SMART CONTRACTS REVIEW

zokyo

February 26th 2025 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING HAVEN1

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 3 High issues: 2 resolved and 1 acknowledged = - 3 points deducted
- 3 Medium issues: 1 resolved and 2 acknowledged = - 4 points deducted
- 7 Low issues: 6 resolved and 1 acknowledged = - 1 point deducted
- 9 Informational issues: 5 resolved and 4 acknowledged = 0 points deducted

Thus, 100 - 3 - 4 - 1 = 92

TECHNICAL SUMMARY

This document outlines the overall security of the Haven1 smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Haven1 smart contract/s codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Haven1 team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The Smart contract's source code was taken from the Haven1 repository.
Repo: https://github.com/haven1network/solidity-core
Last commit - 9bd5c1a
Branch - https://github.com/haven1network/solidity-core/tree/HVN-0000-audit-feedback

Contracts under the scope:

- contracts/airdrop/AirdropClaim.sol
- contracts/bridge/bridge-controller/BridgeController.sol:
- contracts/bridge/bridge-relayer/BridgeRelayer.sol
- contracts/bridge/locked-h1/LockedH1.sol
- contracts/externalChain/eth-mainnet/H1.sol
- contracts/fee/FeeContract.sol
- contracts/fee/channels/fee-distributor-channels/FeeDistributorChannelWH1.sol
- contracts/fee/channels/staking/StakingChannelESH1.sol
- contracts/fee/channels/validator-rewards/ValidatorRewardsBase.sol
- contracts/fee/channels/validator-rewards/ValidatorRewardsConfig.sol
- contracts/fee/channels/validator-rewards/ValidatorRewardsESH1.sol
- contracts/fee/channels/validator-rewards/ValidatorRewardsH1.sol
- contracts/staking/SimpleStaking.sol
- contracts/tokens/HRC20.sol
- contracts/utils/OnChainRouting.sol
- contracts/utils/upgradeable/RecoverableUpgradeable.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Haven1 smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01 Due diligence in assessing the overall code quality of the codebase.

02 Cross-comparison with other, similar smart contract/s by industry leaders.

03 Testing contract/s logic against common and uncommon attack vectors.

Executive Summary

The Haven1 protocol strives to lead the charge on becoming a rekt-free protocol for the web3 ecosystem. In the current landscape of blockchain security which is significantly complicated and continuously changing, Haven1 aims to address the issues revolving around the lack of clarity and consensus on the best approach for securing networks.To address this, Haven1 has built an EVM compatible blockchain that extends GoQuorum. The foundation of this area of the protocol revolves around the bridge contracts which allows users to transfer their funds from Haven1 to external blockchains and vice versa. In addition to this, airdrops through the AirdropClaim contract, SimpleStaking allowing users to earn rewards, validator rewards and vesting are all options for users which will allow the ecosystem to thrive.

Users can transfer their tokens between Haven1 and other blockchains through the BridgeRelayer and the BridgeController contracts. The BridgeRelayer contract is responsible for acting as a task queue for cross-chain transactions. Each supported blockchain manages a separate queue and has functionalities to allow for queuing and dequeuing tasks. On the other hand, the BridgeController contract facilitates cross-chain transfers for the Haven1 network.

In addition to cross-chain transfers, users have the ability to collect airdrop rewards through the AirdropClaim contract. The contract will manage the distribution of native token airdrops for the Haven1 community in the most transparent and fair way possible. Allocations for airdrops to the users are based on earned LP and XP points which are collected through participation in the Haven1 testnet. The contract supports for allocating, claiming, cancelling and handling unclaimed or discarded tokens.

As a cornerstone of DeFI, staking is also available via the SimpleStaking contract which allows users to stake a specific HRC20 token in order to earn rewards. Users stake their tokens through the stake function where the admin will notify the reward amount beforehand. As time moves forward, as long as the user is staked they will be eligible for the proportionate amount of rewards depending on how many tokens they've staked and duration.

Alongside vesting, validator rewards and fee distribution throughout this section of the ecosystem, Zokyo was responsible for conducting the security audit for this portion of Haven1. Overall the code is well thought out, well commented for the user's information on the goals of the contract in question and rigorously engineered. The issues discovered ranged from critical down to informational findings mostly revolving around the handling of fee distribution between channels, business logic errors, front/back running and gas optimisations/best engineering practices.

That being said, whilst the code is quite centralised as one of the Haven1 team's intentional designs - it does little to compromise security. However, while there wasn't a strictly defined security impact within the contracts looked at around the applicationFee modifier within the H1NativeApplicationUpgradeable, when the refundRemainingBalance is set to true, it did create some cause for concern for the security team. It's highly recommended that the Haven1 team carefully reconsiders this logic to refund the entire balance of native tokens to the user as this could impact developer contracts inheriting this dependency who have not given thought to its logic.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Haven1 team and the Haven1 team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Fee Distribution In The FeeContract May Be Unfair Due To The Lack Of Consideration Of Newly Added Channels And Channels Which Have Been Removed	High	Resolved
2	Oracle refreshment can fail without notifying, leading to the use of a stale price affecting the fees	High	Resolved
3	The amount of funds designated as 'gas' will always be transferred to the association	High	Acknowledged
4	Pool manipulation can lead to protocol loosing funds when swapping gas amounts	Medium	Acknowledged
5	The `_discardedAirdrop` variable in `AirdropClaim.sol` is wrongly updated, possibly leading to an incorrect track of discarded tokens	Medium	Resolved
6	FeeContract Operators Have the Ability To Use Fees To Grief Users	Medium	Acknowledged
7	RateLimiting In The BridgeRelayer Can Be Bypassed By Using An Alternative Contract Address	Low	Resolved
8	Unchecked external call to ValidatorConfig contract	Low	Resolved
9	Missing Events after altering important parameters	Low	Resolved
10	Usage Of tx.origin Can Be Used To Bypass Fees By Phishing Attempts	Low	Acknowledged
11	Missing sanity check	Low	Resolved
12	Starting and ending time for airdrop can be set the same	Low	Resolved

#	Title	Risk	Status
13	The setExemptContract() function does not check whether an address is a contract or not	Low	Resolved
14	Use of assert statements instead of require statements	Informational	Resolved
15	Ambiguous failure handling in add functions	Informational	Acknowledged
16	EOAs Can Become A Grace Contract In The FeeContract	Informational	Acknowledged
17	Certain actions are centralized	Informational	Acknowledged
18	The H1 Token's Total Supply Is Minted Directly To The Deployer - Distribution May Cause Instability In Token Price	Informational	Resolved
19	Cancelling An Airdrop Will Render The Contract Useless	Informational	Acknowledged
20	Multiple Instances Of Operating Directly On Storage Variables May Cause Increased Gas Intensiveness	Informational	Resolved
21	`depositAirdrop` can be called 1 block before airdrop ending	Informational	Resolved
22	Floating Solidity Version in pragma Statement	Informational	Resolved

Fee Distribution In The FeeContract May Be Unfair Due To The Lack Of Consideration Of Newly Added Channels And Channels Which Have Been Removed

Description

The FeeContract will distribute its fees to channels which can be added by an address who possesses the operator role. These fees are distributed by multiplying the amount by the channel weights and dividing by the amount of contract shares; however, new channels which are added before distribution will force existing channels to give up a share of their fees to new channels.

Impact

Existing channels will receive an unfair amount of fees because part of their share will be given up to newer channels for the distribution period. In addition to this, channels which have been removed will also miss out on their fees earned during the current distribution period. This was marked as High in severity because this issue results in the loss of earnings or disproportionately paid fees for channels.

Recommendation:

It's recommended that the current implementation of adding and removing channels is refactored to distribute fees before a channel is added and before a channel is removed. This will ensure segregation between preexisting channels and channels which are new to the contract or are being deleted which will ensure that fees are distributed fairly and evenly.

Oracle refreshment can fail without notifying, leading to the use of a stale price affecting the fees.

Description

The `updateFee()` function within the `FeeContract.sol` smart contract is used to update the H1 application fee, the H1 USD price, and associated values:

```
function updateFee() external {
        if (block.timestamp <= _networkFeeResetTimestamp) return;


        _refreshOracle();
        uint256 oracleVal = queryOracle();
        _feePrior = _fee;
        _fee = (oracleVal * _feeUSD) / _SCALE;


        _h1USDPrev = _h1USD;
        _h1USD = oracleVal;


        _networkFeeResetTimestamp = _feeUpdateEpoch + block.timestamp;
        _networkFeeGraceTimestamp = _gracePeriod + block.timestamp;


        emit FeeUpdated(_fee);
    }
```

The key part of the issue is the call to `_refreshOracle()` which is used to refresh the oracle, as its name implies:

```
/**
     * @notice Refreshes the oracle.
     *
```

```
     * @return True if the refresh was successful, false otherwise.
     */
    function _refreshOracle() private returns (bool) {
        return IFeeOracle(_oracle).refreshOracle();
    }
```

However, it can be observed that the refreshment of the oracle can silently fail while returning a boolean (false) variable.

The issue is related to the fact that the `updateFee()` function is not receiving and checking the boolean variable returned by `_refreshOracle()`, which indicates whether the oracle has been correctly updated or not.

Consider the following scenario:

1. The oracle returns X.
2. `networkFeeResetTimestamp` has passed so `updateFee()` can be executed again.
3. The oracle should now return an updated price Y but for any reason the call fails but the transaction does not revert.
4. The value used will be X while it should be Y.
5. The fees are incorrectly calculated, leading to the protocol earning more or less fees than the correct amount.

Impact:
If the oracle silently fails, an incorrect amount of fee will be applied, the amount of received funds will be lower than the correct one.

Recommendation:
There are several possibilities to address this issue. The principal point is handling the case where the `_refreshOracle()` call returns 'false'. Consider the following possibilities/ suggestions:

1. Revert the tx.
2. Use a temporary fee value if the call fails.
3. Use a second oracle if the first one fails.

These are just examples/suggestions. The decision regarding the handling implementation is more of a business decision that should be made by the team.

The amount of funds designated as 'gas' will always be transferred to the association.

Description

The `startWithdrawal` function within the `BridgeController.sol' smart contract, which is used to start a request to withdraw assets from Haven1, reserves a portion of the amount that is wanted to be withdrawn to cover gas fees. The expected gas amount is retrieved from a mapping which can be modified by the action of an admin:

```
// Ensure that the withdrawal amount can cover the gas fee.
        uint256 gasAmt = _gas[chainID_][hrc20_];
        if (amt_ < gasAmt) {
            revert BridgeController__InsufficientGas(
                chainID_,
                hrc20_,
                amt_,
                gasAmt
            );
        }
```

This means that the amount returned by the mapping will not represent the actual amount of consumed gas with a 100% accuracy, which can lead to a scenario where these amounts of gas are always sent to the association when the excess of gas should be returned to the user instead.

Impact:
If the described scenario takes place, an amount of funds, which were expected to be used to cover gas, is going to be unfairly sent to the association.

Recommendation:
We recommend that the contract emit an event capturing the gas usage details for each user transaction. This event should include information on the gas spent by the user, enabling external applications—particularly on cross-chain platforms—to determine the amount of unspent gas that can be claimed. By doing so, the other chain can record and verify the claimable amount based on actual gas consumption. This enhancement would provide greater transparency and facilitate a seamless user experience when claiming unspent gas, ultimately improving interoperability between chains.

Developer comments:
The gas amount retrieved from the mapping ensures that withdrawals can be processed smoothly. While it may not always match the exact gas consumed with 100% precision, any excess serves as a necessary buffer to account for potential fluctuations in operational costs associated with maintaining the service. This ensures that the bridge remains functional and efficient without introducing unnecessary complexity or refund mechanisms that could create additional vulnerabilities or overhead. Importantly, this gas amount is not static—it is updated frequently to reflect current network conditions as accurately as possible.

The withdrawal process on the user interface is fully transparent. Before a user initiates a withdrawal, they are presented with the estimated gas cost. They have the choice to accept the transaction or decide not to proceed. Since users are making an informed decision, there is no hidden or unfair loss of funds. The design priorities clarity and user discretion rather than attempting to refund minor excess amounts, which would introduce additional complexity and potential edge cases.

Pool manipulation can lead to protocol loosing funds when swapping gas amounts

Description

The `getRouteAndSwap()` function within the `OnChainRouting.sol` smart contract is used by users to input the `tokenIn`, `tokenOut`, `amountIn` and `recipient` and perform a swap:

```
function getRouteAndSwap(
        address tokenIn,
        address tokenOut,
        uint256 amountIn,
        address recipient
    ) external payable returns (uint256 amountOut) {
        (uint256 amountOutExpected, bytes memory path) = _getRoute(
            tokenIn,
            tokenOut,
            amountIn
        );
        if (amountOutExpected == 0) {
            return 0;
        }
        return _executeSwap(path, amountIn, amountOutExpected,
recipient);
    }
```

The function is responsible for calculating the path to be followed to execute the transaction and the expected amount, which is later used as `amountOutMin` when performing the swap. This process is highly risky due to the significant possibility of manipulation.

The path, `amountOutMin`, and deadline MUST always be set as input parameters, and therefore should be calculated off-chain. Relying on an on-chain calculation of these parameters will almost certainly lead to manipulation. When a transaction is pending confirmation, it is visible in the mempool to anyone. At this point, attackers can front-run the transaction by performing swaps just before it is executed. This manipulation can result in the selection of an unfavorable path or the calculation of a very low `minAmountIn`,

potentially even zero, causing the protocol to receive less funds than expected.

This also affects the `_swapGasTokens()` and `startWithdrawal()` functions within the `BridgeController.sol' smart contract as these swaps will get manipulated and funds will get lost.

Apart from has been mentioned, there is not enough slippage protection:

The `executeSwap()` function within the `OnChainRouting.sol` smart contract is the one used for swapping tokens. This function receives an `amountOutMinimum` parameter, which is used when calling `_executeSwap()`:

```
function _executeSwap(
        bytes memory path,
        uint256 amountIn,
        uint256 amountOutMinimum,
        address recipient
    ) internal returns (uint256 amountOut) {
        address tokenIn;
        assembly {
            tokenIn := div(
                mload(add(add(path, 0x20), 0)),
                0x100000000000000000000000
            )
        }
        IERC20(tokenIn).transferFrom(msg.sender, address(this), amountIn);
        IERC20(tokenIn).approve(address(swapRouter), amountIn);
        ISwapRouter.ExactInputParams memory params = ISwapRouter
            .ExactInputParams({
                path: path,
                recipient: recipient,
                deadline: block.timestamp,
                amountIn: amountIn,
                amountOutMinimum: amountOutMinimum
            });
        amountOut = swapRouter.exactInput{ value: msg.value }(params);
        return amountOut;
    }
```

The function is responsible for calculating the path to be followed to execute the transaction and the expected amount, which is later used as `amountOutMin` when performing the swap. This process is highly risky due to the significant possibility of manipulation.

The path, `amountOutMin`, and deadline MUST always be set as input parameters, and therefore should be calculated off-chain. Relying on an on-chain calculation of these parameters will almost certainly lead to manipulation. When a transaction is pending confirmation, it is visible in the mempool to anyone. At this point, attackers can front-run the transaction by performing swaps just before it is executed. This manipulation can result in the selection of an unfavorable path or the calculation of a very low `minAmountIn`, potentially even zero, causing the user to lose all their funds.

Another point to add to this issue is that the function only supports 2 hops paths, while in reality some swaps could be larger than 2 hops in order to be performed correctly.

This also affects the `_swapGasTokens()` and `startWithdrawal()` functions within the `BridgeController.sol' smart contract as these swaps will get manipulated and funds will get lost.

Apart from has been mentioned, there is not enough slippage protection:

The `executeSwap()` function within the `OnChainRouting.sol` smart contract is the one used for swapping tokens. This function receives an `amountOutMinimum` parameter, which is used when calling `_executeSwap()`:

```
function _executeSwap(
        bytes memory path,
        uint256 amountIn,
        uint256 amountOutMinimum,
        address recipient
    ) internal returns (uint256 amountOut) {
        address tokenIn;
        assembly {
            tokenIn := div(
                mload(add(add(path, 0x20), 0)),
                0x100000000000000000000000
            )
        }
        IERC20(tokenIn).transferFrom(msg.sender, address(this), amountIn);
        IERC20(tokenIn).approve(address(swapRouter), amountIn);
        ISwapRouter.ExactInputParams memory params = ISwapRouter
            .ExactInputParams({
                path: path,
                recipient: recipient,
                deadline: block.timestamp,
                amountIn: amountIn,
                amountOutMinimum: amountOutMinimum
            });
        amountOut = swapRouter.exactInput{ value: msg.value }(params);
        return amountOut;
    }
```

Even though the `amountOutMinimum` is a recommended safety parameter for protecting against slippage, it is not enough. It can be observed that the `deadline` parameter for performing swaps has been hardcoded to `block.timestamp`. This can lead to serious problems related to front-running attacks and slippage.

Consider the following example:

1. The expected swap output is X at this specific moment.
2. amountOutMinimum has been set to X - Y.
3. An attacker front-runs the transaction to not allow it being included in the current block and as consequence it is executed some blocks ahead.

As a result, the transaction is executed, resulting in X - Y amount out. The transaction could also be executed at step 1 when the output would have been X, therefore the user has lost Y amount of tokens.

Impact:
If the transaction is frontrunner, it could be manipulated, the swap will result in receiving fewer funds and therefore, funds are lost.

Recommendation:
There are several things to fix:

1. The `_getRoute()` and `getRoute()` functions should not be used as it has been mentioned, these calculations should be performed off chain. This also affects to the used functions within the `_getRoute()` scope, `_checkDifferentFeeTiersSingleSwap()` and `_checkDifferentFeeTiersPathSwap()`, `_quoteSingleSwap()`.

1. The `getRouteAndSwap()` function must not rely on `_getRoute` to calculate the `amountOutExpected` and `path`, but instead, receive these parameters as function input parameters, with them being calculated off-chain to avoid manipulation.

For adding a correct slippage protection, instead of setting:

```
deadline: block.timestamp
```

Set `deadline` as a function parameter so that it can not be manipulated due to transaction displacements. If this transaction isn't immediately triggered then it reverts as a validator may "hold the transaction" and then trigger it when it suits them for MEV efforts. Set this `deadline` parameter to the `_executeSwap()` function and the ones calling it.

Developer comments:
This contract is designed solely to enable the Bridge Controller to execute small token swaps to cover gas fees for assets bridged out of the network.

As a result, the transaction is executed, resulting in X - Y amount out. The transaction could also be executed at step 1 when the output would have been X, therefore the protocol has lost Y amount of tokens.

Impact:
If the transaction is frontrunner, it could be manipulated, the swap will result in receiving fewer funds and therefore, funds are lost.

Recommendation:
There are several things to fix:

1. The `_getRoute()` and `getRoute()` functions should not be used as it has been mentioned, these calculations should be performed off chain. This also affects to the used functions within the `_getRoute()` scope, `_checkDifferentFeeTiersSingleSwap()` and `_checkDifferentFeeTiersPathSwap()`, `_quoteSingleSwap()`.

1. The `getRouteAndSwap()` function must not rely on `_getRoute` to calculate the `amountOutExpected` and `path`, but instead, receive these parameters as function input parameters, with them being calculated off-chain to avoid manipulation.

For adding a correct slippage protection, instead of setting:

```
deadline: block.timestamp
```

Set `deadline` as a function parameter so that it can not be manipulated due to transaction displacements. If this transaction isn't immediately triggered then it reverts as a validator may "hold the transaction" and then trigger it when it suits them for MEV efforts. Set this `deadline` parameter to the `_executeSwap()` function and the ones calling it.

Developer comments:
This contract is designed solely to enable the Bridge Controller to execute small token swaps to cover gas fees for assets bridged out of the network.

The access control implementation ensures that only the Bridge Controller can perform swaps, preventing direct user interaction. Additionally, since the Bridge Controller only facilitates minimal swaps to cover gas costs owed to the Association, no user funds are at risk.

Given the Bridge's design - validator consensus mechanisms to approve bridge transactions — the quote and path generation for these small swaps must occur on-chain.

On Haven1, several factors mitigate the concerns raised in this issue. The network operates under a Proof of Authority model, where only pre-approved validators can participate. Additionally, with a gas price of zero and strict contract deployment requirements—including two independent audits and an Association review—MEV and frontrunning attacks, as identified in this issue, are very unlikely. Furthermore, because the network does not rely on economic incentives for block production, there is no block reordering, eliminating common exploit vectors.

The Association and its liquidity partners are committed to maintaining liquidity across relevant token pairs, meaning that swaps for the small token amounts managed by this contract should be executed within two hops.

Finally, in the event of slippage, no user funds will be impacted. The Bridge Controller exclusively swaps assets for gas, and in all cases, these assets are owned by the Association.

MEDIUM-2 | RESOLVED

The `_discardedAirdrop` variable in `AirdropClaim.sol` is wrongly updated, possibly leading to an incorrect track of discarded tokens.

Description:
The `_discardedAirdrop` variable within the `AirdropClaim.sol` smart contract is first updated by the `claimAirdrop()` function. The `_discardedAirdrop` variable is here updated by the % of tokens which represents the penalty for claiming `h1`:

```
uint256 esH1 = (amount * esh1BPS_) / _BPS_SCALE;
uint256 h1 = ((amount - esH1) * _H1_DEDUCTION_BPS) / _BPS_SCALE;

 _availableAirdrop -= amount;
_discardedAirdrop += (amount - (esH1 + h1));
```

This means that if a user is claiming 1000 h1 tokens then discardedAirdrop would be updated to 750 tokens.

These tokens are withdrawn by the association by executing `collectDiscarded()`. This function differentiates the case where the airdrop collection is over and the case where it is still running. There is a wrong update when the airdrop is over as it is adding the whole contract balance to the previously recorded amount on `discardedAirdrop()`:

```
if (block.timestamp > _endTS) {
            // Airdrop collection is over - collect all remaining
            withdrawAmount = address(this).balance;
            _availableAirdrop = 0;
            _discardedAirdrop += withdrawAmount;
```

Consider the following scenario:

1. Airdrop tokens are 5000.
2. The user claims 1000 tokens with a 75% penalty.
3. `collectDiscarded` is updated to 750 (75% of 1000)
4. Now the contract balance is 4250.
5. The airdrop finishes and the admin executes `collectDiscarded()`, `discardedAirdrop` should be updated to 4250 because it is the non-distributed balance. However, it is increased by the whole contract balance, which means 750 + 4250 = 5000.

Impact:
An incorrect track of funds in `_discardedAirdrop` will be produced.

Recommendation:
In order to implement a clean solution, we recommend only allowing claiming discarded amounts once the airdrop is over.

FeeContract Operators Have the Ability To Use Fees To Grief Users

Description:
The FeeContract is responsible for the core functionality of protocol fees within the Haven1 ecosystem. It performs distribution, collection and fee exceptions for particular addresses. There exists a storage variable `_feeUSD` which denotes the application fee in USD to 1e18 precision. An operator has the ability to set unreasonably high fees through this function (by frontrunning or otherwise) which can impact users interacting with the ecosystem.

Impact
The operator has the ability to grief users by setting unreasonably high fees in USD. In addition to this, high fees may also cause denial of service in other areas of the ecosystem for deposits at or below the set fees.

Recommendation:
It's recommended that `_feeUSD` is refactored to be a percentage as opposed to a dollar value so that fees are fairly collected from users deposited. In addition to this, set a reasonable threshold for setting fees in a two-step function fashion so that users are aware that application fees are about to change.

Client's comment: we insisted that fees are to remain an absolute amount and chose boundless fees for governance flexibility. Leaving this issue in the report so users accept that boundless, absolute fees exist in the fee contract.

RateLimiting In The BridgeRelayer Can Be Bypassed By Using An Alternative Contract Address

Description

The BridgeRelayer contract supports rate limiting which ensures that there is a controlled and steady flow of transactions from any single address. This is defined by the `rateLimit` modifier where rate limit + `_reqFreq` is required to be larger than the block.timestamp. A user can use an alternative contract address in order to bypass rate limits because we are assessing the msg.sender as opposed to the transaction creator.

Impact

Alternative contract addresses can be used to bypass rate limits for a single address. This was rated a Medium in severity because using an alternative contract address can break the intended function of this modifier.

Recommendation:

It's recommended that the `rateLimit` modifier is modified to assess the rate limit of tx.origin as opposed to msg.sender so the modifier can validate the transaction creator as opposed to the address directly interacting with the BridgeRelayer.

Unchecked external call to ValidatorConfig contract

Description

The contract `ValidatorRewardsBase` makes an external call to `_validatorConfig.numberOf()` without validating the return value. If `_validatorConfig.numberOf()` returns a zero value, the calculation in the `calculateShare` function could potentially result in division by zero which is an unintended outcome.

```
function calculateShare() public view returns (uint256) {
    uint256 b = address(this).balance;
    if (b == 0) {
        return 0;
    }
    uint256 l = _validatorConfig.numberOf();
    return b / l;
}
```

Impact

Function introduces a risk of division by zero, which would cause the transaction to revert in a way that is difficult to debug.

Recommendation:

It is recommended to validate the return value of `_validatorConfig.numberOf()` before using it in further calculations.

Missing Events after altering important parameters

Description

The functions in the `FeeContract` and `ValidatorRewardsESH1` contracts, such as `setGraceContract`, `setFeeUSD`, `setAssocShare`, and `setESH1Address`, modify important parameters but do not emit events. This can make it difficult for external users or even project managers to track changes in the contract state and potentially lead to misunderstandings.

`FeeContract.sol`:

```
function setGraceContract(bool status_) external;
function setFeeUSD(uint256 feeUSD_) external ;
function setAssocShare(uint256 assocShare_) external ;
```

`ValidatorRewardsESH1.sol`:

```
function setESH1Address(address esH1_) external ;
```

Impact

The absence of event emissions in functions within the FeeContract and ValidatorRewardsESH1 contracts reduces on-chain transparency and makes it difficult to track critical state changes.

Recommendation:

It is recommended to emit the appropriate events within these functions to notify external contracts and users about changes in critical parameters. Emitting events will enhance transparency and provide a clear record of parameter modifications, enabling better monitoring and understanding of the contract state.

Usage Of tx.origin Can Be Used To Bypass Fees By Phishing Attempts

Description
The `getFee()` function in the FeeContract makes an assessment whether special EOAs are exempt from fees or not. An attacker may use a smart contract to make a phishing attempt which may allow them to bypass fees by returning zero.

Impact
A malicious user may utilize phishing attempts in order to force a fee exempted EOA to trigger a transaction which may allow the contract owner to bypass fees. This was rated low in severity due to the unlikeliness and the requirement of user interaction to trigger the issue, but still possesses a security risk.

Recommendation:
It's recommended that the `getFee()` function is refactored to include smart contracts in the assessment. This can be done by refactoring the function to reflect the following:

```
function getFee() public view returns (uint256) {
    if (_feeExemptEOAs[tx.origin] && !_isContract(msg.sender)) {
        return 0;
    }
    if (_graceContracts[msg.sender] && _isGracePeriod()) {
        return _min(_feePrior, _fee);
    }
    return _fee;
}
```

This will ensure that the eligible EOA and only the eligible EOA is exempt from fees.

Client's comment: Acknowledged

Missing sanity check

Description

The `setRewardsDuration()` function within the `SimpleStaking.sol` smart contract is used by the admin to set a new value for `_duration`:

```
function setRewardsDuration(
        uint256 duration_
    ) external onlyRole(DEFAULT_ADMIN_ROLE) {
        if (_finishAt > block.timestamp) {
            revert Staking__RewardDurationNotFinished(_finishAt);
        }


        _duration = duration_;


        emit RewardDurationUpdated(duration_);
    }
```

However, there are no checks to ensure that the new value is not 0, which would lead to incorrect calculations for the rewardRate.

Impact

Setting the `duration_` variable as `0` may lead to incorrect calculations for the rewardRate.

Recommendation:

Add a check to ensure that the new `_duration` is not being set to `0`.

Starting and ending time for airdrop can be set the same.

Description
The `initialize()` function within the `AirdropClaim.sol` smart contract implements a check to ensure that startTS_ is not higher than endTS_:

```
if (startTS_ > endTS_ || startTS_ < block.timestamp) {
            revert AirdropClaim__WrongData();
        }
```

However, it is not ensured that startTS_ is not equal to endTS_.

The same scenario is present within the `setStartTimestamp()` and `setEndTimestamp()` functions.

Impact
An airdrop with duration `0` can be created, leading to users being unable to claim.

Recommendation:
Modify the implementation of the check by using `startTS_ >= endTS_` to ensure that both variables are not set to the same value.

The setExemptContract() function does not check whether an address is a contract or not

Description
The function `setExemptContract()` within `FeeContract.sol` is used to update the fee exemption status of an entire contract. However, it is not checked if the input address is a contract or not, as it is done in other functions.

```
function setExemptContract(
        address contract_,
        bool skipFee_
    ) external onlyRole(OPERATOR_ROLE) {
        _feeExemptContracts[contract_] = skipFee_;
        emit ExemptContractUpdated(contract_, skipFee_);
    }
```

The same situation takes place within the `setExemptCaller()` and `setExemptFunction()` functions.

Impact
An EOA can be set to be exempt when it is not a contract.

Recommendation:
Implement the following check:

```
bool isContract = _isContract(contract_);


        if (!isContract) {
            revert FeeContract__OnlyEOA(eoa_);
        }
```

Use of assert statements instead of require statements

Description

The `BridgeRelayer` contract utilizes assert statements in functions like `enqueue()`, `dequeue()` and `_indexOfExn()`, which can be hit during normal operation. Assert statements are typically used to check for conditions that should never occur and are primarily meant for testing. In this case, require statements should be used instead of assert to handle expected error conditions during contract execution.

Impact

Unlike `require`, which properly handles expected error conditions by reverting with an error message, `assert` consumes all remaining gas and triggers a `Panic` error which is harder to debug.

Recommendation:

Replace the assert statements in the contract functions with require statements to properly handle expected error conditions. Require statements are better suited for checking user input and contract preconditions, ensuring that issues are handled gracefully by reverting and providing a meaningful error message.

Fix Addressed by the client at commit 333422d, assert statements in functions _indexOfExn() are replaced by the proper revert. While other assertions are shown by Manual Review that they are invariants .

Ambiguous failure handling in add functions

Description

The functions `addIndexer()`, `addOperator()`, and `addChain()` in BridgeRelayer.sol include a pattern where the _exists() function is called to check if an element already exists in a collection. When the element exists, the function returns silently without providing any indication of the failure. This leads to ambiguous failure handling as there is no explicit indication of why the operation failed.

```
bool exists = _exists(_supportedChains, chainID);
if (exists) return;
```

Impact

These functions silently return when an element already exists — without reverting with an error message or returning a boolean that indicates the failure which can be consumed by the caller. External observers and smart contract interactions cannot reliably determine if an error occurred, making it difficult to track failures. Hence, a caller may execute a valid transaction that modifies other parts of some contract state, despite the intended operation within these functions failing.

Recommendation:

It is recommended to enhance the failure handling mechanism in these functions by either adding a boolean return value to indicate success or failure, or by reverting the transaction with a specific error message when the addition operation fails due to the existence of the element. This way, the failure reasons will be clearly communicated on-chain, improving transparency and making it easier for users and developers to understand and troubleshoot issues related to adding existing elements.

INFORMATIONAL-3 | ACKNOWLEDGED

EOAs Can Become A Grace Contract In The FeeContract

Description
In the FeeContract `setGraceContract` allows a caller to obtain grace status. This includes EOAs as anybody can pass through the status to the function parameters and insert themselves into the `_graceContracts` as there is insufficient validation.

Recommendation:
It's recommended that `setGraceContract` is amended to `setGraceAddress` in order to reflect the naming conventions of the function.

INFORMATIONAL-4 | ACKNOWLEDGED

Certain actions are centralized

Description
There are several actions within the whole protocol which are centralized, meaning that they depend on a single actor.

- SimpleStaking.sol: `withdraw()` implements a `whenNotGuardianPaused` modifier, meaning that the admin can decide to pause the withdraw functionality so users can not unstake their tokens.
- FeeContract.sol: the 'OPERATOR_ROLE' is allowed to execute `forceDistributeFees()` bypassing any time restrictions.

Impact:
Users will need to trust the centralized actor to operate correctly.

Recommendation:
It is recommended to implement robust governance mechanisms or adopt multi-signature (multi-sig) platforms to manage privileged and centralized functions. These measures help to decentralize authority, increase transparency, and reduce the risk associated with having a single point of failure or control.
Client's comment: this is the intended design

INFORMATIONAL-5 | RESOLVED

The H1 Token's Total Supply Is Minted Directly To The Deployer - Distribution May Cause Instability In Token Price

Description
When deploying the H1 token, the total supply is minted directly to the deployer which may have an impact on the token's price depending on the distribution method.

Recommendation:
It's recommended that the Haven1 team carefully considers the distribution method in a slow and evenly distributed fashion as rapid and concentrated distribution may cause a significant instability in the token's price.

INFORMATIONAL-6 (GAS) | ACKNOWLEDGED

Cancelling An Airdrop Will Render The Contract Useless

Description
The AirdropClaim contract allows for the cancellation of an airdrop. As defined in the comments and natspec, it's intentional to have to redeploy the contract should an airdrop be cancelled but will require the deployer to pay substantially more gas.

Recommendation:
It's recommended that the AirdropClaim contract is refactored to be reusable even after cancelling an airdrop.

Client's comment: Acknowledged

INFORMATIONAL-7 (GAS) | RESOLVED

Multiple Instances Of Operating Directly On Storage Variables May Cause Increased Gas Intensiveness

Description

There are multiple instances throughout the project where functions operate directly on storage variables which can make it more costly to use (2,100 for cold storage access and 100 for every interaction after). On the other hand, mload opcodes only require 2 gas units to operate on.

Recommendation:

It's recommended that these instances create local variables in memory, perform their operations on these local variables, and copy the final values into storage.

INFORMATIONAL-8 | RESOLVED

`depositAirdrop` can be called 1 block before airdrop ending

Description

The function `depositAirdrop()` within the `AirdropClaim.sol` smart contract is used to deposit the airdrop amount to the smart contract. The function implements a `notEnded` modifier to ensure that the function is called before the end of the airdrop period:

```
function depositAirdrop() external payable notEnded {
        if (_ready) {
            revert AirdropClaim__IsReady();
        }

        if (msg.value != _airdropAmount) {
            revert AirdropClaim__WrongAidropAmount(msg.value,
_airdropAmount);
        }

        _ready = true;


        emit AirdropDeposited(msg.value);
    }
```

However, it is not checked if the deposit is being made after the starting period, consider the following example:

- Starting time is set to X.
- Ending time is set to Y.
- depositAirdrop is called at Y-1 (meaning -1 blocks)

The behavior of the function does not make sense for the wanted purpose. It should be ensured that the calling moment is previous to the starting of the airdrop period.

Impact:
The airdrop's deposit can be executed after the starting date, which does not make sense.

Recommendation:
Add a new modifier, for example `notStarted`, which ensures that the function is called before the airdrop starting period.

Floating Solidity Version in pragma Statement

Location: All contracts in scope

Description
The Solidity version specified in the pragma statement is floating rather than fixed to an exact compiler version. This floating version specification allows for unexpected behavior if the contract is compiled with newer or older versions of the Solidity compiler than originally intended.

Impact:
Without a fixed version, updates in the Solidity compiler—including optimizations, security patches, or potential breaking changes—may introduce vulnerabilities or alter the contract's behavior unexpectedly.

Recommendation:
Specify a fixed Solidity version in the pragma statement to avoid the risks associated with floating versions.

• **contracts/airdrop/AirdropClaim.sol** • **contracts/bridge/bridge-controller/BridgeController.sol:** • **contracts/bridge/bridge-relayer/BridgeRelayer.sol** • **contracts/bridge/locked-h1/LockedH1.sol** • **contracts/externalChain/eth-mainnet/H1.sol** • **contracts/fee/FeeContract.sol**	
Reentrance	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

- **contracts/fee/channels/fee-distributor-channels/FeeDistributorChannelWH1.sol**
- **contracts/fee/channels/staking/StakingChannelESH1.sol**
- **contracts/fee/channels/validator-rewards/ValidatorRewardsBase.sol**
- **contracts/fee/channels/validator-rewards/ValidatorRewardsConfig.sol**
- **contracts/fee/channels/validator-rewards/ValidatorRewardsESH1.sol**
- **contracts/fee/channels/validator-rewards/ValidatorRewardsH1.sol**

Reentrance	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

• **contracts/staking/SimpleStaking.sol** • **contracts/tokens/HRC20.sol** • **contracts/utils/OnChainRouting.sol** • **contracts/utils/upgradeable/RecoverableUpgradeable.sol**	
Reentrance	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the Haven1 team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Haven1 team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo