coverage

Hex One Protocol

Smart Contract Security Review

Prepared by: **Coverage Labs**
Date: **March 6th, 2024**
Visit: **coveragelabs.io**

Contents

Disclaimer 1

About Hex One Protocol 2

Executive Summary 3

Methodology 4

Severity Classification 5

Quality Assurance 6

Access Control 6
Code Maturity 7
Architectural Review 9

Stateful Fuzzing 18

Findings Summary 21

[C-01] - Users can claim their airdrop allocation multiple times 24
[C-02] - `HexOneStaking.sol` contract can be permanently bricked when there is no amount staked 26
[H-01] - Lack of TWAP reactivity allows users to borrow more `HEX1` than intended 28
[M-01] - `setBaseData()` method may allow Owner to take malicious action 31
[M-02] - Missing state updates in the `_updatePoolHistory()` method lead to staking reward accounting issues 32
[M-03] - Inaccurate airdrop calculation results in excessive `HEXIT` distribution . 34
[M-04] - Return expression from `getCurrentAirdropDay()` method returning unexpected value 36
[M-05] - Overriding of `stakeIds` can lead to debt erasure 37
[M-06] - `setHexOneBootstrap()` method may allow Owner to take malicious action 39
[M-07] - `setHexOneVault()` method may allow Owner to take malicious action . . 40
[L-01] - Lack of total weight amount check can cause unexpected behavior . . . 41
[L-02] - `teamWallet` is the recipient of a significant portion of the `HEXIT` supply . 42
[L-03] - Lack of minimum `deposit()` amount requirement leads to inaccurate accounting 44
[L-04] - Lack of minimum `stake()` amount requirement leads to inaccurate accounting 47
[L-05] - The growth of balances within the `HexOneStaking.sol` contract does not consistently align with proportional increases in share allocation 52
[L-06] - Price feed may be subject to manipulation on deployment 56
[L-07] - Arbitrary `from()` being used in Bootstrap `deposit()` method 57
[I-01] - Typo in `startAidrop()` function name 58
[I-02] - Several integer ranges do not have proper gas-wise considerations . . . 59
[I-03] - Inefficiencies in storage usage 60
[I-04] - Incorrect TWAP stale time code documentation 61
[I-05] - Consider replacing ERC20 libraries with the Solady implementation . . . 62
[I-06] - `Do-while` cycle more gas efficient than conventional `for` loop 63
[I-07] - Split revert statements in `HexOnePriceFeed.sol` 64
[I-08] - Multiple unnecessary storage readings 65
[I-09] - Unnecessary `contains()` check on `AddressSet` 66
[I-10] - Missing zero address checks 67
[I-11] - Basis point standard not being respected 68
[I-12] - No specific events being emitted in `HexOneStaking.sol` 69
[I-13] - Incorrect sacrifice incentives code documentation 70

[I-14] - Misleading internal method name in `_getHexPrice()` 71

Disclaimer

A Security Review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts.

About Hex One Protocol

`HEX1` is a 100% collateralized stablecoin backed by T-SHARES. To obtain `HEX1`, depositors must create an escrowed `HEX` stake in the protocol, so that they can borrow `HEX1` in a 1:1 ratio against its dollar value. While said `HEX` is escrowed in the protocol T-SHARES are being accrued as per standard `HEX` mechanisms. If the `HEX` collateral value in dollar increases, users are then able to mint more `HEX1` according to the difference in underlying value with the last instance a user borrowed `HEX1`.

Executive Summary

Coverage reviewed Hex One Protocol smart contracts over the course of a 5 week engagement with three engineers. The review was conducted from 29-01-2024 to 06-03-2024.

People Involved

Name	Role	Contact
José Garção	Lead Security Researcher	garcao.random@gmail.com
nexusflip	Security Researcher	0xnexusflip@gmail.com
0xBeirao	Security Researcher	beirao.dev@icloud.com

Application Summary

Name	Hex One Protocol
Repository	`https://github.com/HexOneProtocol/hex1-contracts`
Language	Solidity
Platform	Pulsechain

Code Versioning Control

- Review commit hash - `6e6bb33`.
- Fix review commit hash - `50c32bc`.

Scope

The following smart contracts were within review scope:

- `src/HexitToken.sol`
- `src/HexOneBootstrap.sol`
- `src/HexOnePriceFeed.sol`
- `src/HexOneStaking.sol`
- `src/HexOneToken.sol`
- `src/HexOneVault.sol`
- `src/libraries/UniswapV2Library.sol`
- `src/libraries/UniswapV2OracleLibrary.sol`

Methodology

Our methodology is divided into five different phases, each designed to improve the security and reliability of the contracts in scope. By following this structured approach, we aim to enhance the overall robustness of the codebase.

Context & Cleanup

During this phase, our primary focus was comprehending the intricacies of the codebase and eliminating most known anti-patterns. We started by analysing the storage layout of the contracts using sol2uml and producing detailed user flows and diagrams in order to enhance our understanding of the codebase. Once we had a comprehensive understanding, we performed an initial cleanup targeting areas such as commented code, unused imports, event emission, division before multiplication and unchecked returns, among others. We also ran multiple static analysis tools, such as Slither and Slitherin to ensure that the foundation of the code was free from unnecessary clutter. As a final step we shifted our attention to gas optimizations, where we tried to make the overall codebase more cost-efficient at the deployment and runtime levels.

Stateful Fuzzing

In this phase, our first task was to conduct an assessment of the codebase to identify and establish invariants within the protocol. Once we had a clear understanding of the fundamental properties of the protocol, we proceeded to implement them using Echidna. Once the implementation of invariants was completed we executed an extensive series of fuzz runs to ensure that a wide range of inputs and scenarios were tested.

Manual Review

With the insights gained from previous phases, we delved deeper into the codebase to identify potential edge cases, design flaws and attack vectors that may not be easily detected by automated testing techniques. Additionally, we reviewed the overall business logic of the protocol, ensuring consistency with the protocol's specifications and verifying that said logic aligns with the intended behavior outlined in the documentation.

Quality Assurance

During this phase, we outlined possible improvements to the architecture of the protocol to enhance system monitoring, security and overall design. We also documented the capabilities of privileged actors within the protocol. Furthermore, we classified the maturity of the codebase across different categories.

Fix Review

This phase involved reviewing the client fixes and ensuring their correctness. This process included analyzing the code changes, testing the fixes in an isolated setting and assessing their impact on the overall functionality and security of the protocol.

Severity Classification

Likelihood / Impact	HIGH	MEDIUM	LOW
HIGH	CRITICAL	HIGH	MEDIUM
MEDIUM	HIGH	MEDIUM	LOW
LOW	MEDIUM	LOW	LOW

Impact

- **High** - Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- **Medium** - Global losses (<10%) or losses to only a subset of users.
- **Low** - Minor loss of assets in the protocol or harms a small subset of users.

Likelihood

- **High** - Almost certain to happen, easy to perform, or not easy but highly incentivized.
- **Medium** - Only conditionally possible or incentivized, but still relatively likely.
- **Low** - Requires a highly complex setup, or has little to no incentive in accomplishing the attack.

Action Required for severity levels

- **Critical** - Must fix as soon as possible.
- **High** - Must fix.
- **Medium** - Should fix.
- **Low** - Could fix.

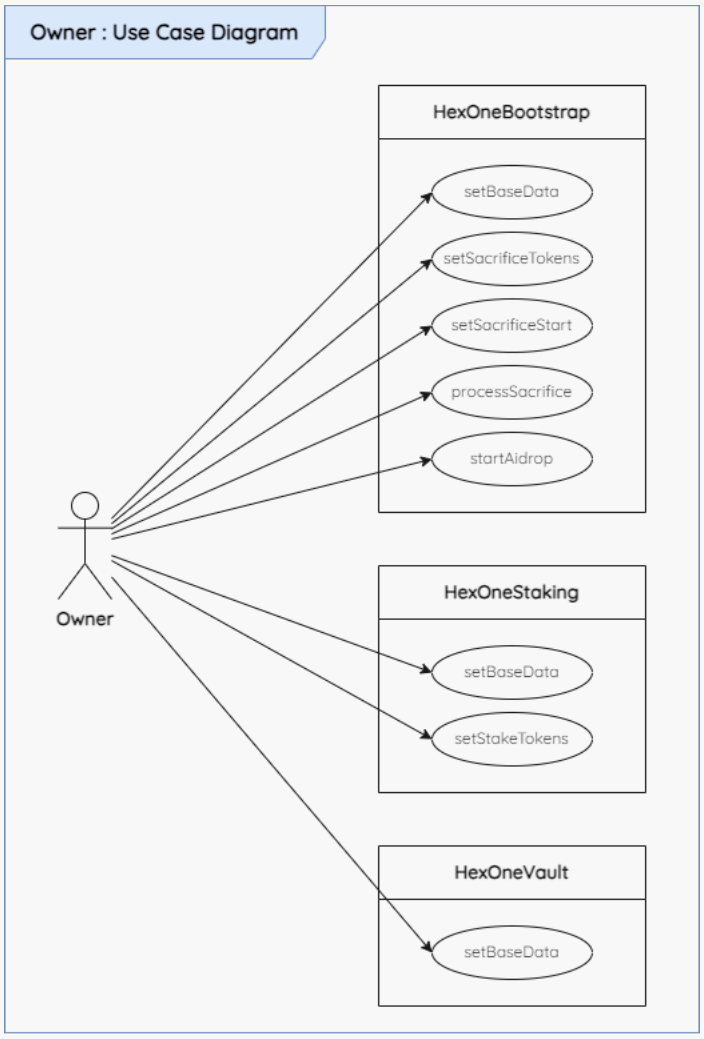
Quality Assurance

Access Control

The protocol employs a simple access control mechanism with only one privileged role: the owner. The owner has the permissions to:

- Set the allowed stake tokens to receive staking rewards, via the `setSacrificeTokens()` method.
- Set the timestamp for the beginning of the sacrifice phase by calling the `setSacrificeStart()` method.
- Process the sacrifice once the 30-day sacrifice period has elapsed, utilizing the `processSacrifice()` method.
- Start the airdrop once the 7-day sacrifice claim period has elapsed, by calling the `startAirdrop()` method.

The owner also has permissions to call certain setter functions that introduced the risk of hotswapping contract implementations, but said issues were mitigated (see findings M-01, M-06 and M-07).



Code Maturity

Code Maturity Evaluation Guidelines

Category	**Description**
Access Control	The use of robust access controls to handle identification and authorization, as well as ensuring safe interactions with the system.
Arithmetic	The proper use of mathematical operations, including addition, subtraction, multiplication, and division, as well as semantics.
Centralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades.
Code Stability	The extent to which the code was altered during the audit and the frequency of changes made over time.
Upgradability	The presence of parametrizations of the system that allow modifications after deployment, ensuring adaptability to future needs.
Front-Running	The system's resistance to front-running attacks, where transactions are manipulated to exploit market conditions.
Monitoring	The presence of events that are emitted whenever there are operations that change the state of the system.
Specification	The presence of comprehensive and readable codebase documentation outlining the purpose, functionality, and design choices of the system.
Testing and Verification	The presence of robust testing procedures, including unit tests, integration tests, and formal verification methods, ensuring the reliability and correctness of the system.

Code Maturity Evaluation Results

Category	**Description**
Access Controls	Satisfactory. All conditional access control functionalities are properly implemented, ensuring secure interactions with the system.
Arithmetic	Moderate. There are several findings that stem from rounding issues, which could be avoided by introducing minimum deposit amount thresholds and refining mathematical operations.
Centralization	Moderate. Certain setter functions introduce the risk of hotswapping contract implementations, which may affect the permissionless nature of the protocol.
Code Stability	Moderate. Some bugs stemming from mathematical outputs hindered testing and thus had to be modified.
Upgradability	Moderate. There is the possibility of changing the implementation of certain contracts, although these have to be fully compatible on an Interface level.
Front-Running	Moderate. There is a possibility that an attacker can front-run the oracle deployment to briefly manipulate the initial price.
Monitoring	Moderate. Although most contracts in scope have comprehensive event emission, the `HexOneStaking.sol` contract lacked context-specific events.
Specification	Weak. The current documentation is outdated in comparison to the actual implementation of the contracts.
Testing and Verification	Weak. There was a significant lack of testing coverage on the codebase (**NOTE:** The client provided an updated testing suite with full test coverage shortly after the review started).

Architectural Review

Improvements can be made to `HEXIT` supply monitoring

The `totalHexitMinted` public variable within the `HexOneBootstrap.sol` contract is supposed to store the total amount of `HEXIT` minted in this given contract, but does not separate mint amounts by sacrifice, airdrop, staking contract mint and team wallet mint.

This makes it difficult to do accounting and/or external composition, as we have noticed when applying invariant testing principles on top of said variable to assess that the minting percentages for both the staking contract and the team wallet were being abided by at all times.

```
/// @dev total amount of HEXIT tokens minted during the bootstrap.
uint256 public totalHexitMinted;
```

We recommend separating the `HEXIT` mint amount logging into `hexitMintedSacrifice`, `hexitMintedAirdropUsers`, `hexitMintedAirdropStaking` and `hexitMintedAirdropTeamWallet` to have an external logging solution that favours external composability.

Consider changing circular dependencies to a hierarchical model

The following snippet of the `HexOneBootstrap.sol` contract is an example of a circular dependency in the scope. We believe upgrading this model to a hierarchical model is more fitting of good Solidity practices. This same issue can be observed in the `HexOneBootstrap.sol`, `HexOneStaking.sol`, `HexOneVault.sol`, `HexitToken.sol` and `HexOneToken.sol` contracts.

```
/// @dev set the address of other protocol contracts.
/// @notice can only be called by the owner.
function setBaseData(address _hexOnePriceFeed, address _hexOneStaking,
 ↪  address _hexOneVault) external onlyOwner {
    if (_hexOnePriceFeed == address(0)) revert
     ↪  InvalidAddress(_hexOnePriceFeed);
    if (_hexOneStaking == address(0)) revert InvalidAddress(_hexOneStaking);
    if (_hexOneVault == address(0)) revert InvalidAddress(_hexOneVault);

    hexOnePriceFeed = _hexOnePriceFeed;
    hexOneStaking = _hexOneStaking;
    hexOneVault = _hexOneVault;
}
```

We recommend changing the above mentioned contracts so that the necessary addresses are declared in the contract constructors, being mindful of the correct deployment sequence to enforce the hierarchical contract dependencies.

Consider a staking implementation with linear reward accrual in a future iteration of the protocol

The following method of the `HexOneStaking.sol` contract is used by the `HexOneBootstrap.sol` contract to feed rewards to the former whenever there is a `deposit` in the `HexOneVault.sol` or the airdrop resulting from the sacrifices starts. Said rewards are available to claim daily depending on the different tokens the user has staked (using different percentual weights for the reward output of the users' stake), up to a maximum daily cap of 1% of the current reward pool both for `HEX` fees resulting from Vault deposits or the additional `HEXIT` supply that is minted whenever the airdrop starts.

```
function purchase(address _poolToken, uint256 _amount) external {
    require(_amount != 0, "Invalid purchase amount");
    require(
        (_poolToken == hexToken && msg.sender == hexOneVault)
            || (_poolToken == hexitToken && msg.sender == hexOneBootstrap),
        "Invalid sender for the specified pool token"
    );

    // if the pool staking day is not sync with the contract staking day
    // there might be gaps in pool history, so we need to updated it
    Pool storage pool = pools[_poolToken];
    if (pool.currentStakingDay < getCurrentStakingDay()) {
        _updatePoolHistory(_poolToken);
    }

    // increment the total assets deposited in the pool
    pool.totalAssets += _amount;

    // transfer tokens from the msg.sender to this contract
    IERC20(_poolToken).safeTransferFrom(msg.sender, address(this),_amount);
}
```

Although we understand the reasoning behind the architectural design decision to distribute 1% of the pooled assets per day to users, being that the `HEXIT` mint post airdrop can have a long enough distribution schedule, gradually inducing `HEXIT` scarcity the longer the rewards are distributed and guaranteeing `HEX` rewards would be available for distribution at each given day, we believe that this distribution system could be linear, thus distributing the full amount of `HEX` fees accrued from the Vault to users and setting a linear distribution and mint rate for `HEXIT` staking rewards, effectively replacing a single `HEXIT` pre-mint after the airdrop which forces dilution of the total token supply with minted linear emissions on demand, proportional to the mint multiplier set by the **Owner** of the Staking contract.

Current iterative logic on calculating rewards:

```
while (lastClaimedDay < getCurrentStakingDay()) {
    // calculate HEX rewards for that day
    PoolHistory storage hexHistory = poolHistory[lastClaimedDay][hexToken];
    emit StakeShares("HEX history total shares", hexHistortotalShares);
    uint256 hexSharesRatio = stakeInfo.hexSharesAmount * FIXED_POINT
     ↪  hexHistory.totalShares;
    hexRewards += (hexHistory.amountToDistribute * hexSharesRatio)
     ↪  FIXED_POINT;

    // calculate HEXIT rewards
    PoolHistory storage hexitHistory =
     ↪  poolHistory[lastClaimedDay][hexitToken];
    uint256 hexitSharesRatio = stakeInfo.hexitSharesAmount FIXED_POINT /
     ↪  hexitHistory.totalShares;
    hexitRewards += (hexitHistory.amountToDistribute hexitSharesRatio) /
     ↪  FIXED_POINT;

    lastClaimedDay++;
}
```

Current iterative logic on updating pools:

```
while (currentStakingDay < getCurrentStakingDay()) {
    // get the pool rewards for each day since it was last updated
    PoolHistory storage history = poolHistory[currentStakingDay][_poolToken];

    // store the total shares emitted by the pool at a specific day
    history.totalShares = pool.totalShares;

    // calculate the amount of pool token to distribute for a specifistaking
     ↪  day
    uint256 availableAssets = pool.totalAssets - pool.distributedAssets;
    uint256 amountToDistribute = (availableAssets * pool.distributionRate) /
     ↪  FIXED_POINT;
    history.amountToDistribute = amountToDistribute;

    // increment the distributedAssets by the pool
    pool.distributedAssets += amountToDistribute;

    // increment the staking day in which the pool rewards were last updated
    currentStakingDay++;
}
```

This change would prevent excessive assymetry in staking output yields which can lead to unpredictability in the percentage of user staking retention over time, potentially reducing the amount of `HEX1/DAI` LP token (being that the liquidity depth of the LP grants stability to the peg) that is staked in the protocol over time, while still granting full control of the distribution of `HEXIT` supply over time and maintaining the staking token weight mechanism.

Adding to that, using a linear mechanism would not need extensive historical storing of daily rewards each user has not claimed in storage, being that it would only be needed to store which tokens the user has staked and their respective amounts without having to iteratively access storage for each day proportionally to the amount of days the user has not claimed rewards for, as well as the last timestamp each user claimed rewards, mitigating potential **Denial-of-Service** issues that the current model may pose on the long run from eventually having to iteratively store a considerable enough amount of data in storage pertaining daily rewards for a user that has not claimed for a very long period of time as well as respective pool data updates.

- **NOTE:** We have calculated the amount of `poolHistory` entries that would need to be updated (in the case of said entries not being updated for a very long period of time) to cause a **Denial-of-Service** issue given the case described above and the results would be negligible on a reasonable decades-long timescale. Nonetheless, this proposed implementation would mitigate any potential issues to the point it is impossible for it to happen in a much greater timescale.
- **NOTE:** We are aware this would require a complete rework of the staking mechanism so this suggestion should only be considered for future iterations of the protocol.

We recommend changing the reward distribution calculations so that all of the pool rewards can be distributed to users linearly and the `HEXIT` distribution rate uses an emissions based model. To achieve that, it would be necessary to:

- Remove the 1% hardcap on rewards to distribute from pools.
- Remove the `_updatePoolHistory()` method and update depending external methods accordingly.
- Add a `setTokenDistributionRate()` method that would allow the **Owner** to change the rate of distribution expressed in `rewardRate` per second for any staking reward tokens.
- Remove the `HEXIT` premint on airdrop start in favour of a `mint()` call inside the `claim()` method that mints the amounts to distribute for a given user after calculating the `HEXIT` said user has not claimed yet through the `_calculateRewards()` method, thus following an emissions-based model for `HEXIT` distribution.
- Simplify the `Pool` and `StakeInfo` structs so that any variables pertaining previous logic would be removed in favour of logging last claimed timestamps, amounts of each staked token of each user, total shares and total amounts claimed of each reward token. Remove the `PoolHistory` struct.
- Change the `_calculateRewards()` method so that it would return the result to the following formula: `rewards = stakedAmount * rewardRate * elapsedTime * tokenWeight`, in which `elapsedTime = block.timestamp - lastUserClaimedTimestamp`.

HEX1/DAI LP peg stability can be ensured by tuning sacrifice processing ratios and increasing incentivization for LP stakers

The `processSacrifice()` method within the `HexOneBootstrap.sol` contract swaps 12.5% of the current sacrificed `HEX` to `DAI` and deposits another 12.5% of `HEX` in the protocol's Vault to mint `HEX1`, in order to deposit both assets in the `HEX1/DAI` PulseX LP in the same ratio.

```
function processSacrifice(uint256 _amountOutMinDai) external onlyOwner {
    if (block.timestamp < sacrificeEnd) revert
    ↪  SacrificeHasNotEndedYet(block.timestamp);
    if (sacrificeProcessed) revert SacrificeAlreadyProcessed();

    // set the sacrifice processed flag to true since the sacrifice was
    ↪  already processed
    sacrificeProcessed = true;

    // update the sacrifice claim period end timestamp
    sacrificeClaimPeriodEnd = block.timestamp + SACRIFICE_CLAIM_DURATION;

    // compute the HEX to swap, corresponding to 12.5% of the total HEX
    ↪  sacrificed
    uint256 hexToSwap = (totalHexAmount * LIQUIDITY_SWAP_RATE) / FIXED_POINT;

    // update the total amount of HEX in the contract by reducing it
    ↪  hexToSwap * 2
    // because 12.5% is used to being swapped to DAI and the other 12.5% are
    ↪  used to mint HEX1
    // that being said 25% of the inital HEX sacrificed should be decremented
    ↪  here!
    totalHexAmount -= hexToSwap * 2;

    // swap 12.5% of inital HEX to DAI from ETH
    address[] memory path = new address[](2);
    path[0] = hexToken;
    path[1] = daiToken;

    IERC20(hexToken).approve(pulseXRouter, hexToSwap);

    uint256[] memory amountOut =
    ↪  IPulseXRouter(pulseXRouter).swapExactTokensForTokens(
        hexToSwap, _amountOutMinDai, path, address(this), block.timestamp
    );

    // enable hex one vault to start working because sacrifice has been
    ↪  processed.
    IHexOneVault(hexOneVault).setSacrificeStatus();

    // create a new deposit for `MAX_DURATION` in the vault with 12.5% of the
    ↪  total minted HEX
    IERC20(hexToken).approve(hexOneVault, hexToSwap);
    (uint256 hexOneMinted,) = IHexOneVault(hexOneVault).deposit(hexToSwap,
    ↪  5555);

    // check if there's an already created HEX1/DAI pair
    address hexOneDaiPair =
    ↪  IPulseXFactory(pulseXFactory).getPair(hexOneToken, daiToken);
    if (hexOneDaiPair == address(0)) {
        hexOneDaiPair = IPulseXFactory(pulseXFactory).createPair(hexOneToken,
         ↪  daiToken);
    }

    // approve router for both amounts
    IERC20(hexOneToken).approve(pulseXRouter, hexOneMinted);
    IERC20(daiToken).approve(pulseXRouter, amountOut[1]);

    // use the newly minted HEX1 + DAI from ETH and create an LP with 1:1
    ↪  ratio
    (uint256 amountHexOneSent, uint256 amountDaiSent, uint256 liquidity) =
    ↪  IPulseXRouter(pulseXRouter).addLiquidity(
        hexOneToken,
        daiToken,
        hexOneMinted,
        amountOut[1],
        hexOneMinted,
        amountOut[1],
        address(this),
        block.timestamp
    );

    emit SacrificeProcessed(hexOneDaiPair, amountHexOneSent, amountDaiSent,
    ↪  liquidity);
}
```

The `stake()` method within the `HexOneStaking.sol` contract allows users to stake tokens and get `HEX` and `HEXIT` rewards which come from distributing 1% of then respective reward token pools daily, depending on the token weights of the ERC20 tokens they are staking. Both represent solutions the protocol has implemented to secure the `HEX1/DAI` pair liquidity depth in order to try and mitigate potential depegs that happen from the ratio of assets in the stable pair being skewed.

```
function stake(address _stakeToken, uint256 _amount) external nonReentrant
↪  onlyWhenStakingEnabled {
    require(stakeTokens.contains(_stakeToken), "Token not allowed");
    require(_amount > 0, "Invalid staking amount");

    // accrue rewards and update history for both the HEX and HEXIT pools
    _accrueRewards(msg.sender, _stakeToken);

    // transfers amount of stake token from the sender to this contract.
    uint256 stakeAmount = _transferToken(_stakeToken, msg.sender,
    ↪  address(this), _amount);

    // update the total amount staked
    totalStakedAmount[_stakeToken] += stakeAmount;

    // calculate the amount of HEX and HEXIT pool shares to give to the user
    uint256 shares = _calculateShares(_stakeToken, stakeAmount);
    require(shares != 0, "Invalid shares amount");

    // update the number of total shares in the HEX and HEXIT pools
    pools[hexToken].totalShares += shares;
    pools[hexitToken].totalShares += shares;
    // update the staking information of the user for a specific stake token
    uint256 currentStakingDay = getCurrentStakingDay();
    StakeInfo storage stakeInfo = stakingInfos[msg.sender][_stakeToken];
    stakeInfo.stakedAmount += stakeAmount;
    if (stakeInfo.initStakeDay == 0) {
        stakeInfo.initStakeDay = currentStakingDay;
    }
    if (stakeInfo.lastClaimedDay == 0) {
        stakeInfo.lastClaimedDay = currentStakingDay;
    }
    stakeInfo.lastDepositedDay = currentStakingDay;
    stakeInfo.hexSharesAmount += shares;
    stakeInfo.hexitSharesAmount += shares;
}
```

As described in finding H-01, there is a possibility of users being able to mint more `HEX1` than expected and selling it against the `HEX1/DAI` PulseX LP that supports the peg, which poses a threat to the protocol's peg stability. Assessing some architectural implications related to this security finding, we can consider that having a frequently updated median of different quotes from multiple sources of truth helps mitigating the issue, but at the same time achieving a consistent dollar cost basis for `HEX1` borrowing without a robust liquidations, LTV and interest rates mechanism is difficult, especially after a significant depeg that stems from a deviation in the ratio of `HEX1` to `DAI` in the PulseX LP pair.

Given that the `HEX1/DAI` pair is the benchmark for the expected dollar valuation of `HEX1`, and although the intention of the protocol is to enable users to have the option of using the underlying value of their `HEX` in external markets while accruing yield from T-Shares and be able to increase their loan output if there is favourable `HEX` price movement, or having the option to buy a long-term discounted bond on `HEX` if the `HEX1` peg breaks by accumulating `HEX1` in a risk-based approach, the peg repeatedly breaking for extensive periods of time might represent an issue for external composability by other DeFi end-points, so there is a necessity of ensuring the `HEX1/DAI` peg is maintained by enforcing more aggressive LP deposit ratios whenever a sacrifice is processed or even more distribution of `HEX` and `HEXIT` incentives for users that stake the LP token.

We recommend ensuring more aggressive LP conversion ratios on `processSacrifice()` (more tokens get deposited on the LP) and/or increase token weight and resulting `HEXIT` and `HEX` yields for `HEX1/DAI` LP token stakes.

No token pre-mint might affect max supply beyond expected

In a scenario where the `deposit()` method of the `HexOneVault.sol` contract is called, unexpected behaviour might happen during its internal transfer call.

In the following snippet of the `HexitToken.sol` contract, there is the possibility to mint tokens without a set threshold or maximum supply. This can lead to an unexpected dilution of the token supply as there might be unintended side effects from external calls leading into the `mint()` call.

```
/// @dev mint HEXIT tokens to a specified account.
/// @notice only bootstrap can call this function.
/// @param _recipient address of the receiver.
/// @param _amount amount of HEX1 being minted.
function mint(address _recipient, uint256 _amount) external
 ↪  onlyHexOneBootstrap {
    _mint(_recipient, _amount);
}
```

Consider changing the token distribution mechanism so that the maximum supply is preminted before any usage, and then change the external `mint()` calls to `transfer()` calls from the supplying contracts which hold the intended portions of token to supply. Alternatively, if there is an intention to have an infinite `HEXIT` supply, we recommend that all mathematical calculations resulting from the protocols' intended architecture that lead to a `mint()` call are thoroughly checked to avoid unintended dilution.

Stateful Fuzzing

HexOneStaking.sol

Property	Runs	Status
Daily distributed HEXIT rewards must be equal to 1%	~102MM	❌
Daily distributed HEX rewards must be equal to 1%	~10MM	❌
HEX pool shares to give are always proportional to the increase in balance of a stake token based on the weight	~10MM	❌
HEXIT pool shares to give are always proportional to the increase in balance of a stake token based on the weight	~10MM	❌
Users cannot unstake more than what they staked (excluding rewards)	~600MM	✅
HEXIT unstake amount is always greater or equal than the stake amount	~600MM	✅
Sum of all users shares must be equal to `pool.totalShares`	~600MM	✅
Users can only unstake 2 days after they've staked	~600MM	✅
The total rewards to be distributed to Alice with N deposits of X total value should be the same for Bob with pN deposits of X same total value	~10MM	❌

HexOneVault.sol

Property	Runs	Status
HEX1 minted must always be equal to the sum of all users `UserInfo.totalBorrowed`	~1MM	❌
Alice must only be able to mint more HEX1 with the same HEX collateral if the HEX price increases	~10MM	✅
Must never be able to mint more `HEX1` with the same collateral if the `HEX` price decreases	~600MM	✅
The amount to withdraw after maturity must always be greater or equal than the HEX collateral deposited	~600MM	✅
Amount and duration on deposit must always be corresponding to the amount minus fee and corresponding set lock on the contract storage	~600MM	✅
The fee taken from user deposits must always be equal to 5%	~600MM	✅
Deposit can not be claimed if maturity has not passed	~600MM	✅
If `hexOneBorrowed` > 0 the amount borrowed must always be burned to claim back HEX	~600MM	✅
The sum of all `DepositInfo.amount` HEX deposited by the user across all its deposits must always be equal to `UserInfo.totalAmount`	~600MM	✅
The sum of all `DepositInfo.borrowed` HEX1 borrowed by the user across all its deposits must always be equal to `UserInfo.totalBorrowed`	~600MM	✅

HexOneBootstrap.sol

Property	Runs	Status
If two users sacrificed the same amount of the same sacrifice token on different days, the one who sacrificed first should always receive more HEXIT	~600MM	✅
If two users are entitled to the same amount of airdrop (HEX staked in USD + sacrificed USD), the one who claimed first should always receive more HEXIT (different days)	~600MM	✅
If two users are entitled to the same amount of airdrop (HEX staked in USD + sacrificed USD), they should always receive the same amount of HEXIT if they claimed the airdrop on the same day	~600MM	✅
The `startAirdrop` function must always mint 33% on top of the total HEXIT minted during the sacrifice phase to the HexOneStaking.sol contract	~10MM	❌
The `startAirdrop` function must always mint 50% on top of the total HEXIT minted during the sacrifice phase to the team wallet	~10MM	❌
The amount of `UserInfo.hexitShares` a user has must always be equal to the amount of HEXIT minted when the user claim its sacrifice rewards via `claimSacrifice` function	~600MM	✅

Findings Summary

ID	Title	Severity	Status
C-01	Users can claim their airdrop allocation multiple times	CRITICAL	Fixed
C-02	`HexOneStaking.sol` contract can be permanently bricked when there is no amount staked	CRITICAL	Fixed
H-01	Lack of TWAP reactivity allows users to borrow more `HEX1` than intended	HIGH	Ack
M-01	`setBaseData()` method may allow owner to take malicious action	MEDIUM	Fixed
M-02	Missing state updates in the `_updatePoolHistory()` method lead to staking reward accounting issues	MEDIUM	Fixed
M-03	Inaccurate airdrop calculation results in excessive `HEXIT` distribution	MEDIUM	Fixed
M-04	Return expression from `getCurrentAirdropDay()` method returning unexpected value	MEDIUM	Fixed
M-05	Overriding of `stakeIds` can lead to debt erasure	MEDIUM	Fixed
M-06	`setHexOneBootstrap()` method may allow Owner to take malicious action	MEDIUM	Fixed
M-07	`setHexOneVault()` method may allow Owner to take malicious action	MEDIUM	Fixed
L-01	Lack of total weight amount check can cause unexpected behavior	LOW	Fixed
L-02	`teamWallet` is the recipient of a significant portion of the `HEXIT` supply	LOW	Ack
L-03	Lack of minimum `deposit()` amount requirement leads to inaccurate accounting	LOW	Fixed
L-04	Lack of minimum `stake()` amount requirement leads to inaccurate accounting	LOW	Fixed
L-05	The growth of balances within the `HexOneStaking.sol` contract does not consistently align with proportional increases in share allocation	LOW	Fixed

ID	Title	Severity	Status
L-06	Price feed may be subject to manipulation on deployment	LOW	Ack
L-07	Arbitrary `from()` being used in Bootstrap `deposit()` method	LOW	Fixed
I-01	Typo in `startAidrop()` function name	INFO	Fixed
I-02	Several integer ranges do not have proper gas-wise considerations	INFO	Ack
I-03	Inefficiencies in storage usage	INFO	Fixed
I-04	Incorrect TWAP stale time code documentation	INFO	Fixed
I-05	Consider replacing ERC20 libraries with the Solady implementation	INFO	Ack
I-06	`Do-while` cycle more gas efficient than conventional `for` loop	INFO	Will not fix
I-07	Split revert statements in `HexOnePriceFeed.sol`	INFO	Ack
I-08	Multiple unnecessary storage readings	INFO	Fixed
I-09	Unnecessary `contains()` check on `AddressSet`	INFO	Fixed
I-10	Missing zero address checks	INFO	Fixed
I-11	Basis points standard is not being respected	INFO	Fixed
I-12	No specific events being emitted in `HexOneStaking.sol`	INFO	Fixed
I-13	Incorrect sacrifice incentives code documentation	INFO	Fixed
I-14	Misleading internal method name in `_getHexPrice()`	INFO	Fixed

Severity	Count	Fixed	Acknowledged	Will not fix
CRITICAL	2	2	0	0
HIGH	1	0	1	0
MEDIUM	7	7	0	0
LOW	7	5	2	0
INFO	14	10	3	1
TOTAL	**31**	**24**	**6**	**1**

[C-01] - Users can claim their airdrop allocation multiple times

ID	Classification	Category	Status
C-01	CRITICAL	Manual Review	Fixed in 46b8401

Impact

The `claimAirdrop()` function in the `HexOneBootstrap.sol` contract serves the purpose of allowing users to withdraw their respective airdrops **once**. However, to safeguard against multiple withdrawals, it checks the `claimedAirdrop()` flag to ensure it's not already set to `true`. Since the flag is never updated, this issue can lead to unintended outcomes such as allowing users to claim the airdrop allocation multiple times.

Proof of Concept

1. Alice is eligible to claim the airdrop, and proceeds calls the `claimAirdrop()` function.
2. Alice mints her `HEXIT` airdrop allocation.
3. Alice notices that the `claimedAirdrop` flag for her `userInfo` is set to `false` and thus calls the `claimAirdrop()` method once more, being able to claim the same airdrop amount.
4. Alice repeats the process until the total amount to be airdropped to all users is drained from the `HexOneBootstrap.sol` contract, siphoning other users' airdrops.

The following snippet shows an example proof of concept where a user can claim its airdrop allocation more than once.

```
function testAirdropNoStateUpdate() public {
    ...
    // users claims airdrop
    vm.prank(user);
    bootstrap.claimAirdrop();

    // get claimedAirdrop bool
    (,,, bool claimedAirdrop) = bootstrap.userInfos(user);

    // assert that although the user claimed the airdrop flag is still
    // set to false
    assertEq(claimedAirdrop, false);

    // assert user HEXIT balance
    uint256 hexitBalanceAfter = IERC20(hexit).balanceOf(user);
    assertGt(hexitBalanceAfter, userHexitBalanceBefore);

    // test that the user can claim it's airdrop more than once
    vm.prank(user);
    bootstrap.claimAirdrop();

    // assert user HEXIT balance exploit
    uint256 hexitBalanceExploit = IERC20(hexit).balanceOf(user);
    assertGt(hexitBalanceExploit, hexitBalanceAfter);
}
```

Recommended Mitigation Steps

Consider adding the following check to the `claimAirdrop()` function.

```
function claimAirdrop() external {
    if (airdropStart == 0) revert AirdropHasNotStartedYet(block.timestamp);
    if (block.timestamp >= airdropEnd)
        revert AirdropAlreadyEnded(block.timestamp);

    // check if the sender already claimed the airdrop
    UserInfo storage userInfo = userInfos[msg.sender];
    if (userInfo.claimedAirdrop) revert AirdropAlreadyClaimed(msg.sender);

    // calculate the amount to airdrop based on the amount
    // that the msg.sender has of staked HEX and sacrificed USD
    uint256 hexitShares = _calculateHexitAirdropShares();
    if (hexitShares == 0) revert IneligibleForAirdrop(msg.sender);

    // increment the total amount of hexit minted by the contract
    totalHexitMinted += hexitShares;

+   // set the claimed airdrop flag to true because airdrop was claimed
+   userInfo.claimedAirdrop = true;

    // mint HEXIT to the sender
    IHexitToken(hexitToken).mint(msg.sender, hexitShares);

    emit AirdropClaimed(msg.sender, hexitShares);
}
```

[C-02] - HexOneStaking.sol contract can be permanently bricked when there is no amount staked

ID	Classification	Category	Status
C-02	CRITICAL	Manual Review	Fixed in 481c50d

Impact

In a scenario where the pool current staking day pool.currentStakingDay is less than the current staking day output in currentStakingDay() and the staking pool contains no staked amount, any attempt to execute the stake() function will consistently revert due to a division by zero error, effectively causing a **Denial-of-Service** that permanently disables users from interacting with the contract.

Proof of Concept

If the HexOneStaking.sol contract remains uncalled for a period exceeding one day, any subsequent calls to the stake() function will trigger an update to both the HEX and HEXIT poolHistory entries. This update is necessary because the poolHistory entries become out-of-sync with the currentStakingDay. When a user increases its' stake rewards, said rewards are accrued to ensure that the new amount of shares does compromise the calculation of rewards earned from previous days. This accrual is crucial for maintaining consistency in reward distribution. However, it's important to note that rewards should not be calculated when the user is staking for the first time. In a scenario where both the HEX and HEXIT pools have no corresponding staked amount to distribute rewards towards, the stake() function will consistently revert due to a division by zero error when computing the rewards to distribute, effectively causing a **Denial-of-Service**.

The following snippet shows an example proof of concept where no users interacted with the staking contract for 2 days since it was enabled, which ultimately lead to a permanent denial of service of the stake() function due to a division by zero error.

```
function test_stake_denialOfService_afterInactivityDays() public {
    // bound the amount of HEX1/DAI to stake
    uint256 amount = 500 * 1e18;

    // deal HEX1/DAI LP to the user
    deal(hexOneDaiPair, user, amount);

    // skip the number of inactivity days after staking is enabled
    skip(2 days);

    // staking contract is bricked because of a division by zero error
    vm.startPrank(user);
    IERC20(hexOneDaiPair).approve(address(staking), amount);
    vm.expectRevert(stdError.divisionError);
    staking.stake(hexOneDaiPair, amount);
    vm.stopPrank();
}
```

Recommended Mitigation Steps

Consider adding a check that ensures rewards are not calculated when the user does not have pool shares (shares are only granted to a user if they have an amount staked).

```
function _accrueRewards(address _user, address _stakeToken) internal {
    ...
    StakeInfo storage stakeInfo = stakingInfos[_user][_stakeToken];
+   if (stakeInfo.hexSharesAmount > 0 && stakeInfo.hexitSharesAmount > 0) {
        // calculate the amount of HEX and HEXIT rewards since the last
        // day claimed
        (
            uint256 hexRewards,
            uint256 hexitRewards
        ) = _calculateRewards(_user, _stakeToken);

        // increment the rewards accrued as unclaimed rewards
        stakeInfo.unclaimedHex += hexRewards;
        stakeInfo.unclaimedHexit += hexitRewards;
    }
}
```

[H-01] - Lack of TWAP reactivity allows users to borrow more `HEX1` than intended

ID	Classification	Category	Status
H-01	HIGH	Manual Review	Acknowledged

Impact

In a scenario where there is a significant price decrease of `HEX` per `DAI`, users are able to mint more `HEX1` than expected. This is due to a delay in the price update process, which stems from the low reactivity of the TWAP. Given the current shallow liquidity of the `HEX/DAI` pool on PulseX, with a TVL of ~227k USD at the time of writing, the amount of `HEX` needed to cause a considerable price drop to capitalize on this issue is rather low.

Proof of Concept

If the `HEX` price drops in comparison to `DAI`, either due to external market conditions or intentional price manipulation, users may be able to borrow `HEX1` before this price drop is reflected in the `HexOnePriceFeed.sol` quote. Due to slow reactivity and lack of accuracy in the price feed, the user may be able to take on an undercollateralized `HEX1` loan. This loan can then be swapped to `DAI` on the `HEX1/DAI` pool in the PulseX DEX, inadvertently creating an arbitrage opportunity that can **severely damage the stablecoin's peg**.

Reactivity, set at one hour in this case, determines how frequently the TWAP is updated. When the oracle's reactivity is excessively slow, as it is here, the price adjustment to the current spot price may be delayed, potentially resulting in price divergences and arbitrage opportunities that may harm the `HEX1` peg and compromise the TWAP's price accuracy. However, if reactivity is excessively high, while the TWAP's price accuracy may increase, it can potentially make prices more susceptible to manipulation. Therefore, finding a balance between reactivity and accuracy is crucial. We believe that the current scope does not benefit from a TWAP implementation, given that the delay in sync with the current spot price is inconvenient.

The following snippet shows an example proof of concept where 2 million `HEX` are exchanged for `DAI`, followed by a deposit of 2 million `HEX` into the `HexOneVault.sol`. This deposit unexpectedly **mints over 50% more `HEX1` than initially expected**. At the time of writing, 2 million `HEX` are worth approximately ~35k `DAI`, which is a rather small liquidity amount for DeFi standards.

```
function testTwapDelay() public {
    // give HEX to the sender for both the swap and the deposit
    uint256 amount = 2000000e8;
    deal(hexToken, user, amount * 2);

    // user sells 2 million HEX to DAI
    vm.startPrank(user);
    IERC20(hexToken).approve(pulseXRouter, amount);
    address[] memory path = new address[](2);
    path[0] = hexToken;
    path[1] = daiToken;
    IPulseXRouter02(pulseXRouter).swapExactTokensForTokens(amount, 1, path,
    ↪  user, block.timestamp);
    vm.stopPrank();

    // check if the HEX to DAI TWAP quote is considerably higher than the
    ↪  spot price
    IPulseXPair pulseXPair = IPulseXPair(hexDaiPair);
    assertEq(hexToken, pulseXPair.token0());

    (uint112 reserve0, uint112 reserve1, uint32 blockTimestampLast) =
    ↪  pulseXPair.getReserves();
    uint256 spotQuote = IPulseXRouter02(pulseXRouter).quote(amount, reserve0,
    ↪  reserve1);
    emit log_named_uint("spotQuote", spotQuote);

    uint256 twapQuote = IHexOnePriceFeed(feed).consult(hexToken, amount,
    ↪  daiToken);
    emit log_named_uint("twapQuote", twapQuote);

    assertLt(spotQuote, twapQuote);

    // enable vault usage
    vm.startPrank(address(bootstrap));
    IHexOneVault(vault).setSacrificeStatus();
    vm.stopPrank();

    // user deposits 2 million HEX in the vault receiving ~50% more HEX1 than
    ↪  expected
    vm.startPrank(user);
    IERC20(hexToken).approve(address(vault), hexBalance);
    (uint256 hex1Balance, ) = IHexOneVault(vault).deposit(hexBalance, 3800);

    // compare expected spot quote vs borrowed HEX1
    assertGt(hex1Balance, spotQuote);
    emit log_named_uint("hex1Balance", hex1Balance);
}
```

```
spotQuote:   20467262326253594781562
twapQuote:   34972723180079698511440
hex1Balance: 33224087021075713585868
```

Recommended Mitigation Steps

Consider replacing the TWAP price feed system with a high-frequency automated price feed that considers the median of all relevant LPs within Pulsechain for the `HEX/DAI` pair, instead of using a single source of truth from PulseX.

[M-01] `setBaseData()` method may allow Owner to take malicious action

ID	Classification	Category	Status
M-01	MEDIUM	Access Control	Fixed in fae2456, a884f78

Impact

In a scenario where an actor has **owner** permissions in the `HexOneBootstrap.sol`, `HexOneStaking.sol` and `HexOneVault.sol` contracts, said actor can replace the delegated `baseData` addresses to contracts that are cross-compatible on an interface level but may have malicious code inserted into its methods.

Proof of Concept

The following snippet of the `HexOneBootstrap.sol` contract is an occurrence where the owner may re-route any of the following input addresses to any contract that has a compatible interface, enabling the possibility of said addresses routing to contracts with malicious code. We believe this adds a centralization risk that counters the intended permissionless nature of the protocol. Consider the following scenario:

1. Alice has **owner** permissions for the `HexOneBootstrap.sol` contract
2. Alice deploys a `HexOneStaking.sol` contract that has an identical interface, but modifies the `transfer()` call in either the `deposit()` or `withdraw()` functions so that any resulting transfer from execution is routed to an EOA controlled by her instead.
3. Alice calls the `setBaseData()` function in the `HexOneBootstrap.sol` contract, changing the address route to her newly deployed contract, thus siphoning funds from the protocol.

```
/// @dev set the address of other protocol contracts.
/// @notice can only be called by the owner.
function setBaseData(
    address _hexOnePriceFeed,
    address _hexOneStaking,
    address _hexOneVault
) external onlyOwner {
    if (_hexOnePriceFeed == address(0))
        revert InvalidAddress(_hexOnePriceFeed);
    if (_hexOneStaking == address(0)) revert InvalidAddress(_hexOneStaking);
    if (_hexOneVault == address(0)) revert InvalidAddress(_hexOneVault);

    hexOnePriceFeed = _hexOnePriceFeed;
    hexOneStaking = _hexOneStaking;
    hexOneVault = _hexOneVault;
}
```

Recommended Mitigation Steps

Change the deployment architecture so that any external addresses are defined on the contract's constructor (avoiding the need for circular contract dependencies but rather a hierarchical model dependant on a sequential deployment order) and remove the `setBaseData()` method.

[M-02] - Missing state updates in the `_updatePoolHistory()` method lead to staking reward accounting issues

ID	Classification	Category	Status
M-02	MEDIUM	Stateful Fuzzing	Fixed in 0d692e6

Impact

In a scenario where the total user shares equal 0 for a given `stakingDay`, rewards might be distributed although no one is able to receive them. That happens because there is no check to ensure that there are shares in the pool, so if there are no shares in the pool (meaning that no one has deposited yet), the pool will still distribute rewards for that `stakingDay` that will ultimately be lost since no one will be able to claim them.

Proof of Concept

The `_updatePoolHistory()` internal method within the `HexOneStaking.sol` contract is not syncing the `poolHistory.amountToDistribute` whenever `pool.totalShares` is equal to 0. This implies that if no users interact with any user-sided methods for the whole duration of a `stakingDay` the rewards for that given `stakingDay` are distributed by the pool but are no longer receivable by any users. Given that the protocol distributes 1% of the available pool shares daily, this issue might create a significant inconsistency in reward distribution efficiency.

```
function _updatePoolHistory(address _poolToken) internal {
    Pool storage pool = pools[_poolToken];

    uint256 currentStakingDay = pool.currentStakingDay;
    while (currentStakingDay < getCurrentStakingDay()) {
        // get the pool rewards for each day since it was last updated
        PoolHistory storage history =
        ↪  poolHistory[currentStakingDay][_poolToken];

        // store the total shares emitted by the pool at a specific day
        history.totalShares = pool.totalShares;

        // calculate the amount of pool token to distribute for a specific
        ↪  staking day
        uint256 availableAssets = pool.totalAssets - pool.distributedAssets;
        uint256 amountToDistribute = (availableAssets *
        ↪  pool.distributionRate) / FIXED_POINT;
        history.amountToDistribute = amountToDistribute;

        // increment the distributedAssets by the pool
        pool.distributedAssets += amountToDistribute;

        // increment the staking day in which the pool rewards were last
        ↪  updated
        currentStakingDay++;
    }
    pool.currentStakingDay = currentStakingDay;
}
```

If Alice deposits on day 1 and on day 2 she withdraws and there are no new user deposits in the HexOneStaking.sol contract, no users will be able to capitalize on day 2 rewards, rendering said rewards as permanently lost.

The following invariant test in the HexOneProperties.sol contract had a **failed assertion** detected during Echidna stateful fuzz runs, indicating that the condition mentioned above was broken.

```
/// @custom:invariant - staking history.amountToDistribute for a given day
↪ must always be == 0 whenever pool.totalShares is also == 0
function poolAmountStateIntegrity() public {
    for (uint256 i = 0; i < stakeTokens.length; i++) {
        (,,, uint256 currentStakingDay,) =
        ↪ hexOneStakingWrap.pools(address(stakeTokens[i]));
        (,, uint256 totalShares,,) =
        ↪ hexOneStakingWrap.pools(address(stakeTokens[i]));
        (, uint256 amountToDistribute) =
        ↪ hexOneStakingWrap.poolHistory(currentStakingDay,
        ↪ address(stakeTokens[i]));


        if (totalShares == 0) {
            assert(totalShares == amountToDistribute);
        }
    }
}
```

Recommended Mitigation Steps

Add the following snippet to the _updatePoolHistory() internal method after accessing the Pool struct in storage:

```
+   if (pool.totalShares == 0) {
        poolHistory.amountToDistribute = 0
    }
    else {
        (...rest of code...)
    }
```

[M-03] - Inaccurate airdrop calculation results in excessive HEXIT distribution

ID	Classification	Category	Status
M-03	MEDIUM	Manual Review	Fixed in c33cbdc

Impact

The inaccurate airdrop calculation mechanism has a significant impact on the HEXIT token supply, resulting in unintended dilution and excessive distribution.

Proof of Concept

In the following snippet, instead of the AIRDROP_DECREASE_FACTOR constant, the SACRIFICE_DECREASE_FACTOR constant is used to calculate the amount of HEXIT per dollar. This leads to a smaller decrease factor and consequently to a distribution of more tokens than expected.

```
/// @dev the sacrifice base hexit amount per dollar decreases 4.76% daily.
uint16 public constant SACRIFICE_DECREASE_FACTOR = 9524;
/// @dev the airdrop base hexit amount per dollar decreases 50% daily.
uint16 public constant AIRDROP_DECREASE_FACTOR = 5000;
```

```
/// @dev computes the amount of HEXIT per dollar to distribute based on the
/// current airdrop day.
function _airdropBaseHexitPerDollar() internal view returns (uint256
↪  baseHexit) {
    uint256 currentAirdropDay = getCurrentAirdropDay();
    if (currentAirdropDay == 1) {
        return AIRDROP_HEXIT_INIT_AMOUNT;
    }

    baseHexit = AIRDROP_HEXIT_INIT_AMOUNT;
    for (uint256 i = 2; i <= currentAirdropDay; ++i) {
        baseHexit = (baseHexit * SACRIFICE_DECREASE_FACTOR) / FIXED_POINT;
    }
}
```

Recommended Mitigation Steps

Change the SACRIFICE_DECREASE_FACTOR constant for the AIRDROP_DECREASE_FACTOR constant in the _airdropBaseHexitPerDollar() method as follows:

```
function _airdropBaseHexitPerDollar() internal view returns (uint256
 ↪  baseHexit) {
    uint256 currentAirdropDay = getCurrentAirdropDay();
    if (currentAirdropDay == 1) {
        return AIRDROP_HEXIT_INIT_AMOUNT;
    }

    baseHexit = AIRDROP_HEXIT_INIT_AMOUNT;
    for (uint256 i = 2; i <= currentAirdropDay; ++i) {
-       baseHexit = (baseHexit * SACRIFICE_DECREASE_FACTOR) / FIXED_POINT;
+       baseHexit = (baseHexit * AIRDROP_DECREASE_FACTOR) / FIXED_POINT;
    }
}
```

[M-04] - Return expression from `getCurrentAirdropDay()` method returning unexpected value

ID	Classification	Category	Status
M-04	MEDIUM	Manual Review	Fixed in c33cbdc

Impact

In a scenario where the `getCurrentAirdropDay()` method is called internally in the protocol, unexpected accounting might occur, leading to incorrect airdrop distribution for users.

Proof of Concept

In the snippet below, the `getCurrentAirdropDay()` method returns 0. This is due to the way the parenthesis priority in the mathematical formula that defines the method output is set.

```
/// @dev returns the current day of the airdrop.
/// @notice if the airdrop had just started this func would return day 1.
function getCurrentAirdropDay() public view returns (uint256) {
    if (block.timestamp < airdropStart) revert
    ↪  AirdropHasNotStartedYet(block.timestamp);
    if (block.timestamp >= airdropEnd) revert
    ↪  AirdropAlreadyEnded(block.timestamp);

    return ((block.timestamp) - airdropStart / 1 days) + 1;
}
```

Recommended Mitigation Steps

Consider changing the return formula to the following:

```
-   return ((block.timestamp) - airdropStart / 1 days) + 1;
+   return ((block.timestamp - airdropStart) / 1 days) + 1;
```

[M-05] - Overriding of `stakeIds` can lead to debt erasure

ID	Classification	Category	Status
M-05	MEDIUM	Manual Review	Fixed in 41aef76

Impact

In a scenario where the `stakeIds` overlap, an attacker can exploit said scenario to circumvent repaying his `HEX1` debt.

Proof of Concept

Each time a user uses the `deposit()` method in the `HexOneVault` contract, a corresponding `stakeId` is associated with the deposit:

```
stakeId = IHexToken(hexToken).stakeCount(address(this)) - 1;
```

However, this snippet of code was added with the assumption that the stake count in `HEX` contract continuously increments, when in fact it represents the total number of active `HEX` stakes associated to a specific address. Consider the following scenario:

- Alice initiates a `HEX` deposit for a period of **5555 days** in the vault, minting `HEX` with a corresponding `stakeId = 0` by calling the `deposit()` function.
- Bob also initiates a `HEX` deposit for a period of **5555 days** in the vault, minting `HEX1` with a corresponding `stakeId = 1` by calling the `deposit()` function.
- Alice claims her `HEX` and yield by repaying her `HEX1` debt whenever the 5555 days period has passed by calling the `claim()` function.
- Bob proceeds to call the `deposit()` function with an amount of `1 HEX`, thus creating a new deposit with `stakeId = 1` and effectively circumventing the payment of his debt to the protocol by overriding the data from his initial deposit.

```
function test_vault_overlappingStakeIds() public {
    // alice deposits in the vault, stakeId = 0 (stakeCount == 1)
    vm.prank(alice);
    (uint256 aliceHex1Borrowed, uint256 aliceStakeId) =
    ↪  vault.deposit(ALICE_HEX_AMOUNT, MIN_DURATION_DAYS);

    assertEq(aliceStakeId, 0);
    assertEq(hexToken.stakeCount(address(vault)), 1);

    // bob deposits in the vault, stakeId = 1 (stakeCount == 2)
    vm.prank(bob);
    (uint256 bobHex1BorrowedBefore, uint256 bobStakeIdBefore) =
        vault.deposit(BOB_HEX_AMOUNT - 1e8, MIN_DURATION_DAYS);

    assertEq(bobStakeIdBefore, 1);
    assertEq(hexToken.stakeCount(address(vault)), 2);

    // advance block.timestamp so that the HEX stakes are mature
    skip(MIN_DURATION_SECONDS);

    // alice claims its stake and reedems it's HEX + yield by repaying the
    ↪  borrowed HEX1 (stakeCount == 1)
    vm.startPrank(alice);
    hex1.approve(address(vault), aliceHex1Borrowed);
    vault.claim(aliceStakeId);
    vm.stopPrank();

    assertEq(hexToken.stakeCount(address(vault)), 1);

    // bob deposits 1 HEX in the vault so his new stake overlaps the old one,
    ↪  stakeId = 1 (stakeCount == 2)
    vm.prank(bob);
    (uint256 bobHex1BorrowedAfter, uint256 bobStakeIdAfter) =
    ↪  vault.deposit(1e8, MIN_DURATION_DAYS);

    assertEq(bobStakeIdBefore, 1);
    assertEq(bobStakeIdAfter, bobStakeIdBefore); // stake id overlapped
    assertEq(hexToken.stakeCount(address(vault)), 2);

    // bob is able replace the information of is first deposit with the
    ↪  information of the second one
    // since they have the same stake id, allowing him to avoid paying back
    ↪  the HEX1 borrowed.
    assertGt(bobHex1BorrowedBefore, bobHex1BorrowedAfter);
}
```

Recommended Mitigation Steps

Consider introducing a state variable representing the `currentStakeId` and increment it every time a `deposit` occurs.

[M-06] - `setHexOneBootstrap()` method may allow Owner to take malicious action

ID	Classification	Category	Status
M-06	MEDIUM	Access Control	Fixed in 50c32bc

Impact

In a scenario where an actor is the **Owner** of the `HexitToken` contract, said actor can replace the delegated `hexOneBootstrap` address to a contract that is cross-compatible on an **Interface** level but may have malicious code inserted into its methods.

Proof of Concept

The following snippet of the **HexitToken.sol** contract is an occurrence where the **Owner** may re-route the **HexOneBootstrap** contract input address to any contract that has a compatible interface, enabling the possibility of said address routing to a contract with malicious code.

```
/// @dev set the address of the bootstrap.
/// @param _hexOneBootstrap address of the bootstrap.
function setHexOneBootstrap(address _hexOneBootstrap) external onlyOwner {
    if (_hexOneBootstrap == address(0)) revert InvalidAddress();
    hexOneBootstrap = _hexOneBootstrap;
    emit BootstrapInitialized(_hexOneBootstrap);
}
```

We believe this adds a centralization risk that counters the intended permissionless nature of the protocol.

Consider the following scenario:

- Alice has **Owner** permissions for the `HexitToken` contract.
- Alice deploys a `HexOneBootstrap` contract that has an identical interface, but modifies the `sacrifice()` method so that any sacrifice deposits are routed to an **EOA** controlled by her instead.
- Alice calls the `setHexOneBootstrap()` function in the `HexitToken` contract, changing the address route to her newly deployed contract, thus being able to siphon funds from the protocol.

Recommended Mitigation Steps

Change the deployment architecture so that any external addresses are defined on the contract's constructor (avoiding the need for circular contract dependencies but rather a hierarchical model dependant on a sequential deployment order) and remove the `setHexOneBootstrap()` method.

[M-07] - `setHexOneVault()` method may allow Owner to take malicious action

ID	Classification	Category	Status
M-07	MEDIUM	Access Control	Fixed in 50c32bc

Impact

In a scenario where an actor is the **Owner** of the `HexOneToken` contract, said actor can replace the delegated `hexOneVault` address to a contract that is cross-compatible on an **Interface** level but may have malicious code inserted into its methods.

Proof of Concept

The following snippet of the `HexOneToken.sol` contract is an occurrence where the **Owner** may re-route the `HexOneVault` contract input address to any contract that has a compatible interface, enabling the possibility of said address routing to a contract with malicious code.

```
/// @dev set the address of the vault.
/// @param _hexOneVault address of the vault.
function setHexOneVault(address _hexOneVault) external onlyOwner {
    if (_hexOneVault == address(0)) revert InvalidAddress();
    hexOneVault = _hexOneVault;
    emit VaultInitialized(_hexOneVault);
}
```

We believe this adds a centralization risk that counters the intended permissionless nature of the protocol.

Consider the following scenario:

- Alice has **Owner** permissions for the `HexOneToken` contract.
- Alice deploys a `HexOneVault` contract that has an identical interface, but modifies the `deposit()` method so that any vault deposits are routed to an **EOA** controlled by her instead.
- Alice calls the `setHexOneVault()` function in the `HexOneToken` contract, changing the address route to her newly deployed contract, thus being able to siphon funds from the protocol.

Recommended Mitigation Steps

Change the deployment architecture so that any external addresses are defined on the contract's constructor (avoiding the need for circular contract dependencies but rather a hierarchical model dependant on a sequential deployment order) and remove the `setHexOneVault()` method.

[L-01] Lack of total weight amount check can cause unexpected behavior

ID	Classification	Category	Status
L-01	LOW	Quality Assurance	Fixed in f6f4f6f

Impact

In the following snippet, there are no checks to verify that the total weight value does not exceed 100%. The absence of this check could cause accounting issues.

```
function setStakeTokens(
    address[] calldata _tokens,
    uint16[] calldata _weights
) external onlyOwner {
    uint256 length = _tokens.length;
    require(length > 0, "Zero length array");
    require(length == _weights.length, "Mismatched array");

    for (uint256 i; i < length; ++i) {
        address token = _tokens[i];
        uint16 rate = _weights[i];

        require(!stakeTokens.contains(token), "Token already added");
        require(
            rate != 0 && rate <= FIXED_POINT,
            "Invalid distribution rate"
        );

        stakeTokens.add(token);
        stakeTokenWeights[token] = rate;
    }
}
```

Recommended Mitigation Steps

Add a check to verify that the **total weight** amount does not exceed **100%**.

[L-02] teamWallet is the recipient of a significant portion of the HEXIT supply

ID	Classification	Category	Status
L-02	LOW	Access Control	Acknowledged

Impact

In a scenario where the teamWallet address defined on the HexOneBootstrap.sol constructor on initialization is not a multisig there might be centralization issues since this wallet is the recipient of a considerable amount of the supply..

Proof of Concept

In the HexOneBootstrap.sol contract constructor, the teamWallet address might not be a multisig. This might influence external trust in the protocol as significant percentage of the HEXIT supply is owned by the team.

```
constructor(
    address _pulseXRouter,
    address _pulseXFactory,
    address _hexToken,
    address _hexitToken,
    address _daiToken,
    address _hexOneToken,
    address _teamWallet
) Ownable(msg.sender) {
    if (_pulseXRouter == address(0)) revert InvalidAddress(_pulseXRouter);
    if (_pulseXFactory == address(0)) revert InvalidAddress(_pulseXFactory);
    if (_hexToken == address(0)) revert InvalidAddress(_hexToken);
    if (_hexitToken == address(0)) revert InvalidAddress(_hexitToken);
    if (_daiToken == address(0)) revert InvalidAddress(_daiToken);
    if (_teamWallet == address(0)) revert InvalidAddress(_teamWallet);

    pulseXRouter = _pulseXRouter;
    pulseXFactory = _pulseXFactory;
    hexToken = _hexToken;
    hexitToken = _hexitToken;
    daiToken = _daiToken;
    hexOneToken = _hexOneToken;
    teamWallet = _teamWallet;
}
```

Consider the following scenario:

1. Alice has sole custody of the teamWallet EOA.
2. Alice gets the 50% of the current HEXIT supply that is minted on top of the total HEXIT minted through the sacrifice phase when the startAirdrop() method gets called.
3. Alice is now in custody of a significant HEXIT amount and proceeds to sell it against a HEXIT LP pair, draining the value of the token.

Recommended Mitigation Steps

Ensure the `teamWallet` address is a multisig EOA.

[L-03] - Lack of minimum `deposit()` amount requirement leads to inaccurate accounting

ID	Classification	Category	Status
L-03	LOW	Stateful Fuzzing	Fixed in af718e2

Impact

In a scenario where a very small amount of `HEX` is deposited into the `HexOneVault.sol` contract there might be accounting issues due to a rounding error.

Proof of Concept

The `deposit()` method within the `HexOneVault.sol` contract allows the user to deposit `HEX`, consults the current `HEX` price in dollar and determines how much `HEX1` the user is allowed to borrow/mint given that quote.

```
/// @dev allows bootstrap to make deposit in name of`_depositor` and mint
 ↪  HEX1.
/// @param _depositor address of the user depositing.
/// @param _amount amount of HEX being deposited.
/// @param _duration of the HEX stake.
function deposit(address _depositor, uint256 _amount, uint16 _duration)
    external
    onlyAfterSacrifice
    onlyHexOneBootstrap
    returns (uint256 hexOneMinted, uint256 stakeId)
{
    if (_duration < MIN_DURATION || _duration > MAX_DURATION) revert
    ↪  InvalidDepositDuration(_duration);
    if (_amount == 0) revert InvalidDepositAmount(_amount);
    if (_depositor == address(0)) revert InvalidDepositor(_depositor);

    IERC20(hexToken).safeTransferFrom(hexOneBootstrap, address(this),
    ↪  _amount);

    return _deposit(_depositor, _amount, _duration);
}
```

During our Echidna fuzz runs we found several cases where the total accounting between the total borrowed amount by users and the possible user ERC20 balances had a very slight discrepancy due to a rounding error in the calculation of the final amount to deposit after taking the 5% fee on deposits imposed by the protocol. We then used trial and error to conclude what would be the minimum amount of outputted `HEX1` to avoid this issue. We concluded that the minimum amount of `HEX1` minted from a user would need to be greater than **1e14**, or **0.0001**.

```
/// @notice takes a 5% fee to be distributed as a staking reward.
/// @param _depositor address of the user depositing.
/// @param _amount amount of HEX being deposited.
/// @param _duration of the HEX stake.
function _deposit(address _depositor, uint256 _amount, uint16 _duration)
    internal
    returns (uint256 hexOneMinted, uint256 stakeId)
{
    // calculate the fee and the real amount being deposited
    uint256 feeAmount = (_amount * DEPOSIT_FEE) / FIXED_POINT;
    uint256 realAmount = _amount - feeAmount;
    (...)
}
```

- Fee calculation snippet in the `_deposit()` internal method

This issue could eventually lead to bricked user positions if the amount of `HEX` they deposited to the Vault would be considerably small. Nonetheless, given the nature of the protocol, such amounts would represent very specific and unlikely edge cases, hence the Low severity classification.

```
function hexOneLiquidationsIntegrity() public {
    uint256 totalHexoneUsersAmount;
    uint256 totalHexoneProtocolAmount;

    for (uint256 i = 0; i < totalNbUsers; i++) {
        (,, uint256 totalBorrowed) =
        ↪  hexOneVault.userInfos(address(users[i]));
        totalHexoneProtocolAmount += totalBorrowed;
        totalHexoneUsersAmount += hex1.balanceOf(address(users[i]));
    }

    require(totalHexoneUsersAmount != 0 && totalHexoneProtocolAmount != 0);

    emit LogUint(totalHexoneUsersAmount);
    emit LogUint(totalHexoneProtocolAmount);
    assert(totalHexoneUsersAmount >= totalHexoneProtocolAmount);
}
```

- This test would fail the Echidna assertions by a varying threshold of 5 to 10 between either the users and the protocol.

```
require(totalHexoneUsersAmount > 1e14);
```

- After the trial and error process through Echidna fuzz runs and testing different precisions for the calculation of the amount to deposit after the 5% fee imposed by the protocol, all solutions for that given accounting formula led to this minimum `HEX1` output value to remediate the precision issue.
- We then calculated an estimate of minimum required `HEX` for a deposit, being mindful that `HEX` is a token with 8 decimals, unlike the 18 decimal standard ERC20 usually employs.
- Being mindful of the current `HEX` price and the considerations made above, we calculated that the minimum deposit amount should be between **0.1** and **1** `HEX` to avoid the issue. The final consideration should be up to the client to decide according to what would be more suitable for the context of the protocol. It is worth noting that greater minimum `HEX` deposit amounts than the ones previously mentioned would reduce potential issues stemming from this root cause should the `HEX` price decrease significantly.

NOTE: During a sacrifice, the minimum sacrifice amount should be bigger than the HEX amount equivalent to 1 DAI to avoid the issue. If the output sacrifice amount is less than **2e16 (0.02 `DAI`)** there is a **Denial-of-Service** issue due to the resulting `HEX` Vault delegate `deposit` amount being less than **1e7**, so adding the following initial check to the `sacrifice()` method mitigates this child issue:

```
// calculate the hexit shares of the token being sacrificed
+ (uint256 hexitShares, uint256 amountSacrificedUSD) =
↪  _calculateHexitSacrificeShares(_token, _amountIn);
+ if (amountSacrificedUSD < 1e18) revert
↪  InvalidSacrificeAmount(amountSacrificedUSD);
```

NOTE: We have deduced that this scenario also applies to the daily distribution rate equalling 1%, as per the following snippet.

Recommended Mitigation Steps

Add the following snippet to the `_deposit()` internal method at the top:

```
+ require(_amount > 1e7, "Deposit less than minimum amount");
```

Alternatively, define a custom error and add it at the top level of the contract:

```
+ error DepositLessThanMinimumAmount(uint256 _amount);
...
+ if (_amount <= 1e7) revert DepositLessThanMinimumAmount(_amount);
```

[L-04] - Lack of minimum `stake()` amount requirement leads to inaccurate accounting

ID	Classification	Category	Status
L-04	LOW	Stateful Fuzzing	Fixed in af718e2

Impact

In a scenario where a very small amount of **eligible ERC20 token** is staked into the `HexOneStaking.sol` contract there might be accounting issues due to a rounding error.

Proof of Concept

The `stake()` function within the `HexOneStaking.sol` contract facilitates users in staking `HEXIT`, `HEX1`, and/or `HEX1/DAI LP` tokens to acquire shares, thereby enabling them to receive daily rewards in both `HEXIT` and `HEX` tokens.

```
function stake(address _stakeToken, uint256 _amount) external nonReentrant
↪   onlyWhenStakingEnabled {
    require(stakeTokens.contains(_stakeToken), "Token not allowed");
    require(_amount > 0, "Invalid staking amount");

    // accrue rewards and update history for both the HEX and HEXIT pools
    _accrueRewards(msg.sender, _stakeToken);

    // transfers amount of stake token from the sender to this contract.
    uint256 stakeAmount = _transferToken(_stakeToken, msg.sender,
    ↪   address(this), _amount);

    // update the total amount staked
    totalStakedAmount[_stakeToken] += stakeAmount;

    // calculate the amount of HEX and HEXIT pool shares to give to the user
    uint256 shares = _calculateShares(_stakeToken, stakeAmount);
    require(shares != 0, "Invalid shares amount");

    // update the number of total shares in the HEX and HEXIT pools
    pools[hexToken].totalShares += shares;
    pools[hexitToken].totalShares += shares;

    // update the staking information of the user for a specific stake token
    uint256 currentStakingDay = getCurrentStakingDay();
    StakeInfo storage stakeInfo = stakingInfos[msg.sender][_stakeToken];
    stakeInfo.stakedAmount += stakeAmount;
    if (stakeInfo.initStakeDay == 0) {
        stakeInfo.initStakeDay = currentStakingDay;
    }
    if (stakeInfo.lastClaimedDay == 0) {
        stakeInfo.lastClaimedDay = currentStakingDay;
    }
    stakeInfo.lastDepositedDay = currentStakingDay;
    stakeInfo.hexSharesAmount += shares;
    stakeInfo.hexitSharesAmount += shares;
}
```

Consider a scenario where Alice initiates **N** deposits of **X** amount, while on the same day, Bob makes **pN** deposits totaling the same amount, **X**, maintaining an equivalent stake. In such a scenario, both Alice and Bob should possess identical shares and consequently receive equal rewards.

During our Echidna fuzz runs, we identified several instances where this fundamental principle is compromised, leading to an inconsistency in the allocation of shares to users and subsequently affecting their reward amounts. This inconsistency arises due to a rounding error within the `_calculateShares()` function, which determines the distribution of shares to users based on their staked amounts.

```
/// @dev calculates the shares to be given to the user depending on the token
↪  staked
/// @notice shares are always 18 decimals, so depending on the token it might
↪  need to be scaled up or down.
/// @param _stakeToken address of the stake token.
/// @param _amount amount of stake token.
function _calculateShares(address _stakeToken, uint256 _amount) internal view
↪  returns (uint256) {
    uint256 shares = (_amount * stakeTokenWeights[_stakeToken]) /
    ↪  FIXED_POINT;
    return _convertToShares(_stakeToken, shares);
}
```

The provided snippet illustrates the scenario previously described, wherein both Alice and Bob stake an amount **X** of tokens on the same day. However, while Alice makes **N** stakes, Bob makes **pN** stakes. The failure of this test highlights an inconsistency in the distribution of shares to Alice and Bob. Despite the expectation of identical rewards for both parties, this consistency does not manifest due to precision loss in the calculation of shares.

```
function totalRewardsToDistToAliceWithNDepositsOfXValueMustEqForBobWithPNDe-
↪  positsOfXValue(
    uint256 randUser,
    uint256 randToken,
    uint256 randAmount,
    uint256 randNDeposits,
    uint256 randP,
    uint256 randWarp
) public {
    // preconditions
    require(hexOneStakingWrap.stakingEnabled());

    // setup
    bool success;
    uint256 nDeposits = clampBetween(randNDeposits, 1, 4);
    uint256 p = clampBetween(randP, 1, 2);
    address token = stakeTokens[randToken % stakeTokens.length];
    User alice = users[randUser % users.length];
    User bob = users[(randUser + 1) % users.length];

    // cleanup
    logState(abi.encode("BEFORE CLEANUP"), alice, bob, token);
    (uint256 aliceAlreadyClaimedRewards, uint256 bobAlreadyClaimedRewards) =
    ↪  setupCleanup(alice, bob, token);
    logState(abi.encode("AFTER CLEANUP"), alice, bob, token);

    // setup amounts
    uint256 maxTokenBalance = ERC20Mock(token).balanceOf(address(alice)) <=
    ↪  ERC20Mock(token).balanceOf(address(bob))
        ? ERC20Mock(token).balanceOf(address(alice))
        : ERC20Mock(token).balanceOf(address(bob));
    require(maxTokenBalance >= 100);
    uint256 bobAmount = clampBetween(randAmount, 1, maxTokenBalance / 100);
    require(bobAmount != 0);
    require(ERC20Mock(token).balanceOf(address(alice)) >= bobAmount * p);
    require(ERC20Mock(token).balanceOf(address(bob)) >= bobAmount);

    // log setup
    emit LogAddress("HEXIT (10%)", address(hexit));
    emit LogAddress("HEX1 (20%)", address(hex1));
    emit LogAddress("HEX1/DAI (70%)", address(hex1dai));
    emit LogAddress("Stake token", token);
    emit LogAddress("Alice", address(alice));
    emit LogAddress("Bob", address(bob));
    emit LogUint256("N", nDeposits);
    emit LogUint256("p", p);
    emit LogUint256("Alice Amount", bobAmount * p);
    emit LogUint256("Bob amount", bobAmount);

    // stake
    // Alice N deposits
    for (uint8 i; i < nDeposits; ++i) {
        (success,) = alice.proxy(
            address(hexOneStakingWrap),
            abi.encodeWithSelector(hexOneStakingWrap.stake.selector, token, p
            ↪   * bobAmount)
        );
        require(success);
    }

    // Bob pN deposits
    for (uint8 j; j < p * nDeposits; ++j) {
        (success,) = bob.proxy(
            address(hexOneStakingWrap),
            ↪   abi.encodeWithSelector(hexOneStakingWrap.stake.selector,
            ↪   token, bobAmount)
        );
        require(success);
    }

    logState(abi.encode("AFTER DEPOSITS"), alice, bob, token);

    // claim
    uint256 warpValue = clampBetween(randWarp, 2 days, 90 days);
    hevm.warp(block.timestamp + warpValue);

    emit LogUint256("After X days", warpValue / 1 days);

    (success,) =
        alice.proxy(address(hexOneStakingWrap),
        ↪   abi.encodeWithSelector(hexOneStakingWrap.claim.selector, token));
    require(success);

    (success,) =
        bob.proxy(address(hexOneStakingWrap),
        ↪   abi.encodeWithSelector(hexOneStakingWrap.claim.selector, token));
    require(success);

    logState(abi.encode("AFTER CLAIM"), alice, bob, token);

    // assert
    (,,,,,,,, uint256 aliceTotalHexClaimed, uint256 aliceTotalHexitClaimed) =
        hexOneStakingWrap.stakingInfos(address(alice), token);
    (,,,,,,,, uint256 bobTotalHexClaimed, uint256 bobTotalHexitClaimed) =
        hexOneStakingWrap.stakingInfos(address(bob), token);

    assertEq(
        aliceTotalHexClaimed + aliceTotalHexitClaimed -
        ↪   aliceAlreadyClaimedRewards,
        bobTotalHexClaimed + bobTotalHexitClaimed - bobAlreadyClaimedRewards,
        "Rewards mismatch"
    );

    // cleanup state for the following invariants
    setupCleanup(alice, bob, token);
}
```

Recommended Mitigation Steps

Implement a **require** statement enforcing a minimum stake amount of at least **1e14** tokens:

```
+ require(_amount > 1e14, "Stake less than minimum amount");
```

Alternatively, define a custom error and add it at the top:

```
+ error StakeLessThanMinimumAmount(uint256 _amount);
...
+ if (_amount <= 1e14) revert StakeLessThanMinimumAmount(_amount);
```

[L-05] - The growth of balances within the `HexOneStaking.sol` contract does not consistently align with proportional increases in share allocation

ID	Classification	Category	Status
L-05	LOW	Stateful Fuzzing	Fixed in af718e2

Impact

The `stake()` function within the `HexOneStaking.sol` contract facilitates users in staking `HEXIT`, `HEX1`, and/or `HEX1/DAI LP` tokens to acquire shares, thereby enabling them to receive daily rewards in both `HEXIT` and `HEX` tokens.

```
/// @dev allow users to stake HEXIT, HEX1 or HEX1/DAI to earn HEX and HEXIT
 ↪  rewards.
/// @param _stakeToken address of the token being staked.
/// @param _amount of token being staked.
function stake(address _stakeToken, uint256 _amount) external nonReentrant
 ↪  onlyWhenStakingEnabled {
    require(stakeTokens.contains(_stakeToken), "Token not allowed");
    require(_amount > 0, "Invalid staking amount");

    // accrue rewards and update history for both the HEX and HEXIT pools
    _accrueRewards(msg.sender, _stakeToken);

    // transfers amount of stake token from the sender to this contract.
    uint256 stakeAmount = _transferToken(_stakeToken, msg.sender,
     ↪  address(this), _amount);

    // update the total amount staked
    totalStakedAmount[_stakeToken] += stakeAmount;

    // calculate the amount of HEX and HEXIT pool shares to give to the user
    uint256 shares = _calculateShares(_stakeToken, stakeAmount);
    require(shares != 0, "Invalid shares amount");

    // update the number of total shares in the HEX and HEXIT pools
    pools[hexToken].totalShares += shares;
    pools[hexitToken].totalShares += shares;

    // update the staking information of the user for a specific stake token
    uint256 currentStakingDay = getCurrentStakingDay();
    StakeInfo storage stakeInfo = stakingInfos[msg.sender][_stakeToken];
    stakeInfo.stakedAmount += stakeAmount;
    if (stakeInfo.initStakeDay == 0) {
        stakeInfo.initStakeDay = currentStakingDay;
    }
    if (stakeInfo.lastClaimedDay == 0) {
        stakeInfo.lastClaimedDay = currentStakingDay;
    }
    stakeInfo.lastDepositedDay = currentStakingDay;
    stakeInfo.hexSharesAmount += shares;
    stakeInfo.hexitSharesAmount += shares;
}
```

In any staking scenario, the factor by which shares increase should always be equal to the factor by which balances increase within the `HexOneStaking.sol` contract.

During our Echidna fuzz runs, we identified several instances where this fundamental property is broken, leading to an inconsistency. This inconsistency arises due to a rounding error within the `_calculateShares()` function, which determines the distribution of shares to users based on their staked amounts.

```
/// @dev calculates the shares to be given to the user depending on the token
↪  staked
/// @notice shares are always 18 decimals, so depending on the token it might
↪  need to be scaled up or down.
/// @param _stakeToken address of the stake token.
/// @param _amount amount of stake token.
function _calculateShares(address _stakeToken, uint256 _amount) internal view
↪  returns (uint256) {
    uint256 shares = (_amount * stakeTokenWeights[_stakeToken]) /
    ↪  FIXED_POINT;
    return _convertToShares(_stakeToken, shares);
}
```

Proof of Concept

The following properties evaluate whether the shares allocated in their respective pools are consistently proportional to the increase in balance of a given stake token based on its weight. By comparing the balance increase factor with the shares increase factor, these functions highlight any discrepancies that might indicate a breakdown in proportionality. Notably, in some cases, these tests have failed, underscoring the presence of inconsistencies in the share allocation mechanism.

HEX pool:

```
function hexPoolSharesToGiveAreAlwaysProportionalToIncreaseInBalance(
    uint256 randUser,
    uint256 randAmount,
    uint256 randStakeToken
) public {
    User user = users[randUser % users.length];
    address stakeToken = stakeTokens[randStakeToken % stakeTokens.length];

    address xToken = stakeTokens[(randStakeToken + 1) % stakeTokens.length];
    uint256 xTokenBalance = hexOneStakingWrap.totalStakedAmount(xToken);
    uint256 xTokenWeight = hexOneStakingWrap.stakeTokenWeights(xToken);

    address yToken = stakeTokens[(randStakeToken + 2) % stakeTokens.length];
    uint256 yTokenBalance = hexOneStakingWrap.totalStakedAmount(yToken);
    uint256 yTokenWeight = hexOneStakingWrap.stakeTokenWeights(yToken);

    uint256 stakeAmount = clampBetween(randAmount, 1,
    ↪  ERC20Mock(stakeToken).balanceOf(address(user)));
    uint256 shares = calculateShares(stakeToken, stakeAmount);

    if (shares > 0) {
        uint256 stakeTokenBalance =
        ↪  hexOneStakingWrap.totalStakedAmount(stakeToken);
        uint256 stakeTokenBalanceIncreaseFactor = ((stakeAmount +
        ↪  stakeTokenBalance) * 10_000) / stakeTokenBalance;

        (,, uint256 totalShares,,) = hexOneStakingWrap.pools(address(hexx));

        uint256 stakeTokenTotalShares =
            totalShares - ((xTokenBalance * xTokenWeight) / 1000) -
            ↪  ((yTokenBalance * yTokenWeight) / 1000);
        uint256 hexSharesIncreaseFactor = ((shares + stakeTokenTotalShares) *
        ↪  10_000) / stakeTokenTotalShares;

        assertEq(stakeTokenBalanceIncreaseFactor, hexSharesIncreaseFactor,
        ↪  "Not proportional");
    }
}
```

HEXIT pool:

```
function hexitPoolSharesToGiveAreAlwaysProportionalToIncreaseInBalance(
    uint256 randUser,
    uint256 randAmount,
    uint256 randStakeToken
) public {
    User user = users[randUser % users.length];
    address stakeToken = stakeTokens[randStakeToken % stakeTokens.length];

    address xToken = stakeTokens[(randStakeToken + 1) % stakeTokens.length];
    uint256 xTokenBalance = hexOneStakingWrap.totalStakedAmount(xToken);
    uint256 xTokenWeight = hexOneStakingWrap.stakeTokenWeights(xToken);

    address yToken = stakeTokens[(randStakeToken + 2) % stakeTokens.length];
    uint256 yTokenBalance = hexOneStakingWrap.totalStakedAmount(yToken);
    uint256 yTokenWeight = hexOneStakingWrap.stakeTokenWeights(yToken);

    uint256 stakeAmount = clampBetween(randAmount, 1,
    ↪  ERC20Mock(stakeToken).balanceOf(address(user)));
    uint256 shares = calculateShares(stakeToken, stakeAmount);

    if (shares > 0) {
        uint256 stakeTokenBalance =
        ↪  hexOneStakingWrap.totalStakedAmount(stakeToken);
        uint256 stakeTokenBalanceIncreaseFactor = ((stakeAmount +
        ↪  stakeTokenBalance) * 10_000) / stakeTokenBalance;

        (,, uint256 totalShares,,) = hexOneStakingWrap.pools(address(hexit));

        uint256 stakeTokenTotalShares =
            totalShares - ((xTokenBalance * xTokenWeight) / 1000) -
            ↪  ((yTokenBalance * yTokenWeight) / 1000);
        uint256 hexitSharesIncreaseFactor = ((shares + stakeTokenTotalShares)
        ↪  * 10_000) / stakeTokenTotalShares;

        assertEq(stakeTokenBalanceIncreaseFactor, hexitSharesIncreaseFactor,
        ↪  "Not proportional");
    }
}
```

Recommended Mitigation Steps

Implement a **require** statement enforcing a minimum stake amount of at least **1e14** tokens:

```
require(_amount >= 1e14, "Stake less than minimum amount");
```

Alternatively, define a custom error and add it at the top:

```
+ error StakeLessThanMinimumAmount(uint256 _amount);
...
+ if (_amount < 1e14) revert StakeLessThanMinimumAmount(_amount);
```

[L-06] - Price feed may be subject to manipulation on deployment

ID	Classification	Category	Status
L-06	LOW	Manual Review	Acknowledged

Impact

In a scenario where an external actor is aware of the deployment schedule for the Hex One Protocol, said actor can frontrun the deployment by organically manipulating the price of HEX, thus being able to mint more HEX1 than expected right after the deployment is concluded.

Proof of Concept

The following snippet within the HexOnePriceFeed.sol contract shows that the first logging of price after a price feed is initialized is fetched directly from the PulseX pool reserves for that given pair.

```
// get the reserves of the pair and the last time the reserves were updated
IPulseXPair pulseXPair = IPulseXPair(pair);
(uint112 reserve0, uint112 reserve1, uint32 blockTimestampLast) =
↪  pulseXPair.getReserves();
```

We believe there might be a minor security consideration to be had here, as an actor with significant HEX holdings might frontrun the deployment, manipulating the price right before the protocol is live to abuse the HEX1 borrowing mechanism, as the reserves output is the actual real-time price and not a cumulative observation.

Consider the following:

- Alice is aware of the deployment schedule for the Hex One Protocol and its respective PulseX pair to get price feeds from and has significant underlying pair holdings for the PulseX pool used for the price feed, as well as significant HEX holdings to lock in the protocol.
- Alice scouts the public mempool for the deployment transactions and proceeds to frontrun said transactions by buying a major amount of HEX against the PulseX pool that is going be logged by the price feed before its initialization, effectively manipulating the underlying value of HEX.
- Alice is then able to mint more HEX1 than initially expected with her share of HEX, immediately followed by a transaction that sells the amount of purchased HEX from the pool (taking advantage of a cyclical swap), thus reducing the intrinsic value of HEX that collateralizes the borrowing of HEX1 and forcing an undercollateralized loan.

Recommended Mitigation Steps

Consider the usage of a private mempool to execute the deployment, especially if the launch date (as in date and hour) of the protocol will be publically known.

[L-07] - Arbitrary `from()` being used in Bootstrap `deposit()` method

ID	Classification	Category	Status
L-07	LOW	Quality Assurance	Fixed in d6eeaae

Impact

In a scenario where the `deposit()` method of the `HexOneVault.sol` contract is called, unexpected behaviour might happen during its internal transfer call.

Proof of Concept

In the following snippet of the `HexOneVault.sol` contract, the `deposit()` function does not respect good practices of transfer calls.

```
/// @dev allows bootstrap to make deposit in name of`_depositor` and mint
↪ HEX1.
/// @param _depositor address of the user depositing.
/// @param _amount amount of HEX being deposited.
/// @param _duration of the HEX stake.
function deposit(address _depositor, uint256 _amount, uint16 _duration)
    external
    onlyAfterSacrifice
    onlyHexOneBootstrap
    returns (uint256 hexOneMinted, uint256 stakeId)
{
    if (_duration < MIN_DURATION || _duration > MAX_DURATION) revert
    ↪   InvalidDepositDuration(_duration);
    if (_amount == 0) revert InvalidDepositAmount(_amount);
    if (_depositor == address(0)) revert InvalidDepositor(_depositor);

    IERC20(hexToken).safeTransferFrom(hexOneBootstrap, address(this),
    ↪   _amount);

    return _deposit(_depositor, _amount, _duration);
}
```

Recommended Mitigation Steps

Change the function name to `depositFromBootstrap()`, in addition to changing the `from` parameter to `msg.sender`.

[I-01] - Typo in `startAidrop()` function name

ID	Classification	Category	Status
I-01	INFORMATIONAL	Quality Assurance	Fixed in 2b4247f

Description

There is a typo in the function name `startAidrop()`, specifically in the word "Aidrop".

```
/// @dev mints 33% on top of the total hexit minted during sacrifice to the
↪   staking
/// contract and an addittional
/// @notice can only be called by the owner of the contract.
function startAidrop() external onlyOwner {
    ...
}
```

Occurrences

- HexOneBootstrap.sol#L360

Recommended Mitigation Steps

Change the method name to `startAirdrop()`.

[I-02] - Several integer ranges do not have proper gas-wise considerations

ID	Classification	Category	Status
I-02	INFORMATIONAL	Quality Assurance	Acknowledged

Description

We assessed that there are several cases (such as the one below) where the maximum bit depth for an integer is being used throughout the scope when not necessary. This causes unnecessary gas costs.

```
/// @dev airdrop phase inital timestamp.
uint256 public airdropStart;
/// @dev airdrop phase final timestamp.
uint256 public airdropEnd;
```

Occurrences

- HexOneBootstrap.sol#L25#L35
- HexOneBootstrap.sol#L78#L96
- HexOneBootstrap.sol#L78#L96
- HexOnePriceFeed.sol#L35#L36
- HexOneStaking.sol#L28#L34
- HexOneStaking.sol#L40
- HexOneVault.sol#L45#L46
- IHexOneBootstrap.sol#L5#L7
- IHexOneBootstrap.sol#L15#L17
- IHexOneStaking.sol#L7#L15
- IHexOneStaking.sol#L22#L52
- IHexOneStaking.sol#L22#L52
- IHexOneVault.sol#L5#L9
- IHexOneVault.sol#L14#L18

Recommended Mitigation Steps

Reassess all integer state variables and reduce their bit depth in a context-fitting solution.

[I-03] - Inefficiencies in storage usage

ID	Classification	Category	Status
I-03	INFORMATIONAL	Quality Assurance	Fixed in d0ccc58

Description

The storage variables within the `HexOneBootstrap.sol`, `HexOneStaking.sol` and `HexOne-Vault.sol` contracts are not optimally declared in terms of storage efficiency.

Occurrences

- HexOneBootstrap.sol#L71-L109
- HexOneStaking.sol#L23-L51
- HexOneVault.sol#L38-L51

Recommended Mitigation Steps

Modify the declaration order to arrange variables from the one occupying the least storage space to the one occupying the most, or vice versa.

[I-04] - Incorrect TWAP stale time code documentation

ID	Classification	Category	Status
I-04	INFORMATIONAL	Quality Assurance	Fixed in 03c8c21

Description

The following constant within the `HexOnePriceFeed.sol` contract indicates that the TWAP period is of one hour.

```
/// @dev period in which the oracle becomes stale.
uint256 public constant PERIOD = 1 hours;
```

Despite this, the following comment indicates that the TWAP period is of two hours. We reached out to the client to rule out either a code or comment inconsistency, to which we concluded was the latter.

```
// if the pair has already been updated in the last 2 hours revert
if (timeElapsed < PERIOD) revert PeriodNotElapsed(pair);
```

Occurrences

- HexOnePriceFeed.sol#L101-L102

Recommended Mitigation Steps

Modify the comment to indicate that the TWAP length is of one hour.

[I-05] - Consider replacing ERC20 libraries with the Solady implementation

ID	Classification	Category	Status
I-05	INFORMATIONAL	Quality Assurance	Acknowledged

Description

The `HexitToken.sol` and `HexOneToken.sol` contracts inherit the ERC20 standard from OpenZeppelin's libraries. Solady is a more gas-optimized alternative, making use of the inline assembler to optimize intermediate calls that accomplish the standard's functionality. Consider replacing the ERC20 inheritance to reduce overall gas costs from token transfers and minting.

Occurrences

- HexOneToken.sol#L4
- HexitToken.sol#L4

Recommended Mitigation Steps

Consider changing the ERC20 inheritance to the corresponding Solady ERC20 library.

[I-06] - Do-while cycle more gas efficient than conventional for loop

ID	Classification	Category	Status
I-06	INFORMATIONAL	Quality Assurance	Will not fix

Description

The following snippet within the HexOnePriceFeed.sol contract is an example of a for loop to iterate over all the possible pairs on the contract's initialization. Do-while loops in Solidity are a more gas efficient implementation, even if you add an if-condition check for the case where the loop doesn't execute at all. It's worthwhile to mention the use of the ++i syntax ensures the incrementing operation is done directly on i, meaning that only one item needs to be stored on the compiler stack.

```
for (uint256 i; i < _pairs.length; i++) {
// check if pair was already added
(...)
}
```

Occurrences

- HexOnePriceFeed.sol#L51-L76
- HexOneBootstrap.sol#L155-L164
- HexOneBootstrap.sol#L490-L492
- HexOneBootstrap.sol#L504-L506
- HexOneBootstrap.sol#L517-L520
- HexOneStaking.sol#L117-L125

Recommended Mitigation Steps

Change the cycle structure wherever possible on the scope to the following:

```
uint256 i;

+ do {
+     unchecked {
+         ++i;
+     }
+ } while (i < _pairs.length);
```

[I-07] - Split revert statements in `HexOnePriceFeed.sol`

ID	Classification	Category	Status
I-07	INFORMATIONAL	Quality Assurance	Fixed in 03d9b43

Description

The following snippet within the `HexOnePriceFeed.sol` contract uses a boolean operator to revert if there are no reserves on either side of the pools' pair. It is more gas efficient to split both conditions and display separate errors for each side of the pool. Furthermore, splitting both revert cases enables a more granular view on the cause of the revert.

```
// check if pair has reserves
if (reserve0 == 0 || reserve1 == 0) revert EmptyReserves(pair);
```

Occurrences

- HexOnePriceFeed.sol#L60-L61

Recommended Mitigation Steps

Consider changing the `if` statement structure to:

```
-   if (reserve0 == 0 || reserve1 == 0) revert EmptyReserves(pair);
+   if (reserve0 == 0) revert EmptyReservesZero(pair);
+   if (reserve1 == 0) revert EmptyReservesOne(pair);
```

[I-08] Multiple unnecessary storage readings

ID	Classification	Category	Status
I-08	INFORMATIONAL	Quality Assurance	Fixed in 48d6f03, d0ccc58, ef52f82

Description

Since Solidity does not cache storage reads, accessing a storage variable incurs a minimum cost of 100 gas. Writes, on the other hand, are notably more costly. Consequently, it's advisable to manually cache the variable to reduce gas expenses.

Occurrences

- HexOneBootstrap.sol#L178-L183
- HexOneBootstrap.sol#L187-L192
- HexOneBootstrap.sol#L252-L311
- HexOneBootstrap.sol#L314-L355
- HexOneBootstrap.sol#L360-L396
- HexOneStaking.sol#L277-L283

Recommended Mitigation Steps

Cache storage variables whenever possible to save gas costs.

[I-09] Unnecessary `contains()` check on `AddressSet`

ID	Classification	Category	Status
I-09	INFORMATIONAL	Quality Assurance	Fixed in 2c26710

Description

It's not necessary to do a `contains()` check on `AddressSet` since it's already checked by `EnumerableSet` library when adding an element.

Occurrences

- HexOnePriceFeed.sol#L54
- HexOneStaking.sol#L120
- HexOneBootstrap.sol#L159

Recommended Mitigation Steps

Consider removing the `contains()` check, as it is redundant given that said check is already done within the `add()` function. Since `add()` returns a boolean value indicating whether the element was added or not, a check could be placed around the `add()` function call instead.

[I-10] - Missing zero address checks

ID	Classification	Category	Status
I-10	INFORMATIONAL	Quality Assurance	Fixed in 7a35247

Description

Some occurrences within the `HexOnePriceFeed.sol`, `HexOneStaking.sol`, and `HexOneBootstrap.sol` contracts lack a **zero address check**. Failing to include this check may lead to unexpected behavior.

Occurrences

- HexOnePriceFeed.sol#L51-L76
- HexOneStaking.sol#L117-L125
- HexOneBootstrap.sol#L155-L164

Recommended Mitigation Steps

Add a zero address check to the occurrences above:

```
+   if (address == address(0)) revert (...)
```

[I-11] - Basis points standard not being respected

ID	Classification	Category	Status
I-11	INFORMATIONAL	Quality Assurance	Fixed in 45a95f1

Description

The constant below, allegedly in basis points, does not correspond to 100%.

```
/// @dev fixed point in basis points
uint16 public constant FIXED_POINT = 1000;
```

Occurrences

- HexOneVault.sol#L27
- HexOneStaking.sol#L54

Recommended Mitigation Steps

Consider changing the `FIXED_POINT` constants value to 10000.

[I-12] - No specific events being emitted in `HexOneStaking.sol`

ID	Classification	Category	Status
I-12	INFORMATIONAL	Quality Assurance	Fixed in 302664b

Description

The `HexOneStaking.sol` contract does not have any direct event emission after purchasing, staking, unstaking or claiming rewards. A `Transfer` event is being emitted every time an ERC20 is transferred, but if it is intended to index specific data from the contracts and/or react to said events in other ends of the stack we would recommend the addition of events that emit specific resulting data from each of these individual methods.

Occurrences

- HexOneStaking.sol

Recommended Mitigation Steps

Consider adding context-fitting events in the `HexOneStaking.sol` contract for each of the aforementioned outcomes.

[I-13] - Incorrect sacrifice incentives code documentation

ID	Classification	Category	Status
I-13	INFORMATIONAL	Quality Assurance	Fixed in 59fbe9f

Description

The following comment within the `HexOneBootstrap.sol` contract indicates that the sacrifice incentive tokens are `HEXIT` and `HEXIT`, when it should be `HEXIT` and `HEX` instead.

```
/// @dev claim HEXIT and HEXIT based on thetotal amount sacrificed.
function claimSacrifice() external returns (
    uint256 stakeId,
    uint256 hexOneMinted,
    uint256 hexitMinted
)
```

Occurrences

- HexOneBootstrap.sol#L313-L314

Recommended Mitigation Steps

Change the code comment to the following:

```
-   /// @dev claim HEXIT and HEXIT based on thetotal amount sacrificed.
+   /// @dev claim HEX and HEXIT based on the total amount sacrificed.
```

[I-14] - Misleading internal method name in `_getHexPrice()`

ID	Classification	Category	Status
I-14	INFORMATIONAL	Quality Assurance	Fixed in 5964ad8

Description

The `_getHexPrice()` function within the `HexOneVault.sol` contract returns the quote (expected tokenOut for a given tokenIn) rather than the actual price.

```
/// @dev tries to consult the price of HEX in DAI (dollars).
/// @notice if consult reverts with PriceTooStale then it needs to
/// update the oracle and only then consult the price again.
function _getHexPrice(uint256 _amountIn) internal returns (uint256)
```

Occurrences

- HexOneVault.sol#L301-L321

Recommended Mitigation Steps

Change the function name to `_getHexQuote()`.