Security Review Report
NM-0419-CLAVE

NETHERMIND
SECURITY

(Feb 24, 2025)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for the smart contracts of Clave. The audit scope comprises smart contracts in the Clagg folder. The smart contracts offer an easy and unified interface in Clave wallet for users to earn rewards from defi protocols.

The user's interactions are simplified to depositing and withdrawing tokens. Users don't have to worry about claiming the accrued rewards in the defi protocols. The Clagg smart contracts handle compounding by claiming and reinvesting for the users.

This security review focuses exclusively on the smart contracts listed in Section 2 (*Audited Files*).

The audited code comprise of 1352 lines of code written in the Solidity language, and the audit was performed using (a) manual analysis of the code base and (b) creation of test cases. **Along this document, we report** 10 points of attention, where two are classified as `Critical`, one is classified as `High`, two are classified as `Medium`, one is classified as `Low`, and 4 are classified as `Informational` and `Best Practices`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.

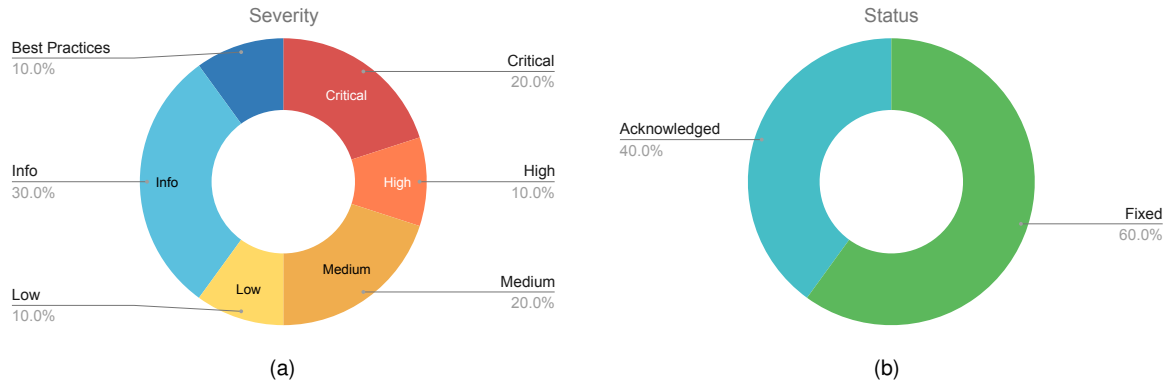

Fig. 1: Distribution of issues: Critical (2), **High** (1), **Medium** (2), **Low** (1), **Undetermined** (0), **Informational** (3), **Best Practices** (1). **Distribution of status: Fixed** (6), **Acknowledged** (4), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Jan 31, 2025
Response from Client	Regular responses during audit engagement
Final Report	Feb 24, 2025
Repository	clagg
Commit (Audit)	6982402ec86d6dcbafbcf2a6305fa3e32d5ea911
Commit (Final)	7325041285e98b2fbab6248ccb8db1de0f1c2234
Documentation Assessment	Medium
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	ClaggMain.sol	81	41	50.6%	22	144
2	IncentiveVault.sol	48	10	20.8%	14	72
3	libraries/OwnerStorageLib.sol	26	6	23.1%	8	40
4	libraries/BaseStorageLib.sol	72	5	6.9%	12	89
5	libraries/MainStorageLib.sol	80	17	21.2%	25	122
6	adapters/ClaggAaveAdapter.sol	118	23	19.5%	35	176
7	adapters/ClaggSyncAdapter.sol	204	47	23.0%	48	299
8	adapters/ClaggMainAdapter.sol	49	20	40.8%	15	84
9	adapters/ClaggZtakeAdapter.sol	72	7	9.7%	20	99
10	adapters/ClaggVenusAdapter.sol	86	13	15.1%	28	127
11	adapters/ClaggBaseAdapter.sol	417	105	25.2%	109	631
12	adapters/ClaggMerklAdapter.sol	14	7	50.0%	4	25
13	utils/SyncSwapper.sol	85	5	5.9%	16	106
	Total	**1352**	**306**	**22.6%**	**356**	**2014**

3 Summary of Issues

	Finding	Severity	Update
1	Config capabilities on adapters will be permanently lost on renouncing owner	Critical	Fixed
2	Improper initialization of `ClaggMainAdapter` prevents functionality access	Critical	Fixed
3	DoS of the Base contract when expiring the incentive immediately	High	Fixed
4	Lack of slippage protection in `deposit`	Medium	Fixed
5	Reading StakeLimit from ZTake adapter could throw error	Medium	Fixed
6	abuse of `compound` function to prevent rewards from being added to liquidity	Low	Acknowledged
7	Accrued rewards/Incentives cannot be withdrawn after full unstaking	Info	Acknowledged
8	Compounding rewards can be lost during migration of staking contract for a pool	Info	Acknowledged
9	TIMELOCK in ClaggMain should be checked against minimum value	Info	Fixed
10	Transfer of owner should be a two setup process	Best Practices	Acknowledged

4 System Overview

ClaggMain is the entry point contract. User can deposit and withdraw tokens to different defi protocols in a unified way by interacting with ClaggMain. The Clagg smart contracts interact with defi protocols through custom adapters. Protocol Owner can manage these adapters to support and extend the seamless integration with various defi protocols.

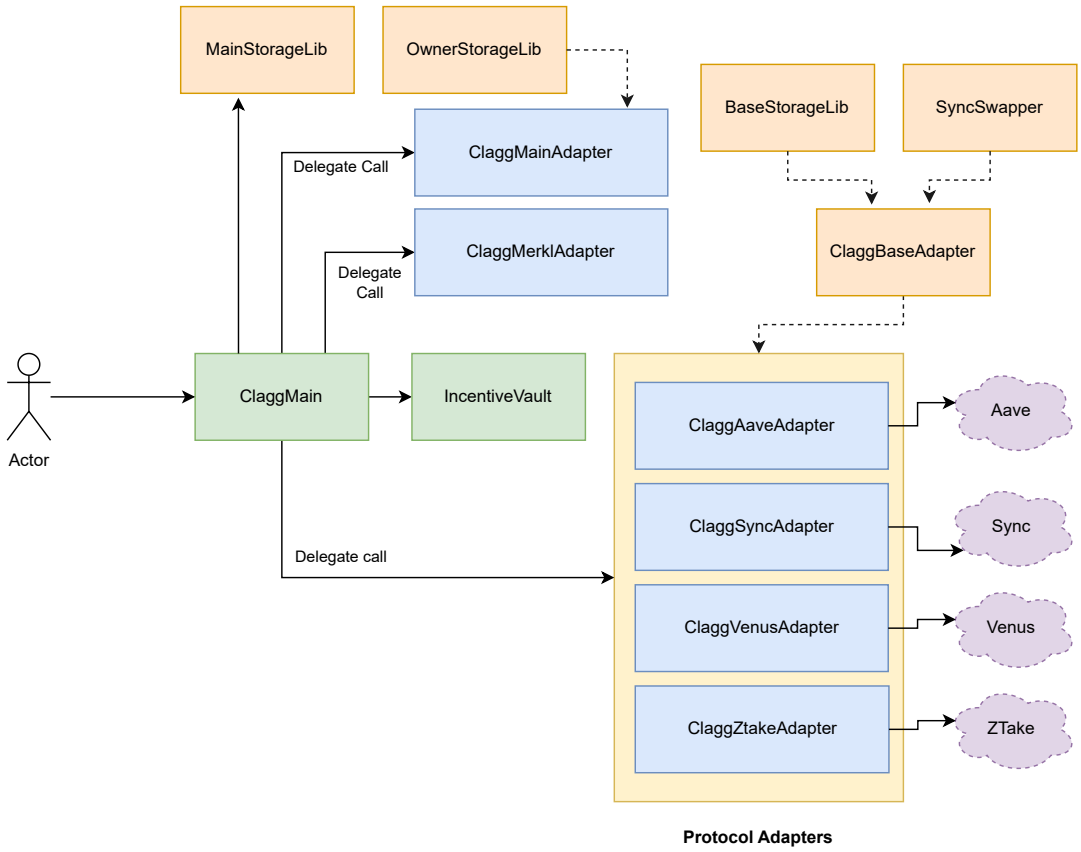

Fig. 2: Clave - Clagg overview

4.1 User

Using Clave wallet, the user can deposit tokens into a pool by calling the ClaggMain's deposit function. The ClaggMain contract delegates the call to the adapter of the Defi protocol. The Clagg smart contracts handle the interactions with the Defi protocol to stake and compound rewards. Adapters are the logic contracts that have the knowledge to interact with the specific defi protocol. Adapters abstracts the protocol specific details and thus provides a unified interaction for the wallet. The user can withdraw their tokens back by calling the withdraw function.

- deposit: The user can deposit tokens into any of the supported pools by calling the deposit function.

```
function deposit(
    address pool,
    uint256 amount,
    uint256 minLiquidity
) external payable virtual override returns (uint256) {
...
}
```

- withdraw: The user can withdraw tokens from the pool by calling the withdraw function.

```
withdraw(
     address pool,
     uint256 shares,
     uint256 minReceived,
     bool skipCompound
 ) external virtual override returns (uint256) {
 ...
 }
```

– compound: Anyone can call this function to claim the accrued rewards and incentives for the deposited tokens.

```
 function compound(address pool) public virtual override {
 ...
 }
```

4.2 Owner

Owner manages the configuration of the Clagg smart contracts. Owner has entitlements to integrate with defi protocols by adding or updating adapters. The owner can also configure pools, rewards and incentives.

– Add Adapter: Owner can add new adapter in ClaggMain to integrate with Defi protocol. As a result the Clave wallet can extend to support new defi protocols in future.

```
function addAdapter(address adapter) external onlyOwner {
  ...
}
```

– Remove Adapter: Owner can remove an existing adapter in ClaggMain to replace with a new adapter or remove support for a defi protocol.

```
 function removeAdapter(address adapter) external onlyOwner {
  ...
 }
```

– Set Owner: Current owner can transfer Owner entitlements to a new account.

```
function setOwner(address _owner) external onlyOwner {
  ...
}
```

– set Fee Vault: Owner can configure a fee vault.

```
function setFeeVault(address newFeeVault) external onlyOwner {
  ...
}
```

– Set Incentive Vault: Owner can configure the incentive vault

```
 function setIncentiveVault(address newIncentiveVault) external onlyOwner {
  ...
 }
```

– Set Pool Configuration: Owner can configure the settings for a pool.

```
 function setPoolConfig(
     address pool,
     address token,
     uint48 performanceFee,
     uint48 nonClaveFee
 ) external override onlyOwner {
   ...
 }
```

– Set Reward Configuration: Owner can configure the reward settings used for claiming the compounding rewards.

```
function setRewardConfigs(
     address pool,
     RewardConfig[] memory _rewardConfigs
 ) external override onlyOwner {
  ...
 }
```

- Set Incentive Configuration: Owner can configure the incentive settings.

```
function setIncentive(
     address pool,
     address token,
     uint256 amount,
     uint256 duration,
     address swapPool
 ) external payable override onlyOwner {
 ...
 }
```

4.3 Adapters

Adapters are logic contracts that are custom developed to integrate with defi protocols. This approach offers flexibility for Clave wallet to onboard new defi protocols or make adjustments to the already integrated defi protocol.

As of now, the below defi protocols are integrated with Clave wallet.

- Aave
- Sync
- Venus
- Ztak

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Critical] Config capabilities on adapters will be permanently lost on renouncing owner in `ClaggMain`

File(s): apps/clave-contracts/contracts/clagg/ClaggMain.sol

Description: `ClaggMainAdapter` contract was created to permanently disable some of the owner functionality in `ClaggMain`. To achieve this goal, the ownership will be renounced in `ClaggMain` by setting the owner variable in `MainStorageLib` to address(0).

`ClaggMainAdapter` contract will have the required entitlements for owner functions that are enabled through an owner variable in `OwnerStorageLib`.

```
library OwnerStorageLib {
    //...
    struct OwnerStorage {
        address owner;
    }
    //...

    function setOwner(address newOwner) internal {
        OwnerStorage storage s = ownerStorage();
        address previousOwner = s.owner;
        s.owner = newOwner;
        emit OwnershipTransferred(previousOwner, newOwner);
    }
    //....
}
```

But, as `ClaggBaseAdapter` contract checks the owner entitlements by looking at `MainStorageLibs` state variable, all the owner functions will also be disabled. After renouncing the owner entitlements, the adapters will not allow the owner to perform any configuration. This is not intended.

```
modifier onlyOwner() {
      MainStorageLib.enforceOwner();
      _;
}
```

The same issue is present in `ClaggMerklAdapter.sol`.

Recommendation(s):The `onlyOwner()` modifier should be updated in `ClaggBaseAdapter` contract to refer to `OwnerStorageLib` instead of `MainStorageLib`.

Also, as `ClaggMerklAdapter` contract does not inherit from `ClaggBaseAdapter`, the `onlyOwner()` should be updated separately to point to `OwnerStorageLib`.

Status: Fixed

Update from the client: Fixed on commit 446cf98

6.2 [Critical] Improper initialization of `ClaggMainAdapter` prevents functionality access

File(s): apps/clave-contracts/contracts/clagg/adapters/ClaggMainAdapter.sol

Description: The `ClaggMainAdapter` is designed to take over administrative functionalities from the `ClaggMain` contract following the renouncement of `ClaggMain`'s ownership. The `ClaggMain` remains the primary entry point, delegating administrative tasks such as `addAdapter` to the `ClaggMainAdapter`.

In the `ClaggMainAdapter` contract, administrative access is granted exclusively to the owner address, which is managed through `OwnerStorageLib`. As the owner is not configured, the owner address in the storage slot should be zero address. Due to this setup, the new owner cannot be set and hence none of the functionality is accessible.

The current implementation of `ClaggMainAdapter` attempts to initialize the state using a constructor:

```
contract ClaggMainAdapter {
    using SafeERC20 for IERC20;

    uint96 public immutable TIMELOCK;

    constructor(uint96 timelock_) {
        TIMELOCK = timelock_;
        // @audit Owner is intended to be initialized here
    }
    // --SNIP
}
```

However, due to the use of `delegatecall`, the `ClaggMainAdapter` executes in the storage context of `ClaggMain`. Consequently, any initialization performed in the constructor of `ClaggMainAdapter` affects only its own storage layout, not that of `ClaggMain`. This results in the owner address remaining unconfigured (defaulting to the zero address), thereby preventing the ability to set a new owner or access any administrative functionality.

Consider initializing the owner state in the context of `ClaggMain`. A recommended approach is to introduce an `initialize` function that will be delegated by `ClaggMain` post deployment. **One important thing to keep in mind** is that the `initialize` function must be called **ONCE**. However, the `ClaggMain` is already deployed and does not have a storage variable to track the initialization status. A possible workaround is to introduce a specific initialize function in the `OwnerStorageLib` that must be called at most once.

Recommendation(s): Consider adding `initialize` function to initialize the owner state in the context of `ClaggMain`.

Status: Fixed

Update from the client: Fixed in commit 7325041

6.3 [High] DoS of the Base contract when expiring the incentive immediately

File(s): /apps/clave-contracts/contracts/clagg/adapters/ClaggBaseAdapter.sol

Description: The `ClaggBaseAdapter`, inherited by all adapters, allows configuration of incentives. These incentives have an expiration date upon which all incentive tokens are released. The contract owner can modify the incentive by either extending its duration or adding more tokens through the `setIncentive` function. Additionally, the owner has the authority to expire the incentive immediately by passing duration as 0.

```
function setIncentive(
        address pool,
        address token,
        uint256 amount,
        uint256 duration,
        address swapPool
    ) external payable override onlyOwner {
        require(token != address(0), 'invalid token');
        require(pool != address(0), 'invalid pool');
        require(amount > 0, 'invalid amount');

        Incentive storage poolIncentive = _getPoolIncentive(pool);

        // If the incentive is still active, update the remaining amount
        if (poolIncentive.expiry > block.timestamp) {
            require(poolIncentive.token == token, 'invalid new incentive');
            poolIncentive.remaining += amount;
        } else {
            require(poolIncentive.remaining == 0, 'incentive remaining');
            // Otherwise, set the new incentive
            poolIncentive.token = token;
            poolIncentive.remaining = amount;
        }

        poolIncentive.expiry = block.timestamp + duration;
        poolIncentive.swapPool = swapPool;
        poolIncentive.lastUpdatedAt = block.timestamp;

        // --SNIP
}
```

The issue arises when the owner opts to expire the incentive immediately. This action triggers a DoS in the `compound` function, specifically within `_handleIncentive`. This is because when the incentive is expired immediately, `poolIncentive.expiry` becomes equal to `poolIncentive.lastUpdatedAt`:

```
poolIncentive.expiry = block.timestamp + duration;
poolIncentive.swapPool = swapPool;
poolIncentive.lastUpdatedAt = block.timestamp;
```

leading to a division by zero in the calculation of `incentiveUsed` within `_handleIncentive`:

```
function _handleIncentive(address pool, address token) internal returns (uint256) {
    // --SNIP
    if (poolIncentive.remaining > 0) {
        address incentiveVault = MainStorageLib.incentiveVault();
        if (incentiveVault == address(0)) {
            return 0;
        }

        // Use expiry timestamp if we're past it, otherwise use current time
        uint256 timestamp = block.timestamp > poolIncentive.expiry
            ? poolIncentive.expiry
            : block.timestamp;

        // Calculate time elapsed since last update
        uint256 timeDiff = timestamp - poolIncentive.lastUpdatedAt;

        // Calculate incentive amount to use based on remaining amount and time proportion
        // Formula: remaining * (elapsed time / total incentive duration)
        uint256 incentiveUsed = (poolIncentive.remaining * timeDiff) /
@>>>            (poolIncentive.expiry - poolIncentive.lastUpdatedAt);
}
    // --SNIP
}
```

Furthermore, recovery from this state is not feasible. Attempts to update the expiry date using `setIncentive` will revert because the incentive still has a remaining amount to distribute:

```
function setIncentive(
    address pool,
    address token,
    uint256 amount,
    uint256 duration,
    address swapPool
) external payable override onlyOwner {
    // --SNIP
    if (poolIncentive.expiry > block.timestamp) {
        // --SNIP
    }else {
@>>>        require(poolIncentive.remaining == 0, 'incentive remaining');
        // --SNIP
    }
    // --SNIP
}
```

As a consequence, a DoS condition is introduced in the `ClaggBaseAdapter`, causing functions that rely on `compound`—such as `deposit` and `withdraw`—to revert.

Recommendation(s): Consider preventing the owner from immediately expiring the incentive by enforcing the `duration` to be greater than zero.

Status: Fixed

Update from the client: Fixed in commit 065ecf0

6.4 [Medium] Lack of slippage protection in `deposit`

File(s): apps/clave-contracts/contracts/clagg/adapters/ClagBaseAdapter.sol

Description: The `deposit` function allows users to deposit to the underlying pool. The users will get shares representing the liquidity they added:

```
function deposit(
    address pool,
    uint256 amount,
    uint256 minLiquidity
) external payable virtual override returns (uint256) {
    PoolConfig storage poolConfig = _getPoolConfig(pool);
    require(poolConfig.token != address(0), 'no token for this pool');

    // Compound before deposit
    compound(pool);

    // Transfer the token from the user to this contract
    _transferFromCaller(poolConfig.token, amount);

    // Deposit tokens
    uint256 liquidity = _addLiquidity(pool, poolConfig.token, amount);

    // Check if the liquidity is greater than the minimum liquidity
    require(liquidity >= minLiquidity, 'liquidity is less than minLiquidity');

    // Update the pool accounting
    uint256 shares = _depositAccounting(pool, liquidity, amount);

    emit Deposit(msg.sender, pool, amount, shares);

    return shares;
}
```

The issue is that the function does not provide a slippage protection against the calculated `shares`. While the function does assert against `minLiquidity`, the `liquidity`'s value (the `shares`) can change before the transaction is executed and after the user submitted his transaction, because of other users deposited just before. In which, the user will get less `shares` than he expected.

Recommendation(s): Consider checking for slippage protection of `shares`.

Status: Fixed

Update from the client: Fixed in commit ada2acd

6.5 [Medium] Reading StakeLimit from ZTake adapter could throw error

File(s): apps/clave-contracts/contracts/clagg/adapters/ClaggZtakeAdapter.sol

Description: `getStakeLimit` function returns the amount of additional stake allowed in the pool. In ClaggZtakeAdapter, the function computes the `stakelimit` taking into account the cumulative limit and the limit applicable per user. This is to ensure that a single user cannot stake beyond the limit per user.

```
function getStakeLimit(address pool) external view override returns (uint256 stakeLimit) {
    PoolInfo storage poolInfo = _getPoolInfo(pool);
    uint256 totalSupply = IZtake(pool).totalSupply();
    uint256 totalLimit = IZtake(pool).totalLimit();
    uint256 limitPerUser = IZtake(pool).limitPerUser();

    uint256 totalLimitRemaining = totalLimit - totalSupply;
@>> uint256 limitPerUserRemaining = limitPerUser - poolInfo.totalLiquidity;// @audit

    stakeLimit = totalLimitRemaining < limitPerUserRemaining
        ? totalLimitRemaining
        : limitPerUserRemaining;
}
```

`limitPerUserRemaining` amount computation looks incorrect. As the `totalLiquidity` in the pool is the cumulative liquidity added by all participants of the pool, it is expected to be much higher than `limitPerUser` which is the limit for one user. As a result, when totalLiquidity in the pool is greater than `limitPerUser`, `getStakeLimit` function will start throwing an underflow error.

As a result, users cannot read the available stake limit.

Recommendation(s):compute the `msg.sender`'s currently staked amount in the pool. If the user's current stake is less than `limitPerUser`, and there is allowance in cumulative stake, allow which ever amount is smaller.

Status: Fixed

Update from the client: Fixed in commit 17100c7

`limitPerUser` can not be higher than `totalLiquidity` because clagg as a whole is a user. Underlying contract is not clagg specific, it's just a regular liquidity staking.

6.6 [Low] abuse of `compound` function to prevent rewards from being added to liquidity

File(s): apps/clave-contracts/contracts/clagg/adapters/ClagBaseAdapter.sol

Description: The `ClagBaseAdapter` is designed to manage rewards by adding them as liquidity, thereby generating additional earnings for depositors. The `compound` function processes this by withdrawing rewards from the underlying pool and swapping the received rewards into the pool's designated token.

```
function _handleCompoundRewards(
    address pool,
    address token
) internal virtual returns (uint256) {
    // --SNIP
    uint256[] memory balancesBefore = new uint256[](rewardConfigs.length);
    for (uint256 i = 0; i < rewardConfigs.length; i++) {
        address reward = rewardConfigs[i].reward;
        if (reward == address(ETH_TOKEN_SYSTEM_CONTRACT)) {
            balancesBefore[i] = address(this).balance;
        } else {
            balancesBefore[i] = IERC20(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++) {
        address reward = rewardConfigs[i].reward;
        // Calculate how much of this reward token was claimed
        uint256 rewardBalance;
        if (reward == address(ETH_TOKEN_SYSTEM_CONTRACT)) {
            rewardBalance = address(this).balance - balancesBefore[i];
        } else {
                rewardBalance = IERC20(reward).balanceOf(address(this)) - balancesBefore[i];
            }

        // Swap reward token to deposit token through up to 2 pools
        // Add swapped amount to running total
        claimed += _handleSwap(
            reward,
            rewardConfigs[i].intermediateToken,
            token,
            rewardConfigs[i].swapPool1,
            rewardConfigs[i].swapPool2,
            rewardBalance,
            rewardConfigs[i].minAmountOut,
            withdrawMode
        );
    }

    // --SNIP
}
```

The `_handleSwap` function is responsible for converting the received reward tokens into the pool's token. However, if the swapped amount is less than the configured minimum (`minAmountOut`), the function skips the swap and returns `0` rewards instead:

```
function _swapFrom(
    address _from,
    address _intermediateToken,
    address _pool1,
    address _pool2,
    uint256 _amountIn,
    uint256 _minAmountOut,
    uint8 _withdrawMode
) internal returns (uint256 _amountOut) {
    // --SNIP
    uint256 amountOut1 = ISyncPool(_pool1).getAmountOut(from, _amountIn, address(this));

    uint256 amountOut2;
    if (_intermediateToken == address(0)) {
        amountOut2 = amountOut1;
    } else {
        amountOut2 = ISyncPool(_pool2).getAmountOut(
            _intermediateToken,
            amountOut1,
            address(this)
        );
    }

    if (amountOut2 < _minAmountOut) {
@>>>    return 0;
    }

    // --SNIP
}
```

When the swapped amount (`amountOut2`) is below the `minAmountOut` threshold, the `_swapFrom` function returns `0`, effectively skipping the addition of these rewards to the liquidity pool. Importantly, these unprocessed rewards are not accounted for or accumulated for future batches. This oversight allows malicious users to repeatedly invoke the compound function when the available rewards are below the minimum required for swapping. By doing so, they prevent the accumulation and addition of rewards to the liquidity pool, thereby denying depositors their entitled rewards.

Recommendation(s): Consider implementing an accumulation mechanism for rewards that do not meet the `minAmountOut` threshold. Specifically, when the received rewards are below the minimum required, these rewards should be stored and combined with future reward batches.

Status: Acknowledged

Update from the client: 29cd0d1 @dev be careful against repeated compound call attacks, I do not think this is a big issue with our low minAmountOut values so I will just add this comment for now.

6.7 [Info] Accrued rewards/Incentives cannot be withdrawn after full unstaking

File(s): apps/clave-contracts/contracts/clagg/adapters/ClaggBaseAdapter.sol

Description:User claims their stake by calling withdraw function on `ClaggAdpater` contract. The function has an optional parameter to skip the accrued rewards and incentives when unstaking.

If the user skips his rewards when withdrawing, he would have no other way to claim the already compounded rewards.

```
function withdraw(
    address pool,
    uint256 shares,
    uint256 minReceived,
    bool skipCompound
) external virtual override returns (uint256) {
    ...
    require(userInfo.balanceOf >= shares, 'not enough balance');

    if (!skipCompound) {
        compound(pool);
    }
    ...

}
```

Recommendation(s): skipCompound should not be allowed when balance is equal to shares being returned.

Status: Acknowledged

Update from the client: zksync tokens staking has limits on the amount that can be withdrawn. As this was preventing the users from withdrawing their tokens, skipCompound is implemented as an option to allow the users to withdraw their tokens. Skipping compound is the user's choice to forgo eligible rewards while having the ability to take their tokens back. Any residual rewards will be distributed among other users.

6.8 [Info] Compounding rewards can be lost during migration of staking contract for a pool

File(s): apps/clave-contracts/contracts/clagg/adapters/ClaggSyncAdapter.sol

Description:ClaggSyncAdapter contract allows the owner to migrate the staking contract for a pool. In the below function, the liquidity currently accounted for in the pool by the old contract is withdrawn from the pool and stake back to the pool by the new contract.

```
function migrate(address pool, address newStaking) external onlyOwner {
    StakingConfig storage stakingConfig = _getStakingConfig(pool);
    PoolInfo storage poolInfo = _getPoolInfo(pool);

    uint256 liquidity = poolInfo.totalLiquidity;

    address oldStaking = stakingConfig.staking;

    // Withdraw the lp tokens from the staking contract if it is set
    if (oldStaking != address(0)) {
        ISyncStaking(oldStaking).withdraw(liquidity, address(this));
    }

    if (newStaking != address(0)) {
        IERC20(pool).forceApprove(newStaking, liquidity);
        ISyncStaking(newStaking).stake(liquidity, address(this));
    }

    stakingConfig.staking = newStaking;

    emit Migrated(pool, newStaking, oldStaking);
}
```

As part of staking, the rewards are accrued over time which are claimed by the _claimCompoundRewards function. There could be unclaimed rewards that should also be accounted for in the pool's totalLiquidity.

The below _claimCompoundRewards function claims the rewards for the staking contract as below.

```
function _claimCompoundRewards(address pool) internal override {
    StakingConfig storage stakingConfig = _getStakingConfig(pool);

    // Claim rewards from the staking contract if it is set
    if (stakingConfig.staking != address(0)) {
        ISyncStaking(stakingConfig.staking).claimRewards(address(this), 1, new bytes(0));
    }
}
```

While migrating the liquidity from the old staking contract to the new contract, the migration logic does not account for the accrued rewards and incentives which are not yet claimed. Such rewards will not be available to the new contract post migration and hence are lost in migration.

Recommendation(s): Migration function should claim the accrued rewards before switching to the new staking contract by calling the compound function.

Status: Acknowledged

Update from the client: e0e14ed @dev Always call compound before calling this function.

6.9 [Info] TIMELOCK in ClaggMain should be checked against minimum value

File(s): apps/clave-contracts/contracts/clagg/ClaggMain.sol

Description:TIMELOCK in `ClaggMain` contract should be initialised with a value greater than a minimum value decided by the team. As `TIMELOCK` is an immutable variable, if TIMELOCK was mistakenly initialized to 0 for example, it will invalidate the time lock feature in the contract allowing adapters to be replaced with new ones immediately.

Recommendation(s): The recommendation is to make TIMELOCK as constant with a time window. Alternatively, add validation to allow setting of TIMELOCK to value greater than a minimum value in the constructor. In this case, there should be a minimum/maximum acceptable value.

Status: Fixed

Update from the client: Fixed in commit 3003b84

6.10 [Best Practices] Transfer of owner should be a two setup process

File(s): apps/clave-contracts/contracts/clagg/ClaggMain.sol, apps/clave-contracts/contracts/clagg/adapters/ClaggMainAdapter.sol

Description: It is recommended to implement a two step process while transferring critical roles between accounts. In the current implementation of `setOwner` function in `ClaggMain` and `ClaggMainAdapter` contract, there is no check on the incoming address to be a `address(0)`. If `address(0)` was set as owner, it would lead to permanent loss of functionality for the protocol.

```
function setOwner(address _owner) external onlyOwner {
    MainStorageLib.setOwner(_owner);
}

function setOwner(address newOwner) external onlyOwner {
    OwnerStorageLib.setOwner(newOwner);
}
```

Recommendation(s):To prevent the risk, the recommended approach is to implement a two step transfer strategy. In the first step, the existing owner will propose the new owner. As a second step, the new owner should accept to claim the ownership.

Status: Acknowledged

Update from the client: we still want to allow renouncement of ownership, so no changes.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about Clave documentation

The `Clave` team was actively present in regular calls, effectively addressing concerns and questions raised by the Nethermind Security team. Additionally, the code includes NatSpec documentation for different functions and their parameters. However, the project documentation could be improved by providing a more comprehensive written overview of the system and an explanation of the different design choices.

8 Test Suite Evaluation

8.1 Compilation Output

```
> npx hardhat compile
Compiling contracts for ZKsync Era with zksolc v1.5.11 and zkvm-solc v0.8.26-1.0.1
Compiling 161 Solidity files
Warning: This declaration shadows an existing declaration.
  --> @matterlabs/zksync-contracts/l2/system-contracts/libraries/Utils.sol:85:9:
   |
85 |         uint256 bytecodeLenInWords = _bytecode.length / 32;
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> @matterlabs/zksync-contracts/l2/system-contracts/libraries/Utils.sol:43:5:
   |
43 |     function bytecodeLenInWords(bytes32 _bytecodeHash) internal pure returns (uint256 codeLengthInWords) {
   |     ^ (Relevant source part starts here and spans across multiple lines).


Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
  --> contracts/paymasters/GaslessPaymaster.sol:53:62:
   |
53 |     ) external payable onlyBootloader returns (bytes4 magic, bytes memory context) {
   |                                                              ^^^^^^^^^^^^^^^^^^^^


Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
  --> contracts/test/MockHook.sol:43:23:
   |
43 |     function isInited(address account) external view returns (bool) {
   |                       ^^^^^^^^^^^^^^^

Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
  --> contracts/test/MockHook.sol:84:23:
   |
84 |     function isInited(address account) external view returns (bool) {
   |                       ^^^^^^^^^^^^^^^


Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
  --> contracts/test/MockModule.sol:34:23:
   |
34 |     function isInited(address account) external view returns (bool) {
   |                       ^^^^^^^^^^^^^^^


Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
 --> contracts/test/MockRegistry.sol:9:22:
  |
9 |     function isClave(address account) external view returns (bool) {
  |                      ^^^^^^^^^^^^^^^


Warning: Function state mutability can be restricted to pure
   --> contracts/test/ERC20PaymasterMock.sol:217:5:
    |
217 |     function callOracle(
    |     ^ (Relevant source part starts here and spans across multiple lines).


Warning: Function state mutability can be restricted to pure
  --> contracts/test/MockAdapter.sol:15:5:
   |
15 |     function withdraw(address, uint256, uint256, bool) external override returns (uint256) {
   |     ^ (Relevant source part starts here and spans across multiple lines).
```

```
Warning: Function state mutability can be restricted to pure
  --> contracts/test/MockHook.sol:43:5:
   |
43 |     function isInited(address account) external view returns (bool) {
   |     ^ (Relevant source part starts here and spans across multiple lines).


Warning: Function state mutability can be restricted to pure
  --> contracts/test/MockHook.sol:84:5:
   |
84 |     function isInited(address account) external view returns (bool) {
   |     ^ (Relevant source part starts here and spans across multiple lines).


Warning: Function state mutability can be restricted to pure
  --> contracts/test/MockModule.sol:34:5:
   |
34 |     function isInited(address account) external view returns (bool) {
   |     ^ (Relevant source part starts here and spans across multiple lines).


Warning: Function state mutability can be restricted to pure
 --> contracts/test/MockRegistry.sol:9:5:
  |
9 |     function isClave(address account) external view returns (bool) {
  |     ^ (Relevant source part starts here and spans across multiple lines).


Warning: The `codegen` setting will become mandatory in future versions of zksolc. Please set it to either `evmla` or
 ↪  `yul`.

Generating typings for: 195 artifacts in dir: typechain-types for target: ethers-v6
Successfully generated 464 typings!
Successfully compiled 161 Solidity files
```

8.2 Tests Output

```
> npx hardhat test --grep "ClaggMain"

ClaggMain

 should set the contract deployer as the owner (1645ms)
 should set the owner (652ms)
 should request to add an adapter (2073ms)
 should not add an adapter if the request is not in the timelock (1593ms)
 should add a new adapter (3056ms)
 should remove an adapter (3719ms)
 should delegate calls to adapters (2846ms)
 should allow the contract to receive Ether (780ms)

8 passing (16s)
```

8.2.1 Slither

All the relevant issues raised by Slither have been incorporated into the issues described in this report.

8.2.2 AuditAgent

All the relevant issues raised by the AuditAgent have been incorporated into this report. The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at https://app.auditagent.nethermind.io.

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.