coverage

Hex One Protocol

Smart Contract Security Review

Prepared by: **Coverage Labs**
Date: **May 31th, 2024**
Visit: **coveragelabs.io**

Contents

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, rewards 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. Users may also be susceptible to liquidations if the underlying stake is mature or its collateral ratio falls below a certain threshold.

Executive Summary

Coverage reviewed Hex One Protocol smart contracts over the course of a 4 week engagement with three engineers. The review was conducted from 2024-05-06 to 2024-05-31.

People Involved

Name	Role	Contact
José Garção	Lead Security Researcher	garcao.random@gmail.com
Rafael Nicolau	Security Researcher	0xrafaelnicolau@gmail.com
nexusflip	Security Researcher	0xnexusflip@gmail.com

Application Summary

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

Code Versioning Control

- Review commit hash - 45514b8.
- Fix review commit hash - 38624eb.

Scope

The following smart contracts were within review scope:

- `src/HexitToken.sol`
- `src/HexOneBootstrap.sol`
- `src/HexOnePriceFeed.sol`
- `src/HexOnePoolManager.sol`
- `src/HexOnePool.sol`
- `src/HexOneToken.sol`
- `src/HexOneVault.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, Slitherin and Aderyn 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 identified invariants was completed we executed an extensive campaign 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 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 role-based access control mechanism with the following roles:

- `OWNER_ROLE`, granted to the deployer address by the `HexitToken.sol`, `HexOnePriceFeed.sol` and `HexOneBootstrap.sol` contracts.
- `MANAGER_ROLE`, granted to the `HexOnePoolManager.sol` contract address by the `HexitToken.sol` and `HexOnePool.sol` contracts.
- `MINTER_ROLE`, granted to the `HexOnePriceFeed.sol` and `HexOnePool.sol` contract addresses by the `HexitToken.sol` contract.
- `VAULT_ROLE`, granted to the `HexOneVault.sol` contract address by the `HexOneToken.sol` contract.
- `BOOTSTRAP_ROLE`, granted to the `HexOneBootstrap.sol` contract address by the `HexOneVault.sol` contract.

OWNER_ROLE

- Grant the `MANAGER_ROLE` to the `HexOnePoolManager.sol` contract by calling the `initManager()` method. This method can only be called once.
- Grant the `MINTER_ROLE` to the `HexOnePriceFeed.sol` contract by calling the `initFeed()` method. This method can only be called once.
- Grant the `MANAGER_ROLE` to the `HexOnePoolBootstrap.sol` contract by calling the `initBootstrap()` method. This method can only be called once.
- Add price feed quote paths by calling the `addPath()` method.
- Change the oracle update period by calling the `changePeriod()` method.
- Deploy a set of `HexOnePool.sol` contracts by calling the `createPools()` method.
- Deploy a single `HexOnePool.sol` contract by calling the `createPool()` method.
- Grant the `VAULT_ROLE` to the `HexOneVault.sol` contract by calling the `initVault()` method. This method can only be called once.
- Process the sacrifice by calling the `processSacrifice()` method. This method can only be called once.
- Starts the airdrop by calling the `startAirdrop()` method. This method can only be called once.

MANAGER_ROLE

- Grant the `MINTER_ROLE` to the `HexOnePool.sol` contract by calling the `initPool()` method.
- Set the pool `rewardPerToken` by calling the `initialize()` method.

MINTER_ROLE

- Mint `HEXIT` by calling the `mint()` method.

VAULT_ROLE

- Mint `HEX1` by calling the `mint()` method.
- Burn `HEX1` by calling the `burn()` method.

BOOTSTRAP_ROLE

- Enable `HEX1` buyback mechanism by calling the `enableBuyback()` method. This method can only be called once.

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 proper use of permissionless principles 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 and end-to-end tests, 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	Satisfactory. All arithmetic operations are properly implemented and do not introduce unexpected outcomes.
Centralization	Moderate. Some initialization functions introduce the risk of hotswapping contract implementations, as well as a setter function that can mutate the oracle's heartbeat, which may affect the permissionless nature of the protocol.
Code Stability	Moderate. One of the contracts in scope exceeded the maximum bytecode deployment size, which hindered the usage of some testing and analysis tools. (NOTE: The client provided a hotfix shortly after the review started)
Upgradability	Moderate. The lack of pair mutability for paths in the scope's price feed hinders adaptability to future conditions.
Front-Running	Moderate. It is possible to front-run the deposit buyback mechanism in the Vault contract.
Monitoring	Satisfactory. All contracts in scope have comprehensive event emission.
Specification	Satisfactory. The NatSpec and external documentation describe the protocol's functionality in a clear fashion.
Testing and Verification	Satisfactory. The provided testing suite covers a wide range of expected behaviours.

Stateful Fuzzing

`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`	~175MM	✅
If two users sacrificed the same amount in USD and have no `HEX` staked, the one who claimed the airdrop first should always receive more `HEXIT`	~175MM	✅
If two users have the same amount of `HEX` staked and did not participate in the sacrifice, the one who claimed the airdrop first should always receive more `HEXIT`	~175MM	✅
Sacrifices can only be made within the predefined timeframe	~175MM	✅
Sacrifice processing is only available after the sacrifice deadline has passed	~175MM	✅
The sacrifice can only be claimed within the claim period	~175MM	✅
The airdrop can only be claimed within the predefined airdrop timeframe	~175MM	✅

`HexOnePool.sol`

Property	Runs	Status
The total `HEXIT` rewards per second must always be equal to `rewardPerToken * totalStaked`	~175MM	✅
User can never unstake more than what he staked, excluding rewards	~175MM	✅
The total rewards to be distributed to Alice with N deposits of X value must always be equal to Bob with p * N deposits of X / p	~175MM	✅
The `totalStaked` amount must always equal the sum of each users' `stakeOf` amount	~175MM	✅

HexOneVault.sol

Property	Runs	Status
The sum of each HDT `stake.debt` must always be equal to the HEX1 total supply	~175MM	✅
If an HDT has `stake.debt == 0` it can not be taken	~175MM	✅
HDT can only be taken if at least 50% of the `stake.debt` is repaid and the `healthRatio` is less than `MIN_HEALTH_RATIO`	-	❌
Users must only be able to borrow more HEX1 with the same HEX collateral if the HEX price in USD increases	~175MM	✅
The number of accrued stake days + estimated stake days must be equal to 5555	~175MM	✅
If `buybackEnabled == true`, the depositing fee must always equal 1%	~175MM	✅
Withdraw must never be possible if HDT has not reached `stake.end`	~175MM	✅
Liquidation must never be possible if HDT has not reached `stake.end + GRACE_PERIOD`	~175MM	✅
Borrowing must never be possible if HDT has reached `stake.end`	~175MM	✅
Borrowing must never be possible if amount exceeds `maxBorrowable()`	~175MM	✅
Borrowing must never be possible if the resulting `healthRatio` is less than `MIN_HEALTH_RATIO`	~175MM	✅
Take must never be possible if HDT has reached `stake.end + GRACE_PERIOD`	~175MM	✅

Findings

ID	Title	Severity	Status
C-01	First feed update mints an unexpectedly large amount of `HEXIT`	CRITICAL	Fixed
L-01	Unsafe ERC20 operations should not be used	LOW	Fixed
L-02	Certain contract address initializations may enable malicious contract backdoors	LOW	Will not fix
L-03	Mutable oracle update period in `HexOnePriceFeed.sol` may allow manipulation of `HEX` quotes	LOW	Ack
L-04	Immutable quote paths may damage resilience of the price feed	LOW	Ack
L-05	Lack of liquidity weighted average on `_hxQuote()` may facilitate `HEX` quote manipulation	LOW	Will not fix
L-06	Debt title can not be taken if 100% of the debt is repaid	LOW	Fixed
L-07	Deposit buyback fee can be front-runned due to incorrect `amountOutMin`	LOW	Fixed
INFO-01	The `nonReentrant` modifier should precede all other modifiers	INFO	Fixed
INFO-02	Missing `address(0)` validation	INFO	Fixed
INFO-03	Write after write	INFO	Fixed
INFO-04	Unclear NatSpec in `HexOnePoolManager.sol`	INFO	Fixed
INFO-05	NatSpec typos in `HexOneBootstrap.sol`	INFO	Fixed
INFO-06	NatSpec typos in `HexOneVault.sol`	INFO	Fixed
INFO-07	`HexOneBootstrap.sol` exceeds the bytecode size limit	INFO	Fixed
INFO-08	Caching storage variables is more gas efficient than reading directly from storage	INFO	Fixed
INFO-09	Split revert statements to save gas costs	INFO	Ack
INFO-10	Bit shifting is cheaper than dividing by a power of two	INFO	Fixed

ID	Title	Severity	Status
INFO-11	Moving revert statement for gas cost savings	INFO	Fixed
INFO-12	Do-while loops are cheaper than for loops	INFO	Fixed
INFO-13	Calldata is cheaper than memory	INFO	Fixed
INFO-14	Use unchecked math to save gas costs	INFO	Fixed
INFO-15	Store `healthRatio(_id)` in a variable to save gas costs	INFO	Fixed
INFO-16	Storage pointers are more efficient than memory when appropriate	INFO	Ack
INFO-17	Storing `keccak256` as constant leads to inefficient extra hashing	INFO	Fixed
INFO-18	Pre-incrementation and decrementation operand saves gas	INFO	Ack
INFO-19	Caching array length prevents unnecessary memory reads	INFO	Fixed
INFO-20	Precision loss in `processSacrifice` could result in a loss of 1e-8 `HEX`	INFO	Fixed
INFO-21	Missing custom errors on accounting underflows	INFO	Fixed
INFO-22	Missing minimum amount requirement in Vault `deposit` function	INFO	Fixed

Severity	Count	Fixed	Acknowledged	Will not fix
CRITICAL	1	1	0	0
HIGH	0	0	0	0
MEDIUM	0	0	0	0
LOW	7	3	2	2
INFO	22	19	3	0
TOTAL	**30**	**23**	**5**	**2**

[C-01] - First feed update mints an unexpectedly large amount of HEXIT

ID	Classification	Category	Status
C-01	CRITICAL	Manual Review	Fixed in 845b8b2

Description

In the HexOnePriceFeed.sol contract, during the initial feed update, an unexpectedly large amount of HEXIT tokens are minted. This occurs due to a flaw in the logic that calculates the amount of HEXIT to mint based on the time elapsed since the last update, which is mistakenly considered as 0 before the first update.

Proof of Concept

The following proof of concept demonstrates the issue by observing the HEXIT tokens received after each feed update:

```
function test_poc() external {
    // first feed update
    uint256 balBefore = hexit.balanceOf(address(this));
    feed.update();
    uint256 balAfter = hexit.balanceOf(address(this));
    console.log("HEXIT received first update  : %e", balAfter - balBefore);

    vm.warp(block.timestamp + feed.period());

    // second feed update
    balBefore = hexit.balanceOf(address(this));
    feed.update();
    balAfter = hexit.balanceOf(address(this));
    console.log("HEXIT received second update : %e", balAfter - balBefore);

    vm.warp(block.timestamp + feed.period());

    // third feed update
    balBefore = hexit.balanceOf(address(this));
    feed.update();
    balAfter = hexit.balanceOf(address(this));
    console.log("HEXIT received third update  : %e", balAfter - balBefore);
}
```

```
HEXIT received first update : 1.713921825e27
HEXIT received second update: 3e20
HEXIT received third update : 3e20
```

Context

- HexOnePriceFeed.sol#L39
- HexOnePriceFeed.sol#L100-L114

Recommendation

Consider adding the following if statement to ensure that `HEXIT` is only minted after the first update occurs.

```
function update() external {
    uint256 timeElapsed = block.timestamp - lastUpdate;
    if (timeElapsed < period) revert PricesUpToDate();
    uint256 length = pairs.length();
    for (uint256 i; i < length; ++i) {
        _update(pairs.at(i));
    }

    lastUpdate = block.timestamp;

-   IHexitToken(hexit).mint(msg.sender, timeElapsed * MULTIPLIER);
+   if (lastUpdate != 0) {
+       IHexitToken(hexit).mint(msg.sender, timeElapsed * MULTIPLIER);
+   }

    emit PricesUpdated(block.timestamp);
}
```

[L-01] - Unsafe ERC20 operations should not be used

ID	Classification	Category	Status
L-01	LOW	Context & Cleanup	Fixed in 274ad4a

Description

Some ERC20 implementations may not behave as expected. It is recommended to use OpenZeppelin's SafeERC20 library to mitigate the issue.

Context

- HexOneBootstrap.sol#L293

Recommendation

Import OpenZeppelins' SafeERC20 library in the HexOneBootstrap.sol contract and modify the occurrence mentioned above as follows:

```
- IERC20(vault.hex1()).transfer(msg.sender, hex1Minted);
+ IERC20(vault.hex1()).safeTransfer(msg.sender, hex1Minted);
```

[L-02] - Certain contract address initializations may enable malicious contract backdoors

ID	Classification	Category	Status
L-02	LOW	Access Control	Will not fix

Description

The deployer address for the `HexOneBootstrap.sol` and the `HexitToken.sol` contracts has permissions to attribute an address routing to contracts that are compatible on an Interface level but do not necessarily mirror the same exact internal logic. This poses a centralization risk as the deployer might route said addresses to versions with malicious code, such as adding a snippet of code to any user-facing function that drains funds from the protocol into an address in their custody whenever it is called.

Context

- HexOneBootstrap.sol#L149-L155
- HexitToken.sol#L59-L67
- HexitToken.sol#L74-82

Recommendation

Consider using an initialization method that is not compatible with function changes inside the method definitions set in the contracts' Interfaces, such as a prior comparison to the bytecode of the aforementioned reference contracts in order to validate that the target address attribution is routed to an exact copy of said reference contract.

[L-03] - Mutable oracle update period in `HexOnePriceFeed.sol` may allow manipulation of `HEX` quotes

ID	Classification	Category	Status
L-03	LOW	Access Control	Acknowledged

Description

The deployer address for the `HexOnePriceFeed.sol` contract has permissions to change the price feed heartbeat within the pre-established minimum to maximum update periods defined in the contract with constants by using the `changePeriod()` function, allowing for unexpected manipulation of the `HEX` price feed output which is used to mint `HEX1` with an equivalent value to the underlying collateral deposited by a user when borrowing.

This might lead to said actor increasing the heartbeat time before the current price feed becomes stale if the `HEX` price goes down significantly, allowing for either the creation of a `HEX` stake or calling a borrow on an already existing stake that mints more `HEX1` than intended by the protocol due to the price feed output being considerably higher than the spot price, thus allowing for either the Owner or any given user to take an undercollateralized loan.

It is important to note that although any given user may take advantage of this scenario, it is the Owner that ultimately has the bigger advantage by being able to deterministically set the necessary conditions for the attack if the `HEX` price decreases.

Context

- HexOnePriceFeed.sol#L91-L95

Recommendation

Remove the `changePeriod()` function as well as the `MIN_PERIOD` and `MAX_PERIOD` constants and the `period` variable in favour of a single `PERIOD` constant that dictates the price feed update period.

[L-04] - Immutable quote paths may damage the resilience of the price feed

ID	Classification	Category	Status
L-04	LOW	Manual Review	Acknowledged

Description

The `addPath()` function within the `HexOnePriceFeed.sol` contract adds a path to the contract's storage so that a quote for a given pair derives from any given number of intermediate pairs between the input and output token in order to add robustness to the price quote. In the eventuality that any external events occur such that the given intermediate pairs lose a significant amount of liquidity there isn't a function to make said paths mutable, thus hurting the price quote's robustness.

Context

- HexOnePriceFeed.sol#L69-84

Recommendation

Consider adding a `changePath()` function that changes the specific pairs for a given path.

[L-05] - Lack of liquidity weighted average on `_hxQuote()` may facilitate `HEX` quote manipulation

ID	Classification	Category	Status
L-05	LOW	Manual Review	Will not fix

Description

The `_hxQuote()` function within the `HexOneVault.sol` calculates the current `HEX` quote based on an average of three different quotes. Given that this average is not weighted by the liquidity depth of each quote in the path, the `HEX` quote may be more easily manipulable given that an attacker can manipulate the price of the pool with the least liquidity for the given path as said pool has a 33% weighting rather than it being proportional to the total liquidity of the path.

Context

- HexOneVault.sol#L348-353

Recommendation

Consider replacing the current arithmetic logic for the average by a weighted average by liquidity. It is important to note that if the weighted liquidity reading is obtained through the `getReserves()` method within the PulseX libraries it might also be subject to manipulation given that this output is of a real-time measure. If the aforementioned risk outweighs the proposed improvements for this finding, consider implementing a time-weighted average liquidity reading for this case (following the TWAP logic currently in use by the contract scope and adapting it to the liquidity reserves).

[L-06] - Debt title can not be taken if 100% of the debt is repaid

ID	Classification	Category	Status
L-06	LOW	Stateful Fuzzing	Fixed in 8d62ee0

Description

When the health ratio of a `HDT` is calculated as 0, it indicates that the stake associated has no outstanding debt (i.e., `stake.debt == 0`). In such cases, if a user attemps to take another user's `HDT`, the transaction incorrectly reverts. This occurs because the current logic does not account for a scenario where the debt will be fully repaid before transferring the corresponding stake.

Context

- HexOneVault.sol#L294

Recommendation

Consider modifying the logic to handle the scenario mentioned above, thus allowing for the debt to be repaid in full:

```
- if (healthRatio(_id) < MIN_HEALTH_RATIO) revert HealthRatioTooLow();
+ uint256 newRatio = healthRatio(_id);
+ if (newRatio != 0 && newRatio < MIN_HEALTH_RATIO) revert HealthRatioTooLow();
```

[L-07] - Deposit buyback fee can be front-runned due to incorrect `amountOutMin`

ID	Classification	Category	Status
L-07	LOW	Manual Review	Fixed in 845b8b2

Description

The depositing buyback fee mechanism in the `HexOneVault.sol` contract can be front-runned due to `amountOutMin` being set to 0. If `amountOutMin` is not properly calculated, an attacker can front-run the fee swap, resulting in suboptimal execution of the buyback mechanism.

Context

- HexOneVault.sol#L316-317

Recommendation

Consider adding a new `amountOutMin` user input from the user. Alternatively, fetch the pool reserves and compute `amountOutMin` on-chain. Please note that the latter is still susceptible to front-running, but in a less severe way.

```
function _buyback(uint256 _fee) private {
    IERC20(HX).approve(ROUTER_V2, _fee);

    address[] memory path = new address[](4);
    path[0] = HX;
    path[1] = WPLS;
    path[2] = DAI;
    path[3] = hex1;

-    uint256[] memory amounts =
        IPulseXRouter(ROUTER_V2).swapExactTokensForTokens(_fee, 0, path,
            address(this), block.timestamp);

+   (uint256 reserveA, uint256 reserveB) = UniswapV2Library.getReserves(
    FACTORY_V2, DAI, hex1);

+   uint256 amountOutMin = UniswapV2Library.getAmountOut(_fee, reserveA,
    reserveB);

+   uint256[] memory amounts =
            IPulseXRouter(ROUTER_V2).swapExactTokensForTokens(_fee,
                _amountOutMin, path, address(this), block.timestamp);

    IHexOneToken(hex1).burn(address(this), amounts[amounts.length - 1]);
}
```

[INFO-01] - The `nonReentrant` modifier should precede all other modifiers

ID	Classification	Category	Status
INFO-01	INFORMATIONAL	Context & Cleanup	Fixed in 6456dbb

Description

The `nonReentrant` modifier should precede all other modifiers. This adheres to good Solidity practices, thus acting as a safeguard against potential reentrancy issues arising from other modifiers in a function.

Context

- HexOneBootstrap.sol#L304

Recommendation

Consider reordering the modifiers for the occurrence mentioned above in the following order:

```
- function startAirdrop(uint64 _airdropStart) external onlyRole(OWNER_ROLE)
    nonReentrant {
+ function startAirdrop(uint64 _airdropStart) external nonReentrant onlyRole(
    OWNER_ROLE) {
```

[INFO-02] - Missing `address(0)` validation

ID	Classification	Category	Status
INFO-02	INFORMATIONAL	Context & Cleanup	Fixed in 69f8718

Description

There is a missing `address(0)` check within the `HexOneBootstrap.sol` constructor.

Context

- HexOnePriceFeed.sol#L55-62

Recommendation

Consider adding the following snippet at the top level of the constructor:

```
+ if (_hexit == address(0)) revert ZeroAddress();
```

[INFO-03] - Write after write

ID	Classification	Category	Status
INFO-03	INFORMATIONAL	Context & Cleanup	Fixed in bf0ef5b

Description

In the `HexOneBootstrap.sol` contract, there is a redundant attribution of the `BASE_HEXIT` variable inside the `_baseDailyHexit()` internal function.

Context

- HexOneBootstrap.sol#L400-405

Recommendation

Remove the following lines (#L401-L404) from the `_baseDailyHexit()` internal function:

```
function _baseDailyHexit(uint256 _day) private pure returns (uint256 baseHexit)
    {
-   if (_day == 1) {
-       baseHexit = BASE_HEXIT;
-   }
-
    baseHexit = BASE_HEXIT;
    for (uint256 i = 2; i <= _day; ++i) {
        baseHexit = (baseHexit * DECREASE_FACTOR) / FIXED_POINT;
    }
}
```

[INFO-04] - Unclear NatSpec in `HexOnePoolManager.sol`

ID	Classification	Category	Status
INFO-04	INFORMATIONAL	Context & Cleanup	Fixed in c089ce2

Description

The `createPool()` function has an unclear NatSpec description.

Context

- HexOnePoolManager.sol#L58

Recommendation

Replace the following line (#L58) within the `createPool()` NatSpec with:

```
 /*
- *  @dev deploy pool a new pool.
+ *  @dev deploys a new pool.
```

[INFO-05] - NatSpec typos in `HexOneBootstrap.sol`

ID	Classification	Category	Status
INFO-05	INFORMATIONAL	Context & Cleanup	Fixed in 8c62ab3

Description

The NatSpec descriptions for the `claimSacrifice()`, `claimAirdrop()` and `_hexitSacrificeShares()` functions have typos.

Context

- HexOneBootstrap.sol#L255-L256
- HexOneBootstrap.sol#L323
- HexOneBootstrap.sol#L357

Recommendation

Replace the following lines (#L255-L256) within the `claimSacrifice()` NatSpec with:

```
 /*
- *  @dev function to claim rewards from the sacrifie period.
- *  @notice creates an hex stake, and max borrow against it.
+ *  @dev function to claim rewards from the sacrifice period.
+ *  @notice creates a hex stake, and max borrow against it.
```

Replace the following line (#L323) within the `claimAirdrop()` NatSpec with:

```
 /*
- *  @dev claim hexit the rewards from the airdrop period.
+ *  @dev claim hexit rewards from the airdrop period.
```

Replace the following line (#L357) within the `_hexitSacrificeShares()` NatSpec with:

```
 /*
- *  @dev returns the number os hexit shares a user receives during the
    sacrifice phase.
+ *  @dev returns the number of hexit shares a user receives during the
    sacrifice phase.
```

[INFO-06] - NatSpec typos in `HexOneVault.sol`

ID	Classification	Category	Status
INFO-06	INFORMATIONAL	Context & Cleanup	Fixed in 627f016

Description

The NatSpec descriptions for the `MIN_HEALTH_RATIO` constant, as well as the `withdraw()`, `borrow()` `take()` and `_hxQuote()` functions have typos.

Context

- HexOneVault.sol#L55
- HexOneVault.sol#L181
- HexOneVault.sol#L253
- HexOneVault.sol#L277
- HexOneVault.sol#L345

Recommendation

Replace the following line (#L55) within the `MIN_HEALTH_RATIO` NatSpec with:

```
- /// @dev minimum healh ratio, if health ratio is below 250% in bps stakes
    become liquidatable.
+ /// @dev minimum health ratio, if health ratio is below 250% in bps stakes
    become liquidatable.
```

Replace the following line (#L181) within the `withdraw()` NatSpec with:

```
 /*
- *  @dev function to end an hex stake if stake is mature.
+ *  @dev function to end a hex stake if stake is mature.
```

Replace the following line (#L253) within the `borrow()` NatSpec with:

```
 /*
- *  @dev function to borrow agaisnt an hex stake.
+ *  @dev function to borrow against a hex stake.
```

Replace the following line (#L277) within the `take()` NatSpec with:

```
 /*
- *  @dev transfers stake ownership to the msg sender if stake healh ratio is
    below the minimium.
+ *  @dev transfers stake ownership to the msg sender if stake health ratio is
    below the minimum.
```

Replace the following line (#L345) within the `_hxQuote()` NatSpec with:

```
 /*
- *  @dev returns an hex quote in usd based on average price of three pairs.
+ *  @dev returns a hex quote in usd based on average price of three pairs.
```

[INFO-07] - `HexOneBootstrap.sol` exceeds the bytecode size limit

ID	Classification	Category	Status
INFO-07	INFORMATIONAL	Context & Cleanup	Fixed in 274ad4a

Description

The bytecode size of the `HexOneBootstrap.sol` contract exceeds the 24 kilobyte limit imposed by Pulsechain and most EVM chains. This issue was identified using the following command:

```
$ forge build --sizes
```

The reported bytecode size is 30,679 bytes, exceeding the limit by 6,103 bytes.

```
| Contract                | Size (B) | Margin (B) |
|-------------------------|----------|------------|
| HexOneBootstrap         |   30,679 |     -6,103 |
| HexOnePool              |    3,352 |     21,224 |
| HexOnePoolManager       |    7,473 |     17,103 |
| HexOnePriceFeed         |    7,190 |     17,386 |
| HexOneToken             |    3,045 |     21,531 |
| HexOneVault             |   14,792 |      9,784 |
| HexitToken              |    4,012 |     20,564 |
```

Since the bytecode size of `HexOneBootstrap.sol` exceeds the limit, any attempt to deploy this contract would inevitably fail.

Context

- HexOneBootstrap.sol

Recommendation

Consider deploying `HexOneVault.sol` separately and supplying its address as an initialization argument when calling the `HexOneBootstrap.sol` constructor.

[INFO-08] - Caching storage variables is more gas efficient than reading directly from storage

ID	Classification	Category	Status
INFO-08	INFORMATIONAL	Gas Optimizations	Fixed in 5e985ae

Description

The `claimSacrifice()` function within the `HexOneBootstrap.sol` contract reads the `user.sacrificedUsd` variable directly from storage more than once. This increases gas costs significantly in comparison to caching said variables when accessing storage due to storage access being the most expensive option for variable access by a considerable margin, thus caching allows for repeated usage of the same variable while only doing one storage read.

Context

- HexOneBootstrap.sol#L269-281

Recommendation

Replace all `user.sacrificeUsd` references with a local variable initialized with `user.sacrificedUsd` as a value:

```
+ uint256 sacrificedUsd = user.sacrificedUsd
- if (user.sacrificedUsd == 0) revert DidNotParticipateInSacrifice();
+ if (sacrificedUsd == 0) revert DidNotParticipateInSacrifice();
(...)
- uint256 shares = (user.sacrificedUsd * 1e18) / info.sacrificedUsd;
+ uint256 shares = (sacrificedUsd * 1e18) / info.sacrificedUsd;
```

[INFO-09] - Split revert statements to save gas costs

ID	Classification	Category	Status
INFO-09	INFORMATIONAL	Gas Optimizations	Acknowledged

Description

Revert statements which have logic conditions combined with boolean operators are more gas expensive than split statements because the latter does not have extra boolean operators. Additionally, splitting the statements allows for more granularity regarding error messages.

Context

- HexOneBootstrap.sol#L121-L123
- HexOneBootstrap.sol#L133-L135
- HexOneBootstrap.sol#L153-L155
- HexOneBootstrap.sol#L327-L329
- HexOnePriceFeed.sol#L56
- HexOnePriceFeed.sol#L92
- HexOneVault.sol#L285-L287

Recommendation

Consider splitting the revert statements as per the pseudocode below:

```
if (condition1) {
    revert Error1();
}
if (condition2) {
    revert Error2();
}
```

[INFO-10] - Bit shifting is cheaper than dividing by a power of two

ID	Classification	Category	Status
INFO-10	INFORMATIONAL	Gas Optimizations	Fixed in 4d10456

Description

When the denominator of a division is a power of two, it is cheaper to use bit shifting instead.

Context

- HexOneBootstrap.sol#L222
- HexOneVault.sol#L289

Recommendation

Consider using bit shifting for the occurrence mentioned above as follows:

```
- uint256 halfHxAmount = hxAmount / 2;
+ uint256 halfHxAmount = hxAmount >> 1;
```

[INFO-11] - Moving revert statement for gas cost savings

ID	Classification	Category	Status
INFO-11	INFORMATIONAL	Gas Optimizations	Fixed in 6132a80

Description

There is no apparent reason not to move this revert statement to the top of the function. In the current scenario, if the transaction reverts at that point, some business logic has already been executed, resulting in unnecessary gas costs.

Context

- HexOneBootstrap.sol#L176

Recommendation

Consider moving the revert statement on the above mentioned occurrence to the top of the function.

[INFO-12] - Do-while loops are cheaper than for loops

ID	Classification	Category	Status
INFO-12	INFORMATIONAL	Gas Optimizations	Fixed in cf187d4

Description

In certain sections of the code, `for` loops are used where there is a probability of a relatively high number of iterations. In such cases, do-while loops are preferred because they are more gas efficient.

Context

- HexOneBootstrap.sol#L390-L393
- HexOneVault.sol#L392-L395

Recommendation

Consider changing the for loops mentioned in the context section to do-while loops compliant with the pseudocode below:

```
uint256 i;
do {
    unchecked {
        ++i;
    }
} while (i < times);
```

[INFO-13] - Calldata is cheaper than memory

ID	Classification	Category	Status
INFO-13	INFORMATIONAL	Gas Optimizations	Fixed in c181fcb

Description

Loading function inputs or data directly from calldata is more cost-effective than retrieving from memory. This is due to the fewer operations and lower gas costs associated with accessing calldata.

Context

- HexOnePoolManager.sol#L42
- HexOnePriceFeed.sol#L69

Recommendation

Consider using calldata over memory if the data does not need to be modified in the function.

[INFO-14] - Use unchecked math to save gas costs

ID	Classification	Category	Status
INFO-14	INFORMATIONAL	Gas Optimizations	Fixed in aaee34f

Description

There are no possibilities of an underflow in this subtraction given the `hxQuote > stake.debt` check which is done beforehand. As such, using unchecked math for this given case will be more gas efficient.

Context

- HexOneVault.sol#L128-L130

Recommendation

Consider doing the aforementioned subtraction inside an unchecked math scope as follows:

```
if (hxQuote > stake.debt) {
-     amount = hxQuote - stake.debt;
+     unchecked {
+         amount = hxQuote - stake.debt;
+     }
}
```

[INFO-15] - Store `healthRatio(_id)` in a variable to save gas costs

ID	Classification	Category	Status
INFO-15	INFORMATIONAL	Gas Optimizations	Fixed in 8658f36

Description

The `healthRatio` function invokes the `_accrued` internal function, which executes a `for` loop that may involve a relatively high number of iterations. Invoking this function twice within the if statement incurs unnecessary gas costs.

Context

- HexOneVault.sol#L128-L130

Recommendation

Consider caching the output from the `healthRatio` function in a variable as follows:

```
- if (healthRatio(_id) == 0 || healthRatio(_id) >= MIN_HEALTH_RATIO) {
+ uint256 healthRatio = healthRatio(_id);
+ if (healthRatio == 0 || healthRatio >= MIN_HEALTH_RATIO) {
```

[INFO-16] - Storage pointers are more efficient than memory when appropriate

ID	Classification	Category	Status
INFO-16	INFORMATIONAL	Gas Optimizations	Acknowledged

Description

There are several instances in the contract scope where access to structs inside functions is being initialized in memory. Although base gas costs for read and write opcodes are significantly cheaper in memory than in storage, it is worth noting that structs default to storage, which means local memory initialization requires a cold storage load call for each member plus a memory read/write call for each following use. With storage, there is no local initialization cost as there are no prior storage load calls before calling a member, so even though read/write operations are extremely expensive in comparison to memory, storage is ultimately more efficient for cases where a function will not need to access every member of a given struct.

Context

- HexOneBootstrap.sol#L132
- HexOneBootstrap.sol#L391
- HexOneVault.sol#L106
- HexOneVault.sol#L125
- HexOneVault.sol#L163
- HexOneVault.sol#L215
- HexOneVault.sol#L262
- HexOneVault.sol#L386
- HexOneVault.sol#L407

Recommendation

Consider replacing all local struct memory initialization cases referenced above with storage as follows:

```
- Struct memory members = output[[...]]
+ Struct storage members = output[[...]]
```

It is worth noting that it is required to use locally cached variables to ensure the total sum of storage access gas costs is cheaper than its memory counterpart for cases in which a given member is directly accessed through the parent struct more than once in a function:

```
- Struct memory members = output[[...]]
+ Struct storage members = output[[...]]

- members.count = members.count + 1
+ uint256 count = members.count
+ count = count + 1
```

[INFO-17] - Storing `keccak256` as constant leads to inefficient extra hashing

ID	Classification	Category	Status
INFO-17	INFORMATIONAL	Gas Optimizations	Fixed in c763a37

Description

The use of `keccak256` as a constant requires the hashing to be performed each time said constant is used on runtime, which adds unnecessary gas fees as the hashing is deterministic. In comparison, storing it as an immutable is more efficient as the hashing is only done once on deployment. Another efficient approach is to compute the `keccak256` result off-chain and store the resulting hash as a constant. This hash can be computed using Chisel Solidity REPL:

```
$ chisel
keccak256("OWNER_ROLE")
Type: bytes32
- Data: 0xb19546dff01e856fb3f010c267a7b1c60363cf8a4664e21cc89c26224620214e
```

Context

- HexOneBootstrap.sol#L28
- HexOnePool.sol#L19
- HexOnePoolManager.sol#L17
- HexOnePriceFeed.sol#L23
- HexOneToken.sol#L15
- HexOneVault.sol#L29
- HexitToken.sol#L15
- HexitToken.sol#L17
- HexitToken.sol#L19

Recommendation

Change the constant to an immutable and change the constructor so that it can take an initialization input for the immutable `keccak256` on deployment as follows:

```
- bytes32 public constant OWNER_ROLE = keccak256("OWNER_ROLE");
+ bytes32 public immutable OWNER_ROLE = keccak256("OWNER_ROLE");
```

Alternatively, for additional gas cost savings and/or preserving the constructor structure of the contract, calculate the `keccak256` result for each occurence off-chain and manually store it as a `bytes32` constant:

```
- bytes32 public constant OWNER_ROLE = keccak256("OWNER_ROLE");
+ bytes32 public constant OWNER_ROLE = "0
    xb19546dff01e856fb3f010c267a7b1c60363cf8a4664e21cc89c26224620214e";
```

[INFO-18] - Pre-incrementation and decrementation operand saves gas

ID	Classification	Category	Status
INFO-18	INFORMATIONAL	Gas Optimizations	Acknowledged

Description

There are several instances within the contract scope where incrementing and decrementing integers is done by either the post-incrementing (`i++`) and decrementing (`i--`) operands, or simple addition and subtraction. It is more gas efficient to use pre-incrementing (`--i`) and post-incrementing (`++i`) operands for these purposes. It is important to note that this optimization is not valid for cases where the iterator variable will be accessed inside the loop as it will increment the variable itself, thus breaking the loops' intended behaviour.

Context

- HexOnePriceFeed.sol#L74
- HexOnePriceFeed.sol#L77
- HexOnePriceFeed.sol#L130
- HexOneVault.sol#L319
- HexOneVault.sol#L416

Recommendation

Change the above mentioned occurrences to the pre-incrementing operand for addition and to the pre-decrementing operand for subtraction as follows:

```
// addition
- var + 1;
+ ++var;

// subtraction
- var - 1;
+ --var;

// incrementing
- var++;
+ ++var;

// decrementing
- var--;
+ --var;
```

[INFO-19] - Caching array length prevents unnecessary memory reads

ID	Classification	Category	Status
INFO-19	INFORMATIONAL	Gas Optimizations	Fixed in f7ab4d7

Description

There are several instances within the contract scope where the length parameter of an array that is stored in memory is directly read in a for loop or is read directly from memory more than once in a given function. To avoid redundant memory reads and resulting gas costs every time the loop iterates it is recommended to cache it locally instead.

Context

- HexOnePriceManager.sol#L43-L47
- HexOnePriceFeed.sol#L77
- HexOnePriceFeed.sol#L124
- HexOnePriceFeed.sol#L130

Recommendation

Change the above mentioned occurrences so that any direct memory calls to length of arrays are using a locally cached variable as follows:

```
- for (uint256 i; i < _array.length; ++i);
+ uint256 length = _array.length;
+ for (uint256 i; i < length; ++i);
```

[INFO-20] - Precision loss in `processSacrifice` could result in a loss of 1e-8 `HEX`

ID	Classification	Category	Status
INFO-20	INFORMATIONAL	Manual Review	Fixed in 38624eb

Description

At the first line of the code below, `hxAmount` is divided by two and the result is stored in the `halfHxAmount` variable. This variable is then used as a function input to swap `HEX` for `DAI` in PulseX and to deposit `HEX` into the vault. If `hxAmount` is an odd number, there will be a precision loss resulting in a loss of 1e-8 `HEX`.

Context

- HexOneBootstrap.sol#L222-L249

Recommendation

Consider adding a `secondHalfHxAmount` variable to store the remaining half as follows:

```
+ uint256 secondHalfHxAmount = hxAmount - halfHxAmount;
```

Subsequently, utilize the `halfHxAmount` variable for executing the PulseX swap and the `secondHalfHxAmount` variable for depositing into the vault, or vice versa.

[INFO-21] - Missing custom errors on accounting underflows

ID	Classification	Category	Status
INFO-21	INFORMATIONAL	Manual Review	Fixed in 4dc619f

Description

There are three cases within the contract scope where the internal user accounting might underflow:

- Inputting a greater amount than whatever's staked by the user when calling the `unstake()` function within the `HexOnePool.sol` contract
- Inputting a greater amount than necessary to fully pay the stake's debt when calling the `take()` function within the `HexOneVault.sol` contract
- Inputting a greater amount than necessary to fully pay the stake's debt when calling the `repay()` function within the `HexOneVault.sol` contract

Although the intrinsic properties of Solidity 0.8.0+ protect against such cases by reverting with an arithmetic error, we believe that defining custom errors is more fitting of good Solidity practices.

Context

- HexOnePool.sol#L95-L96
- HexOneVault.sol#L245
- HexOneVault.sol#L292

Recommendation

Consider adding the following custom error to the respective contracts' interfaces:

```
+ error AmountExceeds();
```

Afterwards, define the following condition before accounting in the aformentioned cases:

```
+ if (_amount > USER_ACCOUNTING_STATE) revert AmountExceeds();
```

[INFO-22] - No proper handling of minimum amount in Vault `deposit` function

ID	Classification	Category	Status
INFO-22	INFORMATIONAL	Manual Review	Fixed in a92f08a

Description

Having such a small minimum deposit amount in the Vaults' `deposit` function leads to two main issues. First, if a user deposits fewer than 100 tokens, the 1% fee whenever `buybackEnabled = true` will not be deduced as it is rounded down to zero. Additionally, there is no custom error handling for cases where the external PulseX call reverts due to the small deposit amount, thus worsening the user experience (UX) of the protocol.

Context

- HexOneVault.sol#L147

Recommendation

Consider adding a minimum deposit requirement, such as 1 `HEX`:

```
+ if (_amount < 1e8) revert InvalidAmount();
```