

MixBytes()

DIA Lumina Staking Security Audit Report

JUNE 20, 2025

Table of Contents

1. Introduction	4
1.1 Disclaimer	4
1.2 Executive Summary	4
1.3 Project Overview	5
1.4 Security Assessment Methodology	8
1.5 Risk Classification	10
1.6 Summary of Findings	11
2. Findings Report	14
2.1 Critical	14
C-1 paidOutReward subtraction may lead to unstake and unstakePrincipal functions reverting	14
C-2 Inflation Attack in DIAExternalStaking Allows Stealing First Depositor's Stake	15
C-3 Missing Reward Accumulator Initialization in Stake Creation	16
C-4 currentStore.principal is not subtracted after it was withdrawn, allowing users to claim it again infinitely	17
C-5 WDIA prohibits zero transfers, which will block rewards distribution to beneficiary address if 100% goes to principal unstaker	18
2.2 High	19
H-1 Rewards are lost if user requests unstake in less than one day	19
H-2 Rewards claiming may fail due to potentially decreasing parameters used in calculation	20
H-3 Pending principal wallet share BPS value is not saved in storage	21
H-4 Incorrect Reward Calculation When Reward Rate Changes	22
H-5 Incorrect Initialization of lastWithdrawalResetDay Causes Withdrawal Limits to Malfunction	24
H-6 Reward Accumulation Manipulation Through Frequent Claims	25
H-7 Incorrect Calculation of Accrued Rewards in updateRewardRatePerDay	27

2.3 Medium	28
M-1 Daily withdrawal limit decreases with totalPoolSize reduction	28
M-2 Principal wallet pending share BPS usage is missed in getRewardForStakingStore function	29
M-3 stakingIndicesByPrincipalUnstaker mapping not updated in updatePrincipalUnstaker function	30
M-4 Whitelist can be bypassed in DIAWhitelistedStaking	31
M-5 Principal withdrawal may be blocked by insufficient rewards wallet balance in DIAWhitelistedStaking	32
2.4 Low	33
L-1 Incorrect event emission in updatePrincipalPayoutWallet in DIAStakingCommons	33
L-2 Lack of slippage protection in unstake function in DIAExternalStaking	34
L-3 Missing tokensStaked update in unstakePrincipal causing accounting drift in DIAWhitelistedStaking	35
L-4 Missing zero-address validation and boundaries check in DIARewardsDistribution constructor	36
L-5 Redundant zero check for unsigned integer in setDailyWithdrawalThreshold	37
L-6 Unused logic related to the daily withdrawal threshold in DIAStakingCommons	38
L-7 Missing zero amount validation in token transfers in DIAWhitelistedStaking	39
L-8 Multiple legacy and safety issues in WDIA inherited from WETH9	40
L-9 Staking Limit Cannot Be Updated in Case of Incorrect Value	41
L-10 Lack of checks in the DIAExternalStaking and DIAWhitelistedStaking constructors	42
L-11 Unused REWARDS_TOKEN Variable	43
L-12 Unlimited reward rate enables overflow DoS attack in DIARewardsDistribution	44
L-13 Wrong role checked in onlyBeneficiaryOrPayoutWallet modifier in DIAExternalStaking	45
L-14 Principal payout wallet can be set to zero address causing permanent fund loss in DIAStakingCommons	46
L-15 Duplicate IERC20 import in DIAExternalStaking	47
L-16 Replace hardcoded 24 * 60 * 60 with SECONDS_PER_DAY in DIAWhitelistedStaking	48

L-17 Potential out-of-gas risk in <code>_removeStakingIndexFromAddressMapping</code> with large stake counts	49
L-18 Shadowed variables in <code>DIAWhitelistedStaking</code> constructor	50
L-19 Unnecessary update of <code>stakingStartTime</code> after full principal withdrawal in <code>unstakePrincipal()</code> and <code>unstake()</code>	51
L-20 Missing event emission in <code>updatePrincipalUnstaker()</code> in <code>DIAExternalStaking</code> and <code>DIAStakingCommons</code>	52
L-21 Incorrect Parameters in Claimed Event	53
L-22 Redundant Return Value from <code>_updateRewardAccumulator</code>	54
3. About MixBytes	55

1. Introduction

1.1 Disclaimer

The audit makes no statements or warranties regarding the utility, safety, or security of the code, the suitability of the business model, investment advice, endorsement of the platform or its products, the regulatory regime for the business model, or any other claims about the fitness of the contracts for a particular purpose or their bug-free status.

1.2 Executive Summary

The protocol implements a flexible, role-based staking system where users can lock tokens to earn rewards, with support for whitelisted staking, external reward funding, and customizable beneficiary configurations. Contracts like `DIAWhitelistedStaking` and `DIAExternalStaking` manage stake tracking, reward accrual, and payout routing, while `DIARewardsDistribution` handles periodic reward injection. The system emphasizes modularity, allowing principal and reward withdrawal paths to be controlled by different roles.

The audit was conducted over 4 days by 3 auditors.

During the audit, in addition to checking well-known attack vectors and those listed in our internal checklist, we carefully investigated the following:

Access Control and State Validation Mechanisms. The staking system implements comprehensive validation to prevent unstake operations on non-existent staking stores. Analysis confirmed bounds checking and state validation across all functions, preventing unauthorized access to staking data structures. The contracts maintain strict validation of store existence before operations by accessing the `structure` field, which may be initialized only during the stake creation.

Authorization Framework and Ownership Model. The system uses role-based access control through beneficiary, principal unstaker, and payout wallet designations. Verification confirmed unauthorized users cannot claim principal or rewards belonging to others, with ownership verification at critical operations.

Principal Recovery and Fund Security. Analysis focused on whether deposited principal can always be claimed back and if external entities can block claims. While the system generally ensures principal recovery, we identified a critical issue (Critical-1) and a high-severity issue (High-2) that can block principal and rewards claims under certain conditions. This highlights the need for additional safeguards to prevent scenarios where users might lose fund access.

Whitelist Implementation and Security. The whitelist mechanism includes comprehensive checks for addition/removal functions with access restrictions. The system validates that only authorized addresses can modify the whitelist, maintaining approved staker list integrity. However, analysis revealed an issue (Medium-4) related to potential bypass scenarios allowing unauthorized participation under specific conditions.

Temporal Security and Sandwich Attack Prevention. The configured `unstakingDuration` effectively prevents sandwich attacks by enforcing mandatory time delays between unstake requests and execution. This temporal barrier prevents staking and unstaking within the same block, eliminating rapid exploitation of reward distribution windows.

Economic Security and Attack Vector Analysis. Examination of economic attack vectors included share inflation attacks, reward rate manipulation, pool donation mechanisms, and first depositor advantages. The share calculation mechanism requires particular attention as a critical component for fair reward distribution. While basic mechanisms are sound, improvements in share calculation precision and consistency would enhance system security.

State Management and System Initialization. Contracts implement initialization procedures for critical variables with default value handling and state consistency across operations. Verification confirmed correct handling of edge cases including staking limit boundaries, zero principal payout and unstaker addresses scenarios, and daily withdrawal threshold bypass conditions.

Secure ERC20 Integration. The system employs the SafeERC20 library for all token transfers, ensuring secure and consistent handling of ERC20 operations. Additionally, the `nonReentrant` modifier is applied to key functions interacting with staking token to prevent potential reentrancy attacks.

Reward Accrual Timing in DIAWhitelistedStaking. In the current implementation, the statement `rewardLastUpdateTime += SECONDS_IN_A_DAY * daysElapsed;` in `_updateRewardAccumulator()` and `updateRewardRatePerDay()` causes any remaining fraction of the day (i.e., less than a full day) to be excluded from reward accrual until the next update. As a result, if `daysElapsed` rounds down to zero, even if nearly a full day has passed, no rewards are added during that update. However, this remaining portion will be accrued later, based on the updated `rewardRatePerDay`. This behavior reflects the design assumption that rewards are accrued in full-day increments and credited at the end of each day using the most current reward rate. It is important to consider this when updating the reward rate, as it directly impacts how rewards for partial days are eventually applied.

Several serious issues were identified during the audit. It is important to improve the current code quality and development approach. All of the issues are listed below and should be addressed to ensure reliable and secure operation of the protocol. It is highly recommended to write tests for all main functions and user scenarios, and to make greater use of code inheritance, as there are multiple places in the protocol with duplicated logic.

1.3 Project Overview

Summary

Title	Description
Client Name	DIA

Title	Description
Project Name	Lumina Staking
Type	Solidity
Platform	EVM
Timeline	22.05.2025 - 18.06.2025

Scope of Audit

File	Link
contracts/DIAExternalStaking.sol	DIAExternalStaking.sol
contracts/DIARewardsDistribution.sol	DIARewardsDistribution.sol
contracts/DIAStakingCommons.sol	DIAStakingCommons.sol
contracts/DIAWhitelistedStaking.sol	DIAWhitelistedStaking.sol
contracts/StakingErrorsAndEvents.sol	StakingErrorsAndEvents.sol
contracts/WDIA.sol	WDIA.sol

Versions Log

Date	Commit Hash	Note
22.05.2025	50b709199461ae48e3186d5a04cd22eff39ebcf5	Initial Commit
09.06.2025	dfd0bb6dc60735ed6983ee2aa4b10c7fac875ff6	Commit for Re-audit
16.06.2025	a29131c47dbe17dfd3a04f1b9bc5b6ad8e65ed26	Commit for Re-audit
17.06.2025	71d6ba3654acd85407c3f83bfeae88f73b38f0a7	Commit for Re-audit
18.06.2025	0de3a1ec0a4cb3e7fd487429b1f89c9441ffe03e	Commit with Updates

Mainnet Deployments

File	Address	Blockchain
DIAExternalStaking.sol	0x677Cf1...059a5919	DIA
DIAWhitelistedStaking.sol	0x530643...d90389c8	DIA
WDIA.sol	0x9F5dA8...15cBdCa7	DIA

1.4 Security Assessment Methodology

Project Flow

Stage	Scope of Work
Interim audit	Project Architecture Review: <ul style="list-style-type: none">• Review project documentation• Conduct a general code review• Perform reverse engineering to analyze the project's architecture based solely on the source code• Develop an independent perspective on the project's architecture• Identify any logical flaws in the design <p>OBJECTIVE: UNDERSTAND THE OVERALL STRUCTURE OF THE PROJECT AND IDENTIFY POTENTIAL SECURITY RISKS.</p>
	Code Review with a Hacker Mindset: <ul style="list-style-type: none">• Each team member independently conducts a manual code review, focusing on identifying unique vulnerabilities.• Perform collaborative audits (pair auditing) of the most complex code sections, supervised by the Team Lead.• Develop Proof-of-Concepts (PoCs) and conduct fuzzing tests using tools like Foundry, Hardhat, and BOA to uncover intricate logical flaws.• Review test cases and in-code comments to identify potential weaknesses. <p>OBJECTIVE: IDENTIFY AND ELIMINATE THE MAJORITY OF VULNERABILITIES, INCLUDING THOSE UNIQUE TO THE INDUSTRY.</p>
	Code Review with a Nerd Mindset: <ul style="list-style-type: none">• Conduct a manual code review using an internally maintained checklist, regularly updated with insights from past hacks, research, and client audits.• Utilize static analysis tools (e.g., Slither, Mythril) and vulnerability databases (e.g., Solodit) to uncover potential undetected attack vectors. <p>OBJECTIVE: ENSURE COMPREHENSIVE COVERAGE OF ALL KNOWN ATTACK VECTORS DURING THE REVIEW PROCESS.</p>

Stage	Scope of Work
	<p>Consolidation of Auditors' Reports:</p> <ul style="list-style-type: none"> • Cross-check findings among auditors • Discuss identified issues • Issue an interim audit report for client review <p>OBJECTIVE: COMBINE INTERIM REPORTS FROM ALL AUDITORS INTO A SINGLE COMPREHENSIVE DOCUMENT.</p>
Re-audit	<p>Bug Fixing & Re-Audit:</p> <ul style="list-style-type: none"> • The client addresses the identified issues and provides feedback • Auditors verify the fixes and update their statuses with supporting evidence • A re-audit report is generated and shared with the client <p>OBJECTIVE: VALIDATE THE FIXES AND REASSESS THE CODE TO ENSURE ALL VULNERABILITIES ARE RESOLVED AND NO NEW VULNERABILITIES ARE ADDED.</p>
Final audit	<p>Final Code Verification & Public Audit Report:</p> <ul style="list-style-type: none"> • Verify the final code version against recommendations and their statuses • Check deployed contracts for correct initialization parameters • Confirm that the deployed code matches the audited version • Issue a public audit report, published on our official GitHub repository • Announce the successful audit on our official X account <p>OBJECTIVE: PERFORM A FINAL REVIEW AND ISSUE A PUBLIC REPORT DOCUMENTING THE AUDIT.</p>

1.5 Risk Classification

Severity Level Matrix

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** – Theft from 0.5% OR partial/full blocking of funds (>0.5%) on the contract without the possibility of withdrawal OR loss of user funds (>1%) who interacted with the protocol.
- **Medium** – Contract lock that can only be fixed through a contract upgrade OR one-time theft of rewards or an amount up to 0.5% of the protocol's TVL OR funds lock with the possibility of withdrawal by an admin.
- **Low** – One-time contract lock that can be fixed by the administrator without a contract upgrade.

Likelihood

- **High** – The event has a 50-60% probability of occurring within a year and can be triggered by any actor (e.g., due to a likely market condition that the actor cannot influence).
- **Medium** – An unlikely event (10-20% probability of occurring) that can be triggered by a trusted actor.
- **Low** – A highly unlikely event that can only be triggered by the owner.

Action Required

- **Critical** – Must be fixed as soon as possible.
- **High** – Strongly advised to be fixed to minimize potential risks.
- **Medium** – Recommended to be fixed to enhance security and stability.
- **Low** – Recommended to be fixed to improve overall robustness and effectiveness.

Finding Status

- **Fixed** – The recommended fixes have been implemented in the project code and no longer impact its security.
- **Partially Fixed** – The recommended fixes have been partially implemented, reducing the impact of the finding, but it has not been fully resolved.
- **Acknowledged** – The recommended fixes have not yet been implemented, and the finding remains unresolved or does not require code changes.

1.6 Summary of Findings

Findings Count

Severity	Count
Critical	5
High	7
Medium	5
Low	22

Findings Statuses

ID	Finding	Severity	Status
C-1	<code>paidOutReward</code> subtraction may lead to <code>unstake</code> and <code>unstakePrincipal</code> functions reverting	Critical	Fixed
C-2	Inflation Attack in <code>DIAExternalStaking</code> Allows Stealing First Depositor's Stake	Critical	Fixed
C-3	Missing Reward Accumulator Initialization in Stake Creation	Critical	Fixed
C-4	<code>currentStore.principal</code> is not subtracted after it was withdrawn, allowing users to claim it again infinitely	Critical	Fixed
C-5	<code>WDIA</code> prohibits zero transfers, which will block rewards distribution to beneficiary address if 100% goes to principal unstaker	Critical	Fixed
H-1	Rewards are lost if user requests unstake in less than one day	High	Acknowledged
H-2	Rewards claiming may fail due to potentially decreasing parameters used in calculation	High	Fixed
H-3	Pending principal wallet share BPS value is not saved in storage	High	Fixed
H-4	Incorrect Reward Calculation When Reward Rate Changes	High	Fixed

H-5	Incorrect Initialization of <code>lastWithdrawalResetDay</code> Causes Withdrawal Limits to Malfunction	High	Fixed
H-6	Reward Accumulation Manipulation Through Frequent Claims	High	Fixed
H-7	Incorrect Calculation of Accrued Rewards in <code>updateRewardRatePerDay</code>	High	Fixed
M-1	Daily withdrawal limit decreases with <code>totalPoolSize</code> reduction	Medium	Acknowledged
M-2	Principal wallet pending share BPS usage is missed in <code>getRewardForStakingStore</code> function	Medium	Fixed
M-3	<code>stakingIndicesByPrincipalUnstaker</code> mapping not updated in <code>updatePrincipalUnstaker</code> function	Medium	Fixed
M-4	Whitelist can be bypassed in <code>DIAWhitelistedStaking</code>	Medium	Fixed
M-5	Principal withdrawal may be blocked by insufficient rewards wallet balance in <code>DIAWhitelistedStaking</code>	Medium	Fixed
L-1	Incorrect event emission in <code>updatePrincipalPayoutWallet</code> in <code>DIAStakingCommons</code>	Low	Fixed
L-2	Lack of slippage protection in <code>unstake</code> function in <code>DIAExternalStaking</code>	Low	Fixed
L-3	Missing <code>tokensStaked</code> update in <code>unstakePrincipal</code> causing accounting drift in <code>DIAWhitelistedStaking</code>	Low	Fixed
L-4	Missing zero-address validation and boundaries check in <code>DIARewardsDistribution</code> constructor	Low	Fixed
L-5	Redundant zero check for unsigned integer in <code>setDailyWithdrawalThreshold</code>	Low	Fixed
L-6	Unused logic related to the daily withdrawal threshold in <code>DIAStakingCommons</code>	Low	Fixed
L-7	Missing zero amount validation in token transfers in <code>DIAWhitelistedStaking</code>	Low	Fixed

L-8	Multiple legacy and safety issues in <code>WDIA</code> inherited from <code>WETH9</code>	Low	Fixed
L-9	Staking Limit Cannot Be Updated in Case of Incorrect Value	Low	Fixed
L-10	Lack of checks in the <code>DIAExternalStaking</code> and <code>DIAWhitelistedStaking</code> constructors	Low	Fixed
L-11	Unused <code>REWARDS_TOKEN</code> Variable	Low	Fixed
L-12	Unlimited reward rate enables overflow DoS attack in <code>DIARewardsDistribution</code>	Low	Fixed
L-13	Wrong role checked in <code>onlyBeneficiaryOrPayoutWallet</code> modifier in <code>DIAExternalStaking</code>	Low	Fixed
L-14	Principal payout wallet can be set to zero address causing permanent fund loss in <code>DIAStakingCommons</code>	Low	Fixed
L-15	Duplicate <code>IERC20</code> import in <code>DIAExternalStaking</code>	Low	Fixed
L-16	Replace hardcoded <code>24 * 60 * 60</code> with <code>SECONDS_PER_DAY</code> in <code>DIAWhitelistedStaking</code>	Low	Fixed
L-17	Potential out-of-gas risk in <code>_removeStakingIndexFromAddressMapping</code> with large stake counts	Low	Fixed
L-18	Shadowed variables in <code>DIAWhitelistedStaking</code> constructor	Low	Fixed
L-19	Unnecessary update of <code>stakingStartTime</code> after full principal withdrawal in <code>unstakePrincipal()</code> and <code>unstake()</code>	Low	Fixed
L-20	Missing event emission in <code>updatePrincipalUnstaker()</code> in <code>DIAExternalStaking</code> and <code>DIAStakingCommons</code>	Low	Fixed
L-21	Incorrect Parameters in Claimed Event	Low	Fixed
L-22	Redundant Return Value from <code>_updateRewardAccumulator</code>	Low	Fixed

2. Findings Report

2.1 Critical

C-1	paidOutReward subtraction may lead to unstake and unstakePrincipal functions reverting		
Severity	Critical	Status	Fixed in dfd0bb6d

Description

In the `DIASWhitelistedStaking` contract, when a user claims rewards for the first time, the `currentStore.reward` value is zeroed out after the rewards are paid out. If the user then initiates an unstaking request and the unstaking period elapses, the `unstake` function may revert when calculating `rewardToSend = currentStore.reward - currentStore.paidOutReward` because there may be not enough time passed since the beginning of staking, so that `currentStore.reward` is less than `currentStore.paidOutReward`.

This will lead to `unstake` and `unstakePrincipal` functions reverting, making it impossible for user to unstake their principal and rewards. It is important to note that the user won't be able to wait for more rewards as they will not be able to request unstaking for the second time if the first request wasn't fulfilled yet.

Recommendation

We recommend removing the internal accounting for the `currentStore.reward` and `currentStore.paidOutReward` variables. `rewardToSend` variable, which is used inside `unstake` and `unstakePrincipal` functions, should be calculated based on the value returned from the `getRewardForStakingStore` function. `currentStore.paidOutReward` can be kept just as an accumulator. It will be enough to have only the `getRewardForStakingStore` to calculate the rewards as the staking start time is updated every time the rewards are paid out.

Client's Commentary:

The issue has been fixed in commit [d151d961](#)

Also, the re-working of the payment logic (Medium-5) has led to the problem becoming largely obsolete.

C-2	Inflation Attack in DIAExternalStaking Allows Stealing First Depositor's Stake		
Severity	Critical	Status	Fixed in dfd0bb6d

Description

This issue has been identified within the [DIAExternalStaking](#) contract.

The contract is vulnerable to a classic inflation attack due to the permissionless nature of the [addRewardToPool](#) function and the way pool shares are calculated. An attacker can exploit this as follows in newly deployed pool:

1. The attacker stakes in an empty pool the minimum allowed amount ([minimumStake](#)), receiving [minimumStake](#) share of the pool.
2. The attacker then unstakes almost all of their tokens, leaving only a single token (i.e., unstakes [minimumStake - 1](#)).
3. At this point, the total pool shares and the total supply are both equal to 1.
4. The attacker sees in the mempool a transaction of victim and frontruns it, calling [addRewardToPool\(\)](#) with `amount == victim_deposit`.
5. After this step there is 1 share in the pool and `victim_deposit + 1` stake. So victim receives:

$$\text{poolSharesGiven} = (\text{amount} * \text{totalShareAmount}) / \text{totalPoolSize} = \text{victim_deposit} * 1 / (\text{victim_deposit} + 1) = 0 \text{ shares}$$
6. The attacker can un stake funds and repeat the attack on an empty pool.

The issue is classified as **Critical** severity because it allows a malicious actor to steal user's funds.

Recommendation

We recommend adding a non-zero shares minted requirement to the [_stake](#) function or add a special variable which will be passed to the function by the user to control for the minimum shares minted. Also, we recommend restricting who can call [addRewardToPool](#) function.

Client's Commentary:

Client: The issue has been fixed in commit [95aca806](#)

C-3	Missing Reward Accumulator Initialization in Stake Creation		
Severity	Critical	Status	Fixed in a29131c4

Description

This issue has been identified within the `_internalStakeForAddress` function of the `DIAStakingCommons` contract.

The reward accumulator (`rewardAccumulator`) is not initialized when a new stake is created. This oversight allows stakers to claim all rewards that have ever been accumulated by the contract, regardless of when they started staking, because of the following line: `stakerDelta = rewardAccumulator - currentStore.rewardAccumulator`. As the `currentStore.rewardAccumulator` wasn't initialized, `stakerDelta` would be equal to the value of the global `rewardAccumulator`. A similar issue exists in the `DIAWhitelistedStaking._getTotalRewards()` function, where `rewards` variable is calculated using only the `rewardAccumulator` without subtracting the `currentStore.rewardAccumulator`.

The issue is classified as **Critical** severity because it could lead to incorrect reward distribution and potential loss of funds through unauthorized reward claims.

Recommendation

We recommend adding the initialization of the `rewardAccumulator` value during stake creation in the `_internalStakeForAddress` function to ensure proper reward tracking from the moment of staking. Also, `reward` calculation in the `_getTotalRewards` should be changed to properly account for the stored `currentStore.rewardAccumulator`.

Client's Commentary:

Both `stake()` and `stakeForAddress()` invoke `_updateRewardAccumulator()` and pass the most recent `rewardAccumulator` to the `_internalStakeForAddress()` function. This value is then used to initialize the `newStore.rewardAccumulator`, & `newStore.initialRewardAccumulator` variables. The former will continue to increment and track reward progression over time, while the latter remains constant to offset any accumulated rewards prior to the stake's `startTime`, ensuring accurate reward calculations from the moment staking begins.

C-4	<code>currentStore.principal</code> is not subtracted after it was withdrawn, allowing users to claim it again infinitely		
Severity	Critical	Status	Fixed in a29131c4

Description

In the `DIASWhitelistedStaking` contract, when a user withdraws their principal through the `unstake` function, the `currentStore.principal` value is not zeroed out or decremented after the withdrawal. This creates a critical vulnerability where users can repeatedly call `unstake` and claim the same principal amount multiple times, as the contract doesn't track that the principal has already been withdrawn.

This can lead to significant losses for the protocol, as an attacker may withdraw more tokens than were initially staked by them, potentially draining the contract's token balance.

Recommendation

We recommend resetting `currentStore.principal` to zero in the `unstake` function. We recommend adding the line `currentStore.principal = 0;` before the token transfer is performed.

Client's Commentary:

fix added as per recommendation

C-5	WDIA prohibits zero transfers, which will block rewards distribution to beneficiary address if 100% goes to principal unstaker		
Severity	Critical	Status	Fixed in a29131c4

Description

In the `DIAExternalStaking` contract, the `unstake` function distributes rewards between the principal wallet and beneficiary based on the `principalWalletShareBps` parameter. When `principalWalletShareBps` is set to 10000 (100%), all rewards go to the principal unstaker and `requestedUnstakeRewardAmount` (which goes to the beneficiary) becomes zero.

The issue arises because the `WDIA` token contract implements a check that reverts on zero-amount transfers (`if (wad == 0) revert ZeroTransferAmount();`). When `requestedUnstakeRewardAmount` is zero, the `safeTransferFrom` call to the beneficiary address will revert, causing the entire `claim` transaction to fail.

This prevents users from claiming any rewards when 100% of rewards are allocated to the principal unstaker, effectively locking the rewards and making the staking system unusable for stakes with maximum principal wallet share.

Recommendation

We recommend adding a check to skip the transfer to beneficiary when the `requestedUnstakeRewardAmount` is zero. The same check should be performed before transferring `requestedUnstakePrincipalAmount` to the `principalPayoutWallet` address to prevent the same issue in case if the transferred amount is zero. Also, consider allowing zero-amount transfers in the `WDIA` token contract by removing the `ZeroTransferAmount` restriction, though this would require careful consideration of the implications for other parts of the system.

Client's Commentary:

0 transfer is allowed in WDIA, added amount transfer check to skip transfer if amount is 0

2.2 High

H-1	Rewards are lost if user requests unstake in less than one day		
Severity	High	Status	Acknowledged

Description

In the `DIASWhitelistedStaking` contract's `getRewardForStakingStore` function, there is a potential loss in rewards calculation if a user requests unstake in less than one day. The function uses the formula:

```
passedDays = passedSeconds / (24 * 60 * 60);
```

If the user requests unstake in less than one day, then `passedDays` will be 0, and the reward calculation will be incorrect, leading to rewards for that period of time being lost.

Recommendation

We recommend adding a check for `passedDays` to be greater than 0 in the `getRewardForStakingStore` function or implementing a different logic for the reward calculation, which will account not only for the whole day, but also for the seconds passed since the start of the staking. Also, we recommend making it impossible to call `requestUnstake` in less than one day in case if the logic with the `passedDays` is left unchanged.

Client's Commentary:

Client: This is intentional, rewards are only paid out after full days

MixBytes: There is a related issue present in the `d4d0bb6dc60735ed6983ee2aa4b10c7fac875ff6` commit, allowing anyone to manipulate `rewardAccumulator` state variable updates using the described rounding issue. This issue is described in the High #6.

H-2	Rewards claiming may fail due to potentially decreasing parameters used in calculation		
Severity	High	Status	Fixed in dfd0bb6d

Description

In the `DIASWhitelistedStaking` contract, there is a potential issue in the reward calculation and claiming logic that could lead to reward claims being blocked. The issue arises from the following sequence:

1. In the `updateReward` function, there's an assertion:

```
assert(reward >= currentStore.reward);
```

2. This assertion can fail due to two factors:

- `rewardRatePerDay` can be decreased by the contract owner
- `currentStore.principal` can be decreased during partial principal unstaking

3. When either of these values decreases, the reward calculation in `getRewardForStakingStore`:

```
return (rewardRatePerDay * passedDays * currentStore.principal) / 10000;
```

may return a value lower than `currentStore.reward`, causing the assertion to fail.

4. This issue is particularly problematic because:

- After the first reward claim, `currentStore.reward` is set to 0
- This blocks both reward and principal claiming for the user

Recommendation

We recommend removing the internal accounting for the `currentStore.reward` and `currentStore.paidOutReward` variables and the mentioned assertion `assert(reward >= currentStore.reward);`. `rewardToSend` variable, which is used inside `unstake` and `unstakePrincipal` functions, should be calculated based on the value returned from the `getRewardForStakingStore` function. `currentStore.paidOutReward` can be kept just as an accumulator. It will be enough to have only the `getRewardForStakingStore` to calculate the rewards as the staking start time is updated every time the rewards are paid out.

Client's Commentary:

This has been fixed in conjunction with Critical-1

H-3	Pending principal wallet share BPS value is not saved in storage		
Severity	High	Status	Fixed in a29131c4

Description

In both `DIAExternalStaking` and `DIABWhitelistedStaking` contracts, the `_getCurrentPrincipalWalletShareBps` function uses the pending value directly from memory without storing it in storage after the timeout period has passed. It will lead to using an outdated `stakingStores[stakingStoreIndex].principalWalletShareBps` value in case of the new pending updates which hasn't passed through the timeout yet.

This issue is important because it prevents beneficiary from fully controlling the proportion of the rewards between the principal and beneficiary wallet. Imagine the case when the principal staker initially sets bps share value to 100%, which is then getting changed by the beneficiary to 50% and after some time they decide to change it again and set bps share value to 70%. It will lead to using the 100% bps value originally set by principal staker (during the `SHARE_UPDATE_GRACE_PERIOD` period), which wouldn't be expected by the beneficiary.

Recommendation

We recommend updating the `_getCurrentPrincipalWalletShareBps` function to store the pending value in storage after the timeout period has passed.

Client's Commentary:

Client: "Added latest bps update to storage from map if time required is elapsed, whenever update bps is called"

Fixed in [4ba9dd8c](#)

MixBytes: This issue is still present in the `DIAExternalStaking` contract

Client: Added fix in `DIAExternalsStaking`

H-4	Incorrect Reward Calculation When Reward Rate Changes		
Severity	High	Status	Fixed in a29131c4

Description

This issue has been identified within the `getRewardForStakingStore` function of the `DIAWhitelistedStaking` contract.

Currently, rewards are calculated using the current `rewardRatePerDay`, regardless of whether the rate has changed between `currentStore.stakingStartTime` and the present moment. As a result, two identical stakes may receive different rewards depending on when they call `unstake()`.

The issue is classified as **High** severity because it can lead to unfair or inconsistent reward distribution among stakers, potentially undermining trust in the staking mechanism.

Recommendation

We recommend implementing an accumulator that tracks the total rewards accrued per unit of stake. This accumulator should be updated during any relevant operation. As an example – global index implementation in `Comptroller.sol` by Compound Protocol.

There may be introduced the following logic:

```
unclaimed += currentStore.principal * (globalIndex - userIndex);
```

where `globalIndex` is updated before each change made to the rewards rate or other parameters affecting the rewards accrual by users. It may be done in such way:

```
timeElapsed = block.timestamp - lastUpdateTime;

// We don't take into account potential precision loss
// here due to the division by SECONDS_PER_DAY
rewardAccrued = rewardRatePerDay * timeElapsed / SECONDS_PER_DAY;

indexDelta = rewardAccrued * 1e18 / totalStaked;
globalIndex += indexDelta;
lastUpdateTime = block.timestamp;
```

And then user rewards can be calculated as follows:

```
userDelta = globalIndex - userIndex[user];
reward = currentStore.principal * userDelta / 1e18;
userUnclaimed[user] += reward;
userIndex[user] = globalIndex;
```

That approach will ensure that the correct rewards rate is applied during the appropriate staking period and won't lead to distributing more or less rewards than it was expected by the protocol users.

Client's Commentary:

Client: We've introduced a `rewardAccumulator` global variable that tracks the cumulative sum of daily reward rates over time for all stakes, reflecting the total rewards accrued per stake. `rewardLastUpdateTime` is another global variable that records the timestamp of the last update to the `rewardAccumulator` variable, ensuring that whenever it is updated, the calculation starts from the last update time.

`rewardAccumulator` is updated with every `unstake`, `claim`, and `updateRewardRatePerDay` function call within `DIASWhitelistedStaking.sol` & `DIARewardsDistribution.sol` contracts.

MixBytes: There is a new issue introduced. It is related to the `rewardAccumulator` variable logic. This issue is described in the Critical #3.

Client: We've introduced a `rewardAccumulator` global variable that tracks the cumulative sum of daily reward rates over time for all stakes, reflecting the total rewards accrued per stake. `rewardLastUpdateTime` is another global variable that records the timestamp of the last update to the `rewardAccumulator` variable, ensuring that whenever it is updated, the calculation starts from the last update time.

`rewardAccumulator` is updated with every `unstake`, `claim`, and `updateRewardRatePerDay` function call within `DIASWhitelistedStaking.sol` & `DIARewardsDistribution.sol`.

H-5	Incorrect Initialization of <code>lastWithdrawalResetDay</code> Causes Withdrawal Limits to Malfunction		
Severity	High	Status	Fixed in <code>dfd0bb6d</code>

Description

This issue has been identified within the constructor of the `DIAExternalStaking` contract. The `lastWithdrawalResetDay` variable is initialized with the current block timestamp, rather than the timestamp divided by `SECONDS_IN_A_DAY`. As a result, the daily withdrawal counters are never properly reset, causing the withdrawal limits to effectively become one-time limits rather than daily limits. This means the intended functionality of daily withdrawal restrictions does not work as designed, and the contract cannot be upgraded to fix this issue. The issue is classified as **High** severity because it breaks a core security mechanism of the contract.

Recommendation

We recommend initializing `lastWithdrawalResetDay` as `block.timestamp / SECONDS_IN_A_DAY` in the constructor to ensure the daily reset logic functions correctly.

Client's Commentary:

This has been fixed in commit `c33748d8`

H-6	Reward Accumulation Manipulation Through Frequent Claims		
Severity	High	Status	Fixed in 71d6ba36

Description

This issue has been identified within the `claim` function of the `DIASWhitelistedStaking` contract. Any user can deprive other stakers of rewards by calling the `claim` function multiple times per day. The reward accumulator is updated on each `claim()` call for any user. Consequently, a malicious user can stake the minimum amount and repeatedly call `claim()` throughout the day. Since `daysElapsed` will be rounded down to 0 each time, the `rewardAccumulator` will remain unchanged despite updating `rewardLastUpdateTime`. This results in no rewards being distributed to other stakers.

The issue is classified as **High** severity because it allows a malicious actor to effectively freeze reward distribution for all other stakers in the system, potentially causing significant financial impact.

Recommendation

We recommend adding a constant multiplier to increase precision when calculating the time interval since the last update (currently `daysElapsed`). This could be implemented as follows:

```
uint256 constant PRECISION = 1e18;

// ...

uint256 timeElapsed = (block.timestamp - rewardLastUpdateTime) * PRECISION / SECONDS_IN_A_DAY
uint256 rewardsAccrued = rewardRatePerDay * timeElapsed;
rewardAccumulator += rewardsAccrued;
rewardLastUpdateTime = block.timestamp;
```

When calculating rewards for a specific user, the same constant should be used as a divisor:

```
stakerDelta = rewardAccumulator - currentStore.rewardAccumulator;
currentStore.rewardAccumulator = rewardAccumulator;
stakerReward = (stakerDelta * currentStore.principal) / PRECISION / 10000;
```

This will solve all precision issues in reward calculations.

We recommend changing all the places where the `daysElapsed` variable is calculated to prevent cases when the division rounds down to zero.

Client's Commentary:

Client: Rewards will continue to be distributed in full-day increments, as previously outlined. To address the potential claim spam, the `rewardAccumulator` is now only updated when a full day has elapsed. This safeguard is implemented within the `_updateRewardAccumulator()` function.

MixBytes: There is still a possibility of griefing. A malicious user can call `claim()` every 1.999 days, resulting in up to half of the rewards being lost for all users. If it's important not to accrue rewards for partial days, there is another solution. The accrual of rewards for partial days can be postponed to a later time by updating `rewardLastUpdateTime` to the nearest past moment when

a full day has passed. For this, in `DIASWhitelistedStaking._updateRewardAccumulator()`, `DIASWhitelistedStaking.sol#L367` should be replaced with:

```
rewardLastUpdateTime += SECONDS_IN_A_DAY * daysElapsed;
```

The same should be done in `DIARewardsDistribution.updateRewardRatePerDay()`.

H-7	Incorrect Calculation of Accrued Rewards in <code>updateRewardRatePerDay</code>		
Severity	High	Status	Fixed in a29131c4

Description

This issue has been identified within the `updateRewardRatePerDay` function of the `DIARewardsDistribution` contract.

The calculation of the `rewardsAccrued` variable contains two problems that lead to incorrect reward distribution:

1. The function incorrectly uses `rewardRatePerDay` (which has already been updated to the new rate) instead of `oldRewardRate` when calculating rewards accrued between `rewardLastUpdateTime` and `block.timestamp`. This means rewards for the elapsed period are calculated using the new rate instead of the rate that was actually in effect during that period.
2. The function divides the value by 10000, resulting in lost rewards, because it is then added to the `rewardAccumulator` variable, which is then used in `_getRewardForStakingStore` function to calculate `stakerDelta`. This `stakerDelta` variable is multiplied by `currentStore.principal` and is divided by 10000, which is the second division by this value. This leads to the less rewards assigned to the user.

The issue is classified as **High** severity because it directly impacts the core functionality of reward distribution, potentially causing significant financial losses for users due to incorrect reward calculations.

Recommendation

We recommend implementing the following changes to the `rewardsAccrued` calculation:

1. Use `oldRewardRate` instead of `rewardRatePerDay` when calculating rewards accrued between the last update and the current timestamp.
2. Remove the division by 10000 from the accumulator update step.

Client's Commentary:

changes done as per recommendation

2.3 Medium

M-1	Daily withdrawal limit decreases with <code>totalPoolSize</code> reduction		
Severity	Medium	Status	Acknowledged

Description

In the `DIAExternalStaking` contract, the daily withdrawal limit calculation in the `checkDailyWithdrawalLimit` modifier is tied to the `totalPoolSize`. The issue arises because:

1. The daily withdrawal limit is calculated as:

```
availableDailyLimit = (totalPoolSize * withdrawalCapBps) / 10000;
```

2. When `totalPoolSize` decreases (due to unstaking), the `availableDailyLimit` also decreases proportionally, even if the remaining pool size is still significant.
3. This creates a situation where:
 - A user may have successfully withdrawn a certain amount earlier in the day
 - Later withdrawals may be blocked due to the reduced `availableDailyLimit`
 - Even if the total pool size is still substantial, the withdrawal limit may be too low to allow further withdrawals

Users may be unable to withdraw their funds even though the total pool size is still significant and it creates inconsistent withdrawal behavior throughout the day.

Recommendation

We recommend calculating the daily limit based on the initial pool size at the start of the day:

```
if (block.timestamp / SECONDS_IN_A_DAY > lastWithdrawalResetDay) {
    totalDailyWithdrawals = 0;
    lastWithdrawalResetDay = block.timestamp / SECONDS_IN_A_DAY;
    availableDailyLimit = (totalPoolSize * withdrawalCapBps) / 10000;
}
```

`availableDailyLimit` variable may be stored in the storage and updated every time the new day starts.

Client's Commentary:

This is by design and the limit decreases until it reaches the minimal cap, which is the not limited. As part of the review for this issue we noticed that in the the unstaking logic the amount selection should come at the request stage, not unstake, therefore it was moved to a step earlier.

M-2	Principal wallet pending share BPS usage is missed in <code>getRewardForStakingStore</code> function		
Severity	Medium	Status	Fixed in dfd0bb6d

Description

In the `DIAExternalStaking` contract, the `getRewardForStakingStore` function uses the static `store.principalWalletShareBps` value for calculating the principal wallet reward, instead of using the dynamic `_getCurrentPrincipalWalletShareBps` function. This means that pending updates to the principal wallet share percentage are not reflected in the value returned from the `getRewardForStakingStore` function, which may be misleading for users attempting to estimate the rewards to be claimed.

Recommendation

We recommend updating the `getRewardForStakingStore` function to use `_getCurrentPrincipalWalletShareBps` for calculating the principal wallet reward:

```
uint256 principalWalletReward =
    (fullReward * _getCurrentPrincipalWalletShareBps(stakingStoreIndex)) / 10000;
```

Client's Commentary:

replaced usage of storage `principalWalletShareBps` with
view function

M-3	stakingIndicesByPrincipalUnstaker mapping not updated in updatePrincipalUnstaker function		
Severity	Medium	Status	Fixed in dfd0bb6d

Description

In both `DIAExternalStaking` and `DIAWhitelistedStaking` contracts, the `stakingIndicesByPrincipalUnstaker` mapping is not updated when a new principal unstaker is set via the `updatePrincipalUnstaker` function. This means that the mapping will not correctly reflect the current stakes for the old and new principal unstaker.

Recommendation

We recommend modifying the `updatePrincipalUnstaker` function to update the `stakingIndicesByPrincipalUnstaker` mapping for both the old and new principal unstaker addresses. The staking store index for the old address should be removed from the corresponding array and added to the new one (identified by the new principal unstaker address).

Client's Commentary:

mapping update has been added

M-4	Whitelist can be bypassed in DIAWhitelistedStaking		
Severity	Medium	Status	Fixed in dfd0bb6d

Description

This issue has been identified within the [stakeForAddress](#) function of the [DIAWhitelistedStaking](#) contract.

It is possible to call [stakeForAddress\(\)](#) and specify any whitelisted beneficiary, assigning 100% of the rewards to the principal. Since the beneficiary can be a contract, the ability for the beneficiary to later change the reward share does not fully mitigate the problem.

The issue is classified as **Medium** severity because it allows any user to exploit the whitelisted staking mechanism.

Recommendation

We recommend implementing additional checks or restrictions to ensure that the reward share cannot be set to 100% for the principal, or to require explicit consent from the beneficiary, especially if the beneficiary is a contract.

Client's Commentary:

all callers for stake should be whitelisted

M-5	Principal withdrawal may be blocked by insufficient rewards wallet balance in <code>DIAWhitelistedStaking</code>		
Severity	Medium	Status	Fixed in <code>dfd0bb6d</code>

Description

The `unstake()` and `unstakePrincipal()` functions in `DIAWhitelistedStaking` require mandatory reward payments from the external `rewardsWallet` before allowing principal withdrawal. When `rewardsWallet` has insufficient token balance or revoked allowances, the `safeTransferFrom()` calls for reward distribution revert, causing the entire withdrawal transaction to fail. This prevents users from accessing their staked principal even though the principal tokens are held within the contract itself. Users can accumulate large rewards over time that exceed the external wallet's capacity, permanently locking access to their own principal funds with no alternative withdrawal path.

Recommendation

We recommend separating principal and reward withdrawal logic, allowing users to withdraw their principal independently of reward payment status, or implement a try-catch mechanism for reward transfers.

Client's Commentary:

Client: added separate function for `unstakeOnlyPrincipalAmount`

MixBytes: There are added `requestUnstake` and `requestUnstakeWithoutClaim` functions, which differ from each other only by the `if (currentStore.paidOutReward != totalRewards)` check, which can be omitted as it doesn't update any state variables or affects any logic. `requestUnstake` function can be removed.

2.4 Low

L-1	Incorrect event emission in <code>updatePrincipalPayoutWallet</code> in <code>DIAStakingCommons</code>		
Severity	Low	Status	Fixed in dfd0bb6d

Description

The `updatePrincipalPayoutWallet()` function in `DIAStakingCommons` incorrectly emits the `PrincipalPayoutWalletUpdated` event with the already-updated wallet address as the "old wallet" parameter. The function correctly stores the old wallet address in a local variable but then updates the storage before emitting the event, using `currentStore.principalPayoutWallet` (which now contains the new wallet) instead of the saved `oldWallet` variable. This makes event logs show wallet updates from the new address to the new address, making it impossible to track the actual change that occurred. In contrast, `DIAExternalStaking` implements this correctly by using the saved `oldWallet` variable in the event emission.

Recommendation

We recommend updating the event emission to use the saved `oldWallet` variable: `emit PrincipalPayoutWalletUpdated(oldWallet, newWallet, stakingStoreIndex);` to properly track wallet address changes.

Client's Commentary:

Emit values are changed

L-2	Lack of slippage protection in <code>unstake</code> function in <code>DIAExternalStaking</code>		
Severity	Low	Status	Fixed in 71d6ba36

Description

The `unstake()` function in `DIAExternalStaking` calculates withdrawal amounts based on the current pool state at execution time, without providing slippage protection for users. The calculation `currentAmountOfPool = (currentStore.poolShares * totalPoolSize) / totalShareAmount` uses real-time pool values that can change between when a user calls `requestUnstake()` and when they execute `unstake()` after the waiting period. Pool size changes through `addRewardToPool()` calls, other users staking/unstaking, or reward distributions can cause users to receive different amounts than expected.

Recommendation

We recommend adding a `minAmountOut` parameter to the `unstake()` function to allow users to specify the minimum acceptable withdrawal amount, reverting if the calculated amount falls below this threshold: `require(amount >= minAmountOut, "Slippage protection: amount too low")`.

Client's Commentary:

Client: Fixed in 33bab77602effccf0831af32b46606c1ab91a259 when we moved the `unstake` logic to `requestUnstake`

MixBytes: It is still possible to dilute how much shares are deducted from user's balance as there is an uncontrolled `poolSharesUnstakeAmount` variable, which depends on the `currentAmountOfPool` variable. This variable may get unexpectedly changed before user calls `unstake` function by someone else's stake/unstake, which affect `totalPoolSize` and `totalShareAmount` global variables. So the user may control the desired amount, but not the amount of the deducted shares. We recommend adding a `maxPoolSharesUnstakeAmount` parameter to the `unstake()` function to allow users to specify the maximum acceptable amount of shares they are willing to spend, reverting if the calculated amount is greater than this threshold: `require(poolSharesUnstakeAmount <= maxPoolSharesUnstakeAmount, "Slippage protection: shares spent amount too high")`.

Client: `maxPoolSharesUnstakeAmount` is added as per recommendation

MixBytes: There is an incorrect check in the commit a29131c47dbe17dfd3a04f1b9bc5b6ad8e65ed26: if `(poolSharesUnstakeAmount <= maxPoolSharesUnstakeAmount) { ... }`. Now execution reverts if the calculated `poolSharesUnstakeAmount` is lower than the specified by user `maxPoolSharesUnstakeAmount` amount, which should be the other way around: `maxPoolSharesUnstakeAmount` should be the upper limit for the shares being burnt.

Client: The check was fixed and a view function was added for the frontend to find out the expected value of `poolSharesUnstakeAmount` before unstaking.

L-3	Missing <code>tokensStaked</code> update in <code>unstakePrincipal</code> causing accounting drift in <code>DIAWhitelistedStaking</code>		
Severity	Low	Status	Fixed in dfd0bb6d

Description

The `unstakePrincipal()` function in `DIAWhitelistedStaking` correctly decreases the individual stake's principal amount but fails to update the global `tokensStaked` counter inherited from `DIAStakingCommons`. The function updates `currentStore.principal = currentStore.principal - amount` but omits the corresponding `tokensStaked -= amount` operation. This creates an accounting inconsistency where `tokensStaked` becomes progressively inflated compared to the actual sum of all individual principals. While `tokensStaked` is not currently used in `DIAWhitelistedStaking` logic (unlike `DIAExternalStaking` which uses it for staking limit checks), maintaining accurate accounting is important for external integrations. In contrast, `DIAExternalStaking.unstake()` correctly updates both values: `currentStore.principal -= principalUnstakeAmount` and `tokensStaked -= principalUnstakeAmount`.

Recommendation

We recommend adding the missing global counter update in `unstakePrincipal()`: `tokensStaked -= amount;` after the line `currentStore.principal = currentStore.principal - amount;` to maintain accounting consistency.

Client's Commentary:

tokensstaked are updated

L-4	Missing zero-address validation and boundaries check in <code>DIARewardsDistribution</code> constructor		
Severity	Low	Status	Fixed in dfd0bb6d

Description

The constructor in `DIARewardsDistribution` does not validate that `rewardsTokenAddress` or `newRewardsWallet` parameters are non-zero addresses. Setting either parameter to `address(0)` during deployment would create a non-functional contract: a zero token address would cause all reward distribution calls to fail, while a zero rewards wallet would prevent any reward transfers. This creates an inconsistency with the `updateRewardsWallet()` function which properly validates against zero addresses using `require(newWalletAddress != address(0))`. While the contract would naturally revert when attempting to interact with zero addresses, explicit validation provides clearer error messages for debugging deployment issues and prevents deploying immediately non-functional contracts.

There is also a `rewardRatePerDay` parameter being set, which should be checked not to exceed some organic values. If this value is mistakenly set too high, it is possible for any staker to claim unproportional amount of rewards before this rate get reset back to normal value by the contract owner.

Recommendation

We recommend adding zero-address validation in the constructor: `require(rewardsTokenAddress != address(0), "Invalid token address")` and `require(newRewardsWallet != address(0), "Invalid wallet address")` to maintain consistency with other validation patterns in the contract.

Also, it is important to check that `rewardRatePerDay` parameter fits into some pre-defined range aligned with the staking logic.

Client's Commentary:

checks are added

L-5	Redundant zero check for unsigned integer in <code>setDailyWithdrawalThreshold</code>		
Severity	Low	Status	Fixed in a29131c4

Description

The `setDailyWithdrawalThreshold()` functions in both `DIAExternalStaking` and `DIAStakingCommons` contain a redundant check `if (newThreshold <= 0)` for a `uint256` parameter. Since `uint256` is an unsigned integer type, it cannot be negative and the minimum possible value is 0. The condition `<= 0` is equivalent to `== 0`, making the check unnecessarily confusing. This pattern suggests either a misunderstanding of unsigned integer behavior or copy-paste from signed integer validation. While functionally correct (preventing zero thresholds), the misleading condition reduces code clarity and may confuse developers about the actual validation being performed.

Recommendation

We recommend replacing the misleading condition with explicit zero check: `if (newThreshold == 0)` to clearly indicate that only zero values are being rejected, improving code readability and developer understanding.

Client's Commentary:

Client: replaced misleading check of 0

MixBytes: The mentioned check was removed from the `DIAWhitelistedStaking` contract, but is still present in the `DIAExternalStaking`. This check should be changed to the strict `if (newThreshold == 0)`.

Client: changes are done as per recommendation

L-6	Unused logic related to the daily withdrawal threshold in <code>DIAStakingCommons</code>		
Severity	Low	Status	Fixed in <code>dfd0bb6d</code>

Description

The `setDailyWithdrawalThreshold()` function allows the owner to modify the daily withdrawal threshold but appears to serve no purpose in the current implementation. Dead code increases contract size, deployment costs, and potential attack surface while providing no functional benefit. Additionally, the function contains the same redundant zero check issue (`if (newThreshold <= 0)`) as other threshold functions, suggesting it was copied without proper review. Maintaining unused functions can lead to confusion during future development and may introduce unexpected behaviors if accidentally called.

There are also unused storage variables like `withdrawalCap`, `totalDailyWithdrawals`, `lastWithdrawalResetDay`, `dailyWithdrawalThreshold`.

And `withdrawalCapBps` variable with the corresponding `setWithdrawalCapBps` function is also never used.

Recommendation

We recommend removing the unused functions and storage variables from `DIAStakingCommons` to reduce contract size, eliminate dead code, and improve code maintainability.

Client's Commentary:

dead code is removed

L-7	Missing zero amount validation in token transfers in <code>DIAWhitelistedStaking</code>		
Severity	Low	Status	Fixed in a29131c4

Description

The `unstakePrincipal()` function in `DIAWhitelistedStaking` performs token transfers without validating that the transfer amounts are non-zero. Specifically, the transfer to the beneficiary `STAKING_TOKEN.safeTransferFrom(rewardsWallet, currentStore.beneficiary, beneficiaryReward)` on line 242 can execute with `beneficiaryReward = 0` when all rewards go to the principal wallet (`principalWalletShareBps = 10000`). While `safeTransferFrom()` with zero amounts typically succeeds without reverting, it represents unnecessary gas consumption and generates misleading transfer events. The same pattern exists in other transfer operations throughout the contract where amounts could be zero in edge cases.

Recommendation

We recommend adding zero amount validation before token transfers: `if (beneficiaryReward > 0) { STAKING_TOKEN.safeTransferFrom(rewardsWallet, currentStore.beneficiary, beneficiaryReward); }` to avoid unnecessary operations.

Client's Commentary:

Client: zero amount checks are added

MixBytes: This issue is still present in the `unstake` function of the `DIAExternalStaking` contract. Both `requestedUnstakePrincipalAmount` and `requestedUnstakeRewardAmount` are not checked for zero values. This also lead to the new finding related to the WDIA implementation. This issue is described in the Critical #5.

Client: checks are added

L-8	Multiple legacy and safety issues in WDIA inherited from WETH9		
Severity	Low	Status	Fixed in dfd0bb6d

Description

The [WDIA](#) contract inherits several known issues from the WETH9 implementation that collectively reduce its safety, clarity, and compatibility with modern standards:

- **Silent fallback behavior:** The fallback function accepts all calls, regardless of calldata. This causes unexpected ETH deposits when non-existent functions are called, a vulnerability observed in prior hacks such as the Multicoins bridge incident.
- **No overflow/underflow protection:** Arithmetic operations in key functions like [deposit\(\)](#), [withdraw\(\)](#), and [transferFrom\(\)](#) are vulnerable due to reliance on Solidity versions prior to 0.8.0, which lack built-in overflow checks.
- **Unrestricted compiler version:** The contract uses `pragma solidity >=0.4.23`, allowing compilation with a wide range of versions. This introduces inconsistencies in behavior, optimization, and gas usage.
- **Missing zero address validation:** Transfers using [transfer\(\)](#) and [transferFrom\(\)](#) lack checks for zero addresses, risking accidental token burns without warning.
- **Front-running risk in [approve\(\)](#):** The [approve\(\)](#) function directly updates allowances without requiring them to be reset to zero, creating a vulnerability where attackers can drain both old and new allowances during front-running scenarios.

Recommendation

We recommend modernizing and securing the [WDIA](#) contract:

1. Adding `require(msg.data.length == 0)` to the fallback function to block unintended calldata-based ETH deposits.
2. Upgrading to Solidity ^0.8.0 or integrate SafeMath to ensure arithmetic safety.
3. Fixing the compiler version (e.g., `pragma solidity 0.4.26`) to maintain predictable compilation behavior.
4. Adding `require(dst != address(0), "Transfer to zero address")` checks in [transfer\(\)](#) and [transferFrom\(\)](#).
5. Modifying [approve\(\)](#) to follow a two-step pattern or implement [increaseAllowance\(\)](#) and [decreaseAllowance\(\)](#) for safer token approvals.

Client's Commentary:

recommended changes are done

L-9	Staking Limit Cannot Be Updated in Case of Incorrect Value		
Severity	Low	Status	Fixed in dfd0bb6d

Description

This issue has been identified within the [DIAExternalStaking](#) contract.

Currently, the [stakingLimit](#) parameter is set only during contract deployment and cannot be changed afterward. If an incorrect value is set, it cannot be updated, and redeploying the contract would be required.

The issue is classified as **Low** severity because it does not directly impact security, but it reduces the contract's flexibility and upgradability.

Recommendation

We recommend allowing the [stakingLimit](#) to be updated after deployment and adding a check in the constructor to ensure that [stakingLimit](#) is not zero.

Client's Commentary:

add setter for staking limit

L-10	Lack of checks in the <code>DIAExternalStaking</code> and <code>DIAWhitelistedStaking</code> constructors		
Severity	Low	Status	Fixed in a29131c4

Description

This issue has been identified within the constructors of the `DIAExternalStaking` and `DIAWhitelistedStaking` contracts.

The `unstakingDuration` parameter is not restricted in the constructor, whereas it is properly validated in the setter function.

The constructors don't include a check to ensure that the `stakingTokenAddress` parameter is not the zero address.

The issue is classified as **Low** severity because it does not directly compromise security, but it may result in undesirable contract configurations that could affect usability.

Recommendation

We recommend adding appropriate validation checks.

Client's Commentary:

Client: checks are added

MixBytes: `_unstakingDuration` parameter is not checked in the `DIAExternalStaking` constructor.

Client: check is added

L-11	Unused <code>REWARDS_TOKEN</code> Variable		
Severity	Low	Status	Fixed in dfd0bb6d

Description

This issue has been identified within the `DIARewardsDistribution` contract.

The `REWARDS_TOKEN` variable is declared as an immutable public variable but is not used anywhere in the contract.

The issue is classified as **Low** severity because unused variables can lead to confusion for future maintainers or auditors.

Recommendation

We recommend removing the `REWARDS_TOKEN` variable from the contract if it is not required.

Client's Commentary:

unused REWARDS_TOKEN is removed

L-12	Unlimited reward rate enables overflow DoS attack in <code>DIARewardsDistribution</code>		
Severity	Low	Status	Fixed in dfd0bb6d

Description

The `updateRewardRatePerDay()` function in `DIARewardsDistribution` lacks upper bounds validation for the `newRewardRate` parameter, allowing the owner to set arbitrarily large values that cause arithmetic overflow in reward calculations. The reward formula $(\text{rewardRatePerDay} * \text{passedDays} * \text{currentStore.principal}) / 10000$ in `getRewardForStakingStore()` can overflow when `rewardRatePerDay` is set to extremely large values. For example, setting `rewardRatePerDay = type(uint256).max` and calculating rewards for any meaningful stake duration and principal amount will cause overflow, resulting in transaction reverts. This creates a denial-of-service condition where all functions depending on reward calculations (`unstake()`, `unstakePrincipal()`, `updateReward()`, and `getRewardForStakingStore()`) become unusable until the owner reduces the reward rate to a reasonable value. While the attack is reversible by lowering the rate, it represents a significant operational risk where the protocol can be temporarily disabled at any time.

Recommendation

We recommend adding maximum bounds validation in `updateRewardRatePerDay()`: `require(newRewardRate <= MAX_REWARD_RATE, "Reward rate exceeds maximum")` where `MAX_REWARD_RATE` is set to a reasonable upper limit that prevents overflow in reward calculations, such as `MAX_REWARD_RATE = 100000` (1000% per day).

Client's Commentary:

added max limit in updateRewardRatePerDay

L-13	Wrong role checked in <code>onlyBeneficiaryOrPayoutWallet</code> modifier in <code>DIAExternalStaking</code>		
Severity	Low	Status	Fixed in <code>dfd0bb6d</code>

Description

The `onlyBeneficiaryOrPayoutWallet` modifier in `DIAExternalStaking` is intended to allow either the stake beneficiary or the principal payout wallet to call certain functions, but it mistakenly checks `principalUnstaker` instead of `principalPayoutWallet`. This prevents the designated payout wallet from requesting or completing unstake operations, while only the `principalUnstaker` address has permission. This creates a denial-of-service for the rightful payout recipient when roles are separated, contradicting the modifier's documented purpose and name.

Recommendation

We recommend updating the modifier to check `principalPayoutWallet` instead of `principalUnstaker` to match the intended functionality and naming convention.

Client's Commentary:

expected behaviour, incorrect modifier name is changed

Fixed in `89d8bebc`

L-14	Principal payout wallet can be set to zero address causing permanent fund loss in DIAStakingCommons		
Severity	Low	Status	Fixed in dfd0bb6d

Description

The `updatePrincipalPayoutWallet()` function in [DIAStakingCommons](#) lacks validation to prevent setting the principal payout wallet to the zero address (0x0). When a [principalUnstaker](#) sets the payout wallet to `address(0)`, all subsequent withdrawals through `unstake()` and `unstakePrincipal()` functions will permanently burn both principal amounts and reward tokens by sending them to the zero address. The `safeTransfer()` calls to `currentStore.principalPayoutWallet` will succeed but result in irreversible token loss. This affects both the principal amount held in the contract and reward distributions, with no recovery mechanism available once tokens are sent to the zero address.

Recommendation

We recommend adding zero address validation in `updatePrincipalPayoutWallet()`: `require(newWallet != address(0), "Cannot set zero address as payout wallet")` to prevent accidental or malicious fund burning.

Client's Commentary:

added Zero address check in updatePrincipalPayoutWallet

L-15	Duplicate IERC20 import in DIAExternalStaking		
Severity	Low	Status	Fixed in dfd0bb6d

Description

The [DIAExternalStaking](#) contract contains a redundant import statement: `import {IERC20} from "@openzeppelin/contracts/interfaces/IERC20.sol";`. This import is unnecessary if [IERC20](#) is already imported from another OpenZeppelin location such as [@openzeppelin/contracts/token/ERC20/IERC20.sol](#). Including duplicate imports can cause confusion, increase compilation time, and lead to inconsistencies if different versions of the same interface are referenced.

Recommendation

We recommend removing the duplicate [IERC20](#) import from [@openzeppelin/contracts/interfaces/IERC20.sol](#) if it is not specifically required. Ensure that only a single, consistent source of the [IERC20](#) interface is used throughout the contract to maintain clarity and avoid potential mismatches.

Client's Commentary:

duplicate import is removed

L-16	Replace hardcoded <code>24 * 60 * 60</code> with <code>SECONDS_PER_DAY</code> in <code>DIAWhitelistedStaking</code>		
Severity	Low	Status	Fixed in dfd0bb6d

Description

The `DIAWhitelistedStaking` contract uses the hardcoded value `24 * 60 * 60` to calculate `passedDays` in its reward logic. However, the constant `SECONDS_PER_DAY` is already defined in the `StakingErrorsAndEvents` file for this exact purpose. Using the hardcoded expression instead of the named constant introduces inconsistency in the codebase and reduces readability.

Recommendation

We recommend replacing the hardcoded `24 * 60 * 60` expression in the `passedDays` calculation with the `SECONDS_PER_DAY` constant defined in `StakingErrorsAndEvents`. This promotes code consistency and improves maintainability across the staking contracts.

Client's Commentary:

hardcoded values are moved to use constant

L-17	Potential out-of-gas risk in <code>_removeStakingIndexFromAddressMapping</code> with large stake counts		
Severity	Low	Status	Fixed in dfd0bb6d

Description

The `_removeStakingIndexFromAddressMapping` functions iterates through the entire list of staking indices associated with a given address in order to remove a specific index. If a wallet has accumulated a large number of stakes, this linear search and removal operation could result in a transaction that exceeds the block gas limit, leading to out-of-gas errors and making it impossible to complete staking-related operations.

Recommendation

We recommend introducing a reasonable upper limit on the number of concurrent stakes a single user can have. This will prevent excessive gas consumption and ensure consistent performance and reliability of staking functionality across all users.

Client's Commentary:

limit is added

L-18	Shadowed variables in <code>DIAMWhitelistedStaking</code> constructor		
Severity	Low	Status	Fixed in dfd0bb6d

Description

In the `DIAMWhitelistedStaking` contract, the constructor parameters `rewardsWallet` and `rewardRatePerDay` shadow the state variables of the same name. This shadowing can lead to confusion and reduce code readability. While the current implementation correctly assigns these parameters to the corresponding state variables, this naming overlap is considered a poor practice and can make future maintenance error-prone.

Recommendation

We recommend renaming the constructor parameters to clearly differentiate them from the state variables, for example by using a prefixed underscore (`_rewardsWallet`, `_rewardRatePerDay`). This will eliminate ambiguity and improve code clarity.

Client's Commentary:

Variables are renamed

L-19	Unnecessary update of <code>stakingStartTime</code> after full principal withdrawal in <code>unstakePrincipal()</code> and <code>unstake()</code>		
Severity	Low	Status	Fixed in a29131c4

Description

Both the `unstakePrincipal()` function in `DIAWhitelistedStaking` and the `unstake()` function in `DIAExternalStaking` unconditionally update the `stakingStartTime` to `uint64(block.timestamp)` at the end of their execution. However, when the user's principal is fully withdrawn (i.e., `currentStore.principal == 0`), resetting the `stakingStartTime` serves no meaningful purpose. This results in an unnecessary state change that wastes gas and may cause confusion in interpreting stake history or when debugging contract behavior.

Recommendation

We recommend adding a conditional check to ensure `stakingStartTime` is only updated if `currentStore.principal > 0`. This will eliminate redundant state writes, reduce gas costs, and maintain cleaner state logic across both functions.

Client's Commentary:

Client: stakingStartTime is used in calculating rewards,

when partial unstake i.e unstakePrincipal

this as to be reset so that rewards

are accumulated based on pending principal amount

MixBytes: Rewards calculation process in the DIAExternalStaking contract doesn't depend on the stakingStartTime now, so it can be removed at all (or it may be set to the block.timestamp as it is done now, but only in case if the remaining principal is not zero).

Client: stakingStartTime is updated only when principal is not 0

L-20	Missing event emission in <code>updatePrincipalUnstaker()</code> in <code>DIAExternalStaking</code> and <code>DIAStakingCommons</code>		
Severity	Low	Status	Fixed in dfd0bb6d

Description

The `updatePrincipalUnstaker()` function in both `DIAExternalStaking` and `DIAStakingCommons` updates the `principalUnstaker` address for a given staking store but does not emit any event to signal this change. The lack of event emission makes it difficult for off-chain systems, indexers, and users to track critical role changes, especially those related to authorization and access control. Events are essential for transparency and auditability in smart contract systems, particularly when updating roles that control fund withdrawals.

Recommendation

We recommend emitting a `PrincipalUnstakerUpdated` event from `updatePrincipalUnstaker()` whenever the `principalUnstaker` is changed. The event should include the old and new addresses along with the associated `stakingStoreIndex` to allow external tracking and verification of updates.

Client's Commentary:

Events are added

L-21	Incorrect Parameters in Claimed Event		
Severity	Low	Status	Fixed in a29131c4

Description

This issue has been identified within the `unstake` function of the `DIAExternalStaking` contract. The function emits a `Claimed` event with zeroed values for `requestedUnstakePrincipalRewardAmount`, `requestedUnstakePrincipalAmount` and `requestedUnstakeRewardAmount`. This occurs because the event is emitted after the state variables have been reset to zero, resulting in incorrect event data being recorded on the blockchain.

The issue is classified as **Low** severity because while it doesn't affect the functionality of the contract, it impacts the accuracy of off-chain tracking and historical data analysis.

Recommendation

We recommend moving the `Claimed` event emission before the state variables are reset to zero, ensuring that the event contains the correct withdrawal amounts. This will maintain accurate historical records of unstaking operations.

Client's Commentary:

moved claim event before updating state

L-22	Redundant Return Value from <code>_updateRewardAccumulator</code>		
Severity	Low	Status	Fixed in a29131c4

Description

The internal function `_updateRewardAccumulator()` always returns `true`, making its return value effectively meaningless. This design choice leads to unnecessary conditional checks in multiple places throughout the contract, such as:

```
bool success = _updateRewardAccumulator();
...
if (success) {
    // Logic that always executes
}
```

Since success is always true, the conditionals serve no functional purpose and can be safely removed.

Recommendation

We recommend removing the `return true;` statement from the `_updateRewardAccumulator()` function and changing its signature to not return any value. And we recommend removing all variable declarations and conditional checks that depend on the return value of `_updateRewardAccumulator()`.

Client's Commentary:

This issue has been fixed.

3. About MixBytes

MixBytes is a leading provider of smart contract audit and research services, helping blockchain projects enhance security and reliability. Since its inception, MixBytes has been committed to safeguarding the Web3 ecosystem by delivering rigorous security assessments and cutting-edge research tailored to DeFi projects.

Our team comprises highly skilled engineers, security experts, and blockchain researchers with deep expertise in formal verification, smart contract auditing, and protocol research. With proven experience in Web3, MixBytes combines in-depth technical knowledge with a proactive security-first approach.

Why MixBytes


- **Proven Track Record:** Trusted by top-tier blockchain projects like Lido, Aave, Curve, and others, MixBytes has successfully audited and secured billions in digital assets.
- **Technical Expertise:** Our auditors and researchers hold advanced degrees in cryptography, cybersecurity, and distributed systems.
- **Innovative Research:** Our team actively contributes to blockchain security research, sharing knowledge with the community.

Our Services

- **Smart Contract Audits:** A meticulous security assessment of DeFi protocols to prevent vulnerabilities before deployment.
- **Blockchain Research:** In-depth technical research and security modeling for Web3 projects.
- **Custom Security Solutions:** Tailored security frameworks for complex decentralized applications and blockchain ecosystems.


MixBytes is dedicated to securing the future of blockchain technology by delivering unparalleled security expertise and research-driven solutions. Whether you are launching a DeFi protocol or developing an innovative dApp, we are your trusted security partner.

Contact Information

 <https://mixbytes.io/>

 https://github.com/mixbytes/audits_public

 hello@mixbytes.io

 <https://x.com/mixbytes>