Clave Monorepo

Security Review

Cantina Managed review by:

MiloTruck, Lead Security Researcher
Víctor Martínez, Security Researcher

January 7, 2025

Contents

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

Clave is an easy-to-use non-custodial smart wallet powered by Account Abstraction and the Hardware Elements (e.g Secure Enclave, Android Trustzone etc...), offering a unique onboarding process.

From Dec 2nd to Dec 8th the Cantina team conducted a review of Clave Monorepo on commit hash 0ff730eb. The team identified a total of extbf20 issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	1	1	0
High Risk	3	2	1
Medium Risk	7	7	0
Low Risk	7	7	0
Gas Optimizations	0	0	0
Informational	2	2	0
Total	**20**	**19**	**1**

The Cantina team reviewed Clave's clave-monorepo holistically on commit hash c3656e53 and concluded that all issues were addressed and no new vulnerabilities were introduced.

3 Findings

3.1 Critical Risk

3.1.1 `ClaggBaseAdapter._handleFee()` incorrectly calculates the user's `gain`

Severity: Critical Risk

Context: ClaggBaseAdapter.sol#L513-L523

Description: In `ClaggBaseAdapter._handleFee()`, the performance fee is charged based on the user's `gain`, which is calculated as such:

```
// Calculate total fee rate | performance fee + non-Clave holder fee if applicable
uint48 fee = poolConfig.performanceFee +
    (_isClave(msg.sender) ? 0 : poolConfig.nonClaveFee);

// Only charge fee if there is a fee rate and user is withdrawing profits
if (fee > 0 && userInfo.withdrawn + amount > userInfo.deposited) {
    // Calculate profit on this withdrawal
    uint256 gain = amount - (userInfo.deposited - userInfo.withdrawn);

    // Calculate fee amount (fee is in basis points - divide by 10000)
    uint256 perfFee = (gain * fee) / 10000;
```

However, it is possible for `withdrawn + amount > deposited` and `deposited < withdrawn` to both be true at the same time. For `withdrawn` to be more than `deposited`, it means that the user previously withdrew some of his deposit for a profit (ie. already has some gains), and is withdrawing more. This would cause `userInfo.deposited - userInfo.withdrawn` to revert due to arithmetic underflow. For example:

- Assume that the withdrawal fee is 0% for simplicity.
- User deposits 100 tokens:
  - `deposited = 100`.
- The deposit makes a 20% return, so his deposit is now 120 tokens.
- User withdraws 110 tokens:
  - `withdrawn` is set to `110`.
- User withdraws the remaining 10 tokens:
  - `withdrawn + amount = 110 + 10 = 120`.
  - `deposited - withdrawn = 100 - 110 = -10`.

As a result, `_handleFee()` will always revert when called, making it impossible for the user to withdraw his funds through `withdraw()`.

Secondly, another issue with this calculation is that it calculates the **lifetime** profit instead of the profit for this withdrawal. Using the example above:

- `deposited = 100`.
- User withdraws 110 tokens (2):
  - `gain = amount + withdrawn - deposited = 110 + 0 - 100 = 10`.
  - A fee is charged on 10 tokens.
  - `withdrawn` is set to `110`.
- User withdraws the remaining 10 tokens (3):
  - `gain = amount + withdrawn - deposited = 10 + 110 - 100 = 20`.
  - A fee is charged on 20 tokens.

In step (3), the performance fee should be charged on the remaining 10 tokens, since a fee was already charged for the previous withdrawal in step (2). However, the performance fee is calculated based on `gain = 20`, which is the user's lifetime profit, and not the profit for this withdrawal alone.

Recommendation: The Clave team has refactored how `gain` is calculated in `_handleFee()`.

Clave: Fixed in commit 46aece01.

Cantina Managed: Verified, `gain` is now calculated based on the following conditions (`amount` is the withdrawal amount):

1. If `withdrawn + amount <= deposited`, there is no profit.
2. If `deposited <= withdrawn`, the entire withdrawal amount is profit.
3. Otherwise (ie. `withdrawn < deposited < withdrawn + amount`), profit is calculated as `withdrawn + amount - deposited`, which is the proportion of the withdrawal amount that is profit.

3.2 High Risk

3.2.1 Using `aToken.balanceOf()` in `ClaggAaveAdapter._compoundAccounting()` allows for inflation attacks

Severity: High Risk

Context: ClaggAaveAdapter.sol#L98-L102, ClaggBaseAdapter.sol#L559-L563

Description: When `compound()` is called, `ClaggAaveAdapter._compoundAccounting()` sets `totalLiquidity` of the pool to the current aToken balance of the contract:

```
function _compoundAccounting(address pool, uint256) internal override {
    PoolInfo storage poolInfo = _getPoolInfo(pool);

    poolInfo.totalLiquidity = IERC20(pool).balanceOf(address(this));
}
```

However, this allows an attacker to perform an inflation attack by directly donating aTokens to the contract. For example:

- `ClaggAaveAdapter` is newly deployed and has no deposits yet.
- Attacker deposits 1 wei of asset:
  - 1 wei of aToken is minted.
  - `totalLiquidity = 1` and `totalSupply = 1`.
- Attacker directly transfers `100e18` of aToken to the contract.
- Attacker calls `compound()`:
  - `totalLiquidity = 100e18 + 1`.
- Afterwards, a user calls `deposit()` to deposit `10e18` of assets:
  - `10e18` aToken is minted.
  - In `_liquidityToShare()`, the amount of shares to mint is calculated as `liquidity * totalSupply / totalLiquidity = 10e18 * 1 / 100e18`, which rounds down to 0.
  - User receives no shares for his deposit.

In the example above, the `10e18` of assets deposited by the user accrues to the attacker's 1 share.

Recommendation: OpenZeppelin has a writeup on defending against such attacks (see their blog post on ERC4626 inflation attacks).

Consider adding "dead" shares to `totalSupply` on the first deposit in `ClaggBaseAdapter._depositAccounting()` as such:

```
  // If pool is empty, shares = liquidity (1:1 ratio)
  if (poolInfo.totalSupply == 0) {
      shares = liquidity;
      poolInfo.totalLiquidity = liquidity;
+     poolInfo.totalSupply = 1e6;
  } else {
```

The downsides to this are:

1. The first depositor loses a tiny portion of his deposit, since they accrue to the `1e6` dead shares.

2. The shares to liquidity ratio will never be 1:1.
3. A small portion of all future gains (from incentives or compounding rewards) will always accrue to these dead shares.

Alternatively, `ClaggAaveAdapter` could be refactored to use the scaled balance of aTokens instead.

Clave: Fixed in commit cc4ff057.

Cantina Managed: Verified, `1e4` "dead" shares are now minted on the first deposit.

3.2.2 Performance fee is not handled in `ClaggBaseAdapter._handleFee()`

Severity: High Risk

Context: ClaggBaseAdapter.sol#L525-L527

Description: In `ClaggBaseAdapter`, whenever a withdrawal occurs, `_handleFee()` is called to deduct the performance fee from the user's withdrawal amount:

```
// Deduct fee from withdrawal amount and profit
amount -= perfFee;
gain -= perfFee;
```

However, there is no logic in `_handleFee()` to transfer the performance fee out to some address (eg. a treasury). In the current implementation, the performance fee is not handled and simply remains in the contract forever. The only way for the performance fee to be retrieved by the protocol owner would be to call `ClaggMain.rescue()`, which is meant to rescue stuck tokens.

Recommendation: In `_handleFee()`, transfer the performance fee out to a protocol-owned address, such as a treasury.

Clave: Fixed in commit d46bce5d.

Cantina Managed: Verified, the performance fee is now transferred out to a protocol-owned fee vault.

3.2.3 Missing slippage protection on SyncSwap swaps

Severity: High Risk

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

Description: The `SyncSwapper` abstract contract implements the logic to process swaps via SyncSwap's `SyncRouter`. Typically, when integrating DEX protocols, a maximum slippage limit is enforced to protect trades from excessive exposure to MEV. In the case of SyncRouter's `swap` function, an `amountOutMin` parameter can be provided to cap the maximum allowable slippage, ensuring that swaps (e.g., for incentives or rewards) execute at efficient rates.

However, in this implementation, slippage is not being accounted for, resulting in both swap executions lacking appropriate `amountOutMin` parameters:

- `ClaggBaseAdapter.sol#L464`: `minAmountOut` hardcoded to a fixed value set in the config.
- `ClaggBaseAdapter.sol#L359`: `minAmountOut == 0` means the protocol is fine with receiving no tokens from the swap.

Impact: This renders the protocol vulnerable to MEV, potentially resulting in the loss of funds from incentives and compounding rewards during swaps.

Recommendation: Introduce a dedicated value to specify the minimum acceptable output for a particular swap.

Clave: Acknowledged.

Cantina Managed: Acknowledged.

3.3 Medium Risk

3.3.1 Native ETH cannot be used as the incentive token

Severity: Medium Risk

Context: ClaggBaseAdapter.sol#L231-L232, IncentiveVault.sol#L32-L45, IncentiveVault.sol#L48-L58

Description: In `ClaggBaseAdapter.setIncentive()`, `safeApprove()` is always called before transferring the incentive token (ie. `token` below) to `IncentiveVault`:

```
IERC20(token).safeApprove(incentiveVault, amount);
IIncentiveVault(incentiveVault).storeTokens{value: msg.value}(pool, token, amount);
```

However, this implementation does not handle the case where the incentive token is native ETH, which makes it impossible to use native ETH as the incentive token. Additionally, `storeTokens()` and `withdrawTokens()` in `IncentiveVault` do not handle the case where the token stored/witthdrawn is native ETH as well.

Recommendation: Modify the logic in `ClaggBaseAdapter.setIncentive()` to handle native ETH as such:

```
- IERC20(token).safeApprove(incentiveVault, amount);
- IIncentiveVault(incentiveVault).storeTokens{value: msg.value}(pool, token, amount);
+ uint256 value;
+ if (token == address(ETH_TOKEN_SYSTEM_CONTRACT)) {
+     value = amount;
+ } else {
+     IERC20(token).safeApprove(incentiveVault, amount);
+ }
+ IIncentiveVault(incentiveVault).storeTokens{value: value}(pool, token, amount);
```

Additionally, `storeTokens()` and `withdrawTokens()` in `IncentiveVault` should be refactored to be able to handle native ETH.

Clave: Fixed in commit 5df88a57.

Cantina Managed: It would be safer to check if `token == ETH_TOKEN_SYSTEM_CONTRACT` to determine if an ETH transfer should take place, rather than `msg.value > 0`. This prevents the case where `ClaggBaseAdapter.setIncentive()` or `IncentiveVault.storeTokens()` is accidentally called with value while `token` is not ETH.

Consider making the following changes:

1. In `ClaggBaseAdapter.setIncentive()`:

```
- if (msg.value > 0) {
+ if (token == address(ETH_TOKEN_SYSTEM_CONTRACT)) {
      IIncentiveVault(incentiveVault).storeTokens{value: amount}(pool, token, amount);
  } else {
      IERC20(token).safeApprove(incentiveVault, amount);
      IIncentiveVault(incentiveVault).storeTokens(pool, token, amount);
  }
```

2. In `IncentiveVault.storeTokens()`:

```
- if (msg.value > 0) {
-     require(token == ETH_TOKEN_SYSTEM_CONTRACT, 'invalid token');
+ if (token == address(ETH_TOKEN_SYSTEM_CONTRACT)) {
      require(msg.value == amount, 'invalid amount');
  } else {
+     require(msg.value == 0, 'invalid msg.value');
      IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
  }
```

A `msg.value == 0` check is added in `storeTokens()` during ERC20 transfers for additional safety.

Clave: Fixed in commit 29f147a0.

Cantina Managed: Verified, `setIncentive()` now handles native ETH appropriately.

3.3.2 `SyncSwap._swapFrom()` does not encode native ETH correctly

Severity: Medium Risk

Context: SyncSwapper.sol#L77-L84, SyncSwapper.sol#L51-L57, SyncSwapper.sol#L75, SyncSwapRouter.sol#L39, SyncSwapRouter.sol#L62-L73

Description: In the protocol, native ETH is represented with the ZkSync's `ETH_TOKEN_SYSTEM_CONTRACT` address. This can be seen in `SyncSwapper._swapFrom()`, where `_from` is checked against `ETH_TOKEN_SYSTEM_CONTRACT` to determine if value should be transferred to the router:

```
if (_from == address(ETH_TOKEN_SYSTEM_CONTRACT)) {
    ISyncRouter.TokenAmount memory tokenAmount = SYNC_ROUTER.swap{value: _amountIn}(
        paths,
        _minAmountOut,
        block.timestamp
    );
    return tokenAmount.amount;
} else {
```

In `SyncSwapper._swapFrom()`, when encoding `SwapPath` and `SwapStep` to be passed to `SYNC_ROUTER.swap()`, the `_from` address (ie. the input token for the swap) is directly encoded as `SwapPath.tokenIn` and in `SwapStep.data`, as shown below:

```
paths[0] = ISyncRouter.SwapPath({steps: steps, tokenIn: _from, amountIn: _amountIn});
```

```
steps[0] = ISyncRouter.SwapStep({
    pool: _pool1,
    data: abi.encode(_from, address(this), _withdrawMode),
    callback: address(0),
    callbackData: new bytes(0),
    useVault: true
});
```

However, in SyncSwap's router, native ETH is represented with `address(0)` and not `ETH_TOKEN_SYSTEM_CONTRACT`:

```
address private constant NATIVE_ETH = address(0);
```

Additionally, in SyncSwap pools with ETH, `token0`/`token1` are set to the WETH address instead. For example, passing `tokenIn = address(0)` to `getAmountOut()` on the ETH/WBTC pool, where `token1` is ETH, returns an incorrect result.

Therefore, if the input token for a swap is ETH, the input token address passed to `SYNC_ROUTER.swap()` is incorrect.

`_swapFrom()` is used to swap tokens when handling incentive tokens and compounding reward tokens for pools. As such, since `_swapFrom()` will revert when swapping tokens from ETH, pools cannot be configured with incentive or reward tokens as native ETH.

Recommendation: In `SyncSwapper._swapFrom()`, when `_from == ETH_TOKEN_SYSTEM_CONTRACT`, wrap ETH to WETH and handle the swap using WETH. This is similar to how ETH deposits are handled in `ClaggSyncAdapter._addLiquidity()`.

Clave: Fixed in commit 1c9594ab and 8ab3722f.

Cantina Managed: Verified, swaps from ETH are now handled appropriately.

3.3.3 `ClaggSyncAdapter._removeLiquidity()` incorrectly withdraws ETH when `token` is WETH

Severity: Medium Risk

Context: ClaggSyncAdapter.sol#L220-L227, ClaggBaseAdapter.sol#L92-L93

Description: In `ClaggSyncAdapter._removeLiquidity()`, when the `token` address is not native ETH, `syncRouter.burnLiquiditySingle()` is called with `withdrawMode = 1`:

```
// data = abi.encode(token, receiver, withdrawMode)
// withdrawMode: 0 = withdraw to vault, 1 = ETH, 2 = WETH
bytes memory data;
if (token == address(ETH_TOKEN_SYSTEM_CONTRACT)) {
    data = abi.encode(address(_weth()), address(this), 1);
} else {
    data = abi.encode(token, address(this), 1);
}
```

`withdrawMode = 1` means that when `token` is either ETH or WETH, assets withdrawn will always be transferred as native ETH.

Therefore, when `token` is WETH, `_removeLiquidity()` will withdraw ETH instead of WETH. This is incorrect as assets withdrawn from `_removeLiquidity()` should always be the `token` itself.

As a result, when `withdraw()` is called for a SyncSwap pool with `token` as WETH, `_transferToCaller()` will revert as the contract will not have any WETH to transfer to the user:

```
// Transfer the tokens to the user
_transferToCaller(poolConfig.token, tokenAmount);
```

Recommendation: In `ClaggSyncAdapter._removeLiquidity()`, when `token != ETH_TOKEN_SYSTEM_CONTRACT`, `withdrawMode = 2` should be used instead:

```
  bytes memory data;
  if (token == address(ETH_TOKEN_SYSTEM_CONTRACT)) {
      data = abi.encode(address(_weth()), address(this), 1);
  } else {
-     data = abi.encode(token, address(this), 1);
+     data = abi.encode(token, address(this), 2);
  }
```

This will withdraw WETH instead of ETH when `token = WETH`.

Clave: Fixed in commit cfb8b0d2.

Cantina Managed: Verified, the recommendation was implemented.

3.3.4 `ClaggSyncAdapter._addLiquidity()` wrongly calls `approve()` instead of `safeApprove()`

Severity: Medium Risk

Context: ClaggSyncAdapter.sol#L186-L196

Description: In `ClaggSyncAdapter._addLiquidity()`, `approve()` is called instead of `safeApprove()` when depositing `token` as liquidity:

```
IERC20(token).approve(address(syncRouter), amount);

liquidity = syncRouter.addLiquidity2(
    pool,
    inputs,
    abi.encode(address(this)),
    0,
    address(0),
    new bytes(0),
    address(0)
);
```

However, `safeApprove()` should be called instead as `token` could be a token that does not comply with the ERC20 specification.

Recommendation: Call `safeApprove()` instead:

```
- IERC20(token).approve(address(syncRouter), amount);
+ IERC20(token).safeApprove(address(syncRouter), amount);
```

Clave: Fixed in commit 84e5179f.

Cantina Managed: Verified, the recommendation was implemented.

3.3.5 Incentives May Become Locked in the System After Switching Incentive Tokens

Severity: Medium Risk

Context: ClaggBaseAdapter.sol#L216

Description: The `setIncentive` function allows the owner to configure and update protocol incentives for a specific pool. It offers two scenarios:

- If the previous incentive round has ended, the entire configuration can be updated.
- If the previous round is still active, additional incentives can only be added.

However, the protocol currently uses only the elapsed time as a criterion to determine if a round has ended, instead of also verifying whether there are any remaining unclaimed incentives in the vault.

If the current `block.timestamp` exceeds the round's expiry and some incentives remain unprocessed, these unclaimed tokens will become locked in the `IncentivesVault` when the owner changes the configuration into another token. This issue arises because the action does not compound before updating the incentive token.

Recommendation: Consider calling compound at the top of the function.

Clave: Fixed at commit 05dfa5fa.

Cantina Managed: Verified, `setIncentive()` now checks `poolIncentive == 0`, which ensures that incentives from the previous round have been fully distributed.

3.3.6 `ClaggSyncAdapter` LP staking cannot be disabled without bricking the vault

Severity: Medium Risk

Context: ClaggSyncAdapter.sol#L81

Description: In `ClaggSyncAdapter`, SyncSwap liquidity can be automatically staked in a contract if this one has been previously configured by the owner. In order to switch staking contracts the owner can call `migrate` on the adapter, effectively migrating the assets to the new staking contract. However, the current migration function does not allow for disabling staking (setting an empty address as staking contract). As a result, staking cannot be disabled without effectively locking the liquidity by changing the configuration.

Recommendation: Consider implementing the ability to migrate from a staking pool to an empty address, allowing the owner to disable staking functionality when necessary without risking the liquidity.

Clave: Fixed at commit 18047bde.

Cantina Managed: Verified, the recommended fix was implemented.

3.3.7 `ClaggBaseAdapter._handleIncentive()` withdraws the wrong token from the incentive vault

Severity: Medium Risk

Context: ClaggBaseAdapter.sol#L350-L354

Description: In `ClaggBaseAdapter._handleIncentive()`, when distributing incentive tokens from the `incentiveVault` to depositors, `token` is passed to `withdrawTokens`:

```
try
    IIncentiveVault(incentiveVault).withdrawTokens(pool, token, incentiveUsed)
{} catch {
    return 0;
}
```

However, `token` here is incorrect as it is the pool's token, not the incentive token. As a result, whenever `_handleIncentive()` is called, it will revert when attempting to withdraw from the incentive vault.

Recommendation: Pass `poolIncentive.token` to `withdrawTokens` instead.

Clave: Fixed in commit c3656e53.

Cantina Managed: Verified, the recommended fix was implemented.

3.4 Low Risk

3.4.1 `ClaggBaseAdapter._handleCompoundRewards()` is incompatible with compounded rewards in native ETH

Severity: Low Risk

Context: ClaggBaseAdapter.sol#L446-L460

Description: In `ClaggBaseAdapter._handleCompoundRewards()`, the difference between the contract's reward token balance before and after calling `_claimCompoundRewards()` is taken as the amount of reward tokens claimed:

```
// Store initial balances of reward tokens to calculate claimed amounts
uint256[] memory balancesBefore = new uint256[](rewardConfigs.length);
for (uint256 i = 0; i < rewardConfigs.length; i++) {
    balancesBefore[i] = IERC20(rewardConfigs[i].reward).balanceOf(address(this));
}

// Claim rewards from the pool
_claimCompoundRewards(pool);

// For each reward token, calculate amount claimed and swap to deposit token
uint256 claimed = 0;
for (uint256 i = 0; i < rewardConfigs.length; i++) {
    // Calculate how much of this reward token was claimed
    uint256 rewardBalance = IERC20(rewardConfigs[i].reward).balanceOf(address(this)) -
        balancesBefore[i];
```

However, since `balanceOf()` is called, this implementation does not work when the reward token (ie. `rewardConfigs[i].reward` above) is native ETH.

Recommendation: Document that the reward token for pools should never be native ETH.

Clave: Fixed in commit b2cccaca. We decided to fix as it is possible for the reward token to be ETH.

Cantina Managed: Verified, `_handleCompoundRewards()` now handles the case where the reward token is ETH.

3.4.2 `adapter` address that collides with a function selector in `ClaggMain` can never be called

Severity: Low Risk

Context: ClaggMain.sol#L119-L128

Description: In the fallback function of `ClaggMain`, the `adapter` address is extracted from the first 20 bytes of calldata:

```
fallback() external payable {
    address adapter;
    assembly {
        adapter := shr(96, calldataload(0))
    }

    require(MainStorageLib.adapterExists(adapter), 'invalid adapter');

    bytes calldata data = msg.data[20:];
    bytes memory result = EfficientCall.delegateCall(gasleft(), adapter, data);
```

However, if the first 4 bytes in the address of an `adapter` happens to be the same as the function selector of any function in this contract, it will not be possible to forward calls to that `adapter` with this proxy pattern (since having the `adapter` address as the first 20 bytes in calldata would end up calling that function).

Recommendation: Modify the fallback function to extract the `adapter` address from the last 20 bytes of calldata instead.

Clave: Fixed in commit 9cc2fcef.

Cantina Managed: Verified, the recommended fix was implemented.

3.4.3 Timelock in `ClaggMain` is ineffective if owner turns malicious

Severity: Low Risk

Context: ClaggMain.sol#L35-L41, ClaggMain.sol#L57-L59

Description: In the `ClaggMain` contract, to add an adapter to the protocol, the owner has to first call `requestAddAdapter()`. After a `timelock` duration has passed, he can then call `addAdapter()` to add the adapter:

```
function addAdapter(address adapter) external onlyOwner {
    uint256 requestDate = MainStorageLib.addRequestDate(adapter);
    require(requestDate != 0, 'adapter not requested');
    require(requestDate + MainStorageLib.timelock() < block.timestamp, 'timelock not passed');
    MainStorageLib.addAdapter(adapter);
    MainStorageLib.setAddRequestDate(adapter, 0);
}
```

This is meant to prevent malicious adapters from being instantly added should the protocol owner turn malicious. However, since the owner can also call `setTimelock()` to set the `timelock` duration, this design is ineffective. If the owner ever turns malicious, he can simply call `setTimelock(0)` followed by `requestAddAdapter()` and `addAdapter()` to instantly add a malicious adapter.

Recommendation: The `timelock` duration either be immutable, or set by another role that is not the owner.

Clave: Fixed in commit 6c7ad863.

Cantina Managed: Verified, the timelock duration is now immutable.

3.4.4 Inability to Handle ERC-1155, ERC-721 Tokens, or Tokens with Callbacks

Severity: Low Risk

Context: ClaggMain.sol#L119-L135

Description: With the current diamond-like proxy pattern, it's not possible for a handler to receive ERC-1155, ERC-721 tokens, or any tokens with callbacks, since calls without the custom adapter address prepended to the calldata will revert.

Worth noting that with this proxy pattern, it's not possible to have a receive function that contains logic in any adapter. With the current adapters, this is not an issue. However, if more protocols are integrated in the future, this problem could arise.

Recommendation: Consider adding the most popular token callback functions in `ClaggMain` to support handling of ERC-1155, ERC-721 tokens, or tokens with callbacks, in order to future-proof the system as more protocols are integrated.

Clave: Fixed at commit f04f57af.

Cantina Managed: Verified, `ClaggMain` now inherits `TokenCallbackHandler`.

3.4.5 Enforce checks on `setStakingConfig` logic

Severity: Low Risk

Context: ClaggSyncAdapter.sol#L63

Description: `setStakingConfig` could benefit from checking the pool values to ensure `isToken0` and `otherToken` are not set as arbitrary values by the owner.

Recommendation: Add a check against the pool for both token0 and otherToken to ensure correct configuration and prevent potential issues when interacting with the pool.

Clave: Fixed at commit fb5dcdef.

Cantina Managed: Verified, the recommended fix was implemented.

3.4.6 Protocol fees percentage can be set to more than 100%

Severity: Low Risk

Context: ClaggBaseAdapter.sol#L158-L174

Description: In contract `ClaggBaseAdapter`, `setPoolConfig` allows the contract's admin to set new values for the `performanceFee` and the `nonClaveFee` storage variables. Both of which are expressed in basis points with 2 fixed decimals.

The current implementation does not validate that the sum of the `performanceFee` and `nonClaveFee` does not exceed 10,000 basis points (100%).

Impact: Low. A compromised or malicious protocol admin can set the protocol fees to any value.

Recommendation: Consider adding a validation check to ensure that `performanceFee + nonClaveFee < 10000` before allowing the update to proceed.

Clave: Fixed at commit eb000f6a.

Cantina Managed: Verified, the recommended fix was implemented.

3.4.7 Missing Validation for `msg.value == 0` When Transferring Non-Native Tokens

Severity: Low Risk

Context: ClaggBaseAdapter.sol#L540

Description: The `_transferFromCaller` function in the Base Adapter does not enforce `msg.value == 0` when transferring non-native tokens. This omission could result in ETH being locked in the protocol. It is considered a best practice to enforce this check, so it is recommended to implement it to prevent potential issues.

Recommendation: Consider adding a check for `msg.value == 0` when the token is not native (ETH) to ensure ETH is not accidentally locked in the protocol.

Clave: Fixed at commit d724825b.

Cantina Managed: Verified, the recommended fix was implemented.

3.5 Informational

3.5.1 `ClaggBaseAdapter.withdraw()` violates the CEI pattern

Severity: Informational

Context: ClaggBaseAdapter.sol#L92-L96, ClaggBaseAdapter.sol#L544-L551

Description: In `ClaggBaseAdapter.withdraw()`, `_transferToCaller()`, which transfers assets to the user, is called before `snapshot()`:

```
// Transfer the tokens to the user
_transferToCaller(poolConfig.token, tokenAmount);

// Take a snapshot of the pool
snapshot(pool);
```

However, when `poolConfig.token` is ETH, `_transferToCaller()` would gives a callback to the user when transferring ETH:

```
function _transferToCaller(address token, uint256 amount) internal virtual {
    if (token == address(ETH_TOKEN_SYSTEM_CONTRACT)) {
        (bool success, ) = payable(msg.sender).call{value: amount}('');
        require(success, 'transfer failed');
    } else {
        IERC20(token).safeTransfer(msg.sender, amount);
    }
}
```

Therefore, this violates the Check Effects Interactions pattern. A malicious user could use the callback in `_transferToCaller()` to re-enter the contract and perform other actions, changing the result of the snapshot taken in `snapshot()`.

Recommendation: Consider calling `snapshot()` before `_transferToCaller()`.

Clave: Fixed in commit 2797fa88.

Cantina Managed: Verified, the recommendation was implemented.

3.5.2 Minor code improvements

Severity: Informational

Context: SyncSwapper.sol#L77-L94, ClaggBaseAdapter.sol#L522-L523, MainStorageLib.sol#L78-L80, LinkedList.sol#L199-L205, ClaggMain.sol#L120-L123, ClaggBaseAdapter.sol#L183-L191

Description/Recommendation:

1. SyncSwapper.sol#L77-L94 - The code can be simplified:

```
uint256 value;
if (_from == address(ETH_TOKEN_SYSTEM_CONTRACT)) {
    value = _amountIn;
} else {
    IERC20(_from).safeApprove(address(SYNC_ROUTER), _amountIn);
}
ISyncRouter.TokenAmount memory tokenAmount = SYNC_ROUTER.swap{value: value}(
    paths,
    _minAmountOut,
    block.timestamp
);
return tokenAmount.amount;
```

2. ClaggBaseAdapter.sol#L522-L523 - `10_000` should be a constant instead of a magic number.
3. MainStorageLib.sol#L78-L80 - `adapterCount()` isn't called anywhere in the codebase and can be removed.
4. LinkedList.sol#L199-L205 - The code can be simplified:

```
+ self[SENTINEL_ADDRESS] = value;
  if (prev == address(0)) {
-     self[SENTINEL_ADDRESS] = value;
      self[value] = SENTINEL_ADDRESS;
  } else {
-     self[SENTINEL_ADDRESS] = value;
      self[value] = prev;
  }
```

5. ClaggMain.sol#L120-L123 - Consider doing this instead to avoid using assembly:

```
- address adapter;
- assembly {
-     adapter := shr(96, calldataload(0))
- }
+ address adapter = address(bytes20(msg.data[:20]));
```

6. ClaggBaseAdapter.sol#L183-L191 - Consider using the standard `.push()` and `.pop()` when writing to storage arrays, instead of directly overwriting its length. If `rewardConfigs` is overwritten with a new array that has a smaller length, it would leave dirty bytes in storage as the old elements aren't cleaned up.

Clave: All issues have been fixed in the following commits:

1. d7e30373.
2. 84e9a1a5.
3. 1c1f6c0a.
4. 5cb1bb24.

5. 9cc2fcef.
6. 7ae9e4b2.

Cantina Managed: Verified, all issues have been fixed as recommended.