for processing users. The function should process a limited number of users per call and provide a way to continue processing the remaining users in subsequent calls.

One common approach:

1. Modify getAndExecuteAutomatedSavingsPlansDue() to accept startIndex and count parameters.
2. The loop would then iterate from startIndex up to startIndex + count (or usersInArray.length, whichever is smaller).
3. Keepers or external callers would be responsible for calling this function multiple times with advancing startIndex values until all users are processed.

Example modification sketch:

```
function getAndExecuteAutomatedSavingsPlansDue(uint256 _startIndex, uint256 _count) external nonReentrant {
    LibDiamond.DiamondStorage storage diamond = LibDiamond.diamondStorage();
    uint256 dueCount = 0;
    uint256 skippedCount = 0;

    address[] storage usersInArray = diamond.automatedSavingsUsers; // Use storage pointer
    uint256 endIndex = _startIndex + _count;
    if (endIndex > usersInArray.length) {
        endIndex = usersInArray.length;
    }

    for (uint256 i = _startIndex; i < endIndex; i++) {
        address user = usersInArray[i];
        // ... rest of the existing logic ...
    }

    emit LibEventsAndErrors.BatchAutomatedSavingsExecuted(
        dueCount,
        skippedCount
    );
}
```

This requires off-chain infrastructure (keepers) to manage the batch calls. Ensure _count is chosen carefully to stay within reasonable gas limits.

Status: Open

MEDIUM SEVERITY ISSUE

[M-04]: Incorrect User Context in AutomatedSavingsFacet::deactivateSafe() Fails to Deactivate User Plans

- **Summary:**

The deactivateSafe() function incorrectly uses msg.sender instead of the _user parameter when attempting to set a plan's isActive flag to false. This is particularly problematic when called by getAndExecuteAutomatedSavingsPlansDue(), where msg.sender is the keeper, resulting in the actual user's plan remaining active in storage.

- **Description**

The deactivateSafe(address _user) function is intended to deactivate an automated savings plan for a given _user. However, a critical line responsible for this deactivation uses the wrong user context.

The issue lies here:

```
// Inside deactivateSafe(address _user)
LibDiamond.AutomatedSavingsPlan storage plan = diamond
    .automatedSavingsPlans[msg.sender]; // Plan for msg.sender is loaded
// ...
plan.isActive = false; // msg.sender's plan is marked inactive, not _user's plan
```

The **plan** storage pointer is initialized to **diamond.automatedSavingsPlans[msg.sender].** Consequently, when **plan.isActive = false;** is executed, it modifies the **isActive** status of **msg.sender's** plan, not the plan belonging to the _user passed as a parameter.

The most significant impact of this bug occurs when **getAndExecuteAutomatedSavingsPlansDue()** calls **deactivateSafe(user).** In this scenario, **msg.sender** is the address of the keeper executing the transaction, while user is the address of the individual whose plan has expired (e.g., **block.timestamp > plan.unlockTime)** and should be deactivated. Because of the bug, the keeper's plan data (which is likely non-existent or irrelevant) is modified, while the actual **user's** plan **(diamond.automatedSavingsPlans[user].isActive)** remains **true.**

While other parts of **deactivateSafe()** correctly use **_user** for removing from **diamond.automatedSavingsUsers,** updating **diamond.isInAutomatedSavingsArray[_user]**, and resetting streak data, the fundamental **plan.isActive** flag for the actual user is never set to false.

Impact Explanation:

High. If a user's plan expires or needs to be deactivated for other reasons (e.g., all tokens removed), it will not be correctly marked as inactive in the contract's storage due to this bug. The plan will continue to appear active. This can lead to several issues:

1. The plan might still be iterated over in getAndExecuteAutomatedSavingsPlansDue(), causing unnecessary checks and gas consumption.
2. Other logic relying on the isActive flag might behave unexpectedly.
3. Users might be unable to create new plans if the system believes an old, undeactivated plan is still active (depending on other checks like userAutomatedPlanExists). Essentially, user plans cannot be properly deactivated by the automated keeper process.

Likelihood Explanation

High. The getAndExecuteAutomatedSavingsPlansDue() function, which is a core component for managing automated savings, directly calls deactivateSafe(user). In this context, msg.sender (the keeper) will invariably be different from user. Therefore, this bug is triggered under normal operational conditions whenever a plan managed by the keeper needs deactivation.

Recommendation

Modify deactivateSafe() to correctly reference and update the isActive flag of the _user's automated savings plan.

```
        if (diamond.isInAutomatedSavingsArray[_user]) {
            LibDiamond.AutomatedSavingsPlan storage plan = diamond
-               .automatedSavingsPlans[msg.sender];
+               .automatedSavingsPlans[_user]; // Corrected to use _user
            LibDiamond.StreakData storage streak = diamond.userStreaks[_user];
            plan.isActive = false;
        }
```

Status: Open

INFORMATIONAL SEVERITY ISSUE

[I-01] - Redundant and Inefficient Storage Access in AutomatedSavingsFacet.sol

- **Description**

The AutomatedSavingsFacet contract frequently accesses storage variables like diamond.acceptedTokens, plan.tokens, and plan.tokenDetails multiple times within the same function. This increases gas costs unnecessarily, especially in functions that involve loops or repeated operations.

- **Imapct**

Higher gas costs for users interacting with the contract, particularly in functions like createAutomatedSavingsPlan, addTokenToAutomatedPlan, and removeTokenFromAutomatedPlan.

- **Proof of Concept (PoC)**

Affected Functions

1. createAutomatedSavingsPlan (Line 34)
2. addTokenToAutomatedPlan (Line 111)
3. removeTokenFromAutomatedPlan (Line 181)

Steps to Reproduce

1. Deploy the contract.
2. Call any of the affected functions with valid parameters.
3. Measure the gas usage.
4. Optimize the function by caching storage variables in local memory and compare the gas usage.

Example

In the removeTokenFromAutomatedPlan function, the plan.tokens array is accessed multiple times within the loop. This can be optimized by caching the array in a local variable:

Before Optimization:

```
for (uint256 i = 0; i < plan.tokens.length; i++) {
    if (plan.tokens[i] == _token) {
        plan.tokens[i] = plan.tokens[plan.tokens.length - 1];
        plan.tokens.pop();
        break;
    }
}
```

After Optimization:

```
address[] storage tokens = plan.tokens;
for (uint256 i = 0; i < tokens.length; i++) {
    if (tokens[i] == _token) {
        tokens[i] = tokens[tokens.length - 1];
        tokens.pop();
        break;
    }
}
```

Proposed Fix / Recommendation

Cache frequently accessed storage variables in local memory to reduce gas costs. For example:

- Cache plan.tokens in a local variable in removeTokenFromAutomatedPlan.
- Cache diamond.acceptedTokens in a local variable in createAutomatedSavingsPlan.

Status: Open

INFORMATIONAL SEVERITY ISSUE

[I-02]: Redundant Zero Value Check in AutomatedSavingsFacet::createAutomatedSavingsPlan()

- **Summary**

The createAutomatedSavingsPlan() function includes a redundant check for _frequency == 0, which is already covered by a preceding compound check, leading to minor code inefficiency.

- **Description**

In the createAutomatedSavingsPlan() function, input validation includes the following checks:

```
if (_amount == 0 || _frequency == 0)
    revert LibEventsAndErrors.ZeroValueNotAllowed();
if (_frequency == 0) revert LibEventsAndErrors.ZeroValueNotAllowed(); // This check is redundant
```

The first if statement checks if either _amount is zero or _frequency is zero. If _frequency is zero, this first condition (_amount == 0 || _frequency == 0) will evaluate to true, and the transaction will revert.
Therefore, the subsequent check if (_frequency == 0) is redundant because execution will never reach this line if _frequency is zero; the first check would have already caused a revert.

Impact Explanation

Low. This redundancy does not introduce any security vulnerabilities. Its impact is minimal, resulting in a very small amount of unnecessary bytecode deployed and a negligible increase in gas cost for an effectively unreachable code path during plan creation.

Recommendation

Remove the redundant check to slightly optimize bytecode size and gas usage, and to improve code clarity.

```
        if (!diamond.acceptedTokens[_token])
            revert LibEventsAndErrors.InvalidTokenAddress();
        if (_amount == 0 || _frequency == 0)
-           revert LibEventsAndErrors.ZeroValueNotAllowed();
-       if (_frequency == 0) revert LibEventsAndErrors.ZeroValueNotAllowed();
+           revert LibEventsAndErrors.ZeroValueNotAllowed(); // Combined check, removed redundant one
        if (_frequency < 1 days) revert LibEventsAndErrors.InvalidInput();
        if (_frequency > 365 days) revert LibEventsAndErrors.InvalidInput();
        if (_duration < 30 days) revert LibEventsAndErrors.DurationTooShort();
```

Status: Open

INFORMATIONAL SEVERITY ISSUE

[I-03]: Incorrect User Context in AutomatedSavingsFacet::activateSafe()

- **Summary**

The activateSafe() function incorrectly uses msg.sender instead of the _user parameter when updating plan status and user tracking arrays. This can lead to incorrect state updates if the function is ever called in a context where msg.sender is different from the intended _user, such as by another contract managing user plans.

- **Description**

The activateSafe(address _user) function is designed to activate an automated savings plan for a specified _user. However, within its logic, several critical state modifications erroneously use msg.sender as the key instead of the _user parameter.

The problematic lines are:

```
// LibDiamond.AutomatedSavingsPlan storage plan = diamond.automatedSavingsPlans[msg.sender];
// Should be:
// LibDiamond.AutomatedSavingsPlan storage plan = diamond.automatedSavingsPlans[_user];

plan.isActive = true; // This will modify diamond.automatedSavingsPlans[msg.sender].isActive

diamond.automatedSavingsUsers.push(msg.sender); // msg.sender is added instead of _user
diamond.isInAutomatedSavingsArray[msg.sender] = true; // msg.sender's status is updated instead of _user's
```

In the provided code, plan is initialized with **diamond.automatedSavingsPlans[msg.sender].** Consequently, **plan.isActive = true;** updates the plan status for msg.sender, not **_user**. Furthermore, msg.sender is added to the diamond.automatedSavingsUsers array, and **diamond.isInAutomatedSavingsArray[msg.sender]** is set to true, rather than performing these actions for the _user.

While current direct call paths to **activateSafe()** (from **addTokenToAutomatedPlan()** and **extendAutomatedPlanDuration())** pass msg.sender as the _user argument, this masks the bug for those specific scenarios. If **activateSafe()** were to be called by another contract acting on behalf of a user (where msg.sender would be the calling contract's address, not the end-user's), or if internal logic changes, the state for the wrong entity **(msg.sender)** would be updated. The intended _user's plan would not be marked as active, and they would not be correctly tracked in **automatedSavingsUsers.**

Impact Explanation

If activateSafe() is invoked where _user != msg.sender, the intended user's plan will not be activated. Instead, msg.sender's plan status (if one exists) will be incorrectly modified, and msg.sender will be added to the active user list. This can prevent the actual user's plan from being processed by functions like getAndExecuteAutomatedSavingsPlansDue(), effectively rendering their newly added token or plan extension non-operational for automated savings until the issue is rectified.

Likelihood Explanation

The bug is currently latent due to the specific ways activateSafe() is called within this facet **(where _user is msg.sender).** However, the function signature **activateSafe(address _user)** explicitly allows for **_user** to be different from msg.sender. Future integrations, new features, or calls from other contracts could easily trigger this bug, making it a potential point of failure as the system evolves.

Recommendation

Modify the activateSafe() function to consistently use the _user parameter for all state changes related to the user's plan and tracking.

```
        if (!diamond.isInAutomatedSavingsArray[_user]) {
            LibDiamond.AutomatedSavingsPlan storage plan = diamond
-               .automatedSavingsPlans[msg.sender];
+               .automatedSavingsPlans[_user]; // Corrected to use _user

            plan.isActive = true;

-           diamond.automatedSavingsUsers.push(msg.sender);
-           diamond.isInAutomatedSavingsArray[msg.sender] = true;
+           diamond.automatedSavingsUsers.push(_user); // Corrected to use _user
+           diamond.isInAutomatedSavingsArray[_user] = true; // Corrected to use _user
        }
```

Status: Open