QuillAudits

AUDIT REPORT

January 2026

For

elsa

Table of Content

Executive Summary

Project Name	HeyElsa
Protocol Type	Staking Protocol
Project URL	https://www.heyelsa.ai/
Overview	This contract allows holders of the HeyElsa token to stake tokens for rewards in multiple tiers with differing rewards. There is a slashing penalty for users who withdraw their stakes before the expiry period.
Audit Scope	The scope of this audit was to analyze the HeyElsa Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/HeyElsa/staking-contract
Branch	main
Contracts in Scope	ELSAStaking.sol MockELSA.sol IELSA.sol IELSAStaking.sol StakingErrors.sol StakingEvents.sol
Commit Hash	25a710d
Language	Solidity
Blockchain	Base
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	17th - 19th January, 2026
Updated Code Received	19th January, 2026
Review 2	19th - 20th January, 2026
Fixed In	58d2d6

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity

Critical	0 (0.0%)
High	1 (11.2%)
Medium	4 (44.4%)
Low	0 (0.0%)
Informational	4 (44.4%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged	0	0	0	0	0
Partially Resolved	0	0	0	0	0
Resolved	0	1	4	0	4

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Frontrunnable Stake Index Reordering Can Cause Unintended Early-Unstake Penalties	High	Resolved
2	Time-Based Reward Accrual Is Not Bounded by Reward Pool, Leading to Severe Reward Insolency	Medium	Resolved
3	Gasless Staking Authorization Is Frontrunnable, Allowing Tier Manipulation	Medium	Resolved
4	Lack of Partial Reward Claims Can Permanently Lock User Rewards	Medium	Resolved
5	Incorrect STAKE_AUTHORIZATION_TYPEHASH Breaks Gasless Staking Authorization	Medium	Resolved
6	Hardcoded 1e18 Should Use Token Decimals	Informational	Acknowledged
7	estimateAPY() Is Misnamed — Rewards Are Not Compounded	Informational	Resolved
8	Unrealistic Assumed Stake Skews APR Estimation	Informational	Resolved

Issue No.	Issue Title	Severity	Status
9	Error message is not descriptive of the issue or the fix	Informational	Acknowledged

Checked Vulnerabilities

- ✔ Access Management
- ✔ Arbitrary write to storage
- ✔ Centralization of control
- ✔ Ether theft
- ✔ Improper or missing events
- ✔ Logical issues and flaws
- ✔ Arithmetic Computations Correctness
- ✔ Race conditions/front running
- ✔ SWC Registry
- ✔ Re-entrancy
- ✔ Timestamp Dependence
- ✔ Gas Limit and Loops
- ✔ Exception Disorder
- ✔ Gasless Send
- ✔ Use of tx.origin
- ✔ Malicious libraries
- ✔ Compiler version not fixed
- ✔ Address hardcoded
- ✔ Divide before multiply
- ✔ Integer overflow/underflow
- ✔ ERC's conformance
- ✔ Dangerous strict equalities
- ✔ Tautology or contradiction
- ✔ Return values of low-level calls

- ✅ Missing Zero Address Validation
- ✅ Private modifier
- ✅ Revert/require functions
- ✅ Multiple Sends
- ✅ Using suicide
- ✅ Using delegatecall
- ✅ Upgradeable safety
- ✅ Using throw
- ✅ Using inline assembly
- ✅ Style guide violation
- ✅ Unsafe type inference
- ✅ Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity Static Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open Security vulnerabilities identified that must be resolved and are currently unresolved.	**Resolved** These are the issues identified in the initial audit and have been successfully fixed.
Acknowledged Vulnerabilities which have been acknowledged but are yet to be resolved.	**Partially Resolved** Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Impact

Likelihood

	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

High Severity Issues

Frontrunnable Stake Index Reordering Can Cause Unintended Early-Unstake Penalties

Resolved

Path

contracts/ElsaStaking.sol

Function Name

`cleanupInactiveStakes(address)`
`unstake(uint256)`

Description

The staking system relies on user-supplied array indices to identify specific stake positions during withdrawal via unstake(uint256 stakeIndex). However, the cleanupInactiveStakes(address) function can be called by any external user and performs in-place array compaction by shifting active stakes forward and removing inactive ones.

Because cleanupInactiveStakes() mutates the order of userStakes[account], an attacker can front-run a user's unstake() transaction and trigger a reordering of stake indices. As a result, the index originally intended by the user may now reference a different active stake with different maturity conditions causing an unintended early-unstake penalty and loss of principal.

Impact

Users can lose a portion of their principal due to unintended early-unstake penalties.

Likelihood

The attack requires only a mempool watcher and a cheap call to cleanupInactiveStakes().

• No special permission is required.
• Likelihood increases in periods of high withdrawal activity or for large stakes.

POC

State A:

index 0 -> active, mature (no penalty) ✅

1 -> active, not mature (penalty) ✅

2 -> inactive, mature (no penalty) ❌

3 -> active, mature (no penalty) ✅

4 -> active, not mature (penalty) ✅

5 -> active, not mature (penalty) ✅

frontrun with calling cleanupInactiveStakes()

State B:
index 0 -> active, mature (no penalty)
1 -> active, not mature (penalty)
2 -> inactive, mature (no penalty)
3 (2) -> active, mature (no penalty)
4 (3) -> active, not mature (penalty) ⇒gets activated, causing user to lose their principal
5 (4) -> active, not mature (penalty)

Say for example, the positions at index 2,4 are inactive but not cleared and the user tries to withdraw at index 3 (active, mature) expecting no penalties, but gets shifted to the new index 3 (formerly 4) — they immediately lose a portion of their principal unintended.

Recommendation

Limit calls to the stake owner (msg.sender == account).

Medium Severity Issues

Time-Based Reward Accrual Is Not Bounded by Reward Pool, Leading to Severe Reward Insolency

Resolved

Path

contracts/ElsaStaking.sol

Function Name

`_updateReward(address), rewardPerToken(), _calculateDynamicRewardRate()`

Description

The reward accounting system accrues rewards purely based on elapsed time and staking rate, without enforcing a hard upper bound tied to the actual size of the rewardPool. As a result, rewards continue to mathematically accrue even when the protocol does not have sufficient funds to cover them.

Although _updateReward() attempts to mitigate this by checking whether the protocol is solvent (rewardPool >= totalAccruedRewards) before crediting rewards, this check occurs after rewardPerToken() has already increased based on elapsed time. This causes a growing discrepancy between:

- rewardPerTokenStored (which increases continuously with time), and
- The actual amount of tokens available in rewardPool.

During periods of insolvency:

- Rewards are not credited to users.
- rewardDebt is intentionally not updated.
- However, rewardPerTokenStored keeps increasing implicitly over time.

Once the protocol becomes solvent again (e.g., through a reward top-up), users can suddenly accrue large backdated rewards, even if the reward pool was insufficient for most of that period. This allows totalPendingRewards to grow to multiple times the size of the reward pool, making insolvency unavoidable.

Impact

- The protocol can accumulate unpayable reward liabilities.
- totalPendingRewards can exceed rewardPool by large multiples.
- Long-term insolvency becomes mathematically guaranteed under sustained usage.
- Honest users may be unable to claim rewards, or claims may revert if transfers fail.

Likelihood - High

- Reward accrual is continuous and time-based.
- No cap or throttle ties emissions to available rewards.
- Even short periods of high reward rates or delayed updates can cause insolvency.
- This can occur naturally without malicious behavior.

Recommendation

Accrue rewards only when rewards are explicitly funded (e.g., epoch-based funding or discrete reward injections).

Gasless Staking Authorization Is Frontrunnable, Allowing Tier Manipulation

Resolved

Path

Gasless Staking Authorization Is Frontrunnable, Allowing Tier Manipulation

Function Name

`stakeWithAuthorization(address,uint256,uint8,uint256,uint256,bytes32,bytes)`

Description

The stakeWithAuthorization() function allows gasless staking using receiveWithAuthorization, where a user (from) signs an authorization permitting token transfer. However, the signed authorization does not bind the tierId parameter, which is supplied by the transaction sender at execution time.

Because the authorization only covers:

- token transfer (from → address(this)),
- amount,
- validity window,
- nonce,

any external caller who obtains the signed authorization can front-run the intended transaction and submit their own call to stakeWithAuthorization() using the same authorization but with a different tierId.

This enables an attacker to force the stake into:

- a longer lock tier than intended (griefing / liquidity lock), or
- a shorter lock tier than intended (economic manipulation), both of which violate the original signer's intent. The stake is created for the original "from" address, making the attack non-obvious and difficult to recover from.

Impact

Users can be forced into unintended lock durations.

POC

Legitimate user signs the transaction
Relayer calls stakeWithAuthorization.
Malicious user front runs this call with a different tierID than intended.

Likelihood: Medium

No special permissions required to execute the front run. Only gas cost for the attacker is consumed.

Recommendation

Include tierId in the signed payload so that it cannot be altered at execution time.

Lack of Partial Reward Claims Can Permanently Lock User Rewards

Resolved

Path

contracts/ElsaStaking.sol

Function Name

`claimRewards()`

Description

The claimRewards() function enforces an all-or-nothing reward claim model, requiring that a user's entire pending reward balance be paid out in a single transaction. If rewards[msg.sender] exceeds the available rewardPool, the call reverts and no rewards can be claimed at all.

Because rewards accrue over time and users are not required to claim frequently, a user's pending rewards can grow significantly. Since reward accrual is mostly time dependent and not based on the amount of reward in the pool, affected users become permanently unable to claim rewards, even though the protocol may continue operating for others. This creates a liveness failure: users with large reward balances are effectively locked out.

Impact

- Users may be unable to ever withdraw earned rewards.
- Rewards become effectively frozen for long-term or inactive users.
- Large stakers are disproportionately affected.

Likelihood - High

- Long-term stakers commonly delay claiming rewards.
- Reward pool underfunding can occur naturally.
- No malicious action is required.

Recommendation

Permit users to claim up to min(rewards[user], rewardPool) and leave the remainder accrued.

Incorrect STAKE_AUTHORIZATION_TYPEHASH Breaks Gasless Staking Authorization

Resolved

Path

contracts/ElsaStaking.sol

Function Name

`stakeWithAuthorization(…), _verifyStakingAuthorization(…)`

Description

The constant STAKE_AUTHORIZATION_TYPEHASH is incorrectly defined. The value:

```
bytes32 public constant STAKE_AUTHORIZATION_TYPEHASH = =
0x8b73c3c69bb8fe3d512ecc4cf759cc79239f7b179b0ffacaa9a75d522b39400f;
```

is not the typehash for:

```
StakeWithAuthorization(
address from,
uint256 amount,
uint8 tierId,
uint256 validAfter,
uint256 validBefore,
bytes32 nonce
)
```

Instead, this value corresponds to the EIP-712 domain separator typehash, meaning the staking authorization signature can never be validated correctly.

As a result:

- _verifyStakingAuthorization() always computes a digest that does not match the signed message
- stakeWithAuthorization() reverts for all valid signatures
- Gasless staking is effectively non-functional

This issue can be confirmed by hashing the struct definition off-chain. The correct typehash is:

`0x182dd0f5adcd848efe9c68cc0ebe2ceabc771cf2b496c6e8d16d3faa563451f0`

Impact

stakeWithAuthorization() is completely broken

POC

```
it("Verify STAKE_AUTHORIZATION_TYPEHASH", async function () {
const expectedTypeHash = ethers.keccak256(
ethers.toUtf8Bytes(
"StakeWithAuthorization(address from,uint256 amount,uint8 tierId,uint256
validAfter,uint256 validBefore,bytes32 nonce)",
),
);
console.log("Expected typehash:", expectedTypeHash);
console.log(
  "Contract typehash:",
  await staking.STAKE_AUTHORIZATION_TYPEHASH(),
);

expect(await staking.STAKE_AUTHORIZATION_TYPEHASH()).to.equal(
  expectedTypeHash,
);
  });
```

Likelihood - Medium

Issue occurs without any restrictions.

Recommendation

Replace the incorrect constant with the correct EIP-712 struct typehash:

```
bytes32 public constant STAKE_AUTHORIZATION_TYPEHASH = =
0x182dd0f5adcd848efe9c68cc0ebe2ceabc771cf2b496c6e8d16d3faa563451f0;
```

Informational Issues

Hardcoded 1e18 Should Use Token Decimals

Acknowledged

Path

contracts/ElsaStaking.sol

Description

Several protocol constants hardcode 1e18 under the assumption that the staking token uses 18 decimals. This reduces flexibility and can cause incorrect calculations if the staking token uses a different decimal configuration.

Hardcoding decimals tightly couples protocol correctness to a single token implementation and increases upgrade and reuse risk.
Instances:

```
uint256 private constant ELSA_TOTAL_SUPPLY = 1_000_000_000 * 1e18;
uint256 public constant MINIMUM_TVL_FOR_REWARDS = 1000 * 1e18;
```

Recommendation

Fetch and normalize values using IERC20Metadata.decimals() and scale calculations dynamically instead of hardcoding 1e18.

estimateAPY() is misnamed, rewards are not compounded

Resolved

Path

contracts/ElsaStaking.sol

Function Name

`estimateAPY(uint8)`

Description

The function estimateAPY() calculates rewards using a simple linear time-based formula and does not account for compounding. As implemented, the calculation represents APR, not APY.
Using incorrect financial terminology can mislead integrators, dashboards, and users regarding expected returns.

Recommendation

Rename the function to estimateAPR() or explicitly implement compounding logic.

Unrealistic assumedStake skews APR estimation

Resolved

Path

contracts/ElsaStaking.sol

Function Name

`estimateAPY(uint8)`

Description

When totalStaked = 0, the function assumes:
assumedStake = ELSA_TOTAL_SUPPLY / 100;

This equates to 10 million tokens, which conflicts with the protocol's bootstrap mechanism (MINIMUM_TVL_FOR_REWARDS = 1000 tokens). As a result, early APR estimates are severely diluted and unrealistic.

Recommendation

Use MINIMUM_TVL_FOR_REWARDS as the assumed stake baseline.

HeyElsa Team's comment

MINIMUM_TVL_FOR_REWARDS has been updated to 5 million tokens.

Error message is not descriptive of the issue or the fix

Acknowledged

Path

contracts/ElsaStaking.sol

Function Name

`_createStake()`

Description

When the stake array exceeds its maximum size, the function reverts with:

`StakingArrayTooLarge(totalStakeCount, MAX_STAKES_PER_USER * 2) which yields StakingArrayTooLarge(100, 100)`

However, the error does not explain why the revert occurred or that the user must first call cleanupInactiveStakes() to proceed.

Recommendation

Enhance the revert reason to explicitly instruct users to clean up inactive stakes before creating new ones (e.g., StakingArrayTooLargeCleanupRequired).

Vital risks to note

- **Centralization Risk**

This audit assumes privileged roles behave honestly. Currently, all roles are assigned to a single admin address. To reduce single–point–of–failure risk, distribute privileges across a multisig and/or timelock, and separate operational roles where possible.

- **Low test coverage**

The existing suite lacks sufficient end–to–end coverage. Increase test breadth and depth to validate critical flows (stake, unstake, compound, reward accrual, emergency paths) under varied states and edge cases.

- **Reward distribution**

Reward emission assumptions should be validated against tokenomics. For example, a rate of 100 tokens/second yields ~8.64M/day (~3.15B/year) – as seen in the tests. Confirm that configured rates and caps align with the intended emissions schedule and available reward reserves.

Functional Tests

Some of the tests performed are mentioned below:

- ✔ Should be able to stake
- ✔ Should be able to unstake
- ✔ Should be able to stake with authorization
- ✔ Should be able to stakeWithPermit
- ✔ Should be able to unstakeAllMature
- ✔ Should be able to withdraw during emergency
- ✔ Should clear inactive entries from the array

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Threat Model

1. External Dependencies & Trust Boundaries

This section enumerates everything the protocol does not fully control.

1.1 External Dependencies Table

Dependency	Type	How It Is Used	Trust Assumption	Associated Risks
ERC20.sol (+ ERC20Permit + ERC3009)	Base Contract + Permits + Gasless Authorization	Core token logic, balances, authorized transfers (permit), gasless transfers	Correct implementation of ERC20 invariants Signature replay and malleability mitigated by EIP712 type-hashing	
Solidity Compiler	Tool	Compilation	Compiler has no known critical bugs	Older version can lead to unsafe defaults

2. Entry & Exit Points Analysis (Function-Level)

This is the core of the threat model. Every externally callable function must appear here.

2.1 Function Entry / Exit Table

Each function gets one table.

Contract Name	Function	Category
ELSAStaking.sol	initialize()	**What this function can do** Provides initial values for state variables of the contract Provide one-time setup for the contract **What this function cannot/should not do** Re-initialize after proxy has been run once **Main invariant(s)** Sets role based access control Sets reward rate and reward distribution factors **Access level** Implicit restriction (once at deployment)
	stake()	**What this function can do** Allows ANY user to increase the staked balances across multiple tiers. **What this function cannot/should not do** Decrease the staked balances in the system **Main invariant(s)** totalStaked, rewardPerTokenStored, rewardPool, lastUpdateTime **Access level** Open to all users, when system is not paused

Contract Name	Function	Category
	stakeWithPermit()	**What this function can do** Allows ANY user (given the signature and tx details for another user) to update that user's staked balances across multiple tiers. **What this function cannot/should not do** Decrease the staked balances in the system Allow any user to re-use the same signature **Main invariant(s)** signature (v, r, s), permit.nonce, totalStaked, rewardPerTokenStored, rewardPool, lastUpdateTime **Access level** Open to users holding valid signatures, when system is not paused
	stakeWithAuthorization()	**What this function can do** Allows the PAYEE to update a user's staked balances across multiple tiers given valid authorization by signing receiveWithAuthorization. **What this function cannot/should not do** Decrease the staked balances in the system. Accept calls not from the staking contract **Main invariant(s)** validBefore, validAfter, nonce, signature, UserStake.totalStaked, UserStake.rewardPerTokenStored, rewardPool, lastUpdateTime **Access level** Open to only PAYEE, when system is not paused

Contract Name	Function	Category
	unstake()	**What this function can do** Allows ANY user to withdraw the staked balances across multiple tiers, slashing non-mature stakes. **What this function cannot/should not do** Increase the staked balances in the system. **Main invariant(s)** UserStake.totalStaked, UserStake.unlockTime, rewardPool **Access level** Open to all staked users, when system is not paused
	unstakeAllMature()	**What this function can do** Allows ANY user to withdraw staked balances from mature stakes across multiple tiers. **What this function cannot/should not do** Increase the staked balances in the system. **Main invariant(s)** UserStake.totalStaked, UserStake.unlockTime **Access level** Open to all staked users, when system is not paused
	emergencyWithdraw()	**What this function can do** Allows ANY user to withdraw all their staked balances with a penalty attached if not mature. **What this function cannot/should not do** Increase the staked balances in the system.

Contract Name	Function	Category
		Main invariant(s) UserStake.emergencyWithdrawEnabled, UserStake.totalStaked, UserStake.earlyUnstakePenalty, UserStake.unlockTime, rewardPool **Access level** Open to all staked users, when system is paused
	cleanupInactiveStakes()	**What this function can do** Compresses the userStakes array by removing inactive stakes from the array to prevent excessively arrays from griefing users **What this function cannot/should not do** Modify the stake activity (change a stake from active to inactive or vice versa) Stake or unstake tokens **Main invariant(s)** UserStake.isActive **Access level** Open to all users, anytime.
	cleanupInactiveStakes()	**What this function can do** Allow staked users to claim accrued rewards **What this function cannot/should not do** Stake or unstake tokens Transfer rewards tokens to any user other than the caller **Main invariant(s)** rewards, rewardPool, totalAccruedRewards

Contract Name	Function	Category
		Access level Open to all staked users, when system is not paused
	compoundRewards()	**What this function can do** Allow users to restake their earned rewards **What this function cannot/should not do** Remove tokens from the system Transfer rewards tokens to any user **Main invariant(s)** rewards, rewardPool, totalAccruedRewards **Access level** Open to all staked users, when system is not paused

3. Asset Flow Mapping (Critical Contracts)

This section identifies where money actually lives and how it moves.

3.1 Asset-Holding Contracts

List only contracts that custody value.

Contract	Asset Type	Custodied Assets
ELSAStaking.sol	ERC20	User deposits, rewards

3.2 Asset Entry & Exit Functions

This table maps money movement, which is where most exploits happen.

Contract	Function	Asset In	Asset Out	Caller	Risk Notes
ElsaStaking.sol	stake	ERC20	-	Public	
ElsaStaking.sol	unstake	-	ERC20	Public	
ElsaStaking.sol	unstakeAllMature	-	ERC20	Public	
ElsaStaking.sol	claimRewards	-	ERC20	Public	

Closing Summary

In this report, we have considered the security of HeyElsa. We performed our audit according to the procedure described above.

1 High and 4 Medium and 4 informational severity issues were found, the HeyElsa Team resolved 7 issues and acknowledged the rest.

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

QuillAudits

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

January 2026

For

elsa

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com