Competitive Security Assessment

Hashkey_Hodlium

Mar 3rd, 2025

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

- Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
- Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
- Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
- Verify the code base is compliant with the most up-to-date industry standards and security best practices.
- Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

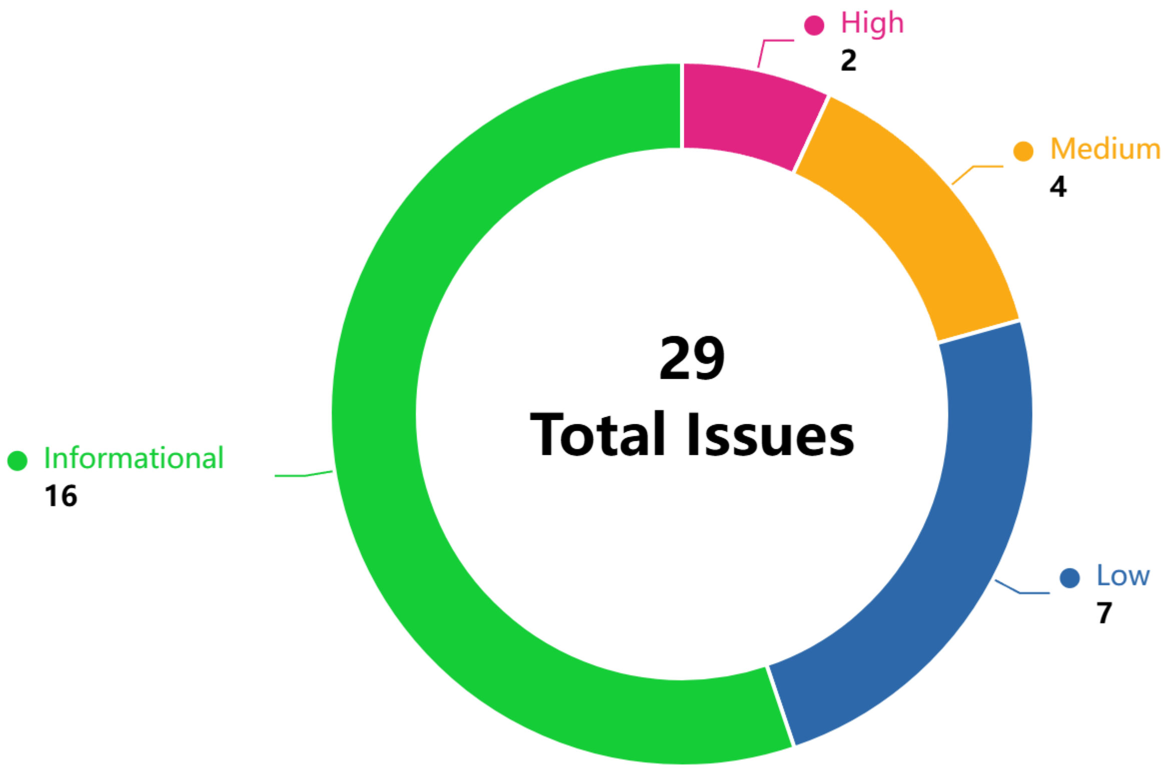
Overview

Project Name	Hashkey_Hodlium
Language	solidity
Codebase	- https://github.com/SpectreMercury/hashkey-hodlium-contract - audit version-efef045edadd382d30950ae0c9be000cc465c76b - final version-be9e017cc1d219059d06101f5545a066e10c546f

Audit Scope

File	SHA256 Hash
contracts/HashKeyChainStaking.sol	bbf6284fe0e0524e20bd962db4af3646c6a11249bc6e83427d7e10b95a039d8a
contracts/HashKeyChainStakingBase.sol	0a91db0f7f20f5aa2d55d79b5eb34fbd29e9037ba1d2b1a745acb9a2aab1aef3
contracts/HashKeyChainStakingOperations.sol	f1fc59b4f620912b6257837e242a9072fb72d9050f2e0c06565e7023e9b3049a
contracts/HashKeyChainStakingAdmin.sol	6863e319fbda345730fde0744f76f527c4c039cc88547bd0cfe77c969e13626f
contracts/HashKeyChainStakingEmergency.sol	123cf84022fe43a0fd2f2d2ded112e9bb9070a9e201ae43171260c06d76ff2a1
contracts/HashKeyChainStakingStorage.sol	1301e35ab0c6582fbaedb4fd089c62071ea148306084d0b92033f0028ce443fa
contracts/HashKeyChainStakingEvents.sol	e61a673937c1f02517700938a40cf1e4c2aaecadc78af5d229f85d3a09f3a6ed
contracts/HashKeyChainStakingProxy.sol	1197e9bf275692252e0aa521cacc887060a43d3daae12eb82101f83d8186cd66
contracts/StHSK.sol	ebe4c5dda921fd1455d446e854c4334d57228cd9739a4e2e5f4e0a1f1c986afd

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
HHM-1	Users will suffer a fund loss after the contract is upgraded	Logical	High	Fixed	***
HHM-2	Missing Lock Bonus Rewards in Unstaking Process	Logical	High	Fixed	***
HHM-3	`totalPooledHSK` Can Be Incorrectly Reduced on Early Unstake, Causing Accounting Imbalance	Logical	Medium	Fixed	***
HHM-4	The safeHskTransfer returns true even when the contract's available balance is insufficient, which will lead to financial losses	Logical	Medium	Fixed	***
HHM-5	The HashKeyChainStakingProxy may not be upgradable	Logical	Medium	Fixed	***

HHM-6	Missing Reward Pool Update in Emergency Withdrawal Functions	Logical	Medium	Fixed	***
HHM-7	if no rewards are added to the contract, users' staked principal (their original msg.value) could be treated as rewards	Logical	Low	Fixed	***
HHM-8	Users who stake via locks, and users who stake without locks gain the same rewards thereby breaking the incentive to lock their ETH in the contract, also since locked stakes incur penalties, the incentive is killed the more.	Logical	Low	Fixed	***
HHM-9	Inconsistent State Between `annualRewardsBudget` and `hskPerBlock` Leading to Incorrect APR Calculations	Logical	Low	Fixed	***
HHM-10	Hardcoded Block Time Assumption in Reward Distribution Calculation	Logical	Low	Fixed	***
HHM-11	First depositor can break minting of shares	Logical	Low	Fixed	***
HHM-12	APR is incorrectly calculated as if all rewards were distributed instantly rather than over a year	Logical	Low	Fixed	***
HHM-13	APR Miscalculation Allows Excessive Reward Distribution	Code Style	Low	Fixed	***
HHM-14	`updateRewardPool` Can Revert Due to Integer Division by Zero	Logical	Informational	Acknowledged	***
HHM-15	`public` functions not called by the contract should be declared `external` instead	Logical	Informational	Acknowledged	***

HHM-16	`getCurrentAPR` Function Lacks Support for Regular (Unlocked) Staking APR Calculation	Logical	Informational	Acknowledged	***
HHM-17	`a = a + b` is more gas effective than `a += b` for state variables (excluding arrays and mappings)	Gas Optimization	Informational	Acknowledged	***
HHM-18	Use Custom Errors instead of Revert Strings to save Gas	Logical	Informational	Acknowledged	***
HHM-19	Upgradeable contract is missing a `__gap[50]` storage variable to allow for new storage variables in later versions	Logical	Informational	Acknowledged	***
HHM-20	Unused code	Gas Optimization	Informational	Acknowledged	***
HHM-21	Unchecked ERC-20 `transfer()` and `transferFrom()` Call	Logical	Informational	Acknowledged	***
HHM-22	The parameters of the InsufficientRewards event may not accurately reflect the intended values	Logical	Informational	Acknowledged	***
HHM-23	The `recoverToken` function does not need to exclude the `stHSK` token	Logical	Informational	Acknowledged	***
HHM-24	Redundant `MAX_APR` Comparison	Logical	Informational	Acknowledged	***
HHM-25	Redundant Contract Inheritance in HashKeyChainStaking Contract	Code Style	Informational	Acknowledged	***
HHM-26	Ownership change should use two-steps process	Logical	Informational	Acknowledged	***
HHM-27	Missing Revert on Invalid Reward Rate Update	Logical	Informational	Acknowledged	***
HHM-28	Lack of APR Validation When Setting Reward Parameters	Logical	Informational	Acknowledged	***

HHM-29	Cache array length outside of loop	Gas Optimization	Informational	Acknowledged	***

HHM-1:Users will suffer a fund loss after the contract is upgraded

Category	Severity	Client Response	Contributor
Logical	High	Fixed	***

Code Reference

- code/contracts/HashKeyChainStaking.sol#L36-L59

```
function initialize(
        uint256 _hskPerBlock,
        uint256 _startBlock,
        uint256 _maxHskPerBlock,
        uint256 _minStakeAmount,
        uint256 _annualBudget
    ) public reinitializer(2) {
        // Validate inputs before proceeding
        require(_hskPerBlock > 0, "HSK per block must be positive");
        require(_startBlock >= block.number, "Start block must be in the future");
        require(_maxHskPerBlock >= _hskPerBlock, "Max HSK per block must be >= HSK per block");
        require(_minStakeAmount > 0, "Min stake amount must be positive");

        __HashKeyChainStakingBase_init(
            _hskPerBlock,
            _startBlock,
            _maxHskPerBlock,
            _minStakeAmount
        );

```

```
        if (_annualBudget > 0) {
            annualRewardsBudget = _annualBudget;
        }
    }
```

- code/contracts/HashKeyChainStakingBase.sol#L27-L73

```
function __HashKeyChainStakingBase_init(
        uint256 _hskPerBlock,
        uint256 _startBlock,
        uint256 _maxHskPerBlock,
        uint256 _minStakeAmount
    ) internal onlyInitializing {
        __Pausable_init();
        __ReentrancyGuard_init();
        __Ownable_init(msg.sender);

        require(_hskPerBlock > 0, "HSK per block must be positive");
        require(_startBlock >= block.number, "Start block must be in the future");
        require(_maxHskPerBlock >= _hskPerBlock, "Max HSK per block must be >= HSK per block");
        require(_minStakeAmount >= 100 ether, "Min stake amount must be >= 100 HSK");

        stHSK = new StHSK();

        hskPerBlock = _hskPerBlock;
        startBlock = _startBlock;
        lastRewardBlock = startBlock;
        maxHskPerBlock = _maxHskPerBlock;
        minStakeAmount = _minStakeAmount;
        totalPooledHSK = 0;
        stakeEndTime = type(uint256).max;  // Default set to maximum value
        version = 1;

        // Calculate and set default annual budget
        uint256 blocksPerYear = (365 * 24 * 3600) / 2; // Blocks per year (assuming 2 seconds per block)
        annualRewardsBudget = _hskPerBlock * blocksPerYear;

        // Set early withdrawal penalties
        earlyWithdrawalPenalty[StakeType.FIXED_30_DAYS] = 500;    // 5%
        earlyWithdrawalPenalty[StakeType.FIXED_90_DAYS] = 1000;   // 10%
        earlyWithdrawalPenalty[StakeType.FIXED_180_DAYS] = 1500;  // 15%
        earlyWithdrawalPenalty[StakeType.FIXED_365_DAYS] = 2000;  // 20%

        // Set bonus for different staking periods
        stakingBonus[StakeType.FIXED_30_DAYS] = 0;      // 0%
        stakingBonus[StakeType.FIXED_90_DAYS] = 80;     // 0.8%
        stakingBonus[StakeType.FIXED_180_DAYS] = 200;   // 2.0%
        stakingBonus[StakeType.FIXED_365_DAYS] = 400;   // 4.0%

        emit StakingContractUpgraded(version);
        emit HskPerBlockUpdated(0, _hskPerBlock);
        emit MaxHskPerBlockUpdated(0, _maxHskPerBlock);
        emit MinStakeAmountUpdated(0, _minStakeAmount);
    }
```

Description

***: ## Summary

The `HashKeyChainStaking` contract is designed to manage staking operations, allowing users to stake `HSK` tokens and earn rewards. It supports upgradability through the OpenZeppelin upgradeable contracts framework. However, a critical vulnerability exists in the initialization logic during contract upgrades. Specifically, the `initialize` function calls `__HashKeyChainStakingBase_init`, which re-deploys the `StHSK` token contract and reset `totalPo-`

`oledHSK` to 0. This results in the loss of all previously minted stHSK tokens and clear all staked amount, causing users to lose access to their staked funds and leading to potential financial losses.

Vulnerability Details

When the initialize function is called during an upgrade, it reinitializes the contract, calling `__HashKeyChainStakingBase_init` function:

```
function initialize(
        uint256 _hskPerBlock,
        uint256 _startBlock,
        uint256 _maxHskPerBlock,
        uint256 _minStakeAmount,
        uint256 _annualBudget
    ) public reinitializer(2) {
        // Validate inputs before proceeding
        require(_hskPerBlock > 0, "HSK per block must be positive");
        require(_startBlock >= block.number, "Start block must be in the future");
        require(_maxHskPerBlock >= _hskPerBlock, "Max HSK per block must be >= HSK per block");
        require(_minStakeAmount > 0, "Min stake amount must be positive");

        __HashKeyChainStakingBase_init(
            _hskPerBlock,
            _startBlock,
            _maxHskPerBlock,
            _minStakeAmount
        );
        ...
        ...
```

The vulnerability lies in the `__HashKeyChainStakingBase_init` function, where a new `StHSK contract` is deployed and `totalPooledHSK` is reset to 0:

```
function __HashKeyChainStakingBase_init(
    uint256 _hskPerBlock,
    uint256 _startBlock,
    uint256 _maxHskPerBlock,
    uint256 _minStakeAmount
) internal onlyInitializing {
    __Pausable_init();
    __ReentrancyGuard_init();
    __Ownable_init(msg.sender);

    require(_hskPerBlock > 0, "HSK per block must be positive");
    require(_startBlock >= block.number, "Start block must be in the future");
    require(_maxHskPerBlock >= _hskPerBlock, "Max HSK per block must be >= HSK per block");
    require(_minStakeAmount >= 100 ether, "Min stake amount must be >= 100 HSK");

    // Vulnerable line: Re-deploys StHSK on every initialization
    stHSK = new StHSK();
    hskPerBlock = _hskPerBlock;
    startBlock = _startBlock;
    lastRewardBlock = startBlock;
    maxHskPerBlock = _maxHskPerBlock;
    minStakeAmount = _minStakeAmount;
    //Vulnerable line: reset to 0
    totalPooledHSK = 0;
    // Other initialization logic...
}
```

Re-deploying `StHSK` during upgrades discards the previous `StHSK` instance. Users' stHSK balances, which represent their staked HSK, are lost. Reseting `totalPooledHSK` to 0 will clear all staked amount in the contract. As a result, users cannot withdraw their funds and will suffer a fund loss.

Recommendation

***: Add a check to prevent re-deployment of StHSK:

```
function __HashKeyChainStakingBase_init(
    uint256 _hskPerBlock,
    uint256 _startBlock,
    uint256 _maxHskPerBlock,
    uint256 _minStakeAmount
) internal onlyInitializing {
    __Pausable_init();
    __ReentrancyGuard_init();
    __Ownable_init(msg.sender);

    require(_hskPerBlock > 0, "HSK per block must be positive");
    require(_startBlock >= block.number, "Start block must be in the future");
    require(_maxHskPerBlock >= _hskPerBlock, "Max HSK per block must be >= HSK per block");
    require(_minStakeAmount >= 100 ether, "Min stake amount must be >= 100 HSK");

    // Only deploy StHSK if it hasn't been deployed yet
    if (address(stHSK) == address(0)) {
        stHSK = new StHSK();
    }
    stHSK = new StHSK();
    hskPerBlock = _hskPerBlock;
    startBlock = _startBlock;
    lastRewardBlock = startBlock;
    maxHskPerBlock = _maxHskPerBlock;
    minStakeAmount = _minStakeAmount;
    //do not reset to 0
    //totalPooledHSK = 0;
    // Other initialization logic...
}
```

Client Response

client response : Fixed. solve here: SpectreMercury/hashkey-hodlium-contract#1

HHM-2:Missing Lock Bonus Rewards in Unstaking Process

Category	Severity	Client Response	Contributor
Logical	High	Fixed	***

Code Reference

- code/contracts/HashKeyChainStakingBase.sol#L100-L102

```
100: totalPooledHSK += hskReward;
101:              reservedRewards -= hskReward;
```

- code/contracts/HashKeyChainStakingStorage.sol#L48

```
48: mapping(StakeType => uint256) public stakingBonus;
```

Description

*****:** The current implementation has a critical flaw in its reward distribution mechanism that can lead to unfair distribution of rewards and potential loss of funds for users. The issue stems from how rewards are calculated and added to the total pool without considering the bonus rewards promised to users with locked stakes.

1. The base reward calculation is done as `multiplier * hskPerBlock`

```
uint256 hskReward = multiplier * hskPerBlock;
```

2. This reward is directly added to totalPooledHSK:

```
totalPooledHSK += hskReward;
```

3. However, the contract also maintains a stakingBonus mapping that provides additional rewards based on stake duration.

```
mapping(StakeType => uint256) public stakingBonus;
```

4. And these bonuses are considered in the APR calculation:

```
uint256 totalApr = baseApr + stakingBonus[_stakeType];
```

This creates a significant discrepancy because:

1. Users are promised a higher APR based on their lock duration (`base + bonus`)
2. But the actual rewards added to the pool only account for the base reward

3. When users withdraw, they receive their share of the `totalPooledHSK`, which doesn't properly account for the promised bonus rewards(the `stakingBonus` is nowhere else to be used)
4. This leads to a situation where users with longer lock periods are effectively underpaid.

This is considered a HIGH severity issue because:

1. It directly impacts the economic model of the protocol
2. Results in financial loss for users who chose longer staking periods
3. Breaks the promised reward structure
4. The impact increases with the total value locked and the duration of staking

Recommendation

***: Modify the reward calculation to account for bonus rewards or Implement a proper tracking system for bonus rewards.

Client Response

client response : Fixed. solve here : SpectreMercury/hashkey-hodlium-contract#1

HHM-3: `totalPooledHSK` Can Be Incorrectly Reduced on Early Unstake, Causing Accounting Imbalance

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	***

Code Reference

- code/contracts/HashKeyChainStakingOperations.sol#L80-L138

```
80: function unstakeLocked(uint256 _stakeId) external nonReentrant {
81:         require(_stakeId < lockedStakes[msg.sender].length, "Invalid stake ID");
82:         LockedStake storage lockedStake = lockedStakes[msg.sender][_stakeId];
83:
84:         require(!lockedStake.withdrawn, "Stake already withdrawn");
85:         require(lockedStake.sharesAmount > 0, "Stake amount is zero");
86:
87:         // Update reward pool
88:         updateRewardPool();
89:
90:         // Check if early withdrawal
91:         bool isEarlyWithdrawal = (block.timestamp < lockedStake.lockEndTime);
92:
93:         // Calculate penalty (if early withdrawal)
94:         uint256 penalty = 0;
95:         uint256 sharesToBurn = lockedStake.sharesAmount;
96:         uint256 hskToReturn = getHSKForShares(sharesToBurn);
97:
98:         if (isEarlyWithdrawal) {
99:             // Determine stake type
```

```
100:            StakeType stakeType;
101:            if (lockedStake.lockDuration == 30 days) stakeType = StakeType.FIXED_30_DAYS;
102:            else if (lockedStake.lockDuration == 90 days) stakeType = StakeType.FIXED_90_DAYS;
103:            else if (lockedStake.lockDuration == 180 days) stakeType = StakeType.FIXED_180_DAYS;
104:            else stakeType = StakeType.FIXED_365_DAYS;
105:
106:            // Calculate elapsed lock period ratio
107:            uint256 elapsedTime = block.timestamp - (lockedStake.lockEndTime - lockedStake.lockDuration);
108:            uint256 completionRatio = (elapsedTime * BASIS_POINTS) / lockedStake.lockDuration;
109:
110:            // Adjust penalty based on completion (higher completion, lower penalty)
111:            uint256 adjustedPenalty = earlyWithdrawalPenalty[stakeType] * (BASIS_POINTS - completionRatio) / BASIS_POINTS;
112:
113:            // Apply penalty
114:            penalty = (hskToReturn * adjustedPenalty) / BASIS_POINTS;
115:            hskToReturn -= penalty;
116:
117:            // Add penalty to reserved rewards
118:            reservedRewards += penalty;
119:        }
```

```

        // Mark stake as withdrawn
        lockedStake.withdrawn = true;

        // Update total staked amount
        totalPooledHSK -= hskToReturn + penalty;

        // Check if user has enough stHSK
        require(stHSK.balanceOf(msg.sender) >= sharesToBurn, "Insufficient stHSK balance");

        // Burn stHSK tokens
        stHSK.burn(msg.sender, sharesToBurn);

        // Return HSK tokens
        bool transferSuccess = safeHskTransfer(payable(msg.sender), hskToReturn);
        require(transferSuccess, "HSK transfer failed");

        emit Unstake(msg.sender, sharesToBurn, hskToReturn, isEarlyWithdrawal, penalty, _stakeId);
    }
```

Description

*****:** In `unstakeLocked`, when a user withdraws their locked stake early, the penalty amount is deducted from their unstake amount and added to `reservedRewards`. However, the contract **incorrectly reduces `totalPooledHSK` by the full amount (including the penalty)**:

```
// Update total staked amount
totalPooledHSK -= hskToReturn + penalty;
```

Here, `penalty` is transferred to `reservedRewards`, meaning it **still belongs to the staking contract**. Since the penalty is not actually removed from the system, deducting it from `totalPooledHSK` **artificially reduces the recorded staking pool size**, creating an accounting imbalance. Over time, as more users unstake early and penalties accumulate, `totalPooledHSK` will become significantly lower than the actual HSK tokens in circulation, leading to incorrect reward calculations and a broken staking system.

Impact:

By subtracting the penalty from `totalPooledHSK` while also adding it to `reservedRewards`, the contract double-counts the penalty as removed, artificially lowering the recorded staking pool size. As more users unstake early, `totalPooledHSK` drifts further from reality, leading to miscalculated APR, incorrect reward distributions, and potential depletion of rewards faster than intended, ultimately destabilizing the staking system.

Recommendation

*****:** Modify the `unstakeLocked` function to reduce `totalPooledHSK` by `hskToReturn` only, while keeping the penalty amount in the system:

```
totalPooledHSK -= hskToReturn;
```

Client Response

client response : Fixed. solve: SpectreMercury/hashkey-hodlium-contract#2

HHM-4:The safeHskTransfer returns true even when the contract's available balance is insufficient, which will lead to financial losses

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	***

Code Reference

- code/contracts/HashKeyChainStakingBase.sol#L150-L165

```
/**
     * @dev Safe HSK transfer function
     * @param _to Recipient address
     * @param _amount Amount
     * @return Whether transfer was successful
     */
    function safeHskTransfer(address payable _to, uint256 _amount) internal returns (bool) {
        uint256 availableBalance = address(this).balance - totalPooledHSK;
        uint256 amountToSend = _amount > availableBalance ? availableBalance : _amount;

        if (amountToSend > 0) {
            (bool success, ) = _to.call{value: amountToSend}("");
            return success;
        }
        return true;
    }
```

- code/contracts/HashKeyChainStakingOperations.sol#L80-L165

```
function unstakeLocked(uint256 _stakeId) external nonReentrant {
        require(_stakeId < lockedStakes[msg.sender].length, "Invalid stake ID");
        LockedStake storage lockedStake = lockedStakes[msg.sender][_stakeId];

        require(!lockedStake.withdrawn, "Stake already withdrawn");
        require(lockedStake.sharesAmount > 0, "Stake amount is zero");

        // Update reward pool
        updateRewardPool();

        // Check if early withdrawal
        bool isEarlyWithdrawal = (block.timestamp < lockedStake.lockEndTime);

        // Calculate penalty (if early withdrawal)
        uint256 penalty = 0;
        uint256 sharesToBurn = lockedStake.sharesAmount;
        uint256 hskToReturn = getHSKForShares(sharesToBurn);

        if (isEarlyWithdrawal) {
            // Determine stake type
            StakeType stakeType;
            if (lockedStake.lockDuration == 30 days) stakeType = StakeType.FIXED_30_DAYS;
            else if (lockedStake.lockDuration == 90 days) stakeType = StakeType.FIXED_90_DAYS;
            else if (lockedStake.lockDuration == 180 days) stakeType = StakeType.FIXED_180_DAYS;
            else stakeType = StakeType.FIXED_365_DAYS;

            // Calculate elapsed lock period ratio
            uint256 elapsedTime = block.timestamp - (lockedStake.lockEndTime - lockedStake.lockDuration);
            uint256 completionRatio = (elapsedTime * BASIS_POINTS) / lockedStake.lockDuration;

            // Adjust penalty based on completion (higher completion, lower penalty)
            uint256 adjustedPenalty = earlyWithdrawalPenalty[stakeType] * (BASIS_POINTS - completionRatio) / BASIS_POINTS;

            // Apply penalty
            penalty = (hskToReturn * adjustedPenalty) / BASIS_POINTS;
            hskToReturn -= penalty;

            // Add penalty to reserved rewards
            reservedRewards += penalty;
        }
```

```

        // Mark stake as withdrawn
        lockedStake.withdrawn = true;

        // Update total staked amount
        totalPooledHSK -= hskToReturn + penalty;

        // Check if user has enough stHSK
        require(stHSK.balanceOf(msg.sender) >= sharesToBurn, "Insufficient stHSK balance");

        // Burn stHSK tokens
        stHSK.burn(msg.sender, sharesToBurn);

        // Return HSK tokens
        bool transferSuccess = safeHskTransfer(payable(msg.sender), hskToReturn);
        require(transferSuccess, "HSK transfer failed");

        emit Unstake(msg.sender, sharesToBurn, hskToReturn, isEarlyWithdrawal, penalty, _stakeId);
    }

    /**
     * @dev Unstake regular (unlocked) stake
     * @param _sharesAmount Share amount to unstake
     */
    function unstake(uint256 _sharesAmount) external nonReentrant {
        require(_sharesAmount > 0, "Cannot unstake 0");
        require(stHSK.balanceOf(msg.sender) >= _sharesAmount, "Insufficient stHSK balance");

        // Update reward pool
        updateRewardPool();

        // Calculate HSK amount to return
        uint256 hskToReturn = getHSKForShares(_sharesAmount);

        // Update total staked amount
        totalPooledHSK -= hskToReturn;

        // Burn stHSK tokens
        stHSK.burn(msg.sender, _sharesAmount);

```

```
        // Return HSK tokens
        bool success = safeHskTransfer(payable(msg.sender), hskToReturn);
        require(success, "HSK transfer failed");

        emit Unstake(msg.sender, _sharesAmount, hskToReturn, false, 0, type(uint256).max);
    }
```

Description

***: ## Summary
The `unstakeLocked` and `unstake` functions in the `HashKeyChainStakingOperations` contract both rely on the `safeHskTransfer` function to transfer funds to users when they withdraw their staked tokens. However, the current implementation of `safeHskTransfer` contains a critical vulnerability: it returns `true` even when the contract's available balance is insufficient to cover the requested transfer amount. This behavior can lead to users not receiving their funds while the contract incorrectly assumes the transfer was successful, resulting in potential financial losses for users.
This issue stems from the logic in `safeHskTransfer` that defaults to returning `true` when the available balance is insufficient, rather than failing explicitly. This vulnerability can be exploited if the contract's balance is depleted, causing users to lose their funds without proper error handling.

Vulnerability Details

Here is the implementation of `safeHskTransfer` function:

```
 function safeHskTransfer(address payable _to, uint256 _amount) internal returns (bool) {
        uint256 availableBalance = address(this).balance - totalPooledHSK;
        uint256 amountToSend = _amount > availableBalance ? availableBalance : _amount;

        if (amountToSend > 0) {
            (bool success, ) = _to.call{value: amountToSend}("");
            return success;
        }
        return true;
    }
```

If the contract's available balance (`address(this).balance - totalPooledHSK`) is less than the requested `_amount` , the function calculates `amountToSend` as the available balance.
If `amountToSend` is `0` (i.e., no funds are available), the function still returns `true` , indicating a successful transfer, even though no funds were sent.
Both `unstakeLocked` and `unstake` functions call `safeHskTransfer` and check its return value:

```
// Return HSK tokens
        bool success = safeHskTransfer(payable(msg.sender), hskToReturn);
        require(success, "HSK transfer failed");
```

Since `safeHskTransfer` returns true even when no funds are sent, the require statement does not revert the transaction, and the contract proceeds as if the transfer was successful. This results in users losing their staked funds without receiving anything in return.

Recommendation

***: Consider following fix:

```
function safeHskTransfer(address payable _to, uint256 _amount) internal returns (bool) {
    uint256 availableBalance = address(this).balance - totalPooledHSK;

    require(availableBalance >= _amount, "Insufficient contract balance");

    (bool success, ) = _to.call{value: _amount}("");
    require(success, "HSK transfer failed");
    return true;
}
```

Client Response

client response : Fixed. solve: SpectreMercury/hashkey-hodlium-contract#2

HHM-5:The HashKeyChainStakingProxy may not be upgradable

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	***

Code Reference

- code/contracts/HashKeyChainStakingProxy.sol#L11-L25

```
11: contract HashKeyChainStakingProxy is TransparentUpgradeableProxy {
12:     constructor(
13:         address _logic,
14:         address _admin,
15:         bytes memory _data
16:     ) payable TransparentUpgradeableProxy(_logic, _admin, _data) {}
17: }
18:
19: /**
20:  * @title HashKeyChainStakingProxyAdmin
21:  * @dev Admin contract for managing proxy upgrades
22:  */
23: contract HashKeyChainStakingProxyAdmin is ProxyAdmin {
24:     constructor() ProxyAdmin(msg.sender) {}
25: }
```

Description

***: In `HashKeyChainStakingProxy.sol`, there is a `HashKeyChainStakingProxyAdmin` contract which commet says it is the admin contract for managing proxy upgrades:

```
/**
 * @title HashKeyChainStakingProxyAdmin
 * @dev Admin contract for managing proxy upgrades
 */
contract HashKeyChainStakingProxyAdmin is ProxyAdmin {
    constructor() ProxyAdmin(msg.sender) {}
}
```

If `HashKeyChainStakingProxyAdmin` is used as the admin of `HashKeyChainStakingProxy`, it may lead to the inability to upgrade the contract in the future.
The `HashKeyChainStakingProxy` extends from `TransparentUpgradeableProxy`:

```
contract HashKeyChainStakingProxy is TransparentUpgradeableProxy {
    constructor(
        address _logic,
        address _admin,
        bytes memory _data
    ) payable TransparentUpgradeableProxy(_logic, _admin, _data) {}
}
```

In OpenZeppelin's `TransparentUpgradeableProxy` implementation(https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v5.2.0/contracts/proxy/transparent/TransparentUpgradeableProxy.sol#L74-L83), the constructor performs the following actions:

```
 /**
     * @dev Initializes an upgradeable proxy managed by an instance of a {ProxyAdmin} with an `initialOwner
`,
     * backed by the implementation at `_logic`, and optionally initialized with `_data` as explained in
     * {ERC1967Proxy-constructor}.
     */
    constructor(address _logic, address initialOwner, bytes memory _data) payable ERC1967Proxy(_logic, _dat
a) {
        _admin = address(new ProxyAdmin(initialOwner));
        // Set the storage value and emit an event for ERC-1967 compatibility
        ERC1967Utils.changeAdmin(_proxyAdmin());
    }
```

The constructor creates a new `ProxyAdmin` instance using new `ProxyAdmin(initialOwner)`.
The `initialOwner` address is passed as the owner of the newly created `HashKeyChainStakingProxyAdmin` which is also a `ProxyAdmin`. As a result, the `HashKeyChainStakingProxyAdmin` is the onwer of the new `ProxyAdmin` instance.
To upgrade the proxy, it must call the `upgradeAndCall` function in `ProxyAdmin` contract(https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v5.2.0/contracts/proxy/transparent/ProxyAdmin.sol#L38-L44):

```
function upgradeAndCall(
        ITransparentUpgradeableProxy proxy,
        address implementation,
        bytes memory data
    ) public payable virtual onlyOwner {
        proxy.upgradeToAndCall{value: msg.value}(implementation, data);
    }
```

Here the onwer is `HashKeyChainStakingProxyAdmin` contract. So we have to call some functions in `HashKeyChainStakingProxyAdmin` contract to call the `ProxyAdmin`'s `upgradeAndCall` function. The execution flow should be:

```
HashKeyChainStakingProxyAdmin::someFunction ---> ProxyAdmin::upgradeAndCall ---> HashKeyChainStakingProxy
```

However, in `HashKeyChainStakingProxyAdmin` contract there is no function to call the `ProxyAdmin`'s `upgradeAndCall` function. As a result, the `HashKeyChainStakingProxy` can not be upgraded any more.

Recommendation

*****:** When deploying `HashKeyChainStakingProxy`, pass the address of EOA address instead of the `HashKeyChainStakingProxyAdmin` as the `initialOwner`.

Client Response

client response : Fixed. solve: SpectreMercury/hashkey-hodlium-contract#2

HHM-6:Missing Reward Pool Update in Emergency Withdrawal Functions

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	***

Code Reference

- code/contracts/HashKeyChainStakingEmergency.sol#L15
- code/contracts/HashKeyChainStakingEmergency.sol#L47-L48

```
15: function emergencyWithdraw() external nonReentrant {
```

```
47: function emergencyWithdrawHSK(uint256 _amount) external onlyOwner {
48:         uint256 availableBalance = address(this).balance - totalPooledHSK;
```

Description

***: The emergencyWithdraw() and emergencyWithdrawHSK() functions fail to call updateRewardPool() before executing withdrawals, which can lead to incorrect reward accounting and potential loss of rewards for users.

```
    function emergencyWithdraw() external nonReentrant {
        uint256 shareBalance = stHSK.balanceOf(msg.sender);
        require(shareBalance > 0, "Nothing to withdraw");
        ...
    }
```

This is particularly problematic because when users perform emergency withdrawals, their accumulated rewards up to that point are not properly accounted for and may be lost.

Recommendation

***: Add `updateRewardPool` calls at the beginning of both emergency withdrawal functions.

Client Response

client response : Fixed. solve: SpectreMercury/hashkey-hodlium-contract#2

HHM-7:if no rewards are added to the contract, users' staked principal (their original msg.value) could be treated as rewards

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/contracts/HashKeyChainStakingBase.sol#L125-L148

```
function getHSKForShares(uint256 _sharesAmount) public view returns (uint256) {
        uint256 totalShares = stHSK.totalSupply();
        if (totalShares == 0) {
            return _sharesAmount; // Initial 1:1 exchange rate
        }
        return (_sharesAmount * totalPooledHSK) / totalShares;
    }

    /**
     * @dev Calculate share amount for specified HSK amount
     * @param _hskAmount HSK amount
     * @return Share amount
     */
    function getSharesForHSK(uint256 _hskAmount) public view returns (uint256) {
        uint256 totalShares = stHSK.totalSupply();

        // Initial 1:1 exchange rate
        if (totalShares == 0 || totalPooledHSK == 0) {
            return _hskAmount;
        }

        // Calculate shares based on the current pool ratio
        return (_hskAmount * totalShares) / totalPooledHSK;
    }
```

- code/contracts/HashKeyChainStakingOperations.sol#L14-L74

```
function stake() external payable nonReentrant whenNotPaused {
        // Strict validation of minimum stake amount
        require(msg.value >= minStakeAmount, "Amount below minimum stake");
        require(block.timestamp < stakeEndTime, "Staking ended");

        // Update reward pool
        updateRewardPool();

        // Calculate shares to mint
        uint256 sharesAmount = getSharesForHSK(msg.value);

        // Update total staked amount
        totalPooledHSK += msg.value;

        // Mint stHSK tokens
        stHSK.mint(msg.sender, sharesAmount);

        emit Stake(msg.sender, msg.value, sharesAmount, StakeType.FIXED_30_DAYS, 0, 0);
    }

```

```
    /**
     * @dev Locked staking, with fixed lock period and additional rewards
     * @param _stakeType Stake type (determines lock period and rewards)
     */
    function stakeLocked(StakeType _stakeType) external payable nonReentrant whenNotPaused {
        // Strict validation of minimum stake amount
        require(msg.value >= minStakeAmount, "Amount below minimum stake");
        require(block.timestamp < stakeEndTime, "Staking ended");

        // Update reward pool
        updateRewardPool();

        // Calculate shares to mint
        uint256 sharesAmount = getSharesForHSK(msg.value);

        // Determine lock period
        uint256 lockDuration;
        if (_stakeType == StakeType.FIXED_30_DAYS) lockDuration = 30 days;
        else if (_stakeType == StakeType.FIXED_90_DAYS) lockDuration = 90 days;
        else if (_stakeType == StakeType.FIXED_180_DAYS) lockDuration = 180 days;
        else if (_stakeType == StakeType.FIXED_365_DAYS) lockDuration = 365 days;
        else revert("Invalid stake type");

        // Create locked stake record
        lockedStakes[msg.sender].push(LockedStake({
            sharesAmount: sharesAmount,
            hskAmount: msg.value,
            lockEndTime: block.timestamp + lockDuration,
            lockDuration: lockDuration,
            withdrawn: false
        }));

        // Update total staked amount
        totalPooledHSK += msg.value;

        // Mint stHSK tokens
        stHSK.mint(msg.sender, sharesAmount);

        uint256 stakeId = lockedStakes[msg.sender].length - 1;
        emit Stake(msg.sender, msg.value, sharesAmount, _stakeType, block.timestamp + lockDuration, stakeId);
```

```
    }
```

Description

***: ## Summary
In `HashKeyChainStakingOperations` contract, the `stake` and `stakeLocked` functions add `msg.value` (the staked principal) to `totalPooledHSK`. The `getHSKForShares` function calculates the HSK amount to return during unstaking based on the ratio of shares to `totalPooledHSK`. If no rewards are added to the contract, the staked principal is treated as part of the rewards pool, leading to the following issues:

1. Users may not receive their full staked principal back during unstaking.
2. The staked principal is incorrectly treated as rewards, leading to unfair distribution.
3. The staking system fails to maintain a clear separation between principal and rewards, undermining its economic model.

Vulnerability Details

The core issue lies in the way `totalPooledHSK` is used to track both staked principal and rewards. When users stake funds, their principal is added to `totalPooledHSK`:

```
function stake() external payable nonReentrant whenNotPaused {
        ...
        ...

        // Update total staked amount
        totalPooledHSK += msg.value;

        // Mint stHSK tokens
        stHSK.mint(msg.sender, sharesAmount);

        emit Stake(msg.sender, msg.value, sharesAmount, StakeType.FIXED_30_DAYS, 0, 0);
    }
```

During unstaking, `getHSKForShares` calculates the HSK amount based on the ratio of shares to `totalPooledHSK`:

```
function unstake(uint256 _sharesAmount) external nonReentrant {
        require(_sharesAmount > 0, "Cannot unstake 0");
        require(stHSK.balanceOf(msg.sender) >= _sharesAmount, "Insufficient stHSK balance");

        // Update reward pool
        updateRewardPool();

        // Calculate HSK amount to return
        uint256 hskToReturn = getHSKForShares(_sharesAmount);

        // Update total staked amount
        totalPooledHSK -= hskToReturn;


        ...
        ...
    }
```

```
function getHSKForShares(uint256 _sharesAmount) public view returns (uint256) {
        uint256 totalShares = stHSK.totalSupply();
        if (totalShares == 0) {
            return _sharesAmount; // Initial 1:1 exchange rate
        }
        return (_sharesAmount * totalPooledHSK) / totalShares;
    }
```

If no rewards are added, the staked principal is effectively treated as rewards, leading to incorrect calculations and potential loss of user funds.

Scenario Breakdown

User A Stakes 1000 HSK:

1. User A stakes 1000 HSK.
2. `totalPooledHSK` becomes 1000 HSK (assuming no rewards yet).

User A Earns 100 HSK in Rewards:

1. After some time, User A earns 100 HSK in rewards.
2. `totalPooledHSK` is 1000 HSK (no rewards added to the contract).

User B Stakes 1000 HSK:

1. User B stakes 1000 HSK.
2. `totalPooledHSK` becomes 2000 HSK (1000 + 1000).

User A Unstakes 1000 HSK + 100 HSK Rewards:

1. User A unstakes their 1000 HSK principal and 100 HSK rewards.
2. `totalPooledHSK` becomes 900 HSK (2000 - 1100).

User B Attempts to Unstake 1000 HSK:

1. User B tries to unstake their 1000 HSK.
   However, the contract only has 900 HSK left, which is less than User B's staked principal.

The issue arises because the contract does not maintain a clear separation between staked principal and rewards. When User A unstakes their principal and rewards, the contract deducts both from `totalPooledHSK`, leaving insufficient funds for User B to unstake their principal and rewards.

Recommendation

***: To fix this issue, separate the tracking of staked principal and rewards. This can be achieved by introducing a new state variable, `totalStakedPrincipal`, to track the total staked principal separately from `totalPooledHSK` (which will now only track rewards). The getHSKForShares function should calculate the HSK amount based on both the staked principal and rewards.

Client Response

client response : Fixed. hashkey-hodlium-contract/pull/3

HHM-8:Users who stake via locks, and users who stake without locks gain the same rewards thereby breaking the incentive to lock their ETH in the contract, also since locked stakes incur penalties, the incentive is killed the more.

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/contracts/HashKeyChainStakingBase.sol#L130C16-L130C63

```
NaN: return (_sharesAmount * totalPooledHSK) / totalShares;
```

Description

***: ## summary
users who lock the ETH, are meant to get more rewards especially based on the amount of time they staked in the contract, however, due to how the rewards are distributed they get exactly the same rewards as those who didn't lock their ETH in the contract.

Details

On the call to:

- `stake()`
- `stakeLocked()`
- `unstake()`
- `unstakeLocked()`

`updateRewardPool();` is called, which adds to the `totalPooledHSK` the amount of rewards that should be distributed to everyone who has a stake based on the number of shares they have.
`(_sharesAmount * totalPooledHSK) / totalShares;` where you everyone with shares will get higher amounts than what they previously staked.
The problem is the formula is the same with exactly the same shared variables for anyone staking or unstaking, so if a user who didn't lock their ETH unstakes after a period of time, they get exactly the same rewards as a person who locked their stake for exactly that same period, and if they decide to unstake there is no penalty whatsoever deducted from their withdrawn ETH, but anyone who locked their stake whilst having same rewards will incur a penalty if users are aware of this, the entire incentive model is broken as no one will see the need to lock their ETH in the contract as there is no added benefit.

Recommendation

***: ## Mitigation
Ensure users who lock their stake gain more rewards than users who don't lock their stake for the same amount of time.

Client Response

client response : Fixed. hashkey-hodlium-contract/pull/3

HHM-9:Inconsistent State Between `annualRewardsBudget` and `hskPerBlock` Leading to Incorrect APR Calculations

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/contracts/HashKeyChainStakingAdmin.sol#L15-L24
- code/contracts/HashKeyChainStakingAdmin.sol#L104-L109

```
15: function updateHskPerBlock(uint256 _hskPerBlock) external onlyOwner {
16:         require(_hskPerBlock <= maxHskPerBlock, "Exceeds maximum HSK per block");
17:
18:         updateRewardPool();
19:         uint256 oldValue = hskPerBlock;
20:         hskPerBlock = _hskPerBlock;
21:
22:         emit HskPerBlockUpdated(oldValue, _hskPerBlock);
23:     }
```

```
104: */
105:     function setAnnualRewardsBudget(uint256 _annualBudget) external onlyOwner {
106:         uint256 oldValue = annualRewardsBudget;
107:         annualRewardsBudget = _annualBudget;
108:
109:         // Calculate corresponding reward per block
```

Description

*****:** The contract maintains two interconnected variables: `annualRewardsBudget` and `hskPerBlock`, which should remain synchronized to ensure accurate APR calculations.
However, the `updateHskPerBlock` function only updates `hskPerBlock` without adjusting the `annualRewardsBudget`, creating a state inconsistency.
In `setAnnualRewardsBudget`, the contract correctly maintains synchronization by:

- Updating `annualRewardsBudget`
- Calculating and updating `hskPerBlock` based on the formula:

```
    uint256 blocksPerYear = (365 * 24 * 3600) / 2;
    uint256 newHskPerBlock = _annualBudget / blocksPerYear;
```

However, `updateHskPerBlock` breaks this synchronization:

```
    function updateHskPerBlock(uint256 _hskPerBlock) external onlyOwner {
        require(_hskPerBlock <= maxHskPerBlock, "Exceeds maximum HSK per block");

        updateRewardPool();
        uint256 oldValue = hskPerBlock;
        hskPerBlock = _hskPerBlock;

        emit HskPerBlockUpdated(oldValue, _hskPerBlock);
    }
```

This is considered a HIGH severity issue because:

- It directly affects the economic calculations (APR) that users rely on for decision-making
- The inconsistency could lead to misleading APR representations

Recommendation

***: Update the function `updateHskPerBlock` to maintain synchronization

Client Response

client response : Fixed. hashkey-hodlium-contract/pull/3

HHM-10:Hardcoded Block Time Assumption in Reward Distribution Calculation

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/contracts/HashKeyChainStakingAdmin.sol#L110

```
110: uint256 blocksPerYear = (365 * 24 * 3600) / 2; // Blocks per year (assuming 2 seconds per block)
```

- code/contracts/HashKeyChainStakingBase.sol#L54

```
54: uint256 blocksPerYear = (365 * 24 * 3600) / 2; // Blocks per year (assuming 2 seconds per block)
```

Description

***: The hardcoded 2-second block time assumption appears in multiple locations:
This assumption is problematic because:

1. Block times can change due to network upgrades
2. Actual block times may vary from target times due to network conditions
3. Different blockchain networks have different block times

Impact

1. Incorrect reward pool updates:
    - APR limiting calculation in updateRewardPool() will use wrong time assumptions
    - Could lead to over or under-distribution of rewards
2. Inaccurate APR calculations:
    - getCurrentAPR() relies on annualRewardsBudget which was calculated incorrectly
    - Users will see incorrect APR estimates

Example Scenario

If deployed on Ethereum (12s block time):

- All reward calculations will be off by a factor of 6
- APR estimates will be incorrect
- Reward distribution will be much slower than intended

Recommendation

***:

1. Add a configurable block time parameter to the base contract:

```
contract HashKeyChainStakingBase {
    uint256 public averageBlockTimeInSeconds;

    function __HashKeyChainStakingBase_init(
        uint256 _hskPerBlock,
        uint256 _startBlock,
        uint256 _maxHskPerBlock,
        uint256 _minStakeAmount,
        uint256 _blockTime
    ) internal onlyInitializing {
        require(_blockTime > 0, "Block time must be positive");
        averageBlockTimeInSeconds = _blockTime;

        uint256 blocksPerYear = (365 * 24 * 3600) / averageBlockTimeInSeconds;
        annualRewardsBudget = _hskPerBlock * blocksPerYear;
        // ... rest of initialization ...
    }

    function updateRewardPool() public {
        // ... existing code ...
        uint256 annualReward = hskPerBlock * (365 days / averageBlockTimeInSeconds);
        // ... rest of function ...
    }
}
```

2. Add admin function to update block time:

```
function updateAverageBlockTime(uint256 _newBlockTime) external onlyOwner {
    require(_newBlockTime > 0, "Block time must be positive");
    uint256 oldValue = averageBlockTimeInSeconds;
    averageBlockTimeInSeconds = _newBlockTime;

    // Recalculate annual budget based on new block time
    uint256 blocksPerYear = (365 * 24 * 3600) / averageBlockTimeInSeconds;
    annualRewardsBudget = hskPerBlock * blocksPerYear;

    emit AverageBlockTimeUpdated(oldValue, _newBlockTime);
}
```

Client Response

client response : Fixed. hashkey-hodlium-contract/pull/3

HHM-11:First depositor can break minting of shares

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/contracts/HashKeyChainStakingOperations.sol#L14
- code/contracts/HashKeyChainStakingOperations.sol#L38

```
14: function stake() external payable nonReentrant whenNotPaused {
```

```
38: function stakeLocked(StakeType _stakeType) external payable nonReentrant whenNotPaused {
```

Description

*****: The attack vector and impact is the same as TOB-YEARN-003, where users may not receive shares in exchange for their deposits if the total asset amount has been manipulated through a large “donation” when calling `stake()` or `stakeLocked` function.
Proof of Concept:

1. Attacker deposits 1 wei (if the minStakeAmount is too small) to mint 1 share
2. Attacker transfers exorbitant amount to `HashKeyChainStaking` to greatly inflate the share’s price,(e.g. totalPooledHSK).
3. Subsequent depositors instead have to deposit an equivalent sum to avoid minting 0 shares. Otherwise, their deposits accrue to the attacker who holds the only share.

```
    function stake() external payable nonReentrant whenNotPaused {
        // Strict validation of minimum stake amount
        require(msg.value >= minStakeAmount, "Amount below minimum stake");
        require(block.timestamp < stakeEndTime, "Staking ended");

        // Update reward pool
        updateRewardPool();

        // Calculate shares to mint
        uint256 sharesAmount = getSharesForHSK(msg.value);

        // Update total staked amount
        totalPooledHSK += msg.value;
        stHSK.mint(msg.sender, sharesAmount);

        emit Stake(msg.sender, msg.value, sharesAmount, StakeType.FIXED_30_DAYS, 0, 0);
}
```

Recommendation

***: 1. Uniswap V2 solved this problem by sending the first 1000 LP tokens to the zero address. The same can be done in this case i.e. when `totalShares == 0`, send the first min liquidity LP tokens to the zero address to enable share dilution.
2. Ensure the number of shares to be minted is non-zero: `require(sharesAmount != 0, "zero shares minted");`

Client Response

client response : Fixed. hashkey-hodlium-contract/pull/3

HHM-12:APR is incorrectly calculated as if all rewards were distributed instantly rather than over a year

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/contracts/HashKeyChainStakingBase.sol#L173-L207

```
function getCurrentAPR(uint256 _stakeAmount, StakeType _stakeType) public view returns (uint256) {
        // Base APR calculation - using annual budget instead of per-block rewards
        uint256 yearlyRewards = annualRewardsBudget;

        uint256 baseApr;
        if (totalPooledHSK == 0) {
            baseApr = MAX_APR;
        } else {
            uint256 newTotal = totalPooledHSK + _stakeAmount;
            baseApr = (yearlyRewards * BASIS_POINTS) / newTotal;

            // Ensure not exceeding maximum APR
            if (baseApr > MAX_APR) {
                baseApr = MAX_APR;
            }
        }

        // Add staking duration bonus
        uint256 totalApr = baseApr + stakingBonus[_stakeType];

        // Get maximum APR for corresponding stake type
        uint256 maxTypeApr;
        if (_stakeType == StakeType.FIXED_30_DAYS) {
            maxTypeApr = 120; // 1.2%
        } else if (_stakeType == StakeType.FIXED_90_DAYS) {
            maxTypeApr = 350; // 3.5%
        } else if (_stakeType == StakeType.FIXED_180_DAYS) {
            maxTypeApr = 650; // 6.5%
        } else {
            maxTypeApr = 1200; // 12.0%
        }

        // Ensure not exceeding this type's maximum APR
        return totalApr > maxTypeApr ? maxTypeApr : totalApr;
    }
```

Description

***: The `getCurrentAPR` function is designed to compute the expected APR based on the staking amount and stake type. However, it incorrectly calculates the base APR using the following logic:

```
uint256 yearlyRewards = annualRewardsBudget;
uint256 baseApr;
if (totalPooledHSK == 0) {
    baseApr = MAX_APR;
} else {
    uint256 newTotal = totalPooledHSK + _stakeAmount;
    baseApr = (yearlyRewards * BASIS_POINTS) / newTotal;
}
```

The issue occurs in the calculation:

```
baseApr = (yearlyRewards * BASIS_POINTS) / newTotal;
```

Here, `yearlyRewards` represents the total annual reward budget in HSK tokens, but `newTotal` (which is `totalPooledHSK + _stakeAmount`) represents the total staked amount. Since the formula does not account for staking duration, the APR is incorrectly calculated as if all rewards were distributed instantly rather than over a year. Additionally, when `totalPooledHSK == 0`, the contract **defaults to `MAX_APR` (30%)**, which can lead to misleadingly high APRs at the start of the contract when only a small amount is staked. This causes unrealistic yield expectations, which may attract users with misleading high returns that later drop sharply as the pool grows.

Impact:

The contract miscalculates APR by treating rewards as if they are distributed instantly rather than over a year, causing early stakers to see artificially high yields. When `totalPooledHSK == 0`, it defaults to `MAX_APR` (30%), misleading users into expecting unsustainable returns. As more users stake, the APR drops sharply, creating a false impression of declining rewards and discouraging long-term participation.

Recommendation

***: Adjust the APR formula to factor in time-based rewards by dividing `yearlyRewards` by the expected staking duration, ensuring a realistic APR projection. Additionally, implement a more gradual APR reduction when `totalPooledHSK == 0` to prevent artificially high returns at launch.

Client Response

client response : Fixed. hashkey-hodlium-contract/pull/3

HHM-13:APR Miscalculation Allows Excessive Reward Distribution

Category	Severity	Client Response	Contributor
Code Style	Low	Fixed	***

Code Reference

- code/contracts/HashKeyChainStakingBase.sol#L78-L118

```
78: function updateRewardPool() public {
79:         if (block.number <= lastRewardBlock) {
80:             return;
81:         }
82: 
83:         if (totalPooledHSK == 0) {
84:             lastRewardBlock = block.number;
85:             return;
86:         }
87: 
88:         uint256 multiplier = block.number - lastRewardBlock;
89:         uint256 hskReward = multiplier * hskPerBlock;
90: 
91:         // Calculate and limit APR
92:         uint256 annualReward = hskPerBlock * (365 days / 2); // 2 seconds per block
93:         uint256 currentAPR = (annualReward * BASIS_POINTS) / totalPooledHSK;
94:         if (currentAPR > MAX_APR) {
95:             hskReward = (totalPooledHSK * MAX_APR * multiplier) / (BASIS_POINTS * (365 days / 2));
96:         }
97: 
```

```
98:         // Check if contract has enough HSK
99:         if (reservedRewards >= hskReward) {
100:            totalPooledHSK += hskReward;
101:            reservedRewards -= hskReward;
102: 
103:            // Update exchange rate
104:            emit ExchangeRateUpdated(totalPooledHSK, stHSK.totalSupply(), getHSKForShares(PRECISION_FACTOR));
105:        } else {
106:            if (reservedRewards > 0) {
107:                totalPooledHSK += reservedRewards;
108:                hskReward = reservedRewards;
109:                reservedRewards = 0;
110: 
111:                // Update exchange rate
112:                emit ExchangeRateUpdated(totalPooledHSK, stHSK.totalSupply(), getHSKForShares(PRECISION_FACTOR));
113:            }
114:            emit InsufficientRewards(hskReward, reservedRewards);
115:        }
116: 
117:        lastRewardBlock = block.number;
```

```
118:    }
```

Description

***: The `updateRewardPool` function is designed to distribute HSK rewards based on staked amounts while capping the APR at `MAX_APR` (30%). However, the APR calculation contains a critical flaw in this section:

```
uint256 annualReward = hskPerBlock * (365 days / 2); // 2 seconds per block
uint256 currentAPR = (annualReward * BASIS_POINTS) / totalPooledHSK;
if (currentAPR > MAX_APR) {
    hskReward = (totalPooledHSK * MAX_APR * multiplier) / (BASIS_POINTS * (365 days / 2));
}
```

The issue arises because `(365 days / 2)` is used as the block count for a year, assuming a 2-second block time. However, Solidity performs integer division, meaning `365 days / 2` evaluates to `15768000 / 2 = 7884000` blocks. The correct block count should be `(365 * 24 * 60 * 60) / 2 = 15768000 / 2 = 15768000` blocks, double the value used in the calculation.
This underestimation causes annualReward to be inflated because it is multiplying hskPerBlock by a block count that is half of the actual yearly block count. As a result, the APR calculation underestimates the total pooled HSK relative to the distributed rewards, leading to excessive reward emissions. The incorrect cap calculation also applies an excessive reduction when enforcing the MAX_APR, leading to irregular and imbalanced reward distributions.

Impact:

Due to the underestimated block count, the staking contract distributes more rewards than it should, depleting the reward pool faster than planned and reducing long-term sustainability.

Recommendation

***: Replace `(365 days / 2)` with an exact block count per year (e.g. `blocksPerYear = (365 * 24 * 60 * 60) / blockTime`).

Client Response

client response : Fixed. hashkey-hodlium-contract/pull/3

HHM-14: `updateRewardPool` Can Revert Due to Integer Division by Zero

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/contracts/HashKeyChainStakingBase.sol#L78-L118

```
78: function updateRewardPool() public {
79:         if (block.number <= lastRewardBlock) {
80:             return;
81:         }
82:
83:         if (totalPooledHSK == 0) {
84:             lastRewardBlock = block.number;
85:             return;
86:         }
87:
88:         uint256 multiplier = block.number - lastRewardBlock;
89:         uint256 hskReward = multiplier * hskPerBlock;
90:
91:         // Calculate and limit APR
92:         uint256 annualReward = hskPerBlock * (365 days / 2); // 2 seconds per block
93:         uint256 currentAPR = (annualReward * BASIS_POINTS) / totalPooledHSK;
94:         if (currentAPR > MAX_APR) {
95:             hskReward = (totalPooledHSK * MAX_APR * multiplier) / (BASIS_POINTS * (365 days / 2));
96:         }
97:
98:         // Check if contract has enough HSK
99:         if (reservedRewards >= hskReward) {
100:            totalPooledHSK += hskReward;
101:            reservedRewards -= hskReward;
102:
103:            // Update exchange rate
104:            emit ExchangeRateUpdated(totalPooledHSK, stHSK.totalSupply(), getHSKForShares(PRECISION_FACTOR));
105:        } else {
106:            if (reservedRewards > 0) {
107:                totalPooledHSK += reservedRewards;
108:                hskReward = reservedRewards;
109:                reservedRewards = 0;
110:
111:                // Update exchange rate
112:                emit ExchangeRateUpdated(totalPooledHSK, stHSK.totalSupply(), getHSKForShares(PRECISION_FACTOR));
113:            }
114:            emit InsufficientRewards(hskReward, reservedRewards);
115:        }
116:
117:        lastRewardBlock = block.number;
118:    }
```

Description

***: In the `updateRewardPool` function, the contract calculates APR and reward distributions using the following logic:

```
uint256 currentAPR = (annualReward * BASIS_POINTS) / totalPooledHSK;
```

However, `totalPooledHSK` represents the total amount of staked HSK in the contract, and if no tokens are staked (`totalPooledHSK == 0`), this line results in a division by zero error, causing the function to revert.
Additionally, the function includes this APR enforcement logic:

```
if (currentAPR > MAX_APR) {
    hskReward = (totalPooledHSK * MAX_APR * multiplier) / (BASIS_POINTS * (365 days / 2));
}
```

Since `totalPooledHSK` is used in the numerator without a prior check for zero, any call to `updateRewardPool` when no funds are staked will cause a revert, preventing the reward pool from updating properly and potentially halting further staking operations.

Impact:

If no HSK is staked, calling `updateRewardPool` immediately reverts due to division by zero, blocking all reward updates and potentially breaking staking-related functions that rely on reward calculations.

Recommendation

***: Before performing any calculations involving `totalPooledHSK`, add a condition to skip computations when `totalPooledHSK == 0`, preventing unintended reverts.

Client Response

client response : Acknowledged.

HHM-15:`public` functions not called by the contract should be declared `external` instead

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/contracts/HashKeyChainStaking.sol#L73
- code/contracts/HashKeyChainStaking.sol#L207

```
73: function isStakingOpen() public view returns (bool) {
```

```
207: function getHSKStakingAPR(uint256 _stakeAmount) public view returns (
```

- code/contracts/HashKeyChainStakingBase.sol#L78

```
78: function updateRewardPool() public {
```

Description

***: There are functions `isStakingOpen()` , `getHSKStakingAPR()` , `updateRewardPool()` which has visibility as `public` but not called internally anywhere, they should be declare as `external`

Recommendation

***: declare functions as `external` instead of `public`

Client Response

client response : Acknowledged.

HHM-16: `getCurrentAPR` Function Lacks Support for Regular (Unlocked) Staking APR Calculation

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/contracts/HashKeyChainStakingBase.sol#L173
- code/contracts/HashKeyChainStakingBase.sol#L194-L203

```
173: function getCurrentAPR(uint256 _stakeAmount, StakeType _stakeType) public view returns (uint256) {
```

```
194: uint256 maxTypeApr;
195:         if (_stakeType == StakeType.FIXED_30_DAYS) {
196:             maxTypeApr = 120; // 1.2%
197:         } else if (_stakeType == StakeType.FIXED_90_DAYS) {
198:             maxTypeApr = 350; // 3.5%
199:         } else if (_stakeType == StakeType.FIXED_180_DAYS) {
200:             maxTypeApr = 650; // 6.5%
201:         } else {
202:             maxTypeApr = 1200; // 12.0%
203:         }
```

- code/contracts/HashKeyChainStakingOperations.sol#L12-L13

```
12: * @dev Regular staking (unlocked), directly receives stHSK
13:      */
```

Description

***: The `getCurrentAPR` function in `HashKeyChainStakingBase.sol` is designed to calculate APR for staking positions but only supports locked staking types.
This is evident from the implementation where APR calculations are strictly tied to fixed staking periods:

```
// ... existing code ...
uint256 maxTypeApr;
if (_stakeType == StakeType.FIXED_30_DAYS) {
    maxTypeApr = 120; // 1.2%
} else if (_stakeType == StakeType.FIXED_90_DAYS) {
    maxTypeApr = 350; // 3.5%
} else if (_stakeType == StakeType.FIXED_180_DAYS) {
    maxTypeApr = 650; // 6.5%
} else {
    maxTypeApr = 1200; // 12.0%
}
// ... existing code ...
```

- Regular stakers cannot determine their expected APR before staking
- The protocol's transparency is compromised as a key economic parameter is not accessible for unlocked staking
- Front-end applications and integrators cannot display APR information for regular staking positions
- Users might make uninformed decisions due to lack of APR visibility

Recommendation

***: Modify the getCurrentAPR function to handle regular (unlocked) staking.

Client Response

client response : Acknowledged.

HHM-17:`a = a + b` is more gas effective than `a += b` for state variables (excluding arrays and mappings)

Category	Severity	Client Response	Contributor
Gas Optimization	Informational	Acknowledged	***

Code Reference

- code/contracts/HashKeyChainStaking.sol#L239

```
239: reservedRewards += msg.value;
```

- code/contracts/HashKeyChainStakingBase.sol#L100-L101

```
100: totalPooledHSK += hskReward;
101:             reservedRewards -= hskReward;
```

- code/contracts/HashKeyChainStakingOperations.sol#L26

```
26: totalPooledHSK += msg.value;
```

Description

***: This saves **16 gas per instance.** There are multiple instances where it can save gas. `a = a + b` is more gas effective than `a += b` for state variables (excluding arrays and mappings)

Recommendation

***: use `a = a + b` instead of `a += b`

Client Response

client response : Acknowledged.

HHM-18:Use Custom Errors instead of Revert Strings to save Gas

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/contracts/HashKeyChainStaking.sol#L44-L47
- code/contracts/HashKeyChainStaking.sol#L66

```
44: require(_hskPerBlock > 0, "HSK per block must be positive");
45:         require(_startBlock >= block.number, "Start block must be in the future");
46:         require(_maxHskPerBlock >= _hskPerBlock, "Max HSK per block must be >= HSK per block");
47:         require(_minStakeAmount > 0, "Min stake amount must be positive");
```

```
66: require(_endTime > block.timestamp, "End time must be in future");
```

Description

***: Custom errors are available from solidity version 0.8.4. Custom errors save **~50 gas** each time they're hit by avoiding having to allocate and store the revert string. Not defining the strings also save deployment gas Additionally, custom errors can be used inside and outside of contracts (including interfaces and libraries). Source: https://blog.soliditylang.org/2021/04/21/custom-errors/:

> *"Starting from Solidity v0.8.4, there is a convenient and gas-efficient way to explain to users why an operation failed through the use of custom errors. Until now, you could already use strings to give more information about failures (e.g., `revert("Insufficient funds.");`), but they are rather expensive, especially when it comes to deploy cost, and it is difficult to use dynamic information in them."*

Recommendation

***: Consider replacing **all revert strings** with custom errors in the solution, and particularly those that have multiple occurrences:

Client Response

client response : Acknowledged.

HHM-19:Upgradeable contract is missing a `__gap[50]` storage variable to allow for new storage variables in later versions

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/contracts/HashKeyChainStakingStorage.sol#L10

```
10: abstract contract HashKeyChainStakingStorage {
```

Description

***: In upgradeable contracts, maintaining a consistent storage layout across versions is crucial to avoid storage collisions, which can lead to data loss or corruption. When using proxy patterns (e.g., OpenZeppelin's upgradeable contracts), new state variables in future versions must align with the existing storage layout. Without proper storage reservation, new variables can overwrite existing slots, causing issues like lost user balances or corrupted mappings. To prevent this, OpenZeppelin's Initializable module suggests including a `__gap` array. This reserved storage acts as a buffer, ensuring that future upgrades don't unintentionally overwrite existing data, especially in contracts that inherit from multiple base contracts.

Recommendation

***: It is strongly recommended to add a `__gap` variable to this contract. This will prevent future storage collisions during upgrades and ensure the contract remains upgradeable without risk of storage corruption. The following line should be added towards the end of the contract to reserve these storage slots:

```
uint256[50] private __gap;
```

Client Response

client response : Acknowledged.

HHM-20:Unused code

Category	Severity	Client Response	Contributor
Gas Optimization	Informational	Acknowledged	***

Code Reference

- code/contracts/HashKeyChainStakingEvents.sol#L14-L15

```
14: event RewardsClaimed(address indexed user, uint256 amount);
15:     event RewardsAdded(uint256 amount, address indexed from);
```

Description

***:

```
    event RewardsClaimed(address indexed user, uint256 amount);
    event RewardsAdded(uint256 amount, address indexed from);
```

These two lines of code are unused and can be deleted.

Recommendation

***:

```
    - event RewardsClaimed(address indexed user, uint256 amount);
    - event RewardsAdded(uint256 amount, address indexed from);
```

Client Response

client response : Acknowledged.

HHM-21:Unchecked ERC-20 `transfer()` and `transferFrom()` Call

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/contracts/HashKeyChainStakingEmergency.sol#L70

```
70: IERC20(_token).transfer(owner(), _amount);
```

Description

***: The `recoverToken` function in `HashKeyChainStakingEmergency.sol` uses a direct transfer call to recover ERC20 tokens, which can be problematic for several reasons:

```
function recoverToken(address _token, uint256 _amount) external onlyOwner {
    require(_token != address(stHSK), "Cannot recover staked token");
    IERC20(_token).transfer(owner(), _amount);  // Unsafe transfer
}
```

The implementation has the following vulnerabilities:

- Some ERC20 tokens (like USDT) don't follow the standard strictly
- Some tokens return false instead of reverting on failure
- The function doesn't check the return value of transfer
- No validation is performed on the `_amount` parameter against the actual balance

Recommendation

***: Use OpenZeppelin's SafeERC20 library with `safeTransfer`

Client Response

client response : Acknowledged.

HHM-22:The parameters of the InsufficientRewards event may not accurately reflect the intended values

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/contracts/HashKeyChainStakingBase.sol#L78-L118

```
78: function updateRewardPool() public {
79:         if (block.number <= lastRewardBlock) {
80:             return;
81:         }
82:
83:         if (totalPooledHSK == 0) {
84:             lastRewardBlock = block.number;
85:             return;
86:         }
87:
88:         uint256 multiplier = block.number - lastRewardBlock;
89:         uint256 hskReward = multiplier * hskPerBlock;
90:
91:         // Calculate and limit APR
92:         uint256 annualReward = hskPerBlock * (365 days / 2); // 2 seconds per block
93:         uint256 currentAPR = (annualReward * BASIS_POINTS) / totalPooledHSK;
94:         if (currentAPR > MAX_APR) {
95:             hskReward = (totalPooledHSK * MAX_APR * multiplier) / (BASIS_POINTS * (365 days / 2));
96:         }
97:
```

```
98:         // Check if contract has enough HSK
99:         if (reservedRewards >= hskReward) {
100:             totalPooledHSK += hskReward;
101:             reservedRewards -= hskReward;
102:
103:             // Update exchange rate
104:             emit ExchangeRateUpdated(totalPooledHSK, stHSK.totalSupply(), getHSKForShares(PRECISION_FACTOR));
105:         } else {
106:             if (reservedRewards > 0) {
107:                 totalPooledHSK += reservedRewards;
108:                 hskReward = reservedRewards;
109:                 reservedRewards = 0;
110:
111:                 // Update exchange rate
112:                 emit ExchangeRateUpdated(totalPooledHSK, stHSK.totalSupply(), getHSKForShares(PRECISION_FACTOR));
113:             }
114:             emit InsufficientRewards(hskReward, reservedRewards);
115:         }
116:
117:         lastRewardBlock = block.number;
```

```
118:     }
```

Description

***: ## Summary
The `updateRewardPool` function in the `HashKeyChainStakingBase` contract contains a potential issue where the parameters of the `InsufficientRewards` event may not accurately reflect the intended values. Specifically, the `hskReward` and `reservedRewards` variables are modified before being emitted in the event, which could lead to misleading or incorrect event data. This issue arises because the event is emitted after the values have been overwritten, rather than using their original values.

Vulnerability Details

In the updateRewardPool function, the following logic is used to handle insufficient rewards:

```
if (reservedRewards >= hskReward) {
    totalPooledHSK += hskReward;
    reservedRewards -= hskReward;
} else {
    if (reservedRewards > 0) {
        totalPooledHSK += reservedRewards;
        hskReward = reservedRewards; // hskReward is overwritten with reservedRewards
        reservedRewards = 0; // reservedRewards is set to 0
    }
    emit InsufficientRewards(hskReward, reservedRewards); // Event is emitted with potentially incorrect values
}
```

In the `else` branch, `hskReward` is overwritten. This means that hskReward no longer represents the originally calculated reward amount but instead reflects the remaining reserved rewards. In the same branch, `reservedRewards` is set to `0`, which means it no longer represents the original available reserved rewards.
The `InsufficientRewards` event is emitted with the modified values of `hskReward` and `reservedRewards`. This can lead to misleading event data, as the required (first parameter) and available (second parameter) values may not accurately reflect the actual state of the contract.

Recommendation

***: Consider following fix:

```
if (reservedRewards >= hskReward) {
    totalPooledHSK += hskReward;
    reservedRewards -= hskReward;
} else {
    uint256 originalHskReward = hskReward; // Save the original hskReward
    uint256 originalReservedRewards = reservedRewards; // Save the original reservedRewards

    if (reservedRewards > 0) {
        totalPooledHSK += reservedRewards;
        hskReward = reservedRewards;
        reservedRewards = 0;
    }

    emit InsufficientRewards(originalHskReward, originalReservedRewards); // Emit event with original values
}
```

Client Response

client response : Acknowledged.

HHM-23:The `recoverToken` function does not need to exclude the `stHSK` token

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/contracts/HashKeyChainStakingEmergency.sol#L68-L71

```
68: function recoverToken(address _token, uint256 _amount) external onlyOwner {
69:         require(_token != address(stHSK), "Cannot recover staked token");
70:         IERC20(_token).transfer(owner(), _amount);
71:     }
```

Description

***: The `recoverToken()` function allows owner to rescure tokens from the contract:

```
    function recoverToken(address _token, uint256 _amount) external onlyOwner {
        require(_token != address(stHSK), "Cannot recover staked token");//@here
        IERC20(_token).transfer(owner(), _amount);
    }
```

However, it is important to recover the `stHSK` tokens send to the contract by mistake.
Lets have a view of the `stake()` function and the `unstake()` function of the `HashKeyChainStakingOperations` contract.
The `stake()` function mints `stHSK` token to users:

```
    function stake() external payable nonReentrant whenNotPaused {
        ...

        // Mint stHSK tokens
        stHSK.mint(msg.sender, sharesAmount);

        ...
    }
```

The `unstake()` function burn `stHSK` token from users:

```
    function unstake(uint256 _sharesAmount) external nonReentrant {
        ...

        // Burn stHSK tokens
        stHSK.burn(msg.sender, _sharesAmount);
```

So, the `stHSK` token will not stay at the contract in normal case.

If someone send the `stHSK` token to the contract by mistake, the `recoverToken()` function is unable to recover the `stHSK` token due to it exclude the `stHSK` token.
Thus, the `recoverToken()` should not exclude the `stHSK` token.

Recommendation

***: Removing the exculde for the `stHSK` token, in the `recoverToken()` function.

Client Response

client response : Acknowledged.

HHM-24:Redundant `MAX_APR` Comparison

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/contracts/HashKeyChainStaking.sol#L216-L217

```
216: baseApr = MAX_APR > 1200 ? 1200 : MAX_APR; // MAX_APR_365_DAYS = 1200
217:             return (baseApr, 120, 1200); // Return default values
```

- code/contracts/HashKeyChainStakingBase.sol#L25-L26

```
25: uint256 internal constant MAX_APR = 3000;          // Maximum APR: 30%
```

Description

***: In the getHSKStakingAPR function, there is a redundant comparison between MAX_APR and 1200:

```
if (totalPooledHSK == 0) {
    baseApr = MAX_APR > 1200 ? 1200 : MAX_APR; // MAX_APR_365_DAYS = 1200
    return (baseApr, 120, 1200); // Return default values
}
```

This comparison is problematic since `MAX_APR` is a constant value, making the ternary operation's result deterministic and the comparison redundant.

```
    uint256 internal constant MAX_APR = 3000;          // Maximum APR: 30%

```

Recommendation

***: Remove the redundant comparison and directly use the intended value.

Client Response

client response : Acknowledged.

HHM-25:Redundant Contract Inheritance in HashKeyChainStaking Contract

Category	Severity	Client Response	Contributor
Code Style	Informational	Acknowledged	***

Code Reference

- code/contracts/HashKeyChainStaking.sol#L19-L21

```
19: HashKeyChainStakingOperations,
20:     HashKeyChainStakingAdmin,
21:     HashKeyChainStakingEmergency
```

- code/contracts/HashKeyChainStakingAdmin.sol#L10-L11

```
10: abstract contract HashKeyChainStakingAdmin is HashKeyChainStakingOperations {
11:     /**
```

Description

***: The `HashKeyChainStaking` contract has a redundant inheritance pattern that could lead to confusion and unnecessary complexity in the codebase.
Specifically, the contract inherits from `HashKeyChainStakingAdmin`, which already inherits from `HashKeyChainStakingOperations`. This creates a redundant inheritance path as follows:

1. `HashKeyChainStaking` inherits from `HashKeyChainStakingOperations` directly
2. `HashKeyChainStaking` also inherits from `HashKeyChainStakingAdmin`
3. `HashKeyChainStakingAdmin` inherits from `HashKeyChainStakingOperations`

```
contract HashKeyChainStaking is
    Initializable,
    HashKeyChainStakingOperations,
    HashKeyChainStakingAdmin,
    HashKeyChainStakingEmergency
{...}
abstract contract HashKeyChainStakingAdmin is HashKeyChainStakingOperations {...}
```

Recommendation

***: Optimize the inheritance.

Client Response

client response : Acknowledged.

HHM-26:Ownership change should use two-steps process

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/contracts/HashKeyChainStakingBase.sol#L15-L20

```
15: abstract contract HashKeyChainStakingBase is
16:     PausableUpgradeable,
17:     ReentrancyGuardUpgradeable,
18:     OwnableUpgradeable,
19:     HashKeyChainStakingEvents
20: {
```

Description

*****:** The contract `HashKeyChainStakingBase` does not implement a 2-Step-Process for transferring ownership.
So ownership of the contract can easily be lost when making a mistake when transferring ownership.
So Consider using the Ownable2StepUpgradeable contract from OZ (https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/access/Ownable2StepUpgradeable.sol) instead.
The way it works is there is a transferOwnership to transfer the ownership and acceptOwnership to accept the ownership. Refer the above Ownable2StepUpgradeable.sol for more details.

Recommendation

*****:** Consider using the Ownable2StepUpgradeable contract from OZ (https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/access/Ownable2StepUpgradeable.sol) instead.

Client Response

client response : Acknowledged.

HHM-27:Missing Revert on Invalid Reward Rate Update

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/contracts/HashKeyChainStakingAdmin.sol#L114-L119

```
114: if (newHskPerBlock <= maxHskPerBlock) {
115:             updateRewardPool();
116:             hskPerBlock = newHskPerBlock;
117:             emit HskPerBlockUpdated(oldValue, hskPerBlock);
118:         }
```

Description

***: The `HashKeyChainStakingAdmin` contract contains a vulnerability in its reward rate update mechanism. When attempting to update `hskPerBlock` with a value that exceeds `maxHskPerBlock`, **the function silently fails without reverting or emitting any events.**

```
        if (newHskPerBlock <= maxHskPerBlock) {
            updateRewardPool();
            hskPerBlock = newHskPerBlock;
            emit HskPerBlockUpdated(oldValue, hskPerBlock);
        }

        emit AnnualBudgetUpdated(oldValue, _annualBudget);
```

It violates the "fail-fast" principle of smart contract development and can cause inconsistency between annual target and reward rate.

Recommendation

***: Implement proper revert behavior when the new reward rate exceeds the maximum allowed value.

Client Response

client response : Acknowledged.

HHM-28:Lack of APR Validation When Setting Reward Parameters

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/contracts/HashKeyChainStakingBase.sol#L55
- code/contracts/HashKeyChainStakingBase.sol#L178-L188

```
55: annualRewardsBudget = _hskPerBlock * blocksPerYear;
```

```
178: if (totalPooledHSK == 0) {
179:             baseApr = MAX_APR;
180:         } else {
181:             uint256 newTotal = totalPooledHSK + _stakeAmount;
182:             baseApr = (yearlyRewards * BASIS_POINTS) / newTotal;
183:
184:             // Ensure not exceeding maximum APR
185:             if (baseApr > MAX_APR) {
186:                 baseApr = MAX_APR;
187:             }
188:         }
```

Description

***: The contract enforces a maximum APR limit of 30% (3000 basis points) through the MAX_APR constant.

```
        if (totalPooledHSK == 0) {
            baseApr = MAX_APR;
        } else {
            uint256 newTotal = totalPooledHSK + _stakeAmount;
            baseApr = (yearlyRewards * BASIS_POINTS) / newTotal;

            // Ensure not exceeding maximum APR
            if (baseApr > MAX_APR) {
                baseApr = MAX_APR;
            }
        }
```

However, when setting critical reward parameters like `hskPerBlock`, `annualRewardsBudget`, or `maxHskPerBlock`, there is no upfront validation to ensure these values won't result in an APR that exceeds the maximum limit. If the value is set to be a large value, after the skimming, some minor HSK will be locked in the contract since they exceed MAX_APR in setp.

Recommendation

***: Add validation when setting reward parameters to ensure they won't exceed MAX_APR.

Client Response

client response : Acknowledged.

HHM-29:Cache array length outside of loop

Category	Severity	Client Response	Contributor
Gas Optimization	Informational	Acknowledged	***

Code Reference

- code/contracts/HashKeyChainStaking.sol#L100

```
100: for (uint256 i = 0; i < userStakes.length; i++) {
```

- code/contracts/HashKeyChainStakingEmergency.sol#L34

```
34: for (uint256 i = 0; i < userStakes.length; i++) {
```

Description

***: If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

Recommendation

***: cache length outside of loop

```
uint256 userStakesLength = userStakes.length;

for (uint256 i = 0; i < userStakes.length; i++) {

}
```

Client Response

client response : Acknowledged.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3’s prior written consent in each instance.

This report is not an “endorsement” or “disapproval” of any particular project or team. This report is not an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3’s position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.