SMART CONTRACTS REVIEW

zokyo

December 31st 2024 | 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: 0 points

Starting with a perfect score of 100:
- 2 Critical issues: 2 resolved = 0 points deducted
- 1 High issue: 1 acknowledged = - 4 points deducted
- 11 Medium issues: 11 resolved = 0 points deducted
- 13 Low issues: 7 resolved and 6 acknowledged = - 6 points deducted
- 9 Informational issues: 6 resolved and 3 acknowledged = 0 points deducted

Thus, 100 - 4 - 6 = 90

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 2 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 5

Executive Summary 7

Structure and Organization of the Document 9

Complete Analysis 10

AUDITING STRATEGY AND TECHNIQUES APPLIED

The Smart contract's source code was taken from the Haven1 repository.
Repo: Repository: https://github.com/haven1network/solidity-core/tree/zokyo-round-one

Last commit: 8d86f972f7542c5e74889efe62a63cc67e12d50e

Contracts under the scope:

- contracts/governance/VotingEscrow.sol
- contracts/h1-developed-application/H1DevelopedApplication.sol
- contracts/h1-developed-application/lib/FnSig.sol
- contracts/h1-native-application/H1NativeApplication.sol
- contracts/h1-native-application/H1NativeApplicationUpgradeable.sol
- contracts/h1-native-application/H1NativeBase.sol
- contracts/network-guardian/NetworkGuardian.sol
- contracts/network-guardian/NetworkGuardianController.sol
- contracts/network-guardian/lib/Array.sol
- contracts/proof-of-identity/ProofOfIdentity.sol
- contracts/proof-of-identity/lib/Attribute.sol
- contracts/proof-of-identity/lib/BytesConversion.sol
- contracts/proof-of-identity/lib/POIType.sol
- contracts/staking/Staking.sol
- contracts/tokens/BackedHRC20.sol
- contracts/tokens/EscrowedH1.sol
- contracts/tokens/WH1.sol
- contracts/utils/Address.sol
- contracts/utils/upgradeable/BlacklistableUpgradeable.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 endeavors to become the premier 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. Haven1 has built an EVM compatible blockchain that extends GoQuorum. The bedrock of the protocol leverages the Proof of Identity Framework and Network Level Guardrails as outlined in the smart contracts within the scope which attempts to provide safe and reliable blockchain solutions. The code gives developers a foundation to build applications in order to assert that secureness is a core part of software engineering on Haven1.

Users have the ability to be issued an identity on chain which is done through the issueIdentity function of the Proof of Identity NFT smart contract. This is usually assigned to the principal account by an administrator with the operator role. Should the user need subsequent identities issued to alternative wallets, an auxiliary identity can be issued to such addresses. Various attributes can be set for such accounts such as the user type, their competency rating, nationality and proof of liveliness. Should there be violations of an account on chain, the operator reserves the right to suspend and unsuspend principal addresses (which includes all auxiliary accounts).

The NetowkrGuardianController is responsible for providing NetworkGuardians with an interface to pause the operation of certain contracts on the Haven1 network. This ability is reserved for the address with the default admin role. The NetworkGuardians on the other hand is a base inherited contract which is a requirement for all native and developed contracts.

Finally, users also have access to staking on Haven1 which allows a user to stake a specific HRC20 token or Native Tokens in order to earn rewards. The default admin user will set the rewards duration and notify the reward amount of the staking period. Users will then be able to stake their tokens where they will have the ability to collect their rewards which are distributed on a curve depending on how long the user has been staked for in accordance to the rewards duration and the reward amount notified by the administrator. Once the staking period is over, users are free to withdraw their tokens or keep them staked for the next staking period. Administrators reserve the right to recover undesired HRC20 tokens which may have been accidentally transferred to the staking contract.

Zokyo was tasked with the security review for Haven1's development live cycle and discovered issues ranging from critical down to informational. Overall the code is very well documented/commented (appreciated by the audit team), well thought out and engineered to a high standard in an effort to achieve Haven1's goals. Most of the issues discovered were revolving around business logic issues, issues around the use of native token and msg.value and upgradeability issues resulting in some kind of security impact. A small amount of informational issues were stage 1 exploits with no resulting security impact to the protocol discovered by the audit team. Zokyo recommends that the development team carefully reviews the issues discovered and implements the suggested fixes in order to further fortify the codebase for the users.

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	Strict Equality In The VotingEscrow Contract Can Lead To A Denial of Service Condition	Critical	Resolved
2	Users can avoid paying fees when withdrawing tokens	Critical	Resolved
3	Payable Vulnerability in Multicall Implementation Resulting In The Bypassing Of Checks Imposed By _msgValue	High	Acknowledged
4	Missing Handling for Contracts Unable to Receive Native Tokens	Medium	Resolved
5	Loss of msg.value if non-zero value sent while deposit token is not h1_address	Medium	Resolved
6	Lack of checking for the new association being the same as the old association in setAssociation function	Medium	Resolved
7	Unsafe Casting In The FeeDistributor May Cause Integer Truncation	Medium	Resolved
8	Resetting Of Attribute Counter In The ProofOfIdentity Contract May Cause The Accidental Overwriting Of Existing Attributes Resulting In The Loss Of Data	Medium	Resolved
9	No Storage Gaps In NetworkGuardianController May Result In Storage Collisions Post Upgrade	Medium	Resolved
10	Proof Of Identity Should Use Safe Minting To Ensure The Receiving Of NFT Tokens	Medium	Resolved
11	Users can get forced to not activate _onlyVeHolderClaimingEnabled mode, which can also lead to them not being able to compound their rewards$	Medium	Resolved

#	Title	Risk	Status
12	Users may lose native tokens if they pay an extra fee	Medium	Resolved
13	`_vestingDuration` can get modified when users are already staking forcing them to stake more time than expected	Medium	Resolved
14	Blacklisting addresses can be bypassed	Medium	Resolved
15	If the linked accounts array grows enough, the suspending/unsuspending functionality will become unavailable	Low	Acknowledged
16	If the auxiliary account's array grows enough, some actions can run out of gas	Low	Resolved
17	Certain actions are centralized	Low	Acknowledged
18	A suspended account can be issued as an auxiliary	Low	Resolved
19	Potential Inflation Of _totalSupply In The Staking Contract Due To Native Token Transfer Callback	Low	Resolved
20	Lack Of Access Controls on _authorizeFeeContractAddressUpdate In H1NativeBase May Cause Unexpected Updates To The Fee Contract Address	Low	Acknowledged
21	Missing Sanity Checks When Initializing The Staking Contrac	Low	Acknowledged
22	Having a non-modifiable number of MAX_ITERS can be risky	Low	Resolved
23	AttributeType can be set to a non-existent id	Low	Resolved
24	Missing `__Context_init` call	Low	Resolved
25	Lack of consideration for transfer fee on tokens in VotingEscrow and Staking contracts	Low	Acknowledged

#	Title	Risk	Status
26	No implementation for increaseAllowance/ decreaseAllowance alongside approve	Low	Resolved
27	Missing event emission in functions that set vital parameters	Low	Resolved
28	Minting and burning are allowed while the contract is paused	Informational	Resolved
29	Floating Solidity Version in pragma Statement	Informational	Resolved
30	Reentrancy Vulnerability in recoverHRC20() and recoverAllHRC20() Functions	Informational	Resolved
31	Spelling Typo Contained In _claim For The EscrowedH1 Contract	Informational	Resolved
32	Lack Of Error Messaging In _create_lock() Function For VotingEscrow	Informational	Resolved
33	If a malicious address gets approved or an approved one gets compromised, it will be able to transfer user's funds to the VotingEscrow	Informational	Acknowledged
34	The WH1 address can be changed after the protocol is initialized	Informational	Acknowledged
35	Incorrect NatSpec comment	Informational	Resolved
36	Some user's attributes may not correspond to the expected fields	Informational	Acknowledged

Strict Equality In The VotingEscrow Contract Can Lead To A Denial of Service Condition

Description

The VotingEscrow contract is responsible for distributing weights depending on time and what users are voting for. Users can deposit and withdraw their tokens which will allow them to be checkpointed and thus determining which user voted for what. There exists a function deposit_for_admin which allows anybody including smart contracts to deposit for someone else but cannot extend their lock time and deposit for a new user.

The issue lies at the bottom of this function where it will assert that the balance of native tokens within the contract is zero. A malicious user can deploy a contract containing the self destruct opcode (whilst deprecated, still works) to bypass the checks when receive is triggered and ultimately force native token into the contract.

The assertion will always fail causing functionalities (including various functionalities for the FeeDistributor contract) relating to deposit_for_admin to be bricked. This was rated a Critical in severity because it's an incredibly easy and cheap attack to trigger where the absolute minimal amount needs to be forced into the target contract.

Proof of Concept

The following Proof of Concept scenario outlines this issue and asserts that the transaction will always be reverted:

https://gist.github.com/chris-zokyo/ab7b8014cb4e8e82dfd87cf56d9c0a40#file-mytest-t-sol-L143-L208

Recommendation:
It is recommended that native tokens being forced into the contract (which were not transferred via a payable function) are ignored when performing mathematical calculations and assertions. In addition to this, removing the strict equality and making proper assertions in the form of an if statement to enforce that the correct deposit token is used will also make the aforementioned function more robust.

Users can avoid paying fees when withdrawing tokens.

The `withdraw()` function within the `VotingEscrow.sol` smart contract, which is used to withdraw every deposited token by the user, implements an `applicationFee` modifier, which is used to charge fees for withdrawing tokens.

There exist another function, `withdraw_and_create_lock()`, which is used to perform 2 actions at the same time, withdrawing funds and creating a lock. However, this function can be used by users to avoid paying fees when withdrawing tokens as it does not implement the `applicationFee` modifier while it does call the internal `_withdraw()` function, which is going to withdraw all tokens, and then it calls `_create_lock()`, which users can use to create a lock of 1 wei of any allowed tokens.

Consider the following scenario:

1. UserA deposited tokenA, tokenB, tokenC.
2. Fees for withdrawing are 1e18.
3. If UserA executes `withdraw()`, he should pay 1e18 fee.
4. However, UserA executes `withdraw_and_create_lock()` and creates a lock with 1 wei.

As a result of the described scenario, userA will receive the tokens and only pay 1 wei of any allowed tokens instead of 1e18 of fee token.

Recommendation:
Add the applicationFee modifier to the `withdraw_and_create_lock()` function.

Payable Vulnerability in Multicall Implementation Resulting In The Bypassing Of Checks Imposed By _msgValue

Location: H1NativeApplicationUpgradeable.sol/H1DevelopedApplication.sol

The implementation of the `applicationFee` modifier in the Haven contracts does not properly protect against a payable vulnerability in multicall scenarios. By bypassing the `_msgValue` abstraction check, an attacker can exploit the contract by executing multiple functions while paying `msg.value` only once.
Haven contracts provide an ecosystem to developers in order to develop smart contracts over the Haven contracts. Two of the Haven contracts `H1NativeApplicationUpgradeable` and `H1DevelopedApplication` attempts to abstract the `msg.value` into new variable `_msgValue` (i.e. mainly to deduct fees from `msg.value`). These modifiers `applicationFee` and `developerFee` that contains this abstraction are also implemented in a way to deal with `mutlicall` (i.e. to be implemented by developers on top of haven contracts). This is shown by how the modifiers handle the fee payment by checking `address(this).balance` which is the known way to encounter the payable vulnerability in multicall.

This way though does not protect the contracts which implement those haven contracts from multicall. Since an attacker can still pay once `msg.value` and bypass the checks that are imposed on `_msgValue` . This scenario is implemented in the following gist: https://gist.github.com/beber89/7ad7560f4007711396bf4510156c1f96

Recommendation:
Developers implementing contracts on top of the Haven contracts should avoid using multicall functions to mitigate the risk of this payable vulnerability. Alternatively, additional security measures such as stricter payment validation could be implemented to enhance the protection against such exploits.

Fix: The client stated that fee payment will be collected once in intended scenarios such as multicalls for token swaps (i.e. uniswap) . Whereas other contracts will be vetted carefully by a centralized association in order to avoid the misuse of the multicall.

Client comment: It was written like this so that we could have one application fee charged per total "swap", not one fee per route chosen.

MEDIUM-1 | RESOLVED

Missing Handling for Contracts Unable to Receive Native Tokens

Location: VotingEscrow.sol

The function `_withdraw()` in VotingEscrow.sol does not account for the scenario where `msg.sender` might be a contract that is unable to receive native tokens, potentially leading to the assets being forever locked if the invocation fails.

Recommendation:
It is recommended to add an argument `recipient` to the function or implement a new external function that allows the sender to specify the recipient address for receiving the assets. This will provide flexibility and ensure that assets are not left locked in case the original `msg.sender` is unable to receive native tokens.

MEDIUM-2 | RESOLVED

Loss of `msg.value` if non-zero value sent while deposit token is not `h1_address`

Location: VotingEscrow.sol

This issue arises in the `deposit_for()` function of VotingEscrow.sol where a non-zero `msg.value` may be lost if it is mistakenly sent when the deposit token is not `h1_address`. The contract currently requires a non-zero value if the `_deposit_token` is `h1_address`, but does not handle the case when `msg.value` is sent erroneously with a different deposit token.

Recommendation:
Update the code logic in the `deposit_for()` function to handle the case where `msg.value` is sent when the deposit token is not `h1_address`. Implement proper validation to check the deposit token before verifying the `msg.value`. If `_deposit_token` is not `h1_address`, revert the transaction if `msg.value` is non-zero to prevent the loss of funds. Additionally, consider providing appropriate error messages to inform users of the correct deposit token and value requirements.

MEDIUM-3 | RESOLVED

Lack of checking for the new association being the same as the old association in setAssociation function

Location: NetworkGuardian.sol

The `setAssociation` function in NetworkGuardian.sol does not prevent the case where the new association address being set is the same as the old association address. This results in revoking roles by accident and preventing these roles from being granted again, as the roles can only be granted by a bearer of the DEFAULT_ADMIN_ROLE which is revoked unintentionally.

Recommendation:
Add a check within the setAssociation function to ensure that the new address `addr` does not equal the old association address `_association` before proceeding with the role changes. This check will prevent unintentional revocation of roles and ensure the roles can be managed properly in the contract.

MEDIUM-4 | RESOLVED

Unsafe Casting In The FeeDistributor May Cause Integer Truncation

Description

The FeeDistributor is responsible for handling the relative fees which are distributed to the VE holders. The `_claimToken` function claims all pending distributions of the provided token for a user. Within the internal claim function there is a conditional where if `amount` is more than zero the cached balance is updated however, amount is defined as a `uint256` but the cached balance is defined as a `uint128`.

Because `amount` is being downcasted unsafely, this may cause an edgecase where the integer is truncated and may result in an unexpected value.

Recommendation:
It's recommended that the SafeCast library is used when attempting to downcast larger values.

Resetting Of Attribute Counter In The ProofOfIdentity Contract May Cause The Accidental Overwriting Of Existing Attributes Resulting In The Loss Of Data

Description

The ProofOfIdentity contract allows the operator to set certain attributes which relate to certain accounts. For this to occur, the `_attributeCount` is heavily relied upon to determine the id of a new attribute; however, there is no check when setting new attributes for ids that have already been set.

Should the operator reset the attribute counter via the `setAttributeCount` function, subsequent attributes may accidentally be overwritten unintentionally.

Recommendation:
It's recommended that before attempting to add an attribute, there is an existence check against the id obtained from `_attributeCount`. Should the attribute already exist, the id should move back to the latest id according to the counter. In addition to this, a force option in the form of a bool can allow the operator to overwrite an attribute intentionally.

No Storage Gaps In H1NativeApplicationUpgradeable May Result In Storage Collisions Post Upgrade

Description

The `H1NativeApplicationUpgradeable` contract is an upgradeable contract implementing `UUPSUpgradeable` which allows the contract's implementation to be modified while retaining the existing state. This contract however, does not implement storage gaps to prevent the overwriting of existing state variables when attempting to upgrade the contract. Since each inherited contract is managing its own storage layout, it should add `__gap` as per the recommendations.

Should the contract be upgraded and new state variables are added, this may result in the collision of storage variables which can have unintended (potentially malicious) consequences.

Case Study:
https://blog.audius.co/article/audius-governance-takeover-post-mortem-7-23-22

Recommendation:
It's recommended that a storage gap is added at the footer of the contract to allow the safe addition of new storage variables in the event of a refactor and the amount of storage slots modified depending on what variables are added. See the below example:

```
uint256[49] __gap;
```

Proof Of Identity Should Use Safe Minting To Ensure The Receiving Of NFT Tokens

Description

The ProofOfIdentity contract is used to issue principal and auxiliary identities to Haven1 users to record the ownership of wallets. The `issueIdentity` and `issueAuxiliary` functions are mostly responsible for such functionalities however, they do not implement the use of `_safeMint()`.

The `account` will be minted a token when the above functions are called however, if the `account` is a contract address (for example receiving an auxiliary token) that does not support the use of ERC-721 tokens, the token may be frozen in the contract resulting in the possible loss of NFTs.
E
IP-721 stipulates the following: `A wallet/broker/auction application MUST implement the wallet interface if it will accept safe transfers.`

Src: https://eips.ethereum.org/EIPS/eip-721

Recommendation:

It's recommended that `_safeMint` is used as opposed to _mint when issuing proof of identity tokens.

Users can get forced to not activate _onlyVeHolderClaimingEnabled mode, which can also lead to them not being able to compound their rewards

The `enableOnlyVeHolderClaiming()` function within the `FeeDistributor.sol` smart contract is used for users to enable/disable other users to claim their rewards for them:

```
function enableOnlyVeHolderClaiming(bool enabled) external {
_onlyVeHolderClaimingEnabled[msg.sender] = enabled;
        emit OnlyVeHolderClaimingEnabled(msg.sender, enabled);
 }
```

However, the `_onlyVeHolderClaimingEnabled` mapping is set to `false` by default. This means that any third party can claim for another user until the user executes `enableOnlyVeHolderClaiming()`.

This design allows any third party to frontrun users who execute `enableOnlyVeHolderClaiming()` and execute `claim()` for them before `_onlyVeHolderClaimingEnabled` is set to true.

The described scenario could impact the legitimate user by allowing a third party to execute the claim and send the funds to the legitimate user, even though the user wanted to compound their rewards instead of receiving them.

Recommendation:
Change the way this concept is designed. Instead of creating a function that enables only the holder to claim, create a function that allows third parties to claim on behalf of the legitimate user, with this option set to false by default. Apply the necessary modifications resulting from this change as well.

Client comment:
This feature is something that we inherited from the original implementation. We agree that the concept should be reworked. There are a few features we have in mind that require a third-party to be able to claim on behalf of a user, but those situations would be special and limited.

MEDIUM-9 | RESOLVED

Users may lose native tokens if they pay an extra fee

The function `startVesting()` within the `EscrowedH1.sol` smart contract implements the `applicationFee()` modifier which is used to charge fees for the action. However, the `refundRemainingBalance` parameter is set to false, which means that if a user pays more than the required amount, they will not get reimbursed.

Recommendation:
The aforementioned function should be refactored to refund the remaining native tokens after the fees have been taken out.

MEDIUM-10 | RESOLVED

`_vestingDuration` can get modified when users are already staking forcing them to stake more time than expected

The variable `_vestingDuration` within the `EscrowedH1.sol` smart contract is set during the contract initialization and determines the amount of time for the vested tokens. However, there is a `setVestingDuration` which can be executed by `OPERATOR_ROLE` at any time

```
function setVestingDuration(
        uint256 newDuration
    ) external onlyRole(OPERATOR_ROLE) {
        _vestingDuration = newDuration;
        emit VestingDurationUpdated(newDuration);
    }
```

If the operator changes the vesting duration after a user has already started staking it will force him to wait more time than expected.
This function neither includes a higher limit for the vesting duration, this means that it can force the user to get his funds locked if the duration is set high.

Recommendation:
Remove the `setVestingDuration` and do not allow changing the duration after the contract deployment.

Blacklisting addresses can be bypassed

The `BlacklistableUpgradeable.sol` smart contract has several functions which disallows blacklisted users from interacting with the contract which implements this dependency.

The `WH1.sol` token inherits from `BlacklistableUpgradeable.sol` and implements its functions and modifiers. One such function is `_assertNotBlacklisted(msg.sender)` which asserts that the msg.sender is not blacklisted from interacting with WH1. This can be bypassed by proxying the transaction with an alternative contract as msg.sender is the address directly interacting with WH1 as opposed to being the transaction creator.

If `msg.sender` is blacklisted and desires to bypass the checks implemented as a part of the BlacklistableUpgradeable dependency, they could initiate the transaction and use an intermediary (alternative) contract so that msg.sender is a different address.

Recommendation:
It's recommended that msg.sender is replaced by tx.origin when determining if the target address black listed as this performs an assessment on the transaction creator as opposed to the contract (or EOA) that is directly interacting with the contract implementing the black list. This removes the possibility for attackers to proxy their transactions by using an alternative contract.

If the linked accounts array grows enough, the suspending/unsuspending functionality will become unavailable

The `ProofOfIdentity.sol` smart contract, allows X amount of 'aux accounts' to be linked to a principal one, however, the added accounts can not get 'unlinked' but only suspended.
The functions for suspending/unsuspending an account implement 'for loops' to suspend all the linked accounts.

```
for (uint256 i; i < l; i++) {
            address a = aux[i];
            _permissionsInterface.updateAccountStatus(ORG, a, 1);
            emit AccountStatusUpdated(AccountStatus.SUSPENDED, a, reason);
        }
```

As linked accounts can not get unlinked, it is technically possible that the amount of linked accounts grows enough that these 'for loops' run out of gas, so suspending/unsuspending operations can not be executed.
There is a maximum limit for liking account, set by `_maxAux`, so the mentioned scenario would take place if `_maxAux` is increased enough for running out of gas.

Recommendation:
Add a function that allows removing accounts from the `_principalToAux` array.

Client comment:
This issue was at the front of our minds when we were creating this contract. For business needs, we cannot remove accounts from the mentioned array. We have made all efforts to ensure out of gas errors won't occur during this and associated operations, and [via the Go Quorum contracts] we do have the ability to direct set per-account status as a backup.
We will have to just acknowledge this issue and leave the implementation as is.

If the auxiliary account's array grows enough, some actions can run out of gas

The `suspendAccount()` and `unsuspendAccount()` within the `ProofOfIdentity.sol` contract are used to suspend and unsuspend all the linked accounts to a principal account.

These functions work with ' for loops' that traverse the whole array of linked accounts. This means that if these arrays grow enough, the mentioned operations can run out of gas and therefore, become impossible to suspend/unsuspend new accounts.

Recommendation:
In order to avoid this edge case, it is recommended to implement 2 new extra functions to suspend/unsuspend a single account from the array and remove it. This would help to reduce the array's size.

Client comment:
This issue was at the front of our minds when we were creating this contract. For business needs, we cannot remove accounts from the mentioned array. We have made all efforts to ensure out of gas errors won't occur during this and associated operations, and [via the Go Quorum contracts] we do have the ability to direct set per-account status as a backup.

We will have to just acknowledge this issue and leave the implementation as is.

Certain actions are centralized

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

- `WH1.sol`: Admin can modify any user's balance by executing `transferFromAdmin()`. User can get blacklisted and become unable to withdraw.
- `EscrowedH1.sol`: claim functionality can get paused leading to users not being able to claim their tokens. Admin can execute `emergencyWithdraw()` and `recoverHRC20()` and withdraw the whole native and erc20 contract's balance. Admin can add/remove any user from the whitelist.
- `BackedHRC20.sol`: admin can burn any user's tokens executing `burnFrom` if they have previously approved the contract. Operator can blacklist/whitelist any user for any reason.
- `ProofOfIdentity.sol`: operator can change any user's token uri and suspend any account.
- Most of the `H1DevelopedApplication.sol` functions.
- `FeeDistributor.sol`: Admin can withdraw any token from the contract by executing `withdrawToken()`. Admin can also disable the claiming of certain tokens by executing `enableTokenClaiming` and setting it to false.
- `VotingEscrow.sol`: Admin can withdraw any non active token from the contract by executing `recover_hrc20`
- `Staking.sol`: admin can withdraw any tokens from the contract by executing `recoverHRC20()`

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 comment:
Acknowledged. It is this way by design. For context, on Mainnet, the DEFAULT_ADMIN for eg will be a multisig.

LOW-4 | RESOLVED

A suspended account can be issued as an auxiliary

The function `issueAuxiliary()` within `ProofOfIdentity.sol` does check if the principal account is suspended or not. However, it does not check if `to` (the auxiliary account) has been previously suspended, it is only checked if `to` is verified, however, an account can get suspended without being verified. Therefore, an auxiliary can be issued while being suspended.

Recommendation:
Implement a check to ensure that `to` is neither suspended before expediting the auxiliary.

LOW-5 | RESOLVED

Potential Inflation Of _totalSupply In The Staking Contract Due To Native Token Transfer Callback

Description

When a user withdraws native tokens from the staking contract (if `h1ToTx` is more than zero), they will be sent native tokens. A user can then redeposit into the contract via a fallback function which can create an inaccuracy in the `_totalSupply`. This was determined to be informational as no security impact was discovered when testing the issue - cross function reentrancy with no security implications however, executing an external call before updating `_totalSupply` makes the protocol vulnerable to a cross-function or read-only reentrancy attacks as `_totalSupply` is a variable used to calculate the rewardPerToken.

```
. . .
uint256 h1ToTx = amountH1_ + _msgValueAfterFee();
```

```
if (h1ToTx > 0) {
    _transferH1Exn(msg.sender, h1ToTx);
}

_totalSupply -= amountToken_ + amountH1_;
. . .
```

It can be observed that the function does not follow the CEI pattern as it first executes the external call and later updates the contract state by decreasing _totalSupply.

Recommendation:
It's recommended that the Checks, Effects and Interactions (CEI) pattern is followed when processing withdrawals if there is a change in the code which may open the opportunity for exploitation. This means that the total supply should be updated before transferring native tokens to the user.

LOW-6 | ACKNOWLEDGED

Lack Of Access Controls on _authorizeFeeContractAddressUpdate In H1NativeBase May Cause Unexpected Updates To The Fee Contract Address

Description

The `updateFeeContract` function in the H1NativeBase contract allows for the changing of the feeContract after deployment. This function is guarded by the `authorizeFeeContractAddressUpdate` which has no existing access controls which can lead to unexpected and unauthorized changes to state variables relating to fees.

A malicious actor may update the fee contract in order to gain more favorable fees for contracts relating to the H1NativeBase. This was determined to be a low in severity because developers are required to implement access controls themselves which can easily be forgotten

Recommendation:
It's recommended that an access control modifier is placed on `_authorizeFeeContractAddressUpdate` by default only authorizing the relative address to make changes to such state variables.

Client comment: Acknowledged

Missing Sanity Checks When Initializing The Staking Contract

Description

When initializing the staking contract there are various sanity checks made such as checks on `stakingToken_` and `rewardToken_` for the zero address; however, additional checks should be made when initializing the staking token because when staking tokens are staked in the `stake()` function, the `msg.value` and `amountToken_` are simply concatenated to create the users balance.

Should a different staking token be used which differs in value from the native token, a disproportionate amount of balance may be given to the user which may allow them to get an unfair amount of rewards.

Recommendation:
It's recommended that additional sanity checks are made when initializing `stakingToken_` to assert that this is the wrapped native token.

Client comment:
We have account for internally (the contract will only be used with "versions" of H1 tokens that can. in some manner. be redeemed 1:1 for native H1)

LOW-8 | RESOLVED

Having a non-modifiable number of MAX_ITERS can be risky

The `NetworkGuardianController.sol` smart contract contains a constant variable `MAX_ITERS` set equal to 500 which represents the maximum number of iterations allowed for a bulk pause or unpause.

These iterations are executed using 'for loops', which can be risky due to the possibility of running out of gas and therefore being unable to execute the action.

Recommendation:
Implement a function which an administrator can execute to change `MAX_ITERS` in case that the set number of iterations is large enough for a transaction running out of gas.

LOW-9 | RESOLVED

AttributeType can be set to a non-existent id

The function `setAttributeType` within the `ProofOfIdentity.sol` smart contract sets an attribute type for a specific id. However, it is not checked if the `id` is lower than `attributeCount`, as it has been done within `setAttributeName()`.

Recommendation:
Implement the same check that is included in the `setAttributeName()` to ensure that `id` is lower than `attributeCount`:

```
if (id >= _attributeCount) {
            revert ProofOfIdentity__InvalidAttribute(id);
        }
```

LOW-10 | RESOLVED

Missing `__Context_init` call

The `__NetworkGuardian_init()` function within `NetworkGuardian.sol` is not calling `__context_init()` from `ContextUpgradeable.sol`

Recommendation:
Add the call to `__context_init()`.

LOW-11 | ACKNOWLEDGED

Lack of consideration for transfer fee on tokens in VotingEscrow and Staking contracts

Location: VotingEscrow.sol, Staking.sol

The contracts do not account for tokens that incur a fee on transfers, which can lead to discrepancies in the accounting as the transferred amounts might be less than expected due to fees being deducted.

Recommendation:
A recommendation is to modify the contracts to adjust the deposited and transferred amounts to account for any fees incurred during token transfers. This can be achieved by either increasing the deposited amount to include the anticipated fee or by subtracting the fee from the deposited amount to ensure accurate accounting. Otherwise, it is required to exclude these tokens from being a deposit asset altogether.

Client comment:
Acknowledged - if any such tokens exist on Haven1 they will not be supported tokens on either contract

No implementation for increaseAllowance/decreaseAllowance alongside approve

Location: WH1.sol

The issue with using the `approve` function in ERC20 tokens lies in how it handles changes to an approved allowance, potentially leading to a double-spend scenario. Here's a scenario to illustrate:

Imagine that a user has set an allowance for a spender (e.g., a decentralized exchange) to transfer 100 tokens on their behalf by calling approve(spender, 100). Now, the user wants to update the allowance to 150 tokens by overriding that approval. To do this, they would need to call approve(spender, 150).

However, this creates a potential race condition if the spender or a malicious actor acts between the two transactions. Here's how:

Initial Setup: The user calls approve(spender, 100), setting the spender's allowance to 100 tokens. Updating Allowance: The user wants to increase the allowance to 150 tokens, so they call approve(spender, 150). Exploit Opportunity: Before the approve call with 150 tokens is mined, the spender sees the 100 tokens approved and immediately transfers them. Result: When the transaction setting the allowance to 150 tokens is mined, the spender has access to transfer 150 more tokens. This means they can withdraw 250 tokens total instead of the intended 150.

Recommendation:
This race condition vulnerability was addressed by introducing two separate functions: `increaseAllowance` and `decreaseAllowance`. These functions allow users to increase or decrease allowances in a safe manner by specifying incremental changes, rather than resetting the value. For instance:

increaseAllowance: Allows the user to incrementally add tokens to the existing allowance, reducing the risk of resetting the allowance mid-transfer. decreaseAllowance: Allows the user to safely reduce the allowance by a specified amount, preventing inadvertent increases due to race conditions. By using these functions, the user avoids resetting the allowance to an arbitrary number, reducing the chances of unintended transfers due to race conditions.

Missing event emission in functions that set vital parameters

Location: VotingEscrow.sol, NetworkGuardian.sol, ProofOfIdentity.sol, Staking.sol

The functions in the provided contracts that set vital parameters do not emit events, which could result in a lack of transparency and tracking changes made by privileged addresses.

In VotingEscrow.sol

```
function set_wh1_address(address) external
function unlock() external
function remove_deposit_token(address addr) external
function add_deposit_token(address addr) external
```

In NetworkGuardian.sol

```
function setAssociation(address addr) external
function setController(address addr) external
```

In ProofOfIdentity.sol

```
function setAttributeCount(uint256 count) external
function setAttributeName(uint256 id, string calldata name) public
function setAttributeType(uint256 id, SupportedAttributeType attrType)
public
function incrementAttributeCount() public
```

In Staking.sol

```
function setRewardsDuration(uint256 duration_) external
function notifyRewardAmount(uint256 amount_) external
```

Recommendation:

It is recommended to modify the functions in the contracts to emit events after setting vital parameters. Emitting events will enhance transparency and allow for better tracking of parameter changes by privileged addresses, enabling better auditing and monitoring of the contract's state.

Minting and burning are allowed while the contract is paused

The `BackedHRC20.sol` smart contract implements a `_beforeTokenTransfer()` function which contains the following NatSpec comment: `The contract must not be paused if it is a mint or burn operation.`.

However, the implementation is checking if the contract is not paused for any operation except minting and burning operations:

```
if (from != address(0) && to != address(0)) {
            _requireNotGuardianPaused();
        }
```

It is possible that the desired behavior is to only check if the contract is paused for any operation except mint / burn operations.

Recommendation:
Change the check to ensure that the contract is not paused for minting / burning operations to fulfill the NatSpec comment:

```
if (from == address(0) || to == address(0)) {
            _requireNotGuardianPaused();
        }
```

In the case of wanting to check if the contract is paused for any operation except mint / burn, fix the NatSpec comment.

INFORMATIONAL-2 | RESOLVED

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. 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.

INFORMATIONAL-3 | RESOLVED

Reentrancy Vulnerability in recoverHRC20() and recoverAllHRC20() Functions

Location: EscrowedH1.sol

The functions recoverHRC20() and recoverAllHRC20() in the EscrowedH1.sol contract are exposed to reentrancy due to the fact that the interactions with external contracts are not protected by a reentrancy guard, and there is a potential risk of reentrancy.

Recommendation:
Modify the _recoverHRC20() function to by guarded from reentrancy by applying a reentrancy guard. Ensure that all checks are performed before any interactions with external contracts.

INFORMATIONAL-4 | RESOLVED

Spelling Typo Contained In _claim For The EscrowedH1 Contract

Recommendation:
In the Internal `_claim` function, `isFinised` should be defined as `isFinished` or `isFinalized` for greater readability.

INFORMATIONAL-5 | RESOLVED

Lack Of Error Messaging In _create_lock() Function For VotingEscrow

Recommendation:
It's recommended that sufficient error messaging is implemented when asserting that _value is greater than zero. Consider implementing an error message such as "_value is required to be greater than zero".

INFORMATIONAL-6 | ACKNOWLEDGED

If a malicious address gets approved or an approved one gets compromised, it will be able to transfer user's funds to the VotingEscrow

The `deposit_for` functions within the VotingEscrow.sol smart contracts allows any caller to deposit funds for other address just by setting the address as parameter. This action is protected by `_can_deposit_for_exn()`.

So only if userA executes `add_to_deposit_whitelist()` with userB's address will allow userB to deposit userA's funds.

However, we would like to remark that if userB is a malicious actor or a legit one that gets compromised he will be able to transfer userA's funds to the VotingEscrow.

Recommendation:
Do only approve accounts that are trusted by you or multisig accounts to ensure that they do not get compromised.

Client comment:
Acknowledged, thank you

INFORMATIONAL-7 | ACKNOWLEDGED

The WH1 address can be changed after the protocol is initialized

The address for the `WH1` token within `VotingEscrow.sol` can be changed by the administrator by executing `set_wh1_address`. The new address can be set at any point in time, meaning that its behavior may differ from the previous one.

Recommendation:
Do only allow setting WH1 one time and not being changed after.

INFORMATIONAL-8 | RESOLVED

Incorrect NatSpec comment

The `assertValidRange()` function in Array.sol the NatSpec comment says: End must be greater than the length. However, the implementation reverts if end is greater than the length.

Recommendation:
Fix the NatSpec comment.

INFORMATIONAL-9 | ACKNOWLEDGED

Some user's attributes may not correspond to the expected fields

The `issueIdentity()` function within the `ProofOfIdentity.sol` smart contract sets by default 4 user's attributes. These attributes are stored in the `expiries` array, in their first 4 positions.

However, the same smart contract implements `getCompetencyRating()`, `getNationality()` and `getIDIssuingCountry()` functions which return positions 4, 5 and 6 from the `expiries` array. These positions are by default empty, so the functions will return null. It is also important to highlight that the admin can set any new attribute to each position, so it is technically possible that on position 4 the nationality is set, instead of the competency rating, as expected, for example.

Recommendation:
Before adding a new attribute, do check if the position for the attribute is free or not and if it does correspond to the expected attribute type.

• contracts/governance/VotingEscrow.sol • contracts/h1-developed-application/H1DevelopedApplication.sol • contracts/h1-developed-application/lib/FnSig.sol • contracts/h1-native-application/H1NativeApplication.sol • contracts/h1-native-application/H1NativeApplicationUpgradeable.sol • contracts/h1-native-application/H1NativeBase.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/network-guardian/NetworkGuardian.sol** • **contracts/network-guardian/NetworkGuardianController.sol** • **contracts/network-guardian/lib/Array.sol** • **contracts/proof-of-identity/ProofOfIdentity.sol** • **contracts/proof-of-identity/lib/Attribute.sol** • **contracts/proof-of-identity/lib/BytesConversion.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/proof-of-identity/lib/POIType.sol** • **contracts/staking/Staking.sol** • **contracts/tokens/BackedHRC20.sol** • **contracts/tokens/EscrowedH1.sol** • **contracts/tokens/WH1.sol** • **contracts/utils/Address.sol** • **contracts/utils/upgradeable/BlacklistableUpgradeable.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