# Security Review For

# DI Protocol

## (Decentralized Interoperability)


AUDITAGENT

September, 2025

Powered by NETHERMIND SECURITY

# Introduction

MetaTx-Contracts presents a production-ready infrastructure for gasless blockchain interactions through meta-transactions with native token support and multi-token gas credit management. This system eliminates the primary barrier to blockchain adoption - the requirement for users to hold native tokens for gas fees - while maintaining security, decentralization, and economic sustainability.

## Scope

| | |
|---|---|
| Repository: | DINetworks/metatx-contracts |
| Branch: | main |
| Audited commit: | 3bbfc3ab70bc5fac2956871a4518e3d6c6b95ef6 |
| Final commit: | e16493662876a49209fd469ce47fa1608d77d2b6 |

## Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities taht need to be fixed

## Code Info

Advanced Scan

| | |
|---|---|
| Scan ID<br>1 | Date<br>September 01, 2025 |
| Organization<br>DINetworks | Repository<br>MetaTx-Contracts |
| Branch<br>main | Commit Hash<br>3bbfc3ab...c6b95ef 6 |

## Contracts in scope

contracts/GasCreditVault.sol
contracts/MetaTxGateway.sol
contracts/TokenPresale.sol
contracts/TokenStaking.sol
contracts/TokenAirdrop.sol
contracts/DI.sol
contracts/DIVote.sol
contracts/KOLAllocation.sol
contracts/TeamAllocation.sol

## Code Statistics

| | | |
|---|---|---|
| Findings<br>32 | Contracts Scanned<br>9 | Lines of Code<br>1879 |

## Findings Summary


Total Findings

- High Risk (2)
- Medium Risk (7)
- Low Risk (13)
- Info (4)
- Best Practices (6)

## Code Summary

The protocol establishes a comprehensive ecosystem centered around the `DI` token, which integrates functionalities for token distribution, decentralized finance (DeFi), governance, and gas abstraction.

The core of the ecosystem is the `DI` token, an ERC20 token that incorporates `ERC20Votes` and `ERC20Permit` functionalities, enabling both on-chain governance and gas-efficient approvals. The protocol manages the token's lifecycle through several distinct mechanisms:

- **Token Distribution**: The initial supply of `DI` tokens is distributed through multiple channels. A `TokenPresale` contract allows users to purchase tokens using various cryptocurrencies, with a dynamic pricing model based on sales progress. A `TokenAirdrop` contract facilitates a Merkle-proof-based distribution to eligible users. Additionally, `TeamAllocation` and `KOLAllocation` contracts manage vesting schedules for team members and key opinion leaders, ensuring a controlled release of tokens over time.
- **Staking**: The `TokenStaking` contract allows `DI` token holders to stake their tokens and earn rewards. The system incentivizes long-term holding by offering higher reward shares for larger stake amounts and longer lock-up durations. This mechanism is designed to reward community members and secure the network.
- **Governance**: A decentralized autonomous organization (DAO) structure is implemented through the `DIVote` contract. `DI` token holders can delegate their voting power and participate in governance by voting on proposals concerning the protocol's development and treasury management. This empowers the community to collectively guide the future of the ecosystem.
- **Gas Abstraction**: A key utility feature is the gas abstraction layer, composed of the `GasCreditVault` and `MetaTxGateway` contracts. Users can deposit whitelisted tokens into the `GasCreditVault` to receive gas credits equivalent to the deposit's USD value. Authorized relayers can then use the `MetaTxGateway` to execute transactions on behalf of users, who sign EIP-712 messages off-chain. The gas fees for these meta-transactions are paid by consuming the user's credits from the vault, providing a seamless, "gasless" user experience.

### Entry Points and Actors

The primary actors interacting with the protocol are Users (Token Holders), Relayers, and Members (Team/KOLs).

- `GasCreditVault`
- `deposit(address token, uint256 amount)`: Executed by **Users** to deposit whitelisted tokens and receive gas credits.
- `withdraw(address token, uint256 creditAmount)`: Executed by **Users** to withdraw their deposited stablecoins by converting credits back to tokens.
- `consumeCredit(address user, uint256 usdValue)`: Executed by **Relayers** to deduct gas credits from a user's account to pay for a transaction.
- `transferCredit(address receiver, uint256 credit)`: Executed by **Users** to transfer their gas credits to another address.
- `MetaTxGateway`

- `executeMetaTransactions(...)`: Executed by **Relayers** to submit and execute a batch of transactions on behalf of a user, using the user's signature.

- **`TokenPresale`**

- `buyTokens(address payToken, uint256 amountIn)`: Executed by **Users** to purchase `DI` tokens using a whitelisted ERC20 token.

- `buyTokens()`: Executed by **Users** to purchase `DI` tokens using the native chain currency (e.g., ETH).

- **`TokenStaking`**

- `stake(uint256 amount, LockDuration duration)`: Executed by **Users** to stake their `DI` tokens for a specific duration.
- `withdraw(uint256 stakeIndex)`: Executed by **Users** to withdraw their staked tokens after the lock period has expired.
- `claimReward(uint256 stakeIndex)`: Executed by **Users** to claim their accumulated staking rewards.

- `updateReward()`: Executed by any actor, typically **Users**, to trigger the reward calculation and distribution update.

- **`TokenAirdrop`**

- `claim(uint256 amount, bytes32[] calldata proof)`: Executed by **Users** to claim their airdropped `DI` tokens using a Merkle proof.

- **`DIVote`**

- `vote(uint256 _proposalId, uint256 _choice)`: Executed by **Token Holders** to cast their vote on an active proposal.

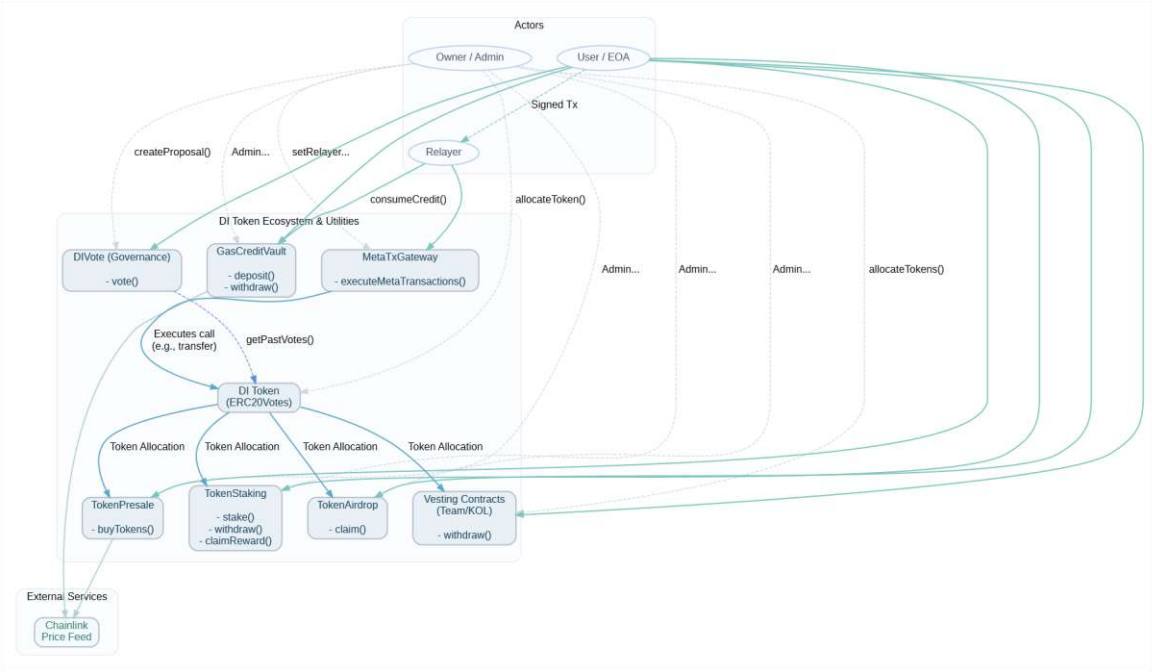
- **`TeamAllocation`** & **`KOLAllocation`**

- `withdraw()`: Executed by **Members** (Team/KOLs) to withdraw their vested tokens according to the predefined schedule.

- **`DI`** **(ERC20 Token)**

- `delegate(address delegatee)`: Executed by **Token Holders** to delegate their voting power to themselves or another address for governance participation.
- `permit(address owner, address spender, uint256 value, uint256 deadline, uint8 v, bytes32 r, bytes32 s)`: Executed by **Users** to approve token spending via an off-chain signature, enabling gasless approvals.

# Code Diagram



1 of 32 Findings

contracts/DI.sol

## `allocateToken` can be called multiple times and mints more than the declared `TOTAL_SUPPLY` while `totalSupply()` always returns a fixed constant

High Risk

The `allocateToken` function is intended to perform the one-time mint that distributes the entire token supply to the various ecosystem actors.

1. The function contains no guard that prevents it from being executed more than once. An attacker that gains the `owner` role (or the current owner by mistake) can call it repeatedly and mint the same allocation over and over again.
2. Even a single call already mints **1 100 000 000 DI** (see the constants inside the function) while `TOTAL_SUPPLY` is declared as 1 000 000 000 DI.
3. The contract overrides `totalSupply()` and returns the constant `TOTAL_SUPPLY`, **ignoring the real value stored in the inherited ERC20's `_totalSupply` variable**.

```
function allocateToken(AllocationAddresses memory addrs) external onlyOwner {
    ...
    uint256 presaleAllocation      = 150_000_000 ether;
    uint256 marketingAllocation    = 100_000_000 ether;
    uint256 kolAllocation          = 50_000_000 ether;
    uint256 teamAllocation         = 50_000_000 ether;
    uint256 treasuryAllocation     = 150_000_000 ether;
    uint256 ecosystemAllocation    = 300_000_000 ether;
    uint256 stakingAllocation      = 100_000_000 ether;
    uint256 liquidityAllocation    = 150_000_000 ether;
    uint256 airdropAllocation      = 50_000_000 ether;

    // sum = 1 100 000 000 > TOTAL_SUPPLY (1 000 000 000)
    _mint(...);
    ...
}

function totalSupply() public pure override returns (uint256) {
    return TOTAL_SUPPLY;   // always 1 000 000 000, even after over-minting
}
```

Consequences:
• The real token supply can silently exceed the advertised cap, breaking all economic assumptions.
• Because `totalSupply()` is hard-coded, any protocol that relies on that function (including `ERC20Votes`-based governance power calculations) will work with wrong numbers, leading to voting-power inflation and accounting inconsistencies.

2 of 32 Findings

contracts/GasCreditVault.sol

### withdrawConsumedCredits withdraws "deltaCredits" worth of EACH token, leading to N-times over-withdrawal

High Risk

`GasCreditVault.withdrawConsumedCredits()` calculates `deltaCredits` only once, but uses the full value independently for every whitelisted token. As a result, the owner receives `deltaCredits` credit-worth of **every** token instead of distributing the amount across tokens, allowing the owner (or a compromised owner key) to drain far more assets than were actually consumed.

```
uint256 deltaCredits = totalConsumedCredits - totalConsumedCreditsWithdrawn;
...
for (uint256 i = 0; i < tokens.length; ++i) {
    uint256 tokenValueInCredits = calculateCreditValue(token, contractBalance);
    uint256 tokenAmount = (deltaCredits * contractBalance) / tokenValueInCredits; // <- deltaCredits reused
    ...
}
```

Because `deltaCredits` is **not** decremented inside the loop, the following holds for every token `t` with balance `B_t`:
`tokenAmount_t = deltaCredits * B_t / tokenValueInCredits_t ≈ deltaCredits` (for stable-coins, exactly `deltaCredits`).
If there are *N* whitelisted tokens the vault sends ≈ `N × deltaCredits` credits worth of assets, while `totalConsumedCreditsWithdrawn` is still increased by only `deltaCredits`, silently violating the economic model and allowing systematic value extraction.

3 of 32 Findings

contracts/GasCreditVault.sol

### calculateTokenValue returns amounts with 18 extra decimals for non-stable tokens

Medium Risk

For non-stable-coins, `calculateTokenValue()` multiplies the incoming credit value by `10^(tokenDecimals+priceFeedDecimals)` and only divides by the price, forgetting to scale the result back down by `10^creditDecimals`. The function therefore returns numbers that are **$10^{18}$ times larger** than the real token amount.

```
uint256 numerator = convertDecimals(creditAmount, creditDecimals, creditDecimals + tokenDecimals + priceFeedDecimals);
return numerator / uint256(price); // missing 10^creditDecimals down-scaling
```

Although the current `withdraw()` path is limited to stable-coins, the faulty routine is publicly exposed (`getTokenValue`) and is used inside `withdrawConsumedCredits`, meaning wrong arithmetic can propagate to owner withdrawals or any future feature that relies on this helper.

4 of 32 Findings

contracts/TokenPresale.sol

### Native-token purchase path in `TokenPresale` is unusable without manual, undocumented configuration

Medium Risk

The payable version of `buyTokens()` forwards the zero-address as payment token to `calculateTokenAmount()`:

```
function buyTokens() public payable ... {
    uint256 tokenAmount = calculateTokenAmount(address(0), msg.value);
    ...
}
```

`calculateTokenAmount()` starts with

```
require(paymentTokens[payToken].isAllowed, "Token not allowed");
```

Unless the owner has previously whitelisted the *zero address* with `addPaymentToken()`, the call will always revert, making the advertised native-coin purchase option unusable. This is easy to overlook and would stall the sale for users intending to pay with the chain's native asset.

5 of 32 Findings

contracts/TokenPresale.sol

## On-Chain Price Calculation Vulnerable to Manipulation in TokenPresale

Medium Risk

The TokenPresale contract calculates token prices on-chain at transaction execution time, making it vulnerable to front-running attacks and price manipulation. The dynamic pricing model, which increases the token price as more tokens are sold, exacerbates this vulnerability.

```
function calculateTokenAmount(address payToken, uint256 amountIn) public view returns
(uint256 tokenAmount) {
    require(paymentTokens[payToken].isAllowed, "Token not allowed");
    PaymentToken memory info = paymentTokens[payToken];
    uint256 tokenUSDPrice = getTokenPriceInUSD(payToken);
    uint256 dynamicRate = getDynamicRate();

    tokenAmount = (convertDecimals(amountIn, info.decimals, 18) * tokenUSDPrice * 1e18) /
(dynamicRate * 1e8); // Normalize decimals
    require(tokensSold + tokenAmount <= totalTokensForSale, "Exceeds sale supply");
}

function getDynamicRate() public view returns (uint256) {
    uint256 remaining = totalTokensForSale - tokensSold;
    uint256 remainingInBps =  remaining * 1e18 / totalTokensForSale;

    uint256 squared = (remainingInBps * remainingInBps) / 1e18;
    uint256 factor = 1e18 + 2e18 * (1e18 - squared) / 1e18;

    return factor / baseRatePerUSD;
}
```

The issue is that the token amount calculation depends on the current state of `tokensSold`, which can be manipulated by front-runners. An attacker can monitor the mempool for large purchase transactions, then execute their own transaction with a higher gas price just before the victim's transaction. This increases `tokensSold` and consequently the `dynamicRate`, resulting in the victim receiving fewer tokens than expected. This is a classic sandwich attack scenario in DeFi.

Additionally, the contract relies on Chainlink price feeds for non-stable tokens, but doesn't include any mechanism to allow users to specify a minimum acceptable amount of tokens, leaving them fully exposed to price fluctuations between transaction submission and execution.

6 of 32 Findings

contracts/TokenPresale.sol

### Insufficient oracle validation in TokenPresale

Medium Risk

The `getTokenPriceInUSD` function in TokenPresale only performs a minimal validation check on the price feed data:

```
(, int256 price,,,) =
AggregatorV3Interface(paymentTokens[token].priceFeed).latestRoundData();
require(price > 0, "Invalid price");
```

Unlike the GasCreditVault contract which performs comprehensive validation, TokenPresale does not validate:

1. If the price feed is stale by checking the timestamp
2. If the round is complete by comparing answeredInRound and roundId
3. If the timestamp is valid

This lack of validation could allow the contract to use outdated, manipulated, or incorrect price data for token purchases. An attacker could exploit this by timing their purchases during oracle price feed anomalies or manipulations, potentially purchasing tokens at a much lower price than intended.

Given that the contract handles token sales, using unvalidated price feed data could lead to significant financial loss for the protocol.

7 of 32 Findings

contracts/MetaTxGateway.sol

### Refund failure can lead to Denial of Service for users

Medium Risk

The `executeMetaTransactions` function handles batches of transactions that may require native tokens. If a transaction in the batch fails, the contract attempts to refund the unused native tokens to the user ( `from` address). The refund is performed using a low-level `call`. If the `from` address is a smart contract that is not designed to receive native tokens (i.e., it lacks a `receive()` or payable `fallback()` function), the refund transfer will fail. This failure will cause the entire `executeMetaTransactions` call to revert. As a result, the user will be unable to execute any meta-transactions through the gateway, effectively locking them out of the system until they can resolve the issue with their receiving contract.

```
function executeMetaTransactions(
    // ...
) external payable nonReentrant whenNotPaused returns (bool[] memory successes) {
    // ...
    if (totalValueRequired > 0) {
        uint256 refundAmount = totalValueRequired - valueUsed;
        if (refundAmount > 0) {
            (bool refundSuccess, ) = payable(from).call{value: refundAmount}("");
            require(refundSuccess, "Refund failed"); // <-- Reverts if 'from' cannot receive
ETH
        }

        emit NativeTokenUsed(batchId, totalValueRequired, valueUsed, refundAmount);
    }
    // ...
}
```

8 of 32 Findings

contracts/GasCreditVault.sol contracts/TokenStaking.sol

### Risky Strict Equality Check

Medium Risk

The contracts contain several instances of strict equality checks (==) that could lead to unexpected behavior:

**GasCreditVault.sol:**
- In `removeToken(address)` at line 184: `require(IERC20(token).balanceOf(address(this)) == 0, "None Zero Balance")` - This check requires the token balance to be exactly zero, which might be difficult to achieve if there are dust amounts left.
- In `withdrawConsumedCredits()` at line 385: `contractBalance == 0` - This condition checks if the contract balance is exactly zero, which might lead to incorrect logic execution if there are dust amounts.

**TokenStaking.sol:**
- In `accrueReward()` at line 153: `totalStakes.lastUpdatedAt == 0` - This check determines if rewards have been initialized, but using a strict equality could cause issues if the value was set to something other than exactly zero.
- In `calculateAPR()` at line 185: `totalStakes.stakingShare == 0` - This check determines if there are any stakes, but using strict equality could lead to division by zero errors if the value is very small but not exactly zero.

Consider using greater than or less than comparisons instead of strict equality where appropriate, especially when dealing with financial calculations or balance checks.

9 of 32 Findings

contracts/TokenPresale.sol

### Permanently Locked Ether

Medium Risk

The TokenPresale contract has payable functions that allow it to receive Ether, but lacks any mechanism to withdraw this Ether if needed:

- `buyTokens()` at line 131-141 is payable
- `receive()` at line 153-155 is payable

Without a withdrawal function, any Ether sent to the contract will be permanently locked. This could happen if:

1. Users accidentally send Ether to the contract outside of the intended functions
2. The contract receives Ether as part of normal operations but has no way to distribute it
3. The contract is deprecated but still has Ether balance

Consider adding a withdrawal function that allows the contract owner or administrator to recover any Ether that might be locked in the contract.

10 of 32 Findings

contracts/GasCreditVault.sol

### Stablecoin Peg Assumption

Low Risk

The GasCreditVault assumes that stablecoins always maintain their 1:1 USD peg, which isn't always true in real-world scenarios:

```
if (info.isStablecoin) {
    return convertDecimals(amount, tokenDecimals, creditDecimals);
}
```

During market stress or for algorithmic stablecoins, this peg can break significantly. The contract has no mechanism to handle de-pegged stablecoins, which could lead to incorrect credit calculations. For example, if a stablecoin depegs to $0.50, users could deposit it at face value and receive twice the credits they should, creating an arbitrage opportunity and potential loss for the protocol.

11 of 32 Findings

contracts/GasCreditVault.sol

### Division by zero risk in GasCreditVault.withdrawConsumedCredits

Low Risk

In the `withdrawConsumedCredits` function, the contract calculates `tokenAmount` by dividing by `tokenValueInCredits`. However, there is no check to ensure that `tokenValueInCredits` is not zero before performing the division:

```
// Calculate the credit value for the token balance in terms of consumed credits
uint256 tokenValueInCredits = calculateCreditValue(token, contractBalance);

// Withdraw the corresponding amount of tokens based on the consumed credits
uint256 tokenAmount = (deltaCredits * contractBalance) / tokenValueInCredits;
```

If `tokenValueInCredits` is 0 (which could happen if the token's price feed returns 0 or if there's an error in the calculation), it would cause a division by zero error. This would revert the transaction and potentially prevent withdrawal of consumed credits, effectively locking them in the contract.

This vulnerability is particularly concerning because it affects the owner's ability to withdraw consumed credits, which is a core functionality of the protocol.

12 of 32 Findings

contracts/GasCreditVault.sol

### Unbounded Loops in Core Functions Can Lead to Denial of Service

Low Risk

Several core functions in the `GasCreditVault` contract, namely `consumeCredit`, `transferCredit`, and `withdrawConsumedCredits`, iterate over the `whitelistedTokens` array to perform their logic. The size of this array is controlled by the owner via the `whitelistToken` function and is not subject to any length restrictions.

If the owner whitelists a large number of tokens, the gas cost of executing these functions will grow linearly. Eventually, the gas required to complete the loop could exceed the block gas limit, causing transactions that call these functions to fail consistently. This would create a Denial of Service condition, preventing key protocol operations:

- Relayers would be unable to call `consumeCredit`, halting the meta-transaction service.
- Users would be unable to call `transferCredit`, freezing their ability to move credits.
- The owner would be unable to call `withdrawConsumedCredits`, preventing fee collection.

13 of 32 Findings

contracts/TokenPresale.sol

### Mismatched array validation in TokenPresale.buyTokens

Low Risk

The TokenPresale contract calculates token amounts based on the payment token and amount, but doesn't validate that the resulting token amount is greater than zero before transferring:

```
function buyTokens(address payToken, uint256 amountIn) external nonReentrant saleActive {
    require(paymentTokens[payToken].isAllowed, "Unsupported payment token");
    require(amountIn > 0, "Invalid amount");

    uint256 tokenAmount = calculateTokenAmount(payToken, amountIn);

    // Transfer stable token to contract
    IERC20(payToken).transferFrom(msg.sender, address(this), amountIn);

    // Transfer sale tokens to buyer
    saleToken.transfer(msg.sender, tokenAmount);
    tokensSold += tokenAmount;

    emit Purchased(msg.sender, payToken, amountIn, tokenAmount);
}
```

If due to rounding errors, precision issues, or extremely small input amounts, the `tokenAmount` could be zero. The contract would still process the payment, transferring tokens from the user to the contract, but the user would receive nothing in return.

While the dynamic rate calculation in `calculateTokenAmount` should typically result in a non-zero amount for any reasonable input, adding an explicit check would prevent potential edge cases where users could lose funds due to precision issues.

14 of 32 Findings

contracts/MetaTxGateway.sol

## No limit on batch transaction size in MetaTxGateway

Low Risk

The `executeMetaTransactions` function in MetaTxGateway does not limit the number of transactions in a batch:

```
function executeMetaTransactions(
    address from,
    MetaTransaction[] calldata metaTxs,
    bytes calldata signature,
    uint256 nonce,
    uint256 deadline
) external payable nonReentrant whenNotPaused returns (bool[] memory successes) {
    // ... validation
    require(metaTxs.length > 0, "Empty batch Txs");
    // ... execution
}
```

According to the Q&A, the developers assume the array length is less than 10, but there is no explicit check for this limit in the code. A malicious relayer could include too many transactions in a batch, causing the transaction to fail due to out-of-gas errors.

This is particularly problematic because a failed batch transaction means the user's signature is wasted (as the nonce doesn't increment), and they would need to create and sign a new meta-transaction batch. This could lead to denial of service where legitimate user transactions are repeatedly blocked by gas limit issues.

15 of 32 Findings

contracts/DI.sol contracts/DIVote.sol contracts/GasCreditVault.sol contracts/KOLAllocation.sol contracts/MetaTxGateway.sol contracts/TeamAllocation.sol contracts/TokenAirdrop.sol contracts/TokenPresale.sol contracts/TokenStaking.sol

### Non-Specific Solidity Pragma Version

Low Risk

All contracts in the project use floating pragma versions instead of fixed versions:

- Most contracts use: `pragma solidity ^0.8.20;`
- DIVote.sol uses: `pragma solidity ^0.8.19;`

Using floating pragmas (^) means the contract will compile with any compiler version that satisfies the constraint. This can lead to inconsistent bytecode when compiled with different versions, potentially introducing bugs or vulnerabilities that weren't present in the tested version.

Recommendation: Lock the pragma to a specific version to ensure consistent compilation results:

```
pragma solidity 0.8.20;
```

This ensures that the contract will always be compiled with the exact same compiler version, reducing the risk of unexpected behavior.

16 of 32 Findings

contracts/DI.sol contracts/GasCreditVault.sol contracts/KOLAllocation.sol contracts/TeamAllocation.sol contracts/TokenPresale.sol contracts/TokenStaking.sol

## Magic Numbers Instead Of Constants

Low Risk

Multiple contracts use magic numbers (hardcoded literals) throughout the codebase instead of named constants:

**DI.sol:**
- `uint256 airdropAllocation = 50_000_000 ether;` (repeated multiple times)

**GasCreditVault.sol:**
- `amount * (10 ** (to - from));` (power calculations)

**KOLAllocation.sol and TeamAllocation.sol:**
- `uint256 linearPart = (member.balance * 4 * (elapsed - CLIFF_DURATION)) / (VESTING_DURATION * 5);`

**TokenPresale.sol:**
-
`tokenAmount = (convertDecimals(amountIn, info.decimals, 18) * tokenUSDPrice * 1e18) / (dynamicRate * 1e8)`

**TokenStaking.sol:**
- `return 100;` (repeated multiple times)
- `if (amount >= 20_000 ether) return 115;`

Using magic numbers makes the code less readable and more error-prone when changes are needed. If the same value needs to be updated in multiple places, it's easy to miss some occurrences.

Recommendation: Define named constants for these values at the contract level:

```
// Examples
uint256 private constant AIRDROP_ALLOCATION = 50_000_000 ether;
uint256 private constant BASE_APR = 100;
uint256 private constant TIER1_THRESHOLD = 20_000 ether;
uint256 private constant TIER1_APR = 115;
```

17 of 32 Findings

contracts/DI.sol contracts/DIVote.sol contracts/GasCreditVault.sol contracts/KOLAllocation.sol contracts/MetaTxGateway.sol contracts/TeamAllocation.sol contracts/TokenAirdrop.sol contracts/TokenPresale.sol contracts/TokenStaking.sol

### PUSH0 Opcode Compatibility Issue

Low Risk

All contracts in the project use Solidity version 0.8.19 or 0.8.20, which target the Shanghai EVM version by default. This generates bytecode that includes the PUSH0 opcode, which may not be supported on all chains:

- Most contracts use: `pragma solidity ^0.8.20;`
- DIVote.sol uses: `pragma solidity ^0.8.19;`

The PUSH0 opcode was introduced in the Shanghai upgrade for Ethereum mainnet, but some Layer 2 solutions, sidechains, or other EVM-compatible blockchains might not have implemented this upgrade yet. Deploying contracts with PUSH0 opcodes on these chains will fail.

Recommendation: If you plan to deploy on chains other than Ethereum mainnet, explicitly specify an EVM version that doesn't use PUSH0 in your compiler settings:

```
// In hardhat.config.js or truffle-config.js
solidity: {
  version: "0.8.20",
  settings: {
    evmVersion: "paris"   // Use Paris instead of Shanghai
  }
}
```

18 of 32 Findings

contracts/GasCreditVault.sol contracts/TokenAirdrop.sol contracts/TokenPresale.sol contracts/TokenStaking.sol

### Unnecessary Modifier Usage

Low Risk

Several contracts define modifiers that are only used once or could be replaced with direct condition checks:

**GasCreditVault.sol:**
- `modifier onlyStablecoin(address token)`

**TokenAirdrop.sol:**
- `modifier airdropActive()`

**TokenPresale.sol:**
- `modifier saleOngoing()`

**TokenStaking.sol:**
- `modifier stakingActive()`

Modifiers that are only used once or twice increase code complexity without providing significant benefits. They can make code harder to follow as the execution jumps to a different location.

Recommendation: For modifiers used only once, consider replacing them with direct condition checks in the function body. For modifiers used in multiple places but with simple logic, consider using private functions instead:

```
// Instead of a modifier
function _requireStablecoin(address token) private view {
    require(stablecoins[token], "Not a stablecoin");
}

// Then in functions
function someFunction(address token) external {
    _requireStablecoin(token);
    // Function logic
}
```

19 of 32 Findings

contracts/GasCreditVault.sol contracts/MetaTxGateway.sol

contracts/TokenStaking.sol

## Empty Code Block Detection

Low Risk

Several contracts contain empty code blocks that serve no purpose:

**GasCreditVault.sol and MetaTxGateway.sol:**

- `function _authorizeUpgrade(address newImplementation) internal override onlyOwner {}`

**TokenStaking.sol:**

- `function updateReward() external accrueReward {}`

Empty code blocks can indicate incomplete implementation, forgotten code, or unnecessary functions. They can confuse readers of the code and may indicate potential issues.

Recommendation:

1. For the empty `_authorizeUpgrade` functions, add a comment explaining that the function is intentionally empty but required by the upgradeable contract pattern:

```
function _authorizeUpgrade(address newImplementation) internal override onlyOwner {
    // This function is required by UUPSUpgradeable but has no additional logic
    // beyond the onlyOwner modifier
}
```

2. For the `updateReward` function in TokenStaking.sol, either add implementation or remove it if it's redundant.

20 of 32 Findings

contracts/KOLAllocation.sol contracts/TeamAllocation.sol contracts/TokenAirdrop.sol contracts/DI.sol contracts/TokenPresale.sol contracts/TokenStaking.sol contracts/DIVote.sol

### Local Variable Shadowing

Low Risk

Multiple contracts have constructor parameters named `_owner` that shadow the state variable with the same name from the inherited Ownable contract:

- KOLAllocation.constructor(address,address)._owner at line 27
- TeamAllocation.constructor(address,address)._owner at line 27
- TokenAirdrop.constructor(address,address)._owner at line 19
- DI.constructor(address)._owner at line 26
- TokenPresale.constructor(address,uint256,address)._owner at line 36
- TokenStaking.constructor(address,address)._owner at line 55
- DIVote.constructor(ERC20Votes,address)._owner at line 32

Variable shadowing occurs when a local variable has the same name as a variable in an outer scope. This can lead to confusion and potential bugs as developers might inadvertently use the wrong variable. In these cases, the constructor parameter `_owner` shadows the `_owner` state variable from the OpenZeppelin Ownable contract.

Consider renaming the constructor parameters to avoid shadowing, for example using `initialOwner` instead of `_owner`.

21 of 32 Findings

contracts/KOLAllocation.sol contracts/TeamAllocation.sol

### Storage Array Modified Using Memory Reference

Low Risk

In both KOLAllocation.sol and TeamAllocation.sol, storage arrays are being accessed but the function is using memory parameters, which means changes won't be persisted to storage:

```
return calculateWithdrawable(membersInfo[index]);
```

This pattern suggests that the `calculateWithdrawable` function takes a memory parameter but is being passed a storage reference. If the function modifies the parameter, those changes won't be reflected in the storage variable.

To fix this issue:

1. If `calculateWithdrawable` should modify storage, change its parameter to use the `storage` keyword
2. If it should only read data, ensure it's properly marked as `view` and doesn't attempt to modify the parameter
3. If modifications are needed but shouldn't affect storage, explicitly create a memory copy before passing it to the function

22 of 32 Findings

contracts/KOLAllocation.sol contracts/TeamAllocation.sol contracts/TokenAirdrop.sol contracts/TokenPresale.sol contracts/TokenStaking.sol

## Unsafe ERC20 Operation Usage

Low Risk

Multiple contracts use unsafe ERC20 operations without checking return values or using SafeERC20:

**KOLAllocation.sol:**
- `token.transfer(msg.sender, withdrawable);`

**TeamAllocation.sol:**
- `token.transfer(msg.sender, withdrawable);`

**TokenAirdrop.sol:**
- `require(token.transfer(to, balance), "Withdraw failed");`

**TokenPresale.sol:**
- `IERC20(token).transfer(to, bal);` (multiple instances)

**TokenStaking.sol:**
- `stakingToken.transfer(msg.sender, pending);` (multiple instances)

Not all ERC20 tokens follow the standard correctly. Some tokens don't return a boolean value, others might return false on failure instead of reverting, and some might not revert on failure at all. This can lead to situations where transfers fail silently, potentially causing loss of funds.

Recommendation: Use OpenZeppelin's SafeERC20 library which handles these inconsistencies:

```
import {SafeERC20} from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol" ;

// In the contract
using SafeERC20 for IERC20;

// Replace transfer calls with safeTransfer
token.safeTransfer(recipient, amount);
```

23 of 32 Findings contracts/GasCreditVault.sol contracts/TokenPresale.sol

### Inadequate Price Feed Validation

Info

While the contracts do include some price feed validation, the checks may not be sufficient in all scenarios. For example, in GasCreditVault:

```
require(price > 0, "Invalid price from feed");
require(updatedAt > 0, "Invalid timestamp from feed");
require(block.timestamp - updatedAt <= PRICE_FEED_TIMEOUT, "Price feed too stale");
require(answeredInRound >= roundId, "Price feed round incomplete");
```

These checks don't account for extreme price volatility or flash crashes. Similarly, in TokenPresale:

```
(, int256 price,,,) =
AggregatorV3Interface(paymentTokens[payToken].priceFeed).latestRoundData();
require(price > 0, "Invalid price");
```

The contract only checks if the price is positive, without additional validation for staleness or anomalies. This could potentially allow transactions to proceed with outdated or manipulated price data, affecting the fairness of token sales or credit calculations.

24 of 32 Findings contracts/GasCreditVault.sol

### Withdrawal mechanism limited to stablecoins in GasCreditVault

Info

The GasCreditVault contract allows users to deposit any whitelisted token (stablecoins and non-stablecoins) through the `deposit` function. However, the `withdraw` function only allows withdrawal of stablecoins, as enforced by the `onlyStablecoin` modifier:

```
function withdraw(address token, uint256 creditAmount) external whenNotPaused
onlyStablecoin(token) {
    require(creditsInToken[msg.sender][token] >= creditAmount, "Insufficient token
balance");
    // ... rest of function
}
```

This asymmetry creates a critical issue: users who deposit non-stablecoin tokens (e.g., ETH, BTC) have no direct way to withdraw these tokens from the vault. Their funds become effectively locked in the contract, as they can only consume them as gas credits or transfer them to other users.

While this might be an intentional design choice, it creates a significant risk for users who may not fully understand this limitation when depositing non-stablecoin tokens.

25 of 32 Findings

contracts/GasCreditVault.sol

### Lack of price feed fallback in GasCreditVault

Info

The GasCreditVault contract relies on Chainlink price feeds for non-stablecoin tokens but does not implement any fallback mechanism if these price feeds fail or become unavailable. If a Chainlink oracle goes down or returns invalid data, users would be unable to deposit or withdraw tokens that rely on that price feed.

```
(uint80 roundId, int256 price, , uint256 updatedAt, uint80 answeredInRound) =
info.priceFeed.latestRoundData();

// Price feed validation
require(price > 0, "Invalid price from feed");
require(updatedAt > 0, "Invalid timestamp from feed");
require(block.timestamp - updatedAt <= PRICE_FEED_TIMEOUT, "Price feed too stale");
require(answeredInRound >= roundId, "Price feed round incomplete");
```

While the contract does implement staleness checks and other validations (which is good practice), it lacks a fallback mechanism such as using alternative price sources or allowing owner-set emergency prices when oracles fail.

This creates a potential single point of failure where oracle issues could disrupt the normal operation of the contract, affecting the core gas credit functionality of the protocol.

26 of 32 Findings

contracts/TokenPresale.sol

### Unchecked multiplication in convertDecimals may silently overflow, breaking presale pricing for huge deposits

Info

`TokenPresale.convertDecimals()` performs `amount * 10 ** (to - from)` inside an `unchecked` block when increasing decimal precision:

```
unchecked {
    return from > to ?
        amount / (10 ** (from - to)) :
        amount * (10 ** (to - from)); // unchecked multiplication
}
```

If a user passes an extremely large `amountIn`, the multiplication can wrap around $2^{256}$ and yield a much smaller number. Because the wrapped value is then multiplied by the USD price and divided later, the buyer receives far fewer tokens than expected, while the presale logic believes the cap is respected. The silent overflow violates arithmetic soundness and can distort token distribution.

27 of 32 Findings

contracts/GasCreditVault.sol

## Lack of validation for zero credit transfer

Best Practices

The `transferCredit` function in GasCreditVault.sol does not validate that the credit amount being transferred is greater than zero. This could lead to unnecessary gas consumption and event emissions for zero-value transfers.

```
function transferCredit(address receiver, uint256 credit) external whenNotPaused {
    address sender = msg.sender;
    require(receiver != address(0), "Invalid receiver address");
    require(credits[sender] >= credit, 'Invalid amount');

    uint256 remaining = credit;
    address[] memory tokens = whitelistedTokens.values();
    for (uint256 i = 0; i < tokens.length && remaining > 0; ++i) {
        address token = tokens[i];
        uint256 userCredited = creditsInToken[sender][token];

        if (userCredited == 0) continue;

        uint256 deduction = userCredited >= remaining ? remaining : userCredited;

        creditsInToken[sender][token] -= deduction;
        creditsInToken[receiver][token] += deduction;
        remaining -= deduction;
    }

    credits[receiver] += credit;
    credits[sender] -= credit;
    emit CreditTransfer(sender, receiver, credit);
}
```

While the function checks that the sender has enough credits (`credits[sender] >= credit`), it doesn't check that `credit > 0`. This means that users can call the function with `credit = 0`, which will execute the function but not actually transfer any credits. This would still emit a `CreditTransfer` event with a zero amount, which could be misleading for off-chain systems monitoring these events.

28 of 32 Findings

contracts/TokenPresale.sol contracts/TokenStaking.sol

### Missing Events for Critical Operations

Best Practices

Some important actions in the contracts don't emit events, making it harder to track changes off-chain and potentially limiting transparency. For example, in TokenPresale, adding payment tokens doesn't emit an event:

```
function addPaymentToken(address token, address priceFeed, uint8 decimals, bool isStable)
external onlyOwner {
    paymentTokens[token] = PaymentToken({
        token: token,
        priceFeed: priceFeed,
        decimals: decimals,
        isStable: isStable,
        isAllowed: true
    });
    // No event emitted
}
```

Adding comprehensive event logging for all significant state changes would improve transparency, facilitate off-chain monitoring, and make it easier to build user interfaces that accurately reflect contract state.

29 of 32 Findings

contracts/TokenStaking.sol contracts/TokenAirdrop.sol contracts/DI.sol

### Lack of Circuit Breakers in Critical Contracts

Best Practices

Several critical contracts in the ecosystem lack circuit breaker mechanisms or pausable functionality that would allow freezing operations in case of detected vulnerabilities or attacks. While the GasCreditVault has pause functionality, other contracts like TokenStaking, TokenAirdrop, and DI don't implement similar safety mechanisms. Adding pause functionality and emergency withdrawal features to these contracts would provide better protection against potential exploits and allow for orderly resolution of any issues that might arise.

30 of 32 Findings

contracts/TeamAllocation.sol contracts/KOLAllocation.sol

### Inefficient member lookup in allocation contracts

Best Practices

Both TeamAllocation and KOLAllocation use linear search in the `getMemberIndex` function to find members:

```
function getMemberIndex(address wallet) internal view returns (uint256) {
    for (uint256 i = 0; i < wallets.length; ++i) {
        if (wallets[i] == wallet) return i;
    }
    return NOT_FOUND;
}
```

This approach has O(n) complexity, which becomes inefficient as the number of members grows. For contracts with many members, this could lead to high gas costs or even transaction failures due to gas limits when calling functions like `withdraw()` or `getWithdrawable()` which depend on this lookup.

A more efficient approach would be to use a mapping from address to index, which would provide O(1) lookup complexity regardless of the number of members.

31 of 32 Findings

contracts/DIVote.sol

### Non-Immutable State Variable

Best Practices

The `voteToken` state variable in DIVote.sol (line 24) should be declared as immutable since it's only set in the constructor and never modified afterward.

Immutable variables are more gas efficient than regular state variables because they're stored in the contract bytecode rather than in storage. When a variable is set once in the constructor and never changed, marking it as immutable can save gas costs during contract execution.

To fix this issue, change the declaration to:

```
ERC20Votes public immutable voteToken;
```

32 of 32 Findings

contracts/TeamAllocation.sol contracts/KOLAllocation.sol

### Inefficient Array Length Usage in Loop

Best Practices

Both TeamAllocation.sol and KOLAllocation.sol access the storage array length in each loop iteration, which is inefficient and increases gas costs:

- In TeamAllocation.sol at line 95: `i < wallets.length`
- In KOLAllocation.sol at line 95: `i < wallets.length`

Accessing storage variables is more expensive than memory variables. When the length of an array is accessed in each iteration of a loop, it results in unnecessary storage reads.

To optimize gas usage, cache the array length in a local variable before entering the loop:

```
uint256 walletsLength = wallets.length;
for (uint256 i = 0; i < walletsLength; i++) {
    // Loop body
}
```

## Disclaimer

Kindly note, the Audit Agent is currently in beta stage and no guarantee is being given as to the accuracy and/or completeness of any of the outputs the Audit Agent may generate, including without limitation this Report. The results set out in this Report may not be complete nor inclusive of all vulnerabilities. The Audit Agent is provided on an 'as is' basis, without warranties or conditions of any kind, either express or implied, including without limitation as to the outputs of the code scan and the security of any smart contract verified using the Audit Agent.

Blockchain technology remains under development and is subject to unknown risks and flaws. This Report does not indicate the endorsement of any particular project or team, nor guarantee its security. Neither you nor any third party should rely on this Report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset.

To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this Report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.