Grindery

Staking Contracts

SMART CONTRACT AUDIT

05.02.2025

Made in Germany by Softstack.io

Table of contents

1. Disclaimer 4
2. About the Project and Company 5
  2.1 Project Overview 6
3. Vulnerability & Risk Level 7
4. Auditing Strategy and Techniques Applied 8
  4.1 Methodology 8
5. Metrics 9
  5.1 Tested Contract Files 9
  5.2 CallGraph 10
  5.3 Source Lines & Risk 11
  5.4 Capabilities 12
  5.5 Source Unites in Scope 13
6. Scope of Work 14
  6.1 Findings Overview 15
  6.2 Manual and Automated Vulnerability Test 16
  6.2.1 Reward Pool Depletion 16
  6.2.2 Shared Reward Pool Can Be Drained By Short-Term Stakers 21
  6.2.3 Missing Lock Period Validation in AddRewards Function 25
  6.2.4 Reward Rate Manipulation 29
  6.2.5 Lack of Slashing Mechanism 32
  6.3 SWC Attacks 36
  6.4 Unit Tests 41

6.5 Verify Claims 42
7. Executive Summary 43
8. About the Auditor 44

1. Disclaimer

The audit makes no statements or warrantees about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of Grindery Inc. If you are not the intended receptor of this document, remember that any disclosure, copying or dissemination of it is forbidden.

Major Versions / Date	Description
0.1 (04.02.2025)	Layout
0.4 (04.02.2025)	Automated Security Testing Manual Security Testing
0.5 (04.02.2025)	Verify Claims
0.6 (05.02.2025)	Testing SWC Checks
0.9 (05.02.2025)	Summary and Recommendation
1.0 (05.02.2025)	Final document

2. About the Project and Company

Company address:

Grindery Inc.
Via España, Delta Bank Tower
Floor 6, Suite 604D
Panama City, Republic of Panama
Public Registry Record: 155739060

Grindery Pte. LTD.
UEN: 202214323C
12 Purvis Street, #02-1620
Singapore 188591

Website: https://www.grindery.com

Twitter (X): https://x.com/grindery_io

Telegram: https://t.me/grinderyai

LinkedIn: https://www.linkedin.com/company/grindery/

YouTube: https://www.youtube.com/@grinderyTV

GitHub: https://github.com/grindery-io

2.1 Project Overview

Grindery is an AI-driven smart wallet platform designed to simplify digital asset management and Web3 interactions. At its core, Grindery integrates AI-powered dApps with a consumer-friendly smart wallet, making blockchain technology more accessible, secure, and efficient for both developers and end users.

The Grindery platform operates through its native token, which fuels transactions, incentivizes development, and ensures economic security through dApp staking. By embedding Web3 functionality directly into messaging platforms like Telegram, Grindery has already onboarded over 3 million users, demonstrating its ability to drive mainstream adoption. Future expansions will integrate the wallet across iOS, Android, and other social platforms, extending its accessibility.

Beyond wallet functionality, Grindery's developer platform enables the creation of AI-powered dApps that offer automation, smart notifications, and enhanced security. The ecosystem leverages account abstraction and advanced automation infrastructure, ensuring an intuitive, secure experience across multiple blockchain networks.

Grindery’s long-term vision is to become the iPhone of Web3 wallets, providing a seamless, AI-enhanced user experience across all major blockchain networks and digital platforms. By bridging decentralized finance with mainstream applications, Grindery aims to accelerate mass adoption of Web3 technologies while maintaining decentralization and user sovereignty.

3. Vulnerability & Risk Level

Risk represents the probability that a certain source-threat will exploit vulnerability, and the impact of that event on the organization or system. Risk Level is computed based on CVSS version 3.0.

Level	Value	Vulnerability	Risk (Required Action)
Critical	9 – 10	A vulnerability that can disrupt the contract functioning in a number of scenarios, or creates a risk that the contract may be broken.	Immediate action to reduce risk level.
High	7 – 8.9	A vulnerability that affects the desired outcome when using a contract, or provides the opportunity to use a contract in an unintended way.	Implementation of corrective actions as soon as possible.
Medium	4 – 6.9	A vulnerability that could affect the desired outcome of executing the contract in a specific scenario.	Implementation of corrective actions in a certain period.
Low	2 – 3.9	A vulnerability that does not have a significant impact on possible scenarios for the use of the contract and is probably subjective.	Implementation of certain corrective actions or accepting the risk.
Informational	0 – 1.9	A vulnerability that have informational character but is not effecting any of the code.	An observation that does not determine a level of risk

4. Auditing Strategy and Techniques Applied

Throughout the review process, care was taken to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices. To do so, reviewed line-by-line by our team of expert auditors and smart contract developers, documenting any issues as there were discovered.

4.1 Methodology

The auditing process follows a routine series of steps:

1. Code review that includes the following:
   i. Review of the specifications, sources, and instructions provided to softstack to make sure we understand the size, scope, and functionality of the smart contract.
   ii. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
   iii. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to softstack describe.
2. Testing and automated analysis that includes the following:
   i. 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.
   ii. Symbolic execution, which is analysing a program to determine what inputs causes each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, actionable recommendations to help you take steps to secure your smart contracts.

5. Metrics

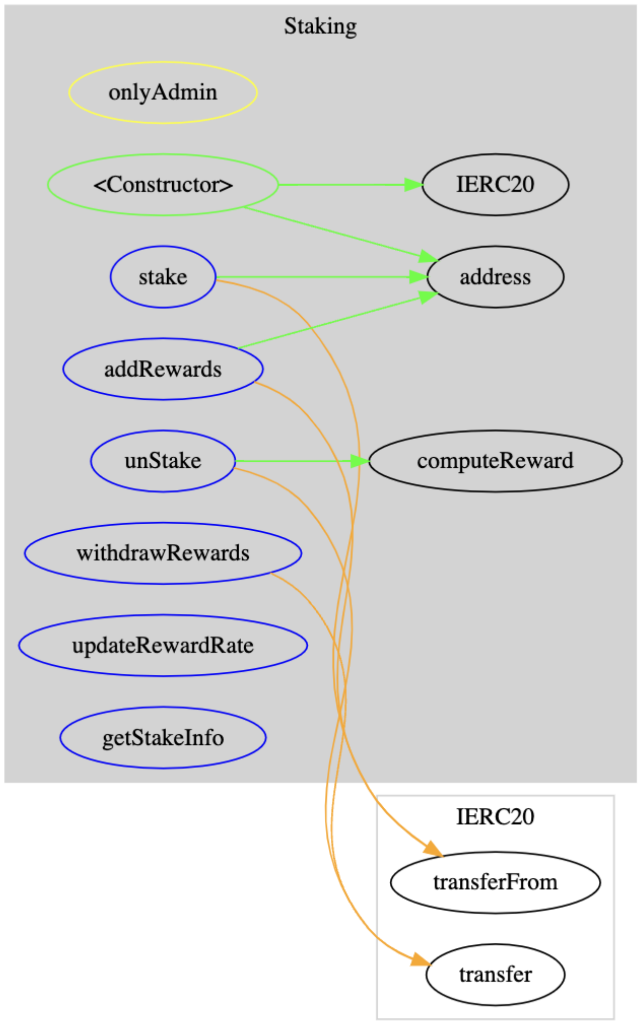
The metrics section should give the reader an overview on the size, quality, flows and capabilities of the codebase, without the knowledge to understand the actual code.

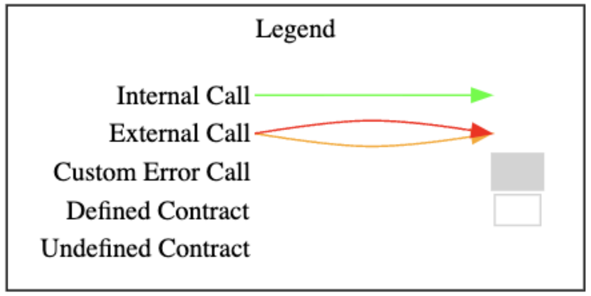
5.1 Tested Contract Files

The following are the MD5 hashes of the reviewed files. A file with a different MD5 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different MD5 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.

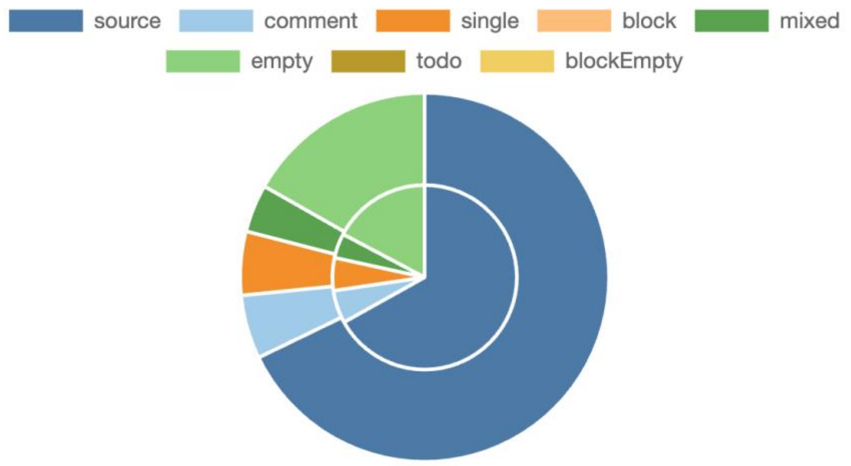
File	Fingerprint (MD5)
./src/Staking.sol	5b37f2e04843da52136c83b71c5ec666

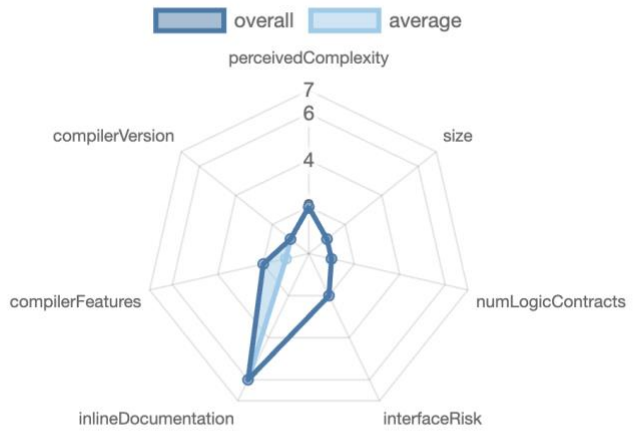
5.2 CallGraph





5.3 Source Lines & Risk





5.4 Capabilities

Solidity Versions observed	Experimental Features	Can Receive Funds	Uses Assembly	Has Destroyable Contracts
0.8.20				

Transfers ETH	Low-Level Calls	DelegateCall	Uses Hash Functions	ECRecover	New/Create/Create2
yes					

Exposed Functions
This section lists functions that are explicitly declared public or payable. Please note that getter methods for public stateVars are not included.

Public	Payable
7	0

External	Internal	Private	Pure	View
6	6	0	0	2

StateVariables

Total	Public
7	7

5.5 Source Unites in Scope

File	Logic Contracts	Interfaces	Lines	nLines	nSLOC	Comment Lines	Complex. Score
src/Staking.sol	1		123	119	93	8	52
Totals	**1**		**123**	**119**	**93**	**8**	**52**

- **Lines**: total lines of the source unit
- **nLines**: normalized lines of the source unit (e.g. normalizes functions spanning multiple lines)
- **nSLOC**: normalized source lines of code (only source-code lines; no comments, no blank lines)
- **Comment Lines**: lines containing single or block comments
- **Complexity Score**: a custom complexity score derived from code statements that are known to introduce code complexity (branches, loops, calls, external interfaces, ...

6. Scope of Work

The Grindery team has provided the necessary smart contract files for auditing. The audit focuses on verifying the staking and reward distribution mechanisms to ensure security, efficiency, and robustness against potential vulnerabilities.

The team has outlined the following security and functionality assumptions for the contract:

1. Adherence to Smart Contract Security Best Practices
   The audit should ensure that the smart contract follow established security standards, identifying vulnerabilities such as reentrancy, integer overflows, underflows, and permission exploits. The goal is to confirm that the contract is resistant to common attack vectors and adhere to industry best practices.

2. Strict Access Control and Administrative Permissions
   The contract should incorporate granular access control mechanisms, ensuring that only authorized entities can modify contract parameters such as reward rates, staking rules, and fund withdrawals.

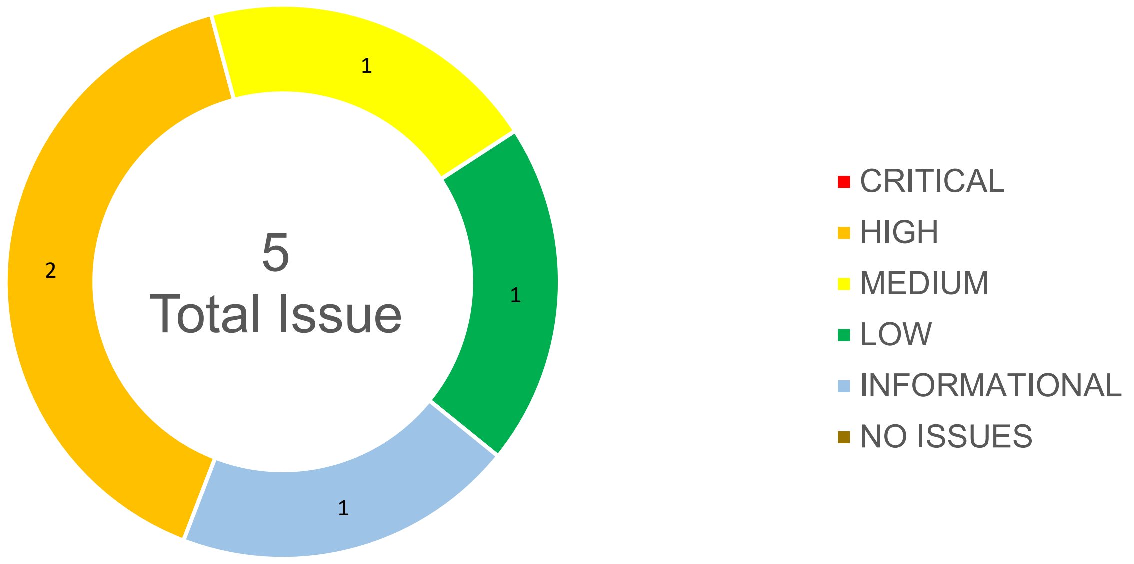
3. Accurate and Fair Reward Distribution
   The audit should verify that staking rewards are correctly computed based on predefined formulas, ensuring that reward distribution is fair, transparent, and resistant to manipulation.

4. Gas Optimization and Transaction Efficiency
   The contract logic should be reviewed for efficiency, minimizing unnecessary gas consumption and reducing costs for users.

5. State Consistency and Edge Case Handling
   The audit should ensure that data integrity is maintained across all staking and reward-related operations. This includes preventing scenarios such as reward miscalculations, unstake exploits, timestamp manipulation, and unintended stake overlaps.

The main goal of this audit will be to verify these claims and ensure that the contracts are secure, efficient, and reliable. Upon the client's request, the audit team can provide further feedback on specific areas of the contract.

6.1 Findings Overview



No	Title	Severity	Status
6.2.1	Reward Pool Depletion	HIGH	OPEN
6.2.2	Shared Reward Pool Can Be Drained By Short-Term Stakers	HIGH	OPEN
6.2.3	Missing Lock Period Validation in AddRewards Function	MEDIUM	OPEN
6.2.4	Reward Rate Manipulation	LOW	OPEN
6.2.5	Lack of Slashing Mechanism	INFORMATIONAL	OPEN

6.2 Manual and Automated Vulnerability Test

CRITICAL ISSUES

During the audit, softstack‘s experts found **no Critical issues** in the code of the smart contract.

HIGH ISSUES

During the audit, softstack‘s experts found **two High issues** in the code of the smart contract.

6.2.1 Reward Pool Depletion

Severity: HIGH
Status: OPEN
File(s) affected: Staking.sol

Attack / Description	The Staking contract has a vulnerability where the reward pool can be depleted, preventing users from claiming their legitimately earned rewards. The issue occurs because: 1. The contract allows admin to withdraw rewards from the pool without considering pending reward obligations. 2. There's no mechanism to ensure sufficient rewards remain for existing stakers. 3. Total pending rewards can exceed the available reward pool balance.

	Proof of Concept:
	
	describe("Reward Pool Depletion", function () {
	  beforeEach(async () => {
	    const Staking = await ethers.getContractFactory("Staking");
	    staking = await Staking.deploy(token.address);
	    await staking.deployed();
	    // Setup multiple users
	    [admin, user1, user2, user3] = await ethers.getSigners();
	    // Transfer tokens to users
	    await token.transfer(user1.address, stakeAmount); await token.transfer(user2.address, stakeAmount);
	    await token.transfer(user3.address, stakeAmount);
	    // Add initial rewards
	    await token.connect(admin).approve(staking.address, rewardAmount);
	    await staking.connect(admin).addRewards(1, rewardAmount);
	  });
	  it("should demonstrate reward pool depletion preventing reward
	  claims", async function () {
	  // Users stake tokens
	  await token.connect(user1).approve(staking.address, stakeAmount);
	  await token.connect(user2).approve(staking.address, stakeAmount);
	  await token.connect(user3).approve(staking.address, stakeAmount);
	  await staking.connect(user1).stake(1, stakeAmount);
	  await staking.connect(user2).stake(1, stakeAmount);
	  await staking.connect(user3).stake(1, stakeAmount);
	  // Admin maliciously withdraws rewards

	```
	  await staking.connect(admin).withdrawRewards(rewardAmount);
	  // Fast forward time
	  await ethers.provider.send("evm_increaseTime", [30 * 24 * 60 *
	    60]);
	  await ethers.provider.send("evm_mine", []);
	  // First user tries to unstake
	  await expect(
	    staking.connect(user1).unStake(1)
	  ).to.be.revertedWith("Reward pool insufficient");
	  // Even though user has valid stake and lock period is over
	  const userStake = await
	  staking.stakesByUserAndPeriod(user1.address, 1);
	  expect(userStake.stakeAmount).to.be.gt(0);
	});
	it("should show total pending rewards can exceed pool balance", async
	  function () {// Users stake tokens
	    await token.connect(user1).approve(staking.address, stakeAmount);
	    await token.connect(user2).approve(staking.address, stakeAmount);
	    await staking.connect(user1).stake(1, stakeAmount);
	    await staking.connect(user2).stake(1, stakeAmount);
	    // Calculate total pending rewards
	    const stake1 = await staking.stakesByUserAndPeriod(user1.address,
	      1);
	    const stake2 = await staking.stakesByUserAndPeriod(user2.address,
	      1);
	```

	```
	  const totalPendingRewards =
	    stake1.pendingReward.add(stake2.pendingReward);
	  // Admin withdraws most of rewards
	  const withdrawAmount =
	    rewardAmount.sub(totalPendingRewards.div(2));
	  await staking.connect(admin).withdrawRewards(withdrawAmount);
	  // Fast forward time
	  await ethers.provider.send("evm_increaseTime", [30 * 24 * 60 *
	    60]);
	  await ethers.provider.send("evm_mine", []);
	  // First user can unstake
	  await staking.connect(user1).unStake(1);
	  // Second user cannot due to insufficient rewards
	  await expect(
	    staking.connect(user2).unStake(1)
	  ).to.be.revertedWith("Reward pool insufficient");
	});
	```

Code	Line 105 – 110 (Staking.sol):
	
	`function withdrawRewards(uint256 amount) external onlyAdmin {`
	`    require(amount <= totalRewardPool, "Amount exceeds reward pool");`
	`    totalRewardPool -= amount;`
	`    stakingToken.transfer(msg.sender, amount);`
	`    emit RewardsWithdrawn(amount);`

	}

Result/Recommendation	We suggest the following fixes:
	
	Implement Reward Reservation:
	mapping(uint8 => uint256) public reservedRewards;
	function addRewards(uint8 lockPeriod, uint256 rewardAmount) external onlyAdmin {
	stakingToken.transferFrom(msg.sender, address(this), rewardAmount);
	availableRewards += rewardAmount;
	}
	
	function withdrawRewards(uint256 amount) external onlyAdmin {
	uint256 availableToWithdraw = totalRewardPool - reservedRewards;
	require(amount <= availableToWithdraw, "Cannot withdraw reserved rewards");
	totalRewardPool -= amount;
	stakingToken.transfer(msg.sender, amount);
	}
	
	Add Reward Tracking:
	function stake(uint8 lockPeriod, uint256 stakeAmount) external {
	// ... existing code ...
	reservedRewards += userStake.pendingReward;

	```
	}
	function unStake(uint8 lockPeriod) external {
	// ... existing code ...
	reservedRewards -= pendingReward;
	}
	```

6.2.2 Shared Reward Pool Can Be Drained By Short-Term Stakers

Severity: HIGH
Status: OPEN
File(s) affected: Staking.sol

Attack / Description	The contract stores all rewards in a single pool via `totalRewardPool`, regardless of which lock period the rewards were intended for. The `addRewards()` function adds rewards to this shared pool. While rewards are added with a `lockPeriod` parameter, this is only used for event emission - the actual rewards go into the shared pool. When users unstake, they can claim from this shared pool regardless of their lock period. **Impact** 1. Long-term stakers (e.g., 12-month/25% APR) can be prevented from claiming their promised higher rewards if short-term stakers (1-month/10% APR) drain the pool first 2. Creates unfair advantage for short-term stakers who can claim rewards earlier, essentially stealing rewards meant for long-term commitments 3. Undermines the incentive structure designed to encourage longer staking periods **Proof of Concept**

```
describe("Shared Reward Pool Vulnerability", function () {
 beforeEach(async () => {
  [admin, user1, user2] = await ethers.getSigners();
  // Deploy contracts
  const Staking = await ethers.getContractFactory("Staking");
  staking = await Staking.deploy(token.address);
  await staking.deployed();
  // Setup initial tokens and approvals
  await token.transfer(user1.address, stakeAmount.mul(2));
  await token.transfer(user2.address, stakeAmount);
  await token.connect(user1).approve(staking.address,
   stakeAmount.mul(2));
  await token.connect(user2).approve(staking.address, stakeAmount);
  // Add rewards to pool - enough for one 12-month stakeconst twelveMonthReward = stakeAmount.mul(2500).div(10000); // 25%
  reward
  await token.connect(admin).approve(staking.address,
   twelveMonthReward);
  await staking.connect(admin).addRewards(12, twelveMonthReward);
 });
 it("should demonstrate short-term stakers can drain rewards meant for
 long - term stakers", async function () {
 // User2 stakes for 12 months (25% reward rate)
 await staking.connect(user2).stake(12, stakeAmount);
 // User1 stakes same amount but for 1 month (10% reward rate)
 await staking.connect(user1).stake(1, stakeAmount);
```

	```
	  // Fast forward 1 month
	  await ethers.provider.send("evm_increaseTime", [30 * 24 * 60 *
	    60]);
	  await ethers.provider.send("evm_mine", []);
	  // User1 can withdraw with 10% reward
	  await staking.connect(user1).unStake(1);
	  // Fast forward 11 more months
	  await ethers.provider.send("evm_increaseTime", [11 * 30 * 24 * 60
	    * 60]);
	  await ethers.provider.send("evm_mine", []);
	  // User2 tries to withdraw but fails due to insufficient rewards
	  await expect(
	    staking.connect(user2).unStake(12)
	  ).to.be.revertedWith("Reward pool insufficient");
	});
	```

Code	Line 99 – 103 (Staking.sol):
	```
	function addRewards(uint8 lockPeriod, uint256 rewardAmount) external onlyAdmin {
	        stakingToken.transferFrom(msg.sender, address(this), rewardAmount);
	        totalRewardPool += rewardAmount;
	        emit RewardsAdded(lockPeriod, rewardAmount);
	    }
	```

Result/Recommendation	We suggest to segregate reward pools by lock period:
	
	`mapping(uint8 => uint256) public rewardPoolsByPeriod;`
	
	Track and reserve rewards when users stake:
	
	`function stake(uint8 lockPeriod, uint256 stakeAmount) external {`
	`  // ... existing checks ...`
	`  uint256 reward = (stakeAmount * rewardRatesPerPeriod[lockPeriod]) /`
	`  10000;`
	`  require(rewardPoolsByPeriod[lockPeriod] >= reward, "Insufficient`
	`  rewards for period");`
	`  rewardPoolsByPeriod[lockPeriod] -= reward;`
	`  // ... rest of function ...`
	
	`  }`
	
	Add rewards to specific period pools:
	
	`function addRewards(uint8 lockPeriod, uint256 rewardAmount) external`
	`onlyAdmin {`
	`stakingToken.transferFrom(msg.sender, address(this), rewardAmount);`
	`rewardPoolsByPeriod[lockPeriod] += rewardAmount;`
	`emit RewardsAdded(lockPeriod, rewardAmount);`
	`}`

MEDIUM ISSUES

During the audit, softstack‘s experts found **one Medium issue** in the code of the smart contract.

6.2.3 Missing Lock Period Validation in AddRewards Function

Severity: MEDIUM
Status: OPEN
File(s) affected: Staking.sol

Attack / Description	The \`addRewards\` function in the Staking contract lacks validation for lock periods, allowing rewards to be added for invalid/unconfigured staking periods. This can lead to funds becoming permanently locked in the contract since no users can stake or claim rewards for these invalid periods.
	
	Proof of Concept
	
	`describe("AddRewards Lock Period Validation", function () {`
	`  beforeEach(async () => {`
	`    const Staking = await ethers.getContractFactory("Staking");`
	`    staking = await Staking.deploy(token.address);`
	`    await staking.deployed();`
	`    // Transfer tokens to admin for rewards`
	`    await token.transfer(admin.address, rewardAmount.mul(2));`
	`    await token.connect(admin).approve(staking.address,`
	`      rewardAmount.mul(2));`
	`  });`
	`  it("should demonstrate rewards can be added for invalid lock periods",`
	`    async function () {`

	```
	  // Add rewards for invalid lock period (2 months - not configured
	  in constructor)
	  const invalidPeriod = 2;
	  // Notice this succeeds despite period=2 not being configuredawait staking.connect(admin).addRewards(invalidPeriod,
	  rewardAmount);
	// Verify rewards were added to pool
	expect(await staking.totalRewardPool()).to.equal(rewardAmount);
	// Try to stake with invalid period - should fail
	await token.transfer(user.address, stakeAmount);
	await token.connect(user).approve(staking.address, stakeAmount);
	await expect(
	  staking.connect(user).stake(invalidPeriod, stakeAmount)
	).to.be.revertedWith("Invalid lock period");
	// Admin cannot withdraw these rewards since they're effectively
	locked
	// Due to no valid staking period to claim them
	const initialAdminBalance = await token.balanceOf(admin.address);
	await staking.connect(admin).withdrawRewards(rewardAmount);
	const finalAdminBalance = await token.balanceOf(admin.address);
	// Verify rewards were withdrawn but are now stuck
	expect(finalAdminBalance).to.equal(initialAdminBalance.add(rewardAmount));
	expect(await staking.totalRewardPool()).to.equal(0);
	  });
	it("should allow adding rewards only for valid lock periods", async
	  function () {
	```

```
  // Test all valid periods from constructor
  const validPeriods = [1, 3, 6, 12];
  for (let period of validPeriods) {
    await staking.connect(admin).addRewards(period,
      rewardAmount.div(4));
    expect(await
staking.totalRewardPool()).to.equal(rewardAmount.div(4).mul(validPeriods.i
ndexOf(period) + 1));
  }
}); it("should demonstrate funds can get stuck with invalid periods",
  async function () {
    // Add rewards for multiple invalid periods
    const invalidPeriods = [2, 4, 5, 7, 8, 9, 10, 11];
    for (let period of invalidPeriods) {
      await staking.connect(admin).addRewards(period,
        rewardAmount.div(8));
    }
    // Total rewards are in pool but can never be claimed
    expect(await staking.totalRewardPool()).to.equal(rewardAmount);
    // No way to stake or claim these rewards since periods are
    invalid
    await token.transfer(user.address, stakeAmount);
    await token.connect(user).approve(staking.address, stakeAmount);
    for (let period of invalidPeriods) {
      await expect(
```

	```
	      staking.connect(user).stake(period, stakeAmount)
	    ).to.be.revertedWith("Invalid lock period");
	  }
	});
	```

Code	Line 99 - 103 (Staking.sol):
	```
	function addRewards(uint8 lockPeriod, uint256 rewardAmount) external onlyAdmin {
	    stakingToken.transferFrom(msg.sender, address(this), rewardAmount);
	    totalRewardPool += rewardAmount;
	    emit RewardsAdded(lockPeriod, rewardAmount);
	  }
	```

Result/Recommendation	We recommend the following changes:
	
	Add lock period validation in the `addRewards` function:
	```
	function addRewards(uint8 lockPeriod, uint256 rewardAmount) external
	onlyAdmin {
	  require(rewardRatesPerPeriod[lockPeriod] != 0, "Invalid lock period");
	  stakingToken.transferFrom(msg.sender, address(this), rewardAmount);
	  totalRewardPool += rewardAmount;
	emit RewardsAdded(lockPeriod, rewardAmount);
	}
	```

LOW ISSUES

During the audit, softstack's experts found **one Low issue** in the code of the smart contract

6.2.4 Reward Rate Manipulation

Severity: LOW
Status: OPEN
File(s) affected: Staking.sol

Attack / Description	The staking contract contains a vulnerability where the admin can manipulate reward rates for existing stakes through the `updateRewardRate` function.
	
	This allows the admin to arbitrarily increase or decrease rewards for locked stakes after users have committed their tokens, breaking the principle of immutable staking terms and potentially causing significant financial impact to users. The root cause is that the `computeReward` function uses the current reward rate from `rewardRatesPerPeriod` mapping instead of storing and using the rate that was active when the user initially staked.
	
	This means any changes to reward rates affect both new and existing stakes.
	
	Proof of Concept
	
	`it("should demonstrate reward rate manipulation impact on unstaking",`
	`  async function () {// User stakes with 10% rate`
	`    await staking.connect(user).stake(1, stakeAmount);`
	`    // Store initial pending reward`
	`    const initialStakeInfo = await staking.getStakeInfo(1);`
	`    // Admin maliciously increases rate before user can unstake`

	```
	  await staking.connect(admin).updateRewardRate(1, 2000);
	  // Fast forward past lock period
	  await ethers.provider.send("evm_increaseTime", [30 * 24 * 60 * 60]);
	  await ethers.provider.send("evm_mine", []);
	  // Try to unstake - this will use the new higher rate
	  await expect(staking.connect(user).unStake(1))
	    .to.emit(staking, "Unstaked")
	    .withArgs(user.address, 1, 0, stakeAmount.mul(2000).div(10000));
	  // 20% reward
	});
	```

Code	Line 112 - 116 (Staking.sol):
	```
	function updateRewardRate(uint8 lockPeriod, uint256 rate) external onlyAdmin {
	        require(rate > 0, "Rate should be greater than 0");
	        rewardRatesPerPeriod[lockPeriod] = rate;
	        emit RewardRateUpdated(lockPeriod, rate);
	  }
	```

Result/Recommendation	Store the reward rate at stake time in the Stake struct:
	```
	struct Stake {
	  uint256 stakeStartTime;
	  uint256 stakeAmount;
	  uint256 pendingReward;
	```

```
  uint256 lockedRewardRate; // Add this
  }
```

Lock in the rate when staking:

```
function stake(uint8 lockPeriod, uint256 stakeAmount) external {
  // ...
  userStake.lockedRewardRate = rewardRatesPerPeriod[lockPeriod];
  // ...
  }
```

Use the locked rate in computeReward:

```
function computeReward(...) {
  // ...uint256 rewardRate = userStake.lockedRewardRate; // Use stored rate
  // ...
  }
```

This ensures that each stake maintains its original reward rate terms, protecting users from potential manipulation while still allowing the admin to update rates for future stakes.

INFORMATIONAL ISSUES

During the audit, softstack‘s experts found **one Informational issue** in the code of the smart contract.

6.2.5 Lack of Slashing Mechanism

Severity: INFORMATIONAL
Status: OPEN
File(s) affected: Staking.sol

Attack / Description	The contract implements staking periods with rewards but does not include any penalty mechanism for:
	
	1. Attempted early withdrawals
	2. Multiple failed unstaking attempts
	3. Other potentially malicious behavior
	
	The proof of concept demonstrates that users can:
	- Make unlimited early unstaking attempts without consequences
	- Receive full rewards despite multiple failed withdrawal attempts
	- Get their entire stake back plus full rewards after the lock period
	
	Proof of Concept
	
	`describe("Lack of Slashing Mechanism", function () {`
	`  beforeEach(async () => {`
	`    const Staking = await ethers.getContractFactory("Staking");`
	`    staking = await Staking.deploy(token.address);`
	`    await staking.deployed();`
	`    await token.transfer(user.address, stakeAmount.mul(2));`
	`    await token.connect(user).approve(staking.address,`
	`      stakeAmount.mul(2));`
	`    // Add rewards to pool`
	`    await token.connect(admin).approve(staking.address, rewardAmount);`

```
  await staking.connect(admin).addRewards(1, rewardAmount);
}); it("should demonstrate no penalty for attempted early unstaking",
  async function () {
    // User stakes tokens
    await staking.connect(user).stake(1, stakeAmount);
    // Get initial stake amount
    const initialStakeInfo = await staking.getStakeInfo(1);
    const initialStakeAmount = initialStakeInfo.stakeAmount;
    // Try to unstake early (should fail)
    await expect(staking.connect(user).unStake(1))
      .to.be.revertedWith("Lock period not over");
    // Check stake amount remains unchanged despite unstaking attempt
    const afterAttemptInfo = await staking.getStakeInfo(1);
    expect(afterAttemptInfo.stakeAmount).to.equal(initialStakeAmount);
    // Fast forward to end of lock period
    await ethers.provider.send("evm_increaseTime", [30 * 24 * 60 *
      60]);
    await ethers.provider.send("evm_mine", []);
    // Successfully unstake
    await staking.connect(user).unStake(1);
    // Verify user gets back full amount despite earlier unstaking
    attempt
    const finalBalance = await token.balanceOf(user.address);
    expect(finalBalance).to.be.gt(stakeAmount); // Gets back stake +
    rewards
  });
 it("should demonstrate lack of cumulative penalties", async function
  () {
  // User stakes tokens
  await staking.connect(user).stake(1, stakeAmount);
  // Make multiple failed unstaking attempts
  for (let i = 0; i < 5; i++) {
    await expect(staking.connect(user).unStake(1))
      .to.be.revertedWith("Lock period not over");
  }
  // Fast forward to end of lock period
  await ethers.provider.send("evm_increaseTime", [30 * 24 * 60 *
    60]);
  await ethers.provider.send("evm_mine", []);
  // Successfully unstake
  const tx = await staking.connect(user).unStake(1);
  // Verify full rewards despite multiple failed attempts
  await expect(tx)
    .to.emit(staking, "Unstaked")
    .withArgs(user.address, 1, 0, stakeAmount.div(10)); // Still
 gets full 10 % reward
 });
});
```

Code	Line 33 - 41 (Staking.sol):
	
	`constructor(address _stakingToken) {`
	`    require(_stakingToken != address(0), "Invalid staking token address");`
	`    admin = msg.sender;`
	`    stakingToken = IERC20(_stakingToken);`
	`    rewardRatesPerPeriod[1] = 1000; // 10% for 1 month`
	`    rewardRatesPerPeriod[3] = 1200; // 12% for 3 months`
	`    rewardRatesPerPeriod[6] = 1800; // 18% for 6 months`
	`    rewardRatesPerPeriod[12] = 2500; // 25% for 12 months`
	`  }`

Result/Recommendation	We suggest the following fixes:
	
	Implement Slashing Mechanism:
	
	`struct Stake {`
	`  uint256 stakeStartTime;`
	`  uint256 stakeAmount;`
	`  uint256 pendingReward;`
	`  uint256 failedAttempts; // Track failed attempts`
	
	`  }`
	
	Add Penalty Logic:
	`function unStake(uint8 lockPeriod) external {`

```
Stake storage userStake =
  stakesByUserAndPeriod[msg.sender][lockPeriod];
if (block.timestamp < userStake.stakeStartTime + (lockPeriod * 30
days)) {
  userStake.failedAttempts++; userStake.slashRate += 5; // 5% penalty per attempt
  revert("Lock period not over");
}
// Apply accumulated penalties
uint256 penalty = (userStake.stakeAmount * userStake.slashRate) / 100;
uint256 finalAmount = userStake.stakeAmount - penalty;
// ... rest of unstaking logic
}
```

6.3 SWC Attacks

ID	Title	Relationships	Test Result
SWC-131	Presence of unused variables	CWE-1164: Irrelevant Code	✓

ID	Title	Relationships	Test Result
SWC-130	Right-To-Left-Override control character (U+202E)	CWE-451: User Interface (UI) Misrepresentation of Critical Information	✓
SWC-129	Typographical Error	CWE-480: Use of Incorrect Operator	✓
SWC-128	DoS With Block Gas Limit	CWE-400: Uncontrolled Resource Consumption	✓
SWC-127	Arbitrary Jump with Function Type Variable	CWE-695: Use of Low-Level Functionality	✓
SWC-125	Incorrect Inheritance Order	CWE-696: Incorrect Behavior Order	✓
SWC-124	Write to Arbitrary Storage Location	CWE-123: Write-what-where Condition	✓
SWC-123	Requirement Violation	CWE-573: Improper Following of Specification by Caller	✓
SWC-122	Lack of Proper Signature Verification	CWE-345: Insufficient Verification of Data Authenticity	✓
SWC-121	Missing Protection against Signature Replay Attacks	CWE-347: Improper Verification of Cryptographic Signature	✓

ID	Title	Relationships	Test Result
SWC-120	Weak Sources of Randomness from Chain Attributes	CWE-330: Use of Insufficiently Random Values	✓
SWC-119	Shadowing State Variables	CWE-710: Improper Adherence to Coding Standards	✓
SWC-118	Incorrect Constructor Name	CWE-665: Improper Initialization	✓
SWC-117	Signature Malleability	CWE-347: Improper Verification of Cryptographic Signature	✓
SWC-116	Timestamp Dependence	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	✓
SWC-115	Authorization through tx.origin	CWE-477: Use of Obsolete Function	✓
SWC-114	Transaction Order Dependence	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	✓
SWC-113	DoS with Failed Call	CWE-703: Improper Check or Handling of Exceptional Conditions	✓
SWC-112	Delegatecall to Untrusted Callee	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	✓
SWC-111	Use of Deprecated Solidity Functions	CWE-477: Use of Obsolete Function	✓

ID	Title	Relationships	Test Result
SWC-110	Assert Violation	CWE-670: Always-Incorrect Control Flow Implementation	✓
SWC-109	Uninitialized Storage Pointer	CWE-824: Access of Uninitialized Pointer	✓
SWC-108	State Variable Default Visibility	CWE-710: Improper Adherence to Coding Standards	✓
SWC-107	Reentrancy	CWE-841: Improper Enforcement of Behavioral Workflow	✓
SWC-106	Unprotected SELFDESTRUCT Instruction	CWE-284: Improper Access Control	✓
SWC-105	Unprotected Ether Withdrawal	CWE-284: Improper Access Control	✓
SWC-104	Unchecked Call Return Value	CWE-252: Unchecked Return Value	✓
SWC-103	Floating Pragma	CWE-664: Improper Control of a Resource Through its Lifetime	✓
SWC-102	Outdated Compiler Version	CWE-937: Using Components with Known Vulnerabilities	✓
SWC-101	Integer Overflow and Underflow	CWE-682: Incorrect Calculation	✓
SWC-100	Function Default Visibility	CWE-710: Improper Adherence to Coding Standards	✓

6.4 Unit Tests

```
Staking Contract Constructor
✓ should deploy with a valid staking token address (52ms)
✓ should revert if the staking token address is zero Staking
✓ should allow users to stake tokens
✓ should not allow staking more than once for the same period (44ms)
✓ should prevent staking with zero amount
✓ should prevent staking for an invalid lock period Unstaking
✓ should revert if the user has no stake
✓ should revert if the reward pool is insufficient
✓ should revert unstaking before the lock period ends
✓ should allow unstaking after the lock period is over (49ms)
✓ should revert unstaking before the lock period ends Reward Pool
✓ should allow admin to add rewards
✓ should not allow non-admin to add rewards
✓ should allow admin to withdraw rewards
✓ should not allow non-admin to withdraw rewards
✓ should not allow withdrawing more rewards than available Reward Rate Update
✓ should allow admin to update reward rates
✓ should not allow non-admin to update reward rates
✓ should revert if the rate is set to 0 Compute Reward
✓ should return 0 reward before the lock period ends
✓ should correctly calculate rewards after the lock period ends

21 passing (2s)
```

6.5 Verify Claims

6.5.1 Adherence to Smart Contract Security Best Practices

The audit should ensure that the smart contract follow established security standards, identifying vulnerabilities such as reentrancy, integer overflows, underflows, and permission exploits. The goal is to confirm that the contract is resistant to common attack vectors and adhere to industry best practices.
Status: tested and verified ✅

6.5.2 Strict Access Control and Administrative Permissions

The contract should incorporate granular access control mechanisms, ensuring that only authorized entities can modify contract parameters such as reward rates, staking rules, and fund withdrawals.
Status: tested and verified ✅

6.5.3 Accurate and Fair Reward Distribution

The audit should verify that staking rewards are correctly computed based on predefined formulas, ensuring that reward distribution is fair, transparent, and resistant to manipulation.
Status: tested and verified ✅

6.5.4 Gas Optimization and Transaction Efficiency

The contract logic should be reviewed for efficiency, minimizing unnecessary gas consumption and reducing costs for users.
Status: tested and verified ✅

6.5.5 State Consistency and Edge Case Handling

The audit should ensure that data integrity is maintained across all staking and reward-related operations. This includes preventing scenarios such as reward miscalculations, unstake exploits, timestamp manipulation, and unintended stake overlaps.
Status: tested and verified ✅

7. Executive Summary

Two independent softstack experts performed an unbiased and isolated audit of the smart contract codebase provided by the Grindery team. The main objective of the audit was to verify the security and functionality claims of the smart contract. The audit process involved a thorough manual code review and automated security testing.

Overall, the audit identified a total of one issue, classified as follows:
- No critical issues were found.
- 2 high severity issues were found.
- 1 medium severity issues were found.
- 1 low severity issues were discovered
- 1 informational issues were identified

The audit report provides detailed descriptions of each identified issue, including severity levels, proof of concepts and recommendations for mitigation. It also includes code snippets, where applicable, to demonstrate the issues and suggest possible fixes. We recommend that the Grindery team review the suggestions.

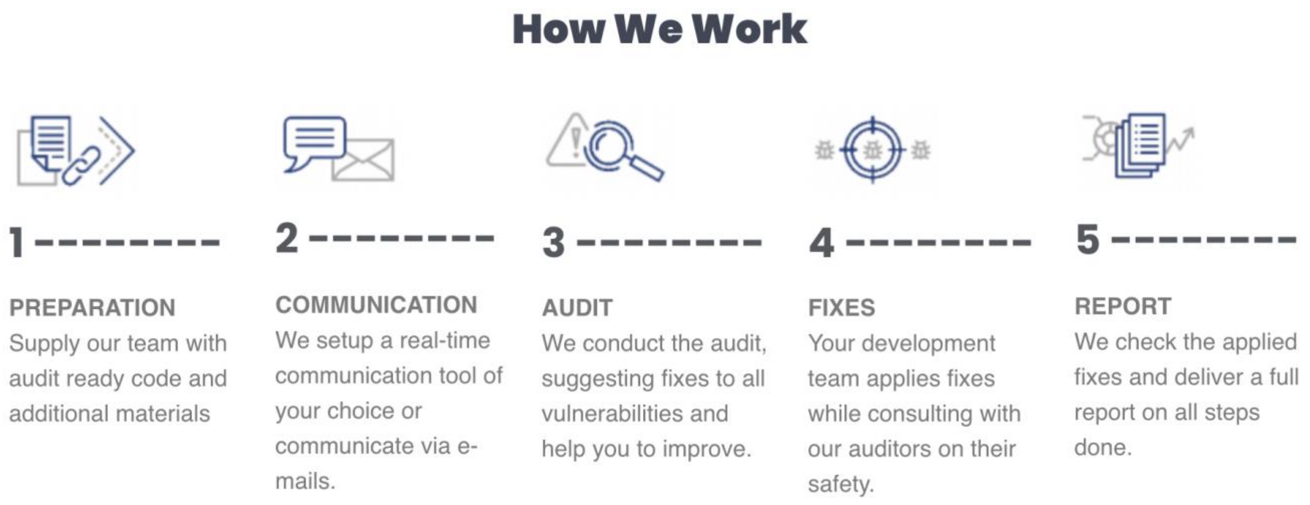
8. About the Auditor

Established in 2017 under the name Chainsulting, and rebranded as softstack GmbH in 2023, softstack has been a trusted name in Web3 Security space. Within the rapidly growing Web3 industry, softstack provides a comprehensive range of offerings that include software development, cybersecurity, and consulting services. Softstack's competency extends across the security landscape of prominent blockchains like Solana, Tezos, TON, Ethereum and Polygon. The company is widely recognized for conducting thorough code audits aimed at mitigating risk and promoting transparency.

The firm's proficiency lies particularly in assessing and fortifying smart contracts of leading DeFi projects, a testament to their commitment to maintaining the integrity of these innovative financial platforms. To date, softstack plays a crucial role in safeguarding over $100 billion worth of user funds in various DeFi protocols.

Underpinned by a team of industry veterans possessing robust technical knowledge in the Web3 domain, softstack offers industry-leading smart contract audit services. Committed to evolving with their clients' ever-changing business needs, softstack's approach is as dynamic and innovative as the industry it serves.

Check our website for further information: https://softstack.io



How We Work

1 --------
PREPARATION
Supply our team with audit ready code and additional materials

2 --------
COMMUNICATION
We setup a real-time communication tool of your choice or communicate via e-mails.

3 --------
AUDIT
We conduct the audit, suggesting fixes to all vulnerabilities and help you to improve.

4 --------
FIXES
Your development team applies fixes while consulting with our auditors on their safety.

5 --------
REPORT
We check the applied fixes and deliver a full report on all steps done.