Quantstamp

Humanity Protocol

Executive Summary

This audit report was prepared by Quantstamp, the leader in blockchain security.

Type	NFT sales and Staking
Timeline	2026-03-12 through 2026-03-17
Language	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	None
Source Code	• humanity-org/node-sc #5cb63ef • humanity-org/humanity-sc #1402bc7
Auditors	• Ibrahim Abouzied Auditing Engineer • Andy Lin Senior Auditing Engineer • Yamen Merhi Auditing Engineer

Documentation quality	Medium
Test quality	Low
Total Findings	11 Fixed: 2 Acknowledged: 9
High severity findings	0
Medium severity findings	1 Fixed: 1
Low severity findings	6 Fixed: 1 Acknowledged: 5
Undetermined severity findings	0
Informational findings	4 Acknowledged: 4

Summary of Findings

The Humanity Node Sale is a set of smart contracts deployed on the Humanity blockchain that facilitates the sale and minting of NFTs. The sale features include tiered pricing, public and whitelist sales, referral rewards, random NFT reward draws using a commit-reveal scheme, and a lockup for NFT transfers.

400 tiers are supported, where each tier will have a unique price, quantity, and public/whitelist allocation. Public users can purchase NFTs at any tier, up to the maximum limit of their wallet. However, whitelisted users can exceed the wallet maximum through purchasing in both the public and whitelist sales. Users can optionally specify a referrer when making a purchase. Referrers receive a portion of the purchase price, depending on whether they are considered a direct or indirect referrer. Referrers can claim their rewards up to the amount they have purchased themselves. After the user is minted their NFT, they are restricted from transferring it for 12 months.

In addition the features above, the audit covered two new features: 1) Users can now stake native tokens with their NFT's to gain rewards. To unstake their tokens, users must submit a request, after which their tokens are claimable after a 14-day cooldown period. 2) Treasury reward distribution is committed on-chain through a merkle tree, from which license holders have their rewards distributed in proportion with their license tier.

The audit uncovered that referral accounting can permanently lock rewards if users' purchased amounts are not enough to claim the full referral rewards by the time the sale ends (HUM-1). Additionally, rewards are not bound to license ownership and modules may behave differently on NFT transfer. Distributions may be redirected to the current owner rather than the owner at the time of reward accrual (HUM-2), and users may unknowingly stake under someone else's license (HUM-3). The auditors have also attached suggestions for general code improvements.

Fix-Review Update: The Humanity team has addressed HUM-1 and HUM-3. They have additionally added a Timelock of 7 days for upgrades to `HStaking.sol` and `TreasuryDistribution.sol`. All other issues have been acknowledged.

ID	DESCRIPTION	SEVERITY	STATUS
HUM-1	**Referral Accounting Can Permanently Lock Reward Tokens After Sales End**	Medium	Fixed
HUM-2	**Transferred Licenses Can Redirect Previously Accrued Distributions**	Low	Acknowledged

ID	DESCRIPTION	SEVERITY	STATUS
HUM-3	`stake()` Does Not Verify that the Caller Owns the Target License	Low	Fixed
HUM-4	`requestUnStake()` Call Re-Lock Matured Funds	Low	Acknowledged
HUM-5	`buyFor()` Allows Third-Party Quota Griefing Impacting Referrals	Low	Acknowledged
HUM-6	Daily Emission Remainders Become Undistributable	Low	Acknowledged
HUM-7	Referral Pause Also Disables License Sales	Low	Acknowledged
HUM-8	`setStartTime(0)` Leaves Reward Distribution Unstarted	Informational	Acknowledged
HUM-9	`setWhitelistRootHash()` Can Invalidate Proofs Mid-Sale	Informational	Acknowledged
HUM-10	Late Tier Creation Inherits a Potentially Expired Sale Window	Informational	Acknowledged
HUM-11	Whitelist Purchases Bypass Recipient Wallet Accounting	Informational	Acknowledged

Assessment Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

Disclaimer

Only features that are contained within the repositories at the commit hashes specified on the front page of the report are within the scope of the audit and fix review. All features added in future revisions of the code are excluded from consideration in this report.

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

1. Code review that includes the following
    1. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
    2. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
    3. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:
    1. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
    2. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarity, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Scope

Files Included

Repo: `https://github.com/humanity-org/node-sc`

Included Paths: `contracts/`

Extensions: `sol`

Repo: `https://github.com/humanity-org/humanity-sc`

Included Paths: `contracts/`

Extensions: `sol`

Operational Considerations

1. `HMSale` contracts are deployed from the `HMSaleFactory` contracts using a beacon proxy. The documentation states that the beacon proxy owner will be renounced to make the contracts immutable. However, if the owner of the beacon contract is not renounced, they can upgrade the `HMSale` implementation contract, which could lead to unforeseen consequences, such as storage collisions, broken functionality, or loss of user funds. Any implementation contract upgrades should be audited and publicly announced prior to execution.
2. The `maxPerWallet` is currently only enforced on non-whitelisted sales. Sales purchased through the whitelist use a separate allocation mechanism via a Merkle tree. If a user has a non-zero allocation in the Merkle tree, their token count can exceed `maxPerWallet`. Additionally, if `maxPerWallet` is reassigned to a smaller value, it will not apply retroactively to previous sales, and there may be users whose wallets exceed `maxPerWallet`.
3. `paymentToken()` should be a stablecoin to ensure the Node licenses are sold at the target price.
4. Users can only assign a referrer once. If future `buy()` or `buyWithWhitelist()` include a new `referrer_`, the contracts will silently ignore the field and continue directing referral rewards to the existing referrer.
5. Whitelisted addresses can purchase from the public pool, even if the whitelisted pool has not been depleted.
6. `TreasuryDistribution.commitMerkleRoot()` assumes `COMMITTER_ROLE` publishes a Merkle root and `tierLicenseCounts_` that are internally consistent, even though the on-chain proof path in `TreasuryDistribution._validateAndRecord()` only verifies membership for `(day, licenseId, tier, contract, chainId)` and does not independently prove that the published per-tier counts match the committed leaf set. A concrete consequence is that the committed root may contain licenses for a tier that was omitted from `tierLicenseCounts_`; proofs for those licenses can still validate, but payout then reverts with `TierRewardNotConfigured` because no allocation was configured for that tier. This is an explicit trusted-operator assumption around payout correctness, not a trustless integrity guarantee.
7. The system relies on the configured paymentToken being a standard non-deflationary ERC20 and on the configured treasury being able to receive both ERC20 tokens and native currency. Fee-on-transfer or otherwise non-standard tokens can break sale/referral accounting, and a misconfigured or reverting treasury can cause purchases to fail.
8. `HMKyc.setKycBatch()` and `HMKyc.revokeKycBatch()` control whether already-accrued referral rewards are currently claimable because rewards accrue in `HMReferral.onPurchased()`, but payout is gated later in `HMReferral.claim()`. The assumption is that `KYC_ROLE` remains a trusted live-compliance authority that may temporarily or indefinitely block withdrawals without clearing the recorded reward balance.
9. The sale architecture relies on the configured treasury address being correct, reachable, and able to receive native currency. Since HMSale._buy() forwards the platform fee using a native call to treasury, a reverting or misconfigured treasury can cause purchases to fail.
10. `HMReferral.pause()` / `HMReferral.unpause()` effectively act as sale-wide operational controls because `HMSale._buy()` unconditionally routes purchase settlement through `HMReferral.onPurchased()`. The assumption is that `DEFAULT_ADMIN_ROLE` is trusted to pause the referral subsystem only when it is acceptable for the sale path to fail closed, and that operators understand this blast radius alongside the explicit sale pause in `HMSaleConfig.pause()`.

Key Actors And Their Capabilities

`HMSaleFactory.sol`

Default Admin

- Has the ability to grant any role to any address. By design, `HMSale` contracts deployed by the `HMSaleFactory` contract are granted the `SALE_CONTRACT_ROLE`. However, if the `DEFAULT_ADMIN_ROLE` grants the `SALE_CONTRACT_ROLE` to an arbitrary address, invalid event emission and execution of mint requests are possible. Alternatively, if the `DEFAULT_ADMIN_ROLE` revokes the `SALE_CONTRACT_ROLE` from a valid `HMSale` contract, nearly all functionality in the `HMSale` contract breaks.
- Create a unique tier sale, where the tier is 0-400 inclusive. By creating a tier sale, a new `HMSale` contract is deployed, and the initial price, public sale amount, and whitelist sale amount are set.
- Updates the maximum purchased amount per wallet via `setMaxPerWallet()`, applicable to all current and future `HMSale` contracts.
- Extends the end time by at most 30 days via `extendEndTime()`, applicable to all current and future `HMSale` contracts.
- Updates the treasury address via `setTreasury()`, applicable to all current and future `HMSale` contracts.
- Pause the contract, preventing public and whitelist purchases in all `HMSale` contracts.
- Unpause the contract, allowing public and whitelist purchases in all `HMSale` contracts.

- Withdraw any ETH or ERC20 tokens from the `HMSaleFactory` contract to the Treasury.
- Transfer or renounce the `DEFAULT_ADMIN_ROLE` in a two-step process after the specified time delay of 1 day. If the role is renounced, all capabilities listed above are no longer available.

Sale Operator

- Set and update the whitelist root hash in any `HMSale` contract.
- Set and update the sale price in any `HMSale` contract before the sale window begins.
- Set and update the public and whitelist sale amounts in any `HMSale` contract before the sale window begins.
- Release the whitelist allocated sales to be purchasable by the public.
- Pause any `HMSale` contract, preventing public and whitelist purchases.
- Unpause any `HMSale` contract, allowing public and whitelist purchases.
- Withdraw any ETH or ERC20 tokens from any `HMSale` contract to the Treasury.

`HMSale.sol`

Public Users

- Buy and request NFT mints up to the maximum public purchase amount per wallet and sale amount for the given tier. Public purchases must occur during the sale window set in the `HMSaleFactory` contract.

Whitelisted Users

- Buy and request NFT mints up to the maximum whitelist purchase amount per wallet and sale amount for the given tier with a valid proof. Whitelist purchases must occur during the sale window set in the `HMSaleFactory` contract.

`HMNodeLicenseNFT.sol`

Default Admin

- Has the ability to grant the `MINTER_ROLE` to any address. While the design is intended to grant the `MINTER_ROLE` only to the `HMSaleFactory` contract, the `DEFAULT_ADMIN_ROLE` can also grant the role to other addresses, allowing them to freely mint NFTs at any tier and node type. Alternatively, the `DEFAULT_ADMIN_ROLE` can revoke the `MINTER_ROLE` from the `HMSaleFactory` contract, preventing users from claiming and minting their purchased NFTs.
- Withdraw any ETH or ERC20 tokens from any `HMNodeLicenseNFT` contract to themself.
- Transfer or renounce the `DEFAULT_ADMIN_ROLE` in a two-step process after the specified time delay of 1 day. If the role is renounced, all capabilities listed above are no longer available.

`HMReferral.sol`

Default Admin

- Update the direct and indirect referral rates.
- Initially set the `HMSaleFactory` contract.
- Withdraw any ETH or ERC20 tokens from any `HMReferral` contract to the treasury.
- Transfer or renounce the `DEFAULT_ADMIN_ROLE` in a two-step process after the specified time delay of 1 day. If the role is renounced, all capabilities listed above are no longer available.

Referrers

- Claim referral rewards as they are accumulated.

`TreasuryDistribution.sol`

Default Admin

- Has the ability to grant any role to any address. In particular, if the `DEFAULT_ADMIN_ROLE` grants the `DISTRIBUTOR_ROLE` to an arbitrary address, that address can execute valid merkle-based distributions for any committed day and transfer treasury source tokens to the current owner of the corresponding license IDs. If the `DEFAULT_ADMIN_ROLE` grants the `COMMITTER_ROLE` to an arbitrary address, that address can commit merkle roots and configure per-tier daily reward allocations for any valid day that is not in the future. Additionally, the `DEFAULT_ADMIN_ROLE` can revoke either role from valid operators, preventing those privileged functions from being performed.
- Update the treasury contract via `setTreasury()`. This changes the contract used to send source tokens for all future distributions.
- Set the distribution start time once via `setStartTime()`. The start time cannot be changed after it is set, and it must not be in the future.
- Set the tier multiplier for any tier from 1-5 inclusive via `setTierMultiplier()`. This affects how future daily reward allocations are calculated when merkle roots are committed.
- Pause the contract, preventing merkle root commitments and token distributions.
- Unpause the contract, allowing merkle root commitments and token distributions.
- Upgrade the contract implementation.
- Withdraw any ERC20 tokens from the `TreasuryDistribution` contract to themself. Native token withdrawal is also authorized through `Withdrawable` unless restricted by the inherited implementation.
- Transfer or renounce the `DEFAULT_ADMIN_ROLE` in a two-step process after the specified time delay of 1 day. If the role is renounced, all capabilities listed above are no longer available.

Distributor

- Distribute rewards for a committed day to the current owner of a license ID via `distribute()`, provided a valid merkle proof is supplied and the license has not already claimed for that day.
- Batch distribute rewards for up to 100 license IDs at once via `batchDistribute()`, provided valid merkle proofs are supplied and the day's cap is not exceeded.
- Cause treasury source tokens to be sent to the current owner of each successfully validated license ID as part of reward distribution.

Committer

- Commit a merkle root for any valid day that is not in the future via `commitMerkleRoot()`.
- Configure the day's tier allocations when committing a merkle root by supplying license counts for up to 5 distinct tiers, which determines the per-tier reward per license and total daily tier allocation.
- Permanently set the merkle root for a given day, as each day's root can only be committed once.

`HStaking.sol`

Default Admin

- Pause the contract, preventing users from staking, requesting unstake, and claiming unstaked amounts.
- Unpause the contract, allowing users to stake, request unstake, and claim unstaked amounts.
- Upgrade the contract implementation.
- Withdraw any ERC20 tokens from the `HStaking` contract to themself. Native token withdrawal is disabled because `_getBalance()` is overridden to always return 0.
- Transfer or renounce the `DEFAULT_ADMIN_ROLE` in a two-step process after the specified time delay of 1 day. If the role is renounced, all capabilities listed above are no longer available.

Public Users

- Stake ETH against any minted license ID via `stake()`. The caller does not need to own the license NFT; the function only requires that the license ID exists.
- Request an unstake of a portion of their staked ETH for a given license ID via `requestUnStake()`, provided they have sufficient staked balance recorded for that license ID.
- Claim previously requested unstaked ETH via `claim()` after the 14 day cooldown has elapsed.
- Query the set and count of license IDs they currently have associated stake under via `getUserStakedLicenses()` and `getUserStakedLicensesCount()`.

Findings

HUM-1

Referral Accounting Can Permanently Lock Reward Tokens After Sales End

Medium | Fixed

Update

Marked as "Fixed" by the client.

Addressed in: `4b092f57cb67bdd2c73703e1d0544185ab315104`, `c48838515ee7f13d96658fdc77797e60ebe6ada8`, and `ee6803b36e58ee1c8bbfa8ca8e77afcad6d89998`.

The client provided the following explanation:

> Fixed per recommendation.

File(s) affected: `HMReferral.sol`

Description: `HMReferral` tracks reserved payment tokens using the global value `totalRewardAmount − totalClaimedAmount`, and `_getTokenBalance()` excludes that entire amount from admin withdrawals.

However, users cannot claim based on rewardAmount alone. In `claim()`, each user's effective reward is capped to:

`min(userInfo.rewardAmount, userInfo.purchasedAmount)`

This means the contract reserves rewards according to the full accumulated `rewardAmount`, while actual user entitlement is limited by `purchasedAmount`. As a result, the reserved amount can exceed the amount that is ever claimable.

For example, a referrer can accumulate referral rewards from other users' purchases without making enough purchases themselves. In that case, their `rewardAmount` increases, but `claim()` only allows up to `purchasedAmount` to be withdrawn. The excess remains included in `totalRewardAmount`, and because `_getTokenBalance()` subtracts `totalRewardAmount − totalClaimedAmount` from the contract balance, the admin cannot withdraw that excess either.

This issue becomes permanent once the sales end. `purchasedAmount` is only increased in `onPurchased()`, which is only invoked by sale contracts when purchases occur. After the sale period is over, affected referrers can no longer increase their `purchasedAmount` to unlock the excess. Any portion of rewards above their final purchase cap therefore becomes permanently unclaimable by users while also remaining permanently reserved against admin withdrawals.

As a result, payment tokens can become permanently locked inside HMReferral.

Recommendation: Introduce a `totalEffectiveRewardAmount` counter that tracks only the actually-claimable portion of rewards (`min(rewardAmount, purchasedAmount)` per user), and use it in `_getTokenBalance()` instead of `totalRewardAmount`. Maintain it with O(1) updates:

- **On reward accrual** (`_updateDirectReferral()` / `_updateIndirectReferral()`): increment `totalEffectiveRewardAmount` by the portion of the new reward that falls under the purchase cap: `min(newRewardAmount, purchasedAmount) − min(oldRewardAmount, purchasedAmount)`. This is zero when the referrer is already fully capped.
- **On purchase** (`onPurchased()`): increment `totalEffectiveRewardAmount` by the portion of rewards newly unlocked by the purchase: `min(newPurchasedAmount, rewardAmount) − min(oldPurchasedAmount, rewardAmount)`.
- **Reserve formula** becomes `totalEffectiveRewardAmount − totalClaimedAmount`, which always equals `Σ min(rewardAmount, purchasedAmount) − claimedAmount` across all users — matching what is actually claimable.

HUM-2
Transferred Licenses Can Redirect Previously Accrued Distributions

Low | Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

> Rewards transfers with NFT to new owner if seller does not withdraw.

File(s) affected: `contracts/TreasuryDistribution.sol`

Description: The `TreasuryDistribution` contract distributes daily rewards for a license through `distribute()` and `batchDistribute()`, with eligibility proven by a Merkle root committed in `commitMerkleRoot()`. However, the payout recipient is not bound into the Merkle leaf and is instead resolved at distribution time in `_validateAndRecord()` using `licenseNFT.ownerOf(licenseId_)`.

As a result, a license can accrue rewards for a given `day_` while owned by one user, then be transferred before `distribute()` is called, causing the reward to be paid to the new owner instead of the owner who accrued it. The root cause is that the leaf only commits to `(day_, licenseId_, tier_, address(this), block.chainid)`, while the receiver is fetched later from the live NFT ownership state. This can misdirect rewards whenever there is a delay between reward accrual and reward distribution.

Recommendation: Bind the intended payout recipient into the reward commitment instead of resolving it from current NFT ownership during distribution. This could be done by including the receiver address in the Merkle leaf and passing it into `distribute()` / `batchDistribute()`.

Alternatively, consider allowing users to claim their distributions themselves rather than awaiting for the `DISTRIBUTOR_ROLE`. This will allow them to claim pending rewards before their transfer.

HUM-3
`stake()` Does Not Verify that the Caller Owns the Target License

Low | Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `6a27a514110673ce14ed4601ea2111c194e9862b`.
The client provided the following explanation:

> Fixed per recommendation.

File(s) affected: `contracts/HStaking.sol`

Description: The `HStaking.stake()` function uses `licenseNft.ownerOf(licenseId)` to confirm that a license exists, but it does not verify that the returned owner matches `msg.sender`. As a result, any user can call `stake()` for any minted `licenseId`, and the stake will be recorded under `userLicenseInfos[msg.sender][licenseId]` even if the caller does not own that NFT.

There is no economic incentive for a user to do this intentionally. However, a user mistakenly staking against another user's license will pass silently, leading to their rewards being misdirected until they notice the issue themselves.

Recommendation: In `HStaking.stake()`, validate that the owner returned by `licenseNft.ownerOf(licenseId)` is `msg.sender` before accepting the deposit.

HUM-4 `requestUnStake()` Call Re-Lock Matured Funds

Low | Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> > As designed, system is designed for simple staking/unstaking.

File(s) affected: `contracts/HStaking.sol`

Description: The `HStaking.requestUnStake()` function allows each user to maintain only a single pending unstake position per `licenseId`. When a user calls `requestUnStake()` again before claiming, the function adds the new amount to `unStakeAmount` and unconditionally overwrites `withdrawableFrom` with `block.timestamp + UNSTAKE_COOLDOWN`.

As a result, a user cannot preserve the original unlock time for previously requested withdrawals. This creates transaction order dependence between `requestUnStake()` and `claim()`, where calling `claim()` before `requestUnstake()` behaves as expected, but calling `requestUnStake()` before `claim()` can unexpectedly delay access to funds by another full cooldown period.

Even if the protocol intentionally prefers a simple unstaking model, the current behavior harms usability and can delay user access to funds in a way that is unrelated to market incentives. Unstaking should be disincentivized through the length of the `UNSTAKE_COOLDOWN` period and missed staking rewards rather than the transaction order dependence of `requestUnstake()` and `claim()`.

Recommendation: Consider separating pending unstake requests so that newly requested withdrawals do not modify the unlock time of older ones.

Alternatively, automatically claim any pending unstakes in `requestUnstake()` if the `UNSTAKE_COOLDOWN` has elapsed, and inform users that calling `requestUnstake()` before the current cooldown period is over will reset the cooldown.

HUM-5
`buyFor()` Allows Third-Party Quota Griefing Impacting Referrals

Low | Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> > No incentive for griefing as attacker would have negative returns.

File(s) affected: `contracts/HMSale.sol`

Description: In `HMSale.buyFor`, any caller can purchase licenses for an arbitrary `recipient_`, and `_buyFor` enforces `maxPerWallet` against `userPublicPurchasedAmounts[recipient_]` rather than against the caller. As a result, a third party can consume another wallet's public-sale quota by buying on its behalf.

This allows an attacker to intentionally send locked licenses to a victim and push the victim to the configured wallet cap. Once that happens, the victim's later direct buy calls will revert with `ExceededMaxPerWallet`, even though the victim never chose to consume their own allocation. Since the minted licenses are also subject to the default transfer lock, the victim cannot simply dispose of them immediately to restore flexibility.

This could lead to targeted griefing and forced exclusion from the public sale for specific wallets. Mainly making the victim lose the chance to use their own preferred referral path.

Recommendation:

- Consider restricting `buyFor` to self-purchases only, or requiring explicit recipient authorization before a third party can buy on someone else's behalf.
- Consider enforcing `recipient_` == `msg.sender` for the unrestricted public path.
- Consider introducing an approval or signature-based gifting flow if third-party purchases are intended.

HUM-6 Daily Emission Remainders Become Undistributable

Low | Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> > Acceptable rounding remainder.

File(s) affected: `contracts/TreasuryDistribution.sol`

Description: In `TreasuryDistribution.commitMerkleRoot()`, the contract computes each tier's `rewardPerLicense` using integer division and stores immutable `dayTierAllocations[day][tier]` values derived from that per-license amount. The intended flow is to split the daily cap returned by `_getCap(day)` across eligible licenses according to tier multipliers.

However, because each tier's per-license reward is rounded down independently, the sum of all tier allocations can be strictly lower than the day cap. Once the merkle root and tier allocations are committed for that day, the undistributed remainder cannot be reassigned because the root is immutable and rewards are fixed by `rewardPerLicense`. As a result, part of the scheduled daily emission becomes permanently unreachable.

This could lead to systematic under-distribution over time, drift between advertised emissions and actual payouts, and treasury balances that remain stranded relative to the intended release schedule. The mismatch may be small per day but can accumulate across many days and periods.

Recommendation:

- Add explicit remainder handling during root commitment, such as assigning the leftover amount to one tier, one designated receiver, or a deterministic balancing rule so the full daily cap is represented.
- If the protocol intentionally accepts under-distribution due to rounding, expose the allocated total and the discarded remainder so operators can account for it transparently.

HUM-7 Referral Pause Also Disables License Sales

Low Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

> Emergency pause will pause referral and sale together.

File(s) affected: `contracts/HMSale.sol`, `contracts/HMReferral.sol`

Description: `HMSale._buy()` unconditionally calls `config().referral().onPurchased(...)` for every purchase, even when `referer_` is the zero address and no referral should ultimately be paid. In `HMReferral`, `onPurchased` is protected by `whenNotPaused`, while pause control is held independently by the referral admin. The intended flow appears to be that pausing the referral subsystem stops referral processing and claims.

However, because every sale purchase must successfully pass through `HMReferral.onPurchased()`, pausing the referral contract halts all license purchases system-wide rather than just referral accrual. There is no fallback path in `HMSale` for purchases without a referrer, and no graceful handling of a paused referral module. A subsystem that looks ancillary becomes a hard dependency for the core sale path.

This could lead to unexpected full-sale outages if operators pause the referral contract to investigate a referral issue, stop claims, or perform maintenance. In theory, the admin which is able to pause is shared across modules, but there is no composability allowing to pause the referral module without pausing the whole sale.

Recommendation: Check whether this behavior is desired and consider documenting it. If not, redesign the module to allow the sale to continue while the referral module is paused. Note: This implies changing the architecture as tracking the purchased amounts while the referral was paused will be harder.

HUM-8 `setStartTime(0)` Leaves Reward Distribution Unstarted

Informational Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

> Admin operational error that would require intentional misconfiguration.

File(s) affected: `node-sc/contracts/TreasuryDistribution.sol`

Description: `TreasuryDistribution.setStartTime()` accepts `0`, even though `currentDay()` treats `startTime == 0` as the unset sentinel. This allows the contract to emit a successful start event while reward publication and distribution remain blocked until a later non-zero start time is set.

Exploit Scenario:

1. A trusted admin calls `setStartTime(0)`.
2. The contract emits a success event and stores `startTime = 0`.
3. Later reward operations that depend on `currentDay()` continue to behave as though start time was never initialized.
4. The system remains paused until the admin sets a valid non-zero start time.

Recommendation:

- Reject `startTime_ == 0` in `setStartTime()`, or
- use a separate initialized flag instead of overloading `0` as both a valid input and an unset sentinel.

HUM-9 `setWhitelistRootHash()` Can Invalidate Proofs Mid-Sale

Informational | Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

> Admin flexibility to update whitelist mid-sale is intentional design, operational responsibility to communicate changes.

File(s) affected: `contracts/HMSale.sol`

Description: In `HMSale`, `setPrice()` and `setSaleAmount()` are restricted by `onlyPreSale`, but `setWhitelistRootHash()` is callable by `onlySaleOperator` at any time. This means the active whitelist merkle root can be replaced even while the sale is live and users are already submitting `buyWithWhitelist()` transactions.

Because `buyWithWhitelist` and `buyWithWhitelistFor` verify proofs against the current `whitelistRootHash`, changing the root immediately invalidates previously valid proofs. A user may prepare a valid proof and submit a purchase transaction, only for that transaction to revert if the operator updates the root before it is mined.

This could lead to selective denial of whitelist purchases, inconsistent user experience, and privileged mid-sale control over which proofs remain usable. While this requires a sale operator, it still creates unnecessary operational and fairness risk during an active sale.

Recommendation: Consider adding `onlyPreSale` to `setWhitelistRootHash`.

HUM-10
Late Tier Creation Inherits a Potentially Expired Sale Window

Informational | Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

> Creating tiers after sale window is admin operational mistake with obvious consequence, not a security issue.

File(s) affected: `contracts/HMSaleFactory.sol`, `contracts/base/HMSaleConfig.sol`, `contracts/HMSale.sol`

Description: `HMSaleFactory` stores a single immutable `startTime` and mutable `endTime` in `HMSaleConfig`, and every `HMSale` proxy reads those same values through `config()`. `createTierSale()` can be called at any time by a sale operator and initializes the new tier sale immediately with its own price and amounts.

However, tier sales do not have an independent sale window and there is no check that a new tier is created before the global sale has started or before it has ended. A tier created late inherits whatever portion of the shared window remains at that moment, which can mean opening instantly with no setup buffer or being created after the sale is already over and therefore being permanently unusable.

This could lead to operator mistakes where a newly created tier launches with unintended timing, loses its pre-sale configuration period, or is deployed into an already expired sale window and can never sell at all.

Recommendation:

- Enforce tier creation before the global sale start, or give each tier sale its own explicit activation window.
- Add a guard in createTierSale that rejects creation after the shared sale window has begun or ended, depending on the intended operating model.
- Alternatively, if the behavior is desired consider documenting it.

HUM-11
Whitelist Purchases Bypass Recipient Wallet Accounting

Informational | Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

> By design to allow whitelist wallets to exceed limit.

File(s) affected: `contracts/HMSale.sol`

Description: `HMSale.buyWithWhitelistFor` allows a whitelisted caller to mint licenses to an arbitrary `recipient_`. The whitelist proof and allocation are verified against `msg.sender`, and `_buy` records the purchase in `userWhitelistPurchasedAmounts[msg.sender]`. By contrast, public purchases are tracked against the actual `recipient_`, and `getUserPurchasedAmount` presents a combined per-user purchase view using `userPublicPurchasedAmounts[user]` + `userWhitelistPurchasedAmounts[user]`.

However, whitelist purchases made for another wallet are never recorded against the receiving wallet. This means the recipient's tracked purchase total is understated, and any sale logic or off-chain process that treats `getUserPurchasedAmount` as the buyer's effective acquisition history becomes inaccurate. It also allows multiple whitelisted accounts to funnel their allocations into one recipient wallet without that wallet's local accounting reflecting the accumulated whitelist mints.

This could lead to concentration-limit bypasses if downstream logic relies on the per-user counters.

Recommendation:

- Decide whether whitelist limits are intended to constrain the paying whitelist account, the receiving wallet, or both, and record purchases consistently with that rule.
- Alternatively, if the behavior is desired consider documenting it.

Auditor Suggestions

S1 `Withdrawable.withdraw()` Double-Reads `_getBalance()`

Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

> Acknowledged.

File(s) affected: `humanity-sc/contracts/base/Withdrawable.sol`, `node-sc/contracts/base/Withdrawable.sol`

Description: `Withdrawable.withdraw()` reads `_getBalance()` twice: once into a cached `balance` variable used for the event, and once again inline in the native-token transfer call. The two reads should not differ in practice because `nonReentrant` is active and no external call occurs between them, but the pattern is wasteful and can create event/value mismatch risk if the function is later refactored. Both the `humanity-sc` and `node-sc` copies of Withdrawable have this same pattern.

Recommendation: Reuse the cached `balance` variable in the transfer call: `call{value: balance}("")`.

S2 Align `userOf()` Behavior with ERC-4907 Reference Implementation

Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

> Acknowledged.

File(s) affected: `humanity-sc/contracts/erc4907/ERC4907.sol`

Description: `ERC4907.userOf()` currently calls `_requireOwned(tokenId)`, which causes `userOf(nonexistentTokenId)` to revert. The ERC-4907 reference implementation does not perform an existence check in `userOf()` and instead returns `address(0)` when there is no active user. This is not a clear spec violation, but it is a compatibility divergence from the reference behavior and from what integrators may expect.

Recommendation:

- Remove `_requireOwned(tokenId)` from `userOf()` so nonexistent tokens return `address(0)`;
- keep `setUser(nonexistent)` reverting as-is;
- add regression tests that `userOf(nonexistent)` returns `address(0)` and `userExpires(nonexistent)` returns `0`;
- if any off-chain component needs existence semantics, use `ownerOf()` rather than `userOf()`.

S3 Reinitializer Guard for Reentrancy State Is Missing

Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> > Acknowledged.

File(s) affected: `contracts/HMSale.sol`

Description: In `HMSale`, the constructor disables initializers and the proxy setup is performed through initialize, which calls `__Pausable_init()` and then sets tier, factory, price, and sale amounts. The purchase entrypoints buy, buyFor, buyWithWhitelist, and buyWithWhitelistFor all rely on nonReentrant from ReentrancyGuard.

However, this file inherits the non-upgradeable ReentrancyGuard rather than the upgradeable variant, and initialize does not perform any explicit initialization for that guard state.

Recommendation: Use the upgradeable reentrancy guard variant and initialize it explicitly in initialize.

S4 Unused Errors Indicate Stale Validation Paths

Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> > Acknowledged.

File(s) affected: `contracts/HMSale.sol`

Description: `HMSale` declares several custom errors including `ExceededSaleAmount`, `InvalidTier`, `ExceededPurchasedAmount`, and `NotEnded`, but the executable functions in this file do not use them.

Recommendation: Remove unused custom errors.

S5 Hardhat-Specific Tier Logic Signals Non-Production Configuration

Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> > Acknowledged.

File(s) affected: `contracts/HMSaleFactory.sol`

Description: `HMSaleFactory` contains a hardcoded `HARDHAT_CHAINID` check in the constructor and assigns `MAX_TIER = 6` when `block.chainid == 31337`. This introduces explicit development-environment behavior directly into production contract logic.

Recommendation: Remove the Hardhat-specific branch

S6 Upgradeable Contracts Do Not Reserve Storage Gaps

Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> > Acknowledged.

File(s) affected: `contracts/HStaking.sol`, `contracts/TreasuryDistribution.sol`

Description: `HStaking` and `TreasuryDistribution` are upgradeable contracts, but neither defines a storage gap such as `uint256[N] private __gap` to reserve storage slots for future upgrades. This reduces flexibility for safely evolving storage layout over time.

Recommendation: Add storage gap reservations to the upgradeable contracts to reduce future storage layout risk during upgrades or consider using namespaced pattern.

S7 `requestUnStake()` Cannot Be Reversed Once Initiated

Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> > Acknowledged.

File(s) affected: `contracts/HStaking.sol`

Description: The contract does not provide any mechanism for a user to cancel a pending unstake request and restore those funds to the staked balance before the cooldown expires.

If the user changes their mind shortly after calling `requestUnStake()`, they must still wait through the full cooldown, claim the funds, and then call `stake()` again. This creates unnecessary friction and can leave user funds idle for the cooldown period even when the user would prefer to continue participating in staking.

Recommendation: Add a function that allows users to move some of their `UnStakeRequested.totalPendingAmount` back to `UnStakeRequested.addedAmount`.

S8 Push-Based Distributions Create a Centralized Claim Bottleneck

Acknowledged

> **Update**
> Marked as "Acknowledged" by the client.
> The client provided the following explanation:
>
> > Acknowledged.

File(s) affected: `contracts/TreasuryDistribution.sol`

Description: The `TreasuryDistribution` contract pays rewards through the privileged `distribute()` and `batchDistribute()` functions, both of which can only be called by an address with `DISTRIBUTOR_ROLE`.

As a result, reward delivery depends entirely on the distributor to actively process payouts for users. If the distributor is delayed, selective, offline, or operationally constrained by `MAX_BATCH_SIZE`, rewards remain claimable in theory but inaccessible in practice.

A pull-based design is markedly more gas-efficient, since it avoids expensive attempts to distribute rewards to multiple participants in a single operation. Although it places the burden of claiming on the user, that tradeoff is usually well worth the gains in scalability and operational simplicity.

Reward allocation is sufficiently privileged given that the Merkle root has already been committed in `commitMerkleRoot()` by the `COMMITTER_ROLE` and all information needed to prove entitlement is available onchain. This should allow removing the `DISTRIBUTOR_ROLE` from `distribute()` and `batchDistribute()` without consequence.

Recommendation: Replace the push-based payout flow with a pull-based claim model that allows the eligible recipient to submit their own Merkle proof and claim directly. If a distributor-assisted flow is still desired, consider supporting both models by keeping batched distributions as an optional convenience while also exposing a permissionless self-claim path.

Definitions

- **High severity** – High-severity issues usually put a large number of users' sensitive information at risk, or are reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
- **Medium severity** – Medium-severity issues tend to put a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or are reasonably likely to lead to moderate financial impact.
- **Low severity** – The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.

- **Informational** – The issue does not pose an immediate risk, but is relevant to security best practices or Defence in Depth.
- **Undetermined** – The impact of the issue is uncertain.
- **Fixed** – Adjusted program implementation, requirements or constraints to eliminate the risk.
- **Mitigated** – Implemented actions to minimize the impact or likelihood of the risk.
- **Acknowledged** – The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).

Appendix

File Signatures

The following are the SHA-256 hashes of the reviewed files. A file with a different SHA-256 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different SHA-256 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

Files

Repo: `https://github.com/humanity-org/node-sc`

- `a6e...23c ./contracts/HStaking.sol`
- `27c...a2e ./contracts/TreasuryDistribution.sol`
- `b41...38e ./contracts/base/Withdrawable.sol`
- `1ed...dd4 ./contracts/interfaces/ITreasury.sol`

Repo: `https://github.com/humanity-org/humanity-sc`

- `b20...5dd ./contracts/HMKyc.sol`
- `ff9...fed ./contracts/HMNodeLicenseNFT.sol`
- `96f...098 ./contracts/HMReferral.sol`
- `159...99d ./contracts/HMSale.sol`
- `1a0...55e ./contracts/HMSaleFactory.sol`
- `aef...79b ./contracts/base/HMSaleConfig.sol`
- `dda...a70 ./contracts/base/HMSaleEvent.sol`
- `bd7...e05 ./contracts/base/HMSaleMinter.sol`
- `ac0...441 ./contracts/base/Withdrawable.sol`
- `eb3...cb3 ./contracts/erc4907/ERC4907.sol`
- `aaa...e5f ./contracts/erc4907/IERC4907.sol`
- `956...f9b ./contracts/interfaces/IHMKyc.sol`
- `6df...e54 ./contracts/interfaces/IHMReferral.sol`
- `776...40b ./contracts/interfaces/IHMSale.sol`
- `3b6...48d ./contracts/interfaces/IHMSaleConfig.sol`
- `1bd...135 ./contracts/interfaces/IHMSaleEvent.sol`
- `b5c...f89 ./contracts/interfaces/IHMSaleFactory.sol`
- `70c...90b ./contracts/interfaces/IHMSaleMinter.sol`
- `87c...83b ./contracts/interfaces/IHmNodeLicenseNFT.sol`
- `054...6e4 ./contracts/libraries/Constants.sol`

Test Suite Results

The following test suite results were gathered by running `npx hardhat test`.

We were unable to gather code coverage results. We strongly advise the Humanity team to add instrumentation for code coverage.

```
HUMANITY-SC
  ERC4907
    ✔ should set user to Bob (657ms)
    ✔ Approved spender can set user (76ms)
```

```
  Supports Interface
    ✔ Supports IERC4907 (118ms)

Factory
  Create Tier
    ✔ Hardhat max tier is 6 (1009ms)
    ✔ Create Tier Sale (303ms)
    ✔ Can not create zero sale amount (264ms)
    ✔ Can not create same tier twice (258ms)
    ✔ Can not create tier 0 (257ms)
    ✔ Can not exceed max tier (259ms)
    ✔ Time check: start time must be in the future (266ms)
    ✔ Time check: endTime <= startTime (236ms)
    ✔ Time check: newEndTime < endTime (239ms)
    ✔ Time check: newEndTime > endTime + MAX_TIME_EXTENSION (237ms)
    ✔ Time check: OK (232ms)
    ✔ Max per wallet check (244ms)
    ✔ Treasury check (239ms)
    ✔ Platform fee check (243ms)
  Withdraw
    ✔ Only admin can withdraw (240ms)
    ✔ Withdraw Token (249ms)

HMKyc
  Set KYC
    ✔ Only admin can set KYC (951ms)
    ✔ Can set KYC (294ms)
    ✔ Can revoke KYC (267ms)
  Withdraw
    ✔ Only admin can withdraw (505ms)
    ✔ Withdraw Token (517ms)

Mint
  Mint direct
    ✔ Can not mint tier 0 (927ms)
    ✔ Can not mint tier > 400 (327ms)

HMNodeLicenseNFT
  Mint
    ✔ Invalid tier (782ms)
    ✔ Invalid node type (309ms)
    ✔ Mint (259ms)
    ✔ Mint untransferable (269ms)
    ✔ Mint OG NFT (265ms)
    ✔ Mint Founder NFT (261ms)
    ✔ Support ERC4907 (256ms)
  Token URI
    ✔ Nonexistent token (494ms)
    ✔ Token json metadata (485ms)
    ✔ Token svg image (513ms)
    ✔ OG NFT (494ms)
    ✔ ounder NFT (461ms)
  supportsInterface
    ✔ ERC721 (417ms)
    ✔ AccessControl (418ms)
  Withdraw
    ✔ Only admin can withdraw (421ms)
    ✔ Withdraw Token (437ms)

Referral
  Check
    ✔ Non-sale contract can not call onPurchased (938ms)
    ✔ Zero factory can not set factory (315ms)
    ✔ Factory already set can not set factory (274ms)
  Refer math
    ✔ Can not self referrer (262ms)
    ✔ Can not referrer loop (272ms)
    ✔ Can not referrer loop 5 levels (304ms)
    ✔ Referral Direct (277ms)
    ✔ Referral Indirect (270ms)
  Claim
    ✔ Only KYC user can claim (250ms)
```

```
    ✔ Can not claim when paused (253ms)
    ✔ Claim (338ms)
  Withdraw
    ✔ Only admin can withdraw (277ms)
    ✔ Withdraw Token (259ms)
    ✔ Withdraw token but remain for claim (282ms)
    ✔ Non-payment token can withdraw all (269ms)

Sale
  Initialize
    ✔ Can not reinitialize (981ms)
  Setters
    ✔ Price: can not set zero (303ms)
    ✔ Price: can set (269ms)
    ✔ Sale amount: saleAmount_ must > whitelistSaleAmount_ (264ms)
    ✔ Sale amount: can set (273ms)
  Global Pause
    ✔ Paused view function (263ms)
    ✔ Can not buy when paused (270ms)
    ✔ Unpause check (256ms)
  Check Buy Before start
    ✔ Cannot buy before start (251ms)
  Check Buy After start
    ✔ Cannot buy with invalid platform fee (246ms)
    ✔ Cannot buy after end (240ms)
    ✔ Cannot change sale amount after start (255ms)
    ✔ Cannot buy 0 (249ms)
    ✔ Cannot buy max per wallet (273ms)
    ✔ Can not buy over public sale amount (252ms)
    ✔ Pause check (269ms)
Whitelist
  ✔ Invalid proof (219ms)
  ✔ Buy whitelist (219ms)
  ✔ Buy over max per wallet (226ms)
  ✔ Can not buy over user allocation (220ms)
  ✔ Can not buy over total whitelist allocation (232ms)
  ✔ User can buy both whitelist and public sale (234ms)
Release Whitelist To Public
  ✔ Can release unsold whitelist to public before sale starts (223ms)
  ✔ Can release unsold whitelist to public during sale (234ms)
  ✔ Can release all unsold whitelist to public (222ms)
  ✔ Cannot release more than unsold whitelist amount (199ms)
  ✔ Cannot release more than remaining unsold after partial whitelist purchase (202ms)
  ✔ Public pool is expanded after release and users can buy from it (197ms)
Withdraw
  ✔ Only admin can withdraw (185ms)
  ✔ Withdraw Token (192ms)
Buy For - Purchase on behalf of another address
      ✔ buyFor: Recipient receives NFT, buyer pays (194ms)
      ✔ buyFor: Public sale limit is tracked per recipient (188ms)
      ✔ buyFor: Different buyers can buy for the same recipient (224ms)
      ✔ buyFor: Events are emitted with recipient as buyer (194ms)
      ✔ buyWithWhitelistFor: Whitelisted buyer can buy for another recipient (191ms)
      ✔ buyWithWhitelistFor: Whitelist allocation is tracked per buyer, not recipient (202ms)
      ✔ buyWithWhitelistFor: Invalid proof when non-whitelisted user tries to buyFor (192ms)
      ✔ buyFor and buyWithWhitelistFor: Can combine both functions (199ms)
      ✔ buyFor: Referral rewards are recorded for referrer (188ms)
      ✔ buyFor: Should revert with zero address recipient (184ms)
      ✔ buyFor: Should work when recipient equals buyer (same as buy) (185ms)
      ✔ getUserPurchasedAmount: Returns different values for buyer vs recipient in proxy purchases
(191ms)

  Withdraw
    ✔ Can not withdraw to zero address (533ms)
    ✔ Withdraw ETH
    ✔ Withdraw Token (128ms)


  102 passing (11703ms)

================================================
NODE-SC
```

```
HStaking
  stake
    ✔ User stakes successfully (399ms)
    ✔ User stakes multiple times on same license (82ms)
    ✔ Should revert when staking with zero amount
    ✔ Should revert when NFT not minted (78ms)
  requestUnStake
    ✔ User requests unstake successfully (76ms)
    ✔ Should revert when unstaking more than staked
    ✔ Should revert when unstaking with zero amount
  claim
    ✔ User claims after cooldown period
    ✔ Should revert when claiming before cooldown period (78ms)
    ✔ Should revert when claiming with no pending request
    ✔ Should not allow double claim
  EnumerableSet View Functions
    ✔ getUserStakedLicenses returns all staked licenses
    ✔ getUserStakedLicenses removes license when fully unstaked and claimed
    ✔ Multiple unstake requests on same license should accumulate
    ✔ getUserStakedLicensesCount returns correct count
    ✔ Unstake resets withdrawable time on accumulation
    ✔ Multiple users have separate staked licenses
    ✔ License remains in list if user has remaining stake after partial unstake and claim (78ms)
  Pause
    ✔ Pause the contract and staking functions should revert
    ✔ Unpause the contract and staking functions should work
  Withdraw
    ✔ Only admin can withdraw
    ✔ Admin withdrawToken successfully
    ✔ Native token withdrawal is disabled

TreasuryDistribution
  constructor
    ✔ Deploys with valid treasury and licenseNFT (346ms)
  commitMerkleRoot
    ✔ All 5 tiers: 5/10/20/25/40 licenses — matches worked example (117ms)
    ✔ Reverts with zero root (95ms)
    ✔ Reverts when day already committed (90ms)
    ✔ Reverts without COMMITTER_ROLE (82ms)
    ✔ Reverts with day = 0 (78ms)
    ✔ Reverts with day > MAX_DAY (80ms)
    ✔ Reverts with empty tierLicenseCounts (84ms)
    ✔ Reverts with too many tiers (82ms)
    ✔ Reverts with invalid tier (tier = 0) (87ms)
    ✔ Reverts with invalid tier (tier > MAX_TIERS) (79ms)
    ✔ Reverts with duplicate tier (77ms)
    ✔ Reverts with zero license count (80ms)
  distribute
    ✔ Distributes with valid proof, emits Distributed with tier and amount (86ms)
    ✔ Reverts when tier reward not configured (80ms)
    ✔ Reverts with wrong tier (invalid proof) (86ms)
    ✔ Reverts on double claim (AlreadyClaimed) (82ms)
    ✔ Reverts without DISTRIBUTOR_ROLE (81ms)
    ✔ Reverts when day not committed (79ms)
    ✔ Different tiers get different rewards (91ms)
    ✔ Reward per license decreases with more licenses in same tier
  batchDistribute
    ✔ Batch distributes with valid proofs and correct balance changes
    ✔ Reverts if empty batch
    ✔ Reverts if batch exceeds max size
    ✔ Reverts if any item in batch has invalid proof
    ✔ Reverts if batch contains duplicate licenseId (AlreadyClaimed)
  setTreasury
    ✔ Updates treasury, emits TreasuryUpdated
    ✔ Reverts with zero address
    ✔ Reverts without DEFAULT_ADMIN_ROLE
  pause/unpause
    ✔ Admin can pause and unpause (80ms)
  cross-day isolation
    ✔ Same license can claim on different days (76ms)
    ✔ Proof from day 1 cannot be used on day 2
  batchDistribute day not committed
```

```
    ✔ Reverts when day not committed
  getCap view
    ✔ Reverts with day = 0
    ✔ Reverts with day > MAX_DAY
  setStartTime
    ✔ Sets startTime only once and emits StartTimeUpdated (91ms)
    ✔ Reverts when startTime already set
    ✔ Reverts with future startTime (90ms)
    ✔ Reverts without DEFAULT_ADMIN_ROLE
  setTierMultiplier
    ✔ Successfully sets tier multiplier as admin
    ✔ Can set multipliers for all 5 tiers (82ms)
    ✔ Can update existing tier multiplier
    ✔ Reverts with InvalidTier when tier is 0
    ✔ Reverts with InvalidTier when tier exceeds MAX_TIERS (5)
    ✔ Reverts without DEFAULT_ADMIN_ROLE
  currentDay
    ✔ Returns day 1 when block.timestamp == startTime
    ✔ Returns correct day after time passes
    ✔ Reverts when startTime is not set (zero) (88ms)
  onlyNotFuture modifier
    ✔ commitMerkleRoot reverts with DayInFuture for future day
    ✔ distribute reverts with DayInFuture for future day
    ✔ batchDistribute reverts with DayInFuture for future day
    ✔ Allows operations on current day after time advance (75ms)

  Withdraw
    ✔ Can not withdraw to zero address (338ms)
    ✔ Withdraw ETH
    ✔ Withdraw Token


  78 passing (5054ms)
```

Changelog

- 2026-03-18 - Initial report
- 2026-04-01 - Final Report

About Quantstamp

Quantstamp is a global leader in blockchain security. Founded in 2017, Quantstamp's mission is to securely onboard the next billion users to Web3 through its best-in-class Web3 security products and services.

Quantstamp's team consists of cybersecurity experts hailing from globally recognized organizations including Microsoft, AWS, BMW, Meta, and the Ethereum Foundation. Quantstamp engineers hold PhDs or advanced computer science degrees, with decades of combined experience in formal verification, static analysis, blockchain audits, penetration testing, and original leading-edge research.

To date, Quantstamp has performed more than 1300 audits and secured over $500 billion in digital asset risk from hackers. Quantstamp has worked with a diverse range of customers, including startups, category leaders and financial institutions. Brands that Quantstamp has worked with include Ethereum 2.0, Binance, Visa, PayPal, Solana, Canton, Polymarket, PandcakeSwap, Virtuals Protocol, Wayfinder, Lido, TrustWallet, Alchemy, World Economic Forum.

Quantstamp's collaborations and partnerships showcase our commitment to world-class research, development and security. We're honored to work with some of the top names in the industry and proud to secure the future of web3.

Notable Collaborations & Customers:

- Blockchains: Ethereum 2.0, Solana, Polygon, TON, Cardano, Binance Smart Chain, Avalanche, Arbitrum, Flow
- DeFi: Lido, Ethena, Compound, Curve, Venus, PancakeSwap, Polymarket
- Infra/Staking: TrustWallet, Alchemy, Liquid Collective, Kiln, Galxe, API3, ssv.network, Luganodes
- Immersive: Virtuals Protocol, Wayfinder, OpenSea, Square Enix, Parallel, Camp, Decentraland
- Institutional: Visa, Circle, Revolut, Anthropic, OpenAI, Canton, Republic, Arkham, Sequoia

Timeliness of content

The content contained in the report is current as of the date appearing on the report and is subject to change without notice, unless indicated otherwise by Quantstamp; however, Quantstamp does not guarantee or warrant the accuracy, timeliness, or completeness of any report you access using the internet or other means, and assumes no obligation to update any information following publication or other making available of the report to you by Quantstamp.

Notice of confidentiality

This report, including the content, data, and underlying methodologies, are subject to the confidentiality and feedback provisions in your agreement with Quantstamp. These materials are not to be disclosed, extracted, copied, or distributed except to the extent expressly authorized by Quantstamp.

Links to other websites

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Quantstamp. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that Quantstamp are not responsible for the content or operation of such web sites, and that Quantstamp shall have no liability to you or any other person or entity for the use of third-party web sites. Except as described below, a hyperlink from this web site to another web site does not imply or mean that Quantstamp endorses the content on that web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the report. Quantstamp assumes no responsibility for the use of third-party software on any website and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any output generated by such software.

Disclaimer

The review and this report are provided on an as-is, where-is, and as-available basis. To the fullest extent permitted by law, Quantstamp disclaims all warranties, expressed implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. You agree that access and/or use of the report and other results of the review, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE. This report is based on the scope of materials and documentation provided for a limited review at the time provided. You acknowledge that Blockchain technology remains under development and is subject to unknown risks and flaws and, as such, the report may not be complete or inclusive of all vulnerabilities. The review is limited to the materials identified in the report and does not extend to the compiler layer, or any other areas beyond the programming language, or programming aspects that could present security risks. The report does not indicate the endorsement by Quantstamp of any particular project or team, nor guarantee its security, and may not be represented as such. No third party is entitled to rely on the report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. Quantstamp does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party, or any open source or third-party software, code, libraries, materials, or information to, called by, referenced by or accessible through the report, its content, or any related services and products, any hyperlinked websites, or any other websites or mobile applications, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third party. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate.