TaoFi

Security Review

Cantina Managed review by:

Phaze, Lead Security Researcher
Slowfi, Security Researcher

August 15, 2025

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
- 3.1 High Risk . . . 4
  - 3.1.1 Arbitrary external calls enable token theft through approved allowances . . . 4
  - 3.1.2 Fungible AlphaTokens enable unauthorized stake removal from arbitrary hotkeys . . . 5
  - 3.1.3 Static AlphaToken amounts cause permanent loss of accrued staking rewards . . . 6
  - 3.1.4 Incorrect unit conversion between TAO and RAO in staking operations . . . 7
- 3.2 Medium Risk . . . 8
  - 3.2.1 `RemoteCall` function bypasses protocol fees and lacks call restrictions . . . 8
  - 3.2.2 Missing Minimum Amount Check When Unstaking TAO . . . 10
- 3.3 Low Risk . . . 10
  - 3.3.1 Potential token balance accumulation without recovery mechanism . . . 10
  - 3.3.2 Incomplete swap parameter overrides and missing documentation . . . 11
  - 3.3.3 Interchain account router not automatically protected from arbitrary calls . . . 11
  - 3.3.4 Implementation Contract Missing `_disableInitializers` in Constructor . . . 12
  - 3.3.5 Missing Check That `swapParams.tokenOut` Matches Expected `usdc` Token . . . 12
- 3.4 Gas Optimization . . . 12
  - 3.4.1 Unused Internal Function: `_transferStake` . . . 12
- 3.5 Informational . . . 13
  - 3.5.1 Use forceApprove instead of deprecated `safeApprove` pattern . . . 13
  - 3.5.2 Insufficient msg.value validation for ETH swaps with bridge costs . . . 13
  - 3.5.3 Incorrect Comment Refers to WETH Instead of WTAO . . . 14
  - 3.5.4 Staking Receiver Is Fixed to `msg.sender` Instead of a Configurable Address . . . 14
  - 3.5.5 Msg.value sent to `StakingPrecompileV2` is not utilized and may be lost . . . 14

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

TaoFi is a suite of foundational DeFi products on Bittensor, the home of decentralized AI

From Jul 4th to Jul 8th the Cantina team conducted a review of taofi-private on commit hash 27572f76. The team identified a total of **17** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	4	4	0
Medium Risk	2	2	0
Low Risk	5	4	1
Gas Optimizations	1	1	0
Informational	5	4	1
Total	**17**	**15**	**2**

3 Findings

3.1 High Risk

3.1.1 Arbitrary external calls enable token theft through approved allowances

Severity: High Risk

Context: SwapBridgeAndCallFromMain.sol#L270-L284

Summary: The `SwapBridgeAndCallFromMain` contract allows arbitrary external calls through the `_executeExternalCall()` function, which can be exploited to steal tokens from users who have granted approvals to the contract. An attacker can craft malicious swap data to call `transferFrom()` on any token contract, draining approved tokens from victims.

Description: The `lifiSwapBridgeAndCall()` function accepts user-provided swap parameters including a target address and arbitrary calldata, which are passed to `_executeExternalCall()`. While the function implements a blacklist mechanism, it does not prevent calls to token contracts by default:

```
function _executeExternalCall(address target, uint256 value, bytes calldata data) internal {
    if (isTargetAddressBlacklisted[target]) {
        revert INVALID_TARGET();
    }

    bool success;
    if (value == 0) {
        (success,) = target.call(data);
    } else {
        (success,) = target.call{value: value}(data);
    }

    if (!success) revert EXTERNAL_CALL_FAILED();
}
```

Since users must approve tokens for legitimate swaps, an attacker can exploit this by:

1. Targeting any ERC20 token contract as the `target`.
2. Crafting calldata to invoke `transferFrom(victim, attacker, amount)`.
3. Stealing approved tokens from any user who has granted allowances to the contract.

The current blacklist approach is insufficient because it would require manually adding every valuable token contract, which is impractical and error-prone.

Impact Explanation: The impact is high as this vulnerability allows attackers to steal any tokens that users have approved to the contract. This puts all user funds at risk that have been approved for legitimate swap operations.

Likelihood Explanation: The likelihood is medium to high because the attack requires no special conditions beyond user approvals, which are necessary for normal contract operation. Any malicious actor can easily exploit this vulnerability to steal approved tokens.

Recommendation: Replace the blacklist approach with a whitelist mechanism that only allows calls to pre-approved swap router contracts:

```
  contract SwapBridgeAndCallFromMain is Initializable, OwnableUpgradeable, ReentrancyGuardUpgradeable {
      // ... existing code ...

-     mapping(address => bool) public isTargetAddressBlacklisted;
+     mapping(address => bool) public isTargetAddressWhitelisted;

      // ... existing code ...

      /**
-      * @dev Set the target address to be blacklisted.
+      * @dev Set the target address to be whitelisted.
       * @param _target - The address of the target.
-      * @param _isBlacklisted - The boolean value to determine if the target is blacklisted.
+      * @param _isWhitelisted - The boolean value to determine if the target is whitelisted.
       */
-     function setTargetAddressBlacklisted(address _target, bool _isBlacklisted) external payable onlyOwner {
-         isTargetAddressBlacklisted[_target] = _isBlacklisted;
+     function setTargetAddressWhitelisted(address _target, bool _isWhitelisted) external payable onlyOwner {
+         isTargetAddressWhitelisted[_target] = _isWhitelisted;

-         emit TargetAddressBlacklisted(_target);
+         emit TargetAddressWhitelisted(_target);
      }

      function _executeExternalCall(address target, uint256 value, bytes calldata data) internal {
-         if (isTargetAddressBlacklisted[target]) {
+         if (!isTargetAddressWhitelisted[target]) {
              revert INVALID_TARGET();
          }

          // ... rest of function unchanged ...
      }
  }
```

This approach provides several advantages:

- Default security: Only explicitly approved contracts can be called.
- Reduced attack surface: Eliminates the possibility of calling arbitrary token contracts.
- Easier maintenance: Only legitimate swap routers need to be whitelisted.
- Clear security model: Makes the intended usage explicit and auditable.

TaoFi: Fixed in commit 26b76a63.

Cantina Managed: Fix verified.

3.1.2 Fungible AlphaTokens enable unauthorized stake removal from arbitrary hotkeys

Severity: High Risk

Context: StakingManager.sol#L177-L209

Summary: The StakingManager contract's design allows users to unstake from any hotkey within a subnet, regardless of which hotkey they originally staked to. This occurs because AlphaTokens are fungible across all hotkeys in the same subnet, enabling users to effectively steal stake delegations from other users' chosen validators.

Description: The StakingManager contract mints fungible AlphaTokens when users stake to any hotkey within a subnet. However, during unstaking, users can specify any hotkey in that subnet to remove stake from, not necessarily the one they originally staked to. This creates a significant vulnerability:

1. Fungible tokens across hotkeys: AlphaTokens for a subnet are identical regardless of which hotkey was staked to.
2. Arbitrary unstaking: The `unstake()` function allows users to remove stake from any hotkey by specifying it in the parameters.
3. No ownership tracking: The contract doesn't track which user staked to which specific hotkey.

Impact Explanation: The impact is high as this vulnerability allows users to effectively steal stake delegations from others. Users can redirect stake from validators they didn't choose to validators they prefer,

undermining the intended staking mechanism and potentially affecting validator rewards and network security.

Likelihood Explanation: The likelihood is high because the attack requires no special conditions - any user with AlphaTokens can unstake from any hotkey in the subnet. This fundamental design flaw will consistently allow unauthorized stake movements.

Proof of Concept:

1. Bob delegates 10 TAO to hotkey B.
2. Alice delegates 10 TAO to hotkey A.
3. Alice unstakes 10 TAO from hotkey B (using her AlphaTokens).
4. Alice stakes 10 more TAO to hotkey A.

Result:

- Bob's intended delegation to B is removed without his consent.
- Alice gains double delegation to A using Bob's original stake.
- Hotkey A receives 20 TAO total delegation.
- Hotkey B loses all delegation.

Recommendation: Consider implementing one of the following approaches to maintain proper stake ownership:

1. Separate AlphaTokens per hotkey: Deploy different AlphaToken contracts for each hotkey (or using an ERC1155 token), making tokens non-fungible across different validators:

```
mapping(uint256 netuid => mapping(bytes32 hotkey => address alphaToken)) public alphaTokens;

function _deployNewAlphaToken(uint256 netuid, bytes32 hotkey) internal returns (address) {
    string memory name = string(abi.encodePacked("Alpha-", Strings.toString(netuid), "-",
    ↪  Strings.toHexString(uint256(hotkey))));
    // ... deploy logic
    alphaTokens[netuid][hotkey] = alphaToken;
    return alphaToken;
}
```

2. Whitelist validator hotkeys: Implement a controlled set of approved validator hotkeys that users can stake to, managed by the protocol team:

```
mapping(uint256 netuid => mapping(bytes32 hotkey => bool)) public approvedHotkeys;

function setApprovedHotkey(uint256 netuid, bytes32 hotkey, bool approved) external onlyOwner {
    approvedHotkeys[netuid][hotkey] = approved;
}

function stake(bytes32 hotkey, uint256 netuid, address receiver, uint256 minAlphaToReceive) external
↪  payable {
    require(approvedHotkeys[netuid][hotkey], "Hotkey not approved for staking");
    // ... rest of staking logic
}
```

This approach would limit the attack surface by restricting staking to a curated list of trusted validators, while maintaining token fungibility within the approved set. Either approach would ensure better control over validator delegations and prevent unauthorized stake redistribution.

TaoFi: Fixed in commit aee5afa3.

Cantina Managed: Fix verified. AlphaTokens are not fungible between different hotkeys anymore.

3.1.3 Static AlphaToken amounts cause permanent loss of accrued staking rewards

Severity: High Risk

Context: StakingManager.sol#L129

Summary: The StakingManager contract mints a fixed amount of AlphaTokens based on the initial stake amount, but these tokens do not automatically reflect accrued staking rewards over time. This design

causes users to permanently lose any rewards that accumulate in their staking positions, as they cannot unstake more than their original static token balance.

Description: The current staking mechanism has a fundamental flaw in how it handles reward accrual:

1. Static minting: When users stake, they receive a fixed amount of AlphaTokens based on their initial stake:

   ```
   IAlphaToken(alphaTokens[netuid]).mint(receiver, alphaReceived);
   ```

2. Reward accumulation: Over time, staked positions accrue rewards in the underlying Subtensor network, increasing the actual stake amount associated with the user's position (split between validator hotkey and delegator coldkey).

3. Limited unstaking: Users can only unstake up to their original AlphaToken balance, meaning accrued rewards become permanently inaccessible:

   ```
   IERC20(alphaToken).transferFrom(msg.sender, address(this), unstakeParams.amount);
   ```

   This creates a scenario where users lose access to legitimate staking rewards that should rightfully belong to them.

Impact Explanation: The impact is high as users will permanently lose all accrued staking rewards over time. In a staking system where rewards are a primary incentive for participation, this represents a significant loss of expected value for users.

Likelihood Explanation: The likelihood is high because reward accrual is a fundamental feature of staking systems. As long as users maintain stakes over time, rewards will accumulate and become inaccessible under the current design.

Recommendation: Consider redesigning the system using an ERC4626-style vault approach that properly accounts for reward accrual.

TaoFi: Fixed in commit aee5afa3.

Cantina Managed: Fix verified. AlphaToken shares are now minted/redeemed in proportion to the growth/decline in stake.

There is the risk of front-running deposits and receiving too little shares or shares that don't reflect the provided value. However, the `minAlphaToReceive` parameter in `stake()` mitigates the issue as long as it is set properly. Likewise, if the `ExactInputSingleParams.amountOutMinimum` is set properly for the `unstakeSwapAndBridge()` action, this should properly protect rate changes when unstaking.

3.1.4 Incorrect unit conversion between TAO and RAO in staking operations

Severity: High Risk

Context: StakingManager.sol#L177-L191

Summary: The `_addStake()` function passes an incorrect unit to the staking precompile, causing a massive overstatement of stake amounts. The function receives `taoAmount` in wei (where 1 TAO = 1e18 wei) but passes it directly to the precompile's `addStake()` function, which expects an amount in RAO units (where 1 TAO = 1e9 RAO). This results in staking amounts that are 1e9 times larger than intended.

Description: The `StakingManager` contract has a critical unit conversion error in the `_addStake()` function:

```
function _addStake(bytes32 hotkey, uint256 netuid, uint256 taoAmount) internal returns (uint256 alphaAmount) {
    // ...
    (bool success,) = stakingPrecompile.call{value: taoAmount}(
        abi.encodeWithSelector(IStakingV2.addStake.selector, hotkey, taoAmount, netuid)
    );
    // ...
}
```

The issue occurs because:

- The `taoAmount` parameter represents TAO in wei units (1 TAO = 1e18 wei).
- The StakingPrecompileV2's `addStake()` function expects the amount parameter in RAO units (1 TAO = 1e9 RAO).

- Passing wei directly as RAO results in a 1e9 multiplier error.

For example, if a user intends to stake 1 TAO (1e18 wei), the precompile interprets this as 1e18 RAO, which equals 1e9 TAO - a billion times more than intended.

Impact Explanation: The impact is medium as this vulnerability prevents users from successfully staking through the system. While no funds are directly lost, the precompile calls will consistently revert due to the incorrect unit conversion, making the staking functionality unusable. This creates operational challenges where bridge operations may have already occurred but subsequent staking fails, requiring manual intervention by the team to resolve these sensitive cross-chain situations.

Likelihood Explanation: The likelihood is high because this error occurs on every staking operation. Any user attempting to stake through the StakingManager will experience this unit conversion bug.

Recommendation: Convert TAO amounts from wei to RAO before passing to the precompile:

```
  function _addStake(bytes32 hotkey, uint256 netuid, uint256 taoAmount) internal returns (uint256 alphaAmount) {
      uint256 alphaBalanceBefore = IStakingV2(stakingPrecompile).getStake(hotkey, pubKey, netuid);

+     // Convert TAO (wei) to RAO for precompile
+     uint256 amountInRao = taoAmount / 1e9;
+
      (bool success,) = stakingPrecompile.call{value: taoAmount}(
-         abi.encodeWithSelector(IStakingV2.addStake.selector, hotkey, taoAmount, netuid)
+         abi.encodeWithSelector(IStakingV2.addStake.selector, hotkey, amountInRao, netuid)
      );

      if (!success) {
          revert AddStakeFailed();
      }

      uint256 alphaBalanceAfter = IStakingV2(stakingPrecompile).getStake(hotkey, pubKey, netuid);
      return alphaBalanceAfter - alphaBalanceBefore;
  }
```

Apply the same conversion to the `_removeStake()` function to ensure consistency across all staking operations (if desired). This conversion ensures that the precompile receives the correct RAO amounts as expected by its interface.

TaoFi: Fixed in commit dc407581.

Cantina Managed: Fix verified.

3.2 Medium Risk

3.2.1 `RemoteCall` function bypasses protocol fees and lacks call restrictions

Severity: Medium Risk

Context: SwapBridgeAndCallFromMain.sol#L207-L217

Summary: The `remoteCall()` function allows users to make arbitrary cross-chain calls without paying protocol fees, effectively bypassing the fee mechanism implemented in the main `lifiSwapBridgeAndCall()` workflow. This undermines the protocol's revenue model and creates an unfair advantage for users who discover this alternative pathway.

Description: The contract implements two pathways for making cross-chain calls:

1. Fee-paying pathway (`lifiSwapBridgeAndCall()`): Charges fees on swapped tokens before bridging.
2. Fee-bypassing pathway (`remoteCall()`): Allows direct cross-chain calls without any fee collection.

The `remoteCall()` function performs the same cross-chain call functionality as the fee-paying pathway but without any restrictions or fee collection:

```
function remoteCall(RemoteCallsParams calldata _params) public payable nonReentrant {
    bytes32 userSpecificSalt = bytes32(uint256(uint160(msg.sender)));
    IInterchainAccountRouterWithOverrides(interchainAccountRouter).callRemoteWithOverrides{value: msg.value}(
        DESTINATION_CHAIN_ID,
        _params.router,
        _params.ism,
        _params.calls,
        _params.hookMetadata,
        userSpecificSalt
    );
}
```

This creates an inconsistency where users can achieve the same cross-chain functionality while avoiding protocol fees entirely.

Impact Explanation: The impact is medium as this primarily affects protocol revenue rather than user fund security. However, it undermines the fee model and creates unfair treatment between users who use different pathways.

Likelihood Explanation: The likelihood is medium to high once users discover this alternative pathway. The function is publicly accessible and provides clear economic incentives for users to bypass fees.

Recommendation: Consider implementing one of the following approaches to address this inconsistency:

1. Apply fees to `remoteCall`: Charge a fee on the native value being sent:

```
function remoteCall(RemoteCallsParams calldata _params) public payable nonReentrant {
    uint256 feeAmount = (msg.value * fee) / PERCENTAGE_FACTOR;
    uint256 remainingValue = msg.value - feeAmount;

    // Transfer fee to treasury
    (bool success, ) = treasury.call{value: feeAmount}("");
    require(success, "Fee transfer failed");

    bytes32 userSpecificSalt = bytes32(uint256(uint160(msg.sender)));
    IInterchainAccountRouterWithOverrides(interchainAccountRouter).callRemoteWithOverrides{value:
    ↪  remainingValue}(
        DESTINATION_CHAIN_ID,
        _params.router,
        _params.ism,
        _params.calls,
        _params.hookMetadata,
        userSpecificSalt
    );
}
```

2. Restrict call types: Limit the types of calls that can be made through `remoteCall()`:

```
function remoteCall(RemoteCallsParams calldata _params) public payable nonReentrant {
    // Validate that calls are only for specific whitelisted operations
    for (uint256 i = 0; i < _params.calls.length; i++) {
        require(isAllowedRemoteCall(_params.calls[i]), "Unauthorized call type");
    }

    // ... rest of function
}
```

3. Remove the function entirely: If there's no specific need for fee-free remote calls, consider removing this function to ensure all cross-chain operations go through the fee-paying pathway. The chosen approach should align with the protocol's intended business model and use cases for cross-chain functionality.

TaoFi: Fixed in commit 193803dc.

Cantina Managed: Verified. Fixed by including a allowed remote call target and function whitelist:

```
    // to => selector => allowed
    mapping(bytes32 => mapping(bytes4 => bool)) public allowedRemoteCalls;
```

3.2.2 Missing Minimum Amount Check When Unstaking TAO

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: In `StakingManager.sol`, the `unstake` function removes a staking position and returns TAO to the user.

However, there is no validation that the `taoReceived` meets a minimum expected amount. While the unstaking logic is internal to the contract the user might receive less TAO than intended given changes on the constant product formula of the subnet on withdrawal.

Implementing a `minAmountOut` check, as is common in DeFi protocols (e.g., swaps, redemptions), allows users to determine the minimum expected amount to receive.

Recommendation: Consider adding a `minAmountTaoReceived` parameter to the unstaking interface and reverting the transaction if `taoReceived < minAmountTaoReceived`. Alternatively, document that this check is expected to be handled off-chain or delegated to the receiver, if that's an explicit design decision.

TaoFi: Fixed in commit f27844bc.

Cantina Managed: Fix verified.

3.3 Low Risk

3.3.1 Potential token balance accumulation without recovery mechanism

Severity: Low Risk

Context: SwapBridgeAndCallFromMain.sol#L194

Description: The `SwapBridgeAndCallFromMain` contract performs token swaps through external calls to LiFi but lacks mechanisms to handle scenarios where tokens remain in the contract after swap operations. This could occur due to:

1. Partial swaps: User errors in swap parameter configuration that result in incomplete token consumption.
2. Rounding differences: Small amounts left due to precision differences in swap calculations.
3. Exact output swaps: When swapping for a specific output amount, excess input tokens may remain unused.

Currently, the contract has no way to recover these potentially stuck tokens, and there's no validation to ensure swap operations consume the expected token amounts.

Recommendation: Consider implementing the following improvements:

1. Add balance validation after swaps to detect remaining tokens:

```
// After external call
uint256 remainingBalance = IERC20(_fromToken).balanceOf(address(this));
if (remainingBalance > 0) {
    // Either revert or transfer back to user
    IERC20(_fromToken).safeTransfer(msg.sender, remainingBalance);
}
```

2. Add an admin recovery function for emergency token retrieval:

```
function emergencyTokenRecovery(address token, address to, uint256 amount) external onlyOwner {
    IERC20(token).safeTransfer(to, amount);
}
```

3. Add native ETH recovery for completeness:

```
function emergencyETHRecovery(address to, uint256 amount) external onlyOwner {
    (bool success, ) = to.call{value: amount}("");
require(success, "ETH transfer failed");
}
```

These additions would provide safeguards against token accumulation and offer recovery mechanisms for edge cases.

TaoFi: Fixed in commit e663ab8d.

Cantina Managed: Verified. by reverting if any balance remains from the input token.

```
uint256 remainingBalance = IERC20(_fromToken).balanceOf(address(this));
if (remainingBalance > 0) revert SWAP_FAILED();
```

We are wondering whether this could cause issues for exact output swaps using Lifi that would naturally leave some dust.

3.3.2 Incomplete swap parameter overrides and missing documentation

Severity: Low Risk

Context: SwapAndStake.sol#L110

Description: The `swapAndStake()` function in SwapAndStake contract modifies user-provided swap parameters without comprehensive validation or clear documentation. Two issues are present:

1. Undocumented balance override: The function overrides `swapParams.amountIn` to use the user's entire token balance:

   ```
   swapParams.amountIn = IERC20(swapParams.tokenIn).balanceOf(address(msg.sender));
   ```

   This behavior is not documented in the function's natspec, potentially confusing users who expect to control the exact amount.

2. Missing recipient override: The function assumes `swapParams.recipient` is correctly set to `address(this)` but doesn't enforce this requirement. If users provide an incorrect recipient address, the swap tokens will be sent elsewhere, causing the subsequent staking operation to fail.

Recommendation:

1. Add clear documentation about both parameter overrides:

   ```
       /**
        * @notice Swaps a specified amount of an input token for TAO and stakes it.
   -    * @dev The caller must have approved this contract to spend their input tokens.
   +    * @dev The caller must have approved this contract to spend their input tokens.
   +    *      Note: This function will override swapParams.amountIn to use the user's entire
   +    *      balance of the input token and swapParams.recipient to ensure tokens are received
   +    *      by this contract for staking.
        * @param swapParams The parameters for the Uniswap V3 swap.
        * @param stakeParams The parameters for staking, including hotkey and netuid.
        */
   ```

2. Override the recipient parameter to ensure tokens are received by the contract:

   ```
       swapParams.amountIn = IERC20(swapParams.tokenIn).balanceOf(address(msg.sender));
   +   swapParams.recipient = address(this);
   ```

   These changes ensure that the function behaves predictably and users understand that their entire token balance will be used for staking.

TaoFi: Fixed in commit 5d60cf09.

Cantina Managed: Fix verified.

3.3.3 Interchain account router not automatically protected from arbitrary calls

Severity: Low Risk

Context: SwapBridgeAndCallFromMain.sol#L118-L123

Description: The `setInterchainAccountRouter()` function allows the owner to update the interchain account router address but does not automatically add the previous router to the blacklist or ensure the new router is protected from arbitrary external calls. This creates a potential security gap where critical infrastructure contracts could be targeted through the `_executeExternalCall()` function.

The interchain account router is a sensitive contract that handles cross-chain operations and should never be callable through the arbitrary call mechanism used for LiFi swaps.

Recommendation: If continuing with the blacklist approach, automatically manage the router's blacklist status when updating:

```
  function setInterchainAccountRouter(address _interchainAccountRouter) external payable onlyOwner {
      require(_interchainAccountRouter.code.length > 0, "Invalid interchain account router address");
      interchainAccountRouter = _interchainAccountRouter;
+
+     // Automatically blacklist the new router to prevent arbitrary calls
+     isTargetAddressBlacklisted[_interchainAccountRouter] = true;

      emit InterchainAccountRouterUpdated(_interchainAccountRouter);
  }
```

TaoFi: Acknowledged.

Cantina Managed: Acknowledged.

3.3.4 Implementation Contract Missing `_disableInitializers` in Constructor

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In `SwapBridgeAndCallFromMain.sol` the contract is upgradeable but does not call `_disableInitializers()` in the constructor. For upgradeable contracts deployed behind a proxy (e.g., using OpenZeppelin's UUPS or Transparent pattern), it is critical to prevent the implementation contract from being initialized directly. Otherwise, a malicious actor could call the `initialize()` function on the implementation contract itself and take control, especially if it's publicly accessible and not yet initialized. Calling `_disableInitializers()` in the constructor of the implementation contract mitigates this by locking it permanently.

Recommendation: Call `_disableInitializers()` in the constructor to prevent the implementation contract from being initialized directly.

TaoFi: Fixed in commit 54578af6.

Cantina Managed: Fix verified.

3.3.5 Missing Check That `swapParams.tokenOut` Matches Expected `usdc` Token

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In `SwapAndStake.sol`, the contract approves `usdc` to the bridge after a token swap. However, there is no check that `swapParams.tokenOut == usdc`, meaning the contract assumes that the swap resulted in USDC tokens. If `tokenOut` was misconfigured or manipulated (e.g., via user input or misrouted data), the contract might attempt to approve or transfer a token it doesn't actually hold, or worse, misrepresent another token as USDC.

This assumption could break downstream logic or, in certain cases, lead to lost funds or failed bridging.

Recommendation: Add a check before approval to ensure the output token from the swap matches the expected USDC address.

TaoFi: Fixed in commit c515fabd.

Cantina Managed: Fix verified.

3.4 Gas Optimization

3.4.1 Unused Internal Function: `_transferStake`

Severity: Gas Optimization

Context: *(No context files were provided by the reviewer)*

Description: The function `_transferStake` defined in `StakingManager.sol` is currently unused. Having unused code increases maintenance overhead and may cause confusion during future development or upgrades.

Recommendation: Consider removing this function if it's not part of an upcoming feature.

Note: If implemented in the future, transferring stake between coldkeys, especially across subnets, would likely require corresponding logic to update synthetic token balances to maintain redeemability guarantees.

TaoFi: Fixed in commit aee5afa3.

Cantina Managed: Fix verified.

3.5 Informational

3.5.1 Use forceApprove instead of deprecated `safeApprove` pattern

Severity: Informational

Context: SwapBridgeAndCallFromMain.sol#L237-L238

Description: The `SwapBridgeAndCallFromMain` contract uses an outdated and potentially gas-inefficient approval pattern. In the `lifiSwapBridgeAndCall()` function, the contract manually sets allowances to zero before setting them to the desired amount:

```
// Approve
IERC20(_fromToken).safeApprove(_approvalAddress, 0);
IERC20(_fromToken).safeApprove(_approvalAddress, _fromAmount);
```

OpenZeppelin's SafeERC20 library now provides the `forceApprove()` function, which handles this two-step approval process internally and is the recommended approach for setting token allowances. The current implementation uses an outdated pattern that was necessary for tokens like USDT that required allowances to be reset to zero before setting a new value.

Recommendation: Replace the manual two-step approval process with OpenZeppelin's `forceApprove()` function:

```
- // Approve
- IERC20(_fromToken).safeApprove(_approvalAddress, 0);
- IERC20(_fromToken).safeApprove(_approvalAddress, _fromAmount);
+ // Approve using the recommended forceApprove method
+ IERC20(_fromToken).forceApprove(_approvalAddress, _fromAmount);
```

This change provides several benefits:

- Gas efficiency: Reduces the number of external calls from two to one when the token doesn't require the zero-reset pattern.
- Code simplicity: Eliminates the need for manual two-step approval logic.
- Future compatibility: Uses the current OpenZeppelin recommended pattern.
- Automatic handling: The function internally handles both standard ERC20 tokens and those requiring zero-reset.

TaoFi: Fixed in commit d6576410.

Cantina Managed: Fix verified.

3.5.2 Insufficient msg.value validation for ETH swaps with bridge costs

Severity: Informational

Context: SwapBridgeAndCallFromMain.sol#L181

Description: In the `lifiSwapBridgeAndCall()` function, when processing ETH swaps (where `_fromToken` is `address(0)`), the contract only validates that `msg.value` covers the swap amount but fails to account for the required bridge cost:

```
if (msg.value < _fromAmount) revert SWAP_FAILED();
```

The function later uses `msg.value - valueSpent - _bridgeCost` for the interchain call, which could result in an underflow if the total `msg.value` is insufficient to cover both the swap amount and bridge costs. This validation gap could cause transaction failures or unexpected behavior during the bridging process.

Recommendation: Update the validation to ensure `msg.value` covers both the swap amount and bridge cost:

```
- if (msg.value < _fromAmount) revert SWAP_FAILED();
+ if (msg.value < _fromAmount + _bridgeCost) revert SWAP_FAILED();
```

This change ensures that users provide sufficient ETH to cover both the LiFi swap operation and the subsequent bridge transaction costs.

TaoFi: Fixed in commit fbe52348.

Cantina Managed: Fix verified.

3.5.3 Incorrect Comment Refers to WETH Instead of WTAO

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In `SwapAndStake.sol`, the following comment inaccurately references WETH:

```
// If asset out is WETH, unwrap it
```

However, the context of the protocol and surrounding logic suggest that the unwrapping applies to **WTAO**, not WETH. Leaving inaccurate comments in the codebase may lead to confusion for developers or auditors reviewing the logic.

Recommendation: Update the comment to accurately reflect the unwrapping of WTAO.

TaoFi: Fixed in commit cd8b53c2.

Cantina Managed: Fix verified.

3.5.4 Staking Receiver Is Fixed to `msg.sender` Instead of a Configurable Address

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In `SwapAndStake.sol`, the `stakeAlpha` function is called with the recipient of the synthetic tokens hardcoded as `msg.sender`. This approach limits flexibility by enforcing that only the transaction sender can receive the staked representation. In contrast, the `StakingManager` contract itself allows specifying an arbitrary receiver address. Supporting a `stakeParams.receiver` field would enable relayed interactions (e.g., via smart wallets or frontends) and better composability with other protocols.

Recommendation: Consider modifying the interface to allow specifying a custom `receiver` in `stakeParams`, and forward it to `StakingManager.stakeAlpha` instead of forcing `msg.sender`.

TaoFi: Acknowledged.

Cantina Managed: Acknowledged.

3.5.5 Msg.value sent to `StakingPrecompileV2` is not utilized and may be lost

Severity: Informational

Context: StakingManager.sol#L177-L191

Summary: The StakingManager contract sends ETH via `msg.value` to the StakingPrecompileV2's `addStake()` function, but this value is not utilized by the precompile and may be permanently lost. Unlike the original StakingPrecompile which refunded unused `msg.value`, the V2 version lacks this refund mechanism while still being marked as payable.

Description: The `_addStake()` function in StakingManager forwards TAO (as `msg.value`) to the StakingPrecompileV2:

```
(bool success,) = stakingPrecompile.call{value: taoAmount}(
    abi.encodeWithSelector(IStakingV2.addStake.selector, hotkey, taoAmount, netuid)
);
```

However, based on the precompile implementations:

- Original StakingPrecompile (V1):
  - Used `msg.value` for staking amount.
  - Included refund logic: `Self::transfer_back_to_caller(&account_id, amount).`
  - The amount was derived from `handle.context().apparent_value`.
- `StakingPrecompileV2`:
  - Takes amount as an explicit parameter (`amount_rao`).
  - Still marked as `payable` but contains no refund logic.
  - The `msg.value` sent is ignored and may be trapped in the precompile.

This means any ETH sent to the V2 precompile via `msg.value` will not be refunded to the caller and becomes inaccessible.

Impact Explanation: The impact is high as this results in permanent loss of user funds. Every staking operation through the StakingManager will lose the entire TAO amount sent via `msg.value`, as these funds become trapped in the precompile contract.

Likelihood Explanation: The likelihood is high because this occurs on every staking operation. The current implementation will consistently result in fund loss for all users who stake through the system.

Recommendation: Remove the `msg.value` parameter from the precompile call since StakingPrecompileV2 doesn't utilize it:

```
  function _addStake(bytes32 hotkey, uint256 netuid, uint256 taoAmount) internal returns (uint256 alphaAmount) {
      uint256 alphaBalanceBefore = IStakingV2(stakingPrecompile).getStake(hotkey, pubKey, netuid);

-     (bool success,) = stakingPrecompile.call{value: taoAmount}(
+     (bool success,) = stakingPrecompile.call(
          abi.encodeWithSelector(IStakingV2.addStake.selector, hotkey, taoAmount, netuid)
      );

      if (!success) {
          revert AddStakeFailed();
      }

      uint256 alphaBalanceAfter = IStakingV2(stakingPrecompile).getStake(hotkey, pubKey, netuid);
      return alphaBalanceAfter - alphaBalanceBefore;
  }
```

> **Important note:**
> This recommendation assumes that the V2 precompile operates differently from V1 and doesn't require TAO to be sent. However, this needs to be verified with the precompile implementation team, as the V2 precompile may need to be updated to either:
> 1. Handle the `msg.value` appropriately (if TAO is needed for staking), or...
> 2. Remove the `payable` modifier if no TAO should be sent.

TaoFi: Fixed in commit dc407581.

Cantina Managed: Fix verified.