TRAVESSIA CREDIT
Security Review

FEBRUARY, 2026

www.chaindefenders.xyz
https://x.com/DefendersAudits

Lead Auditors

PeterSR

0x539.eth

Table of Contents

Protocol Summary

Travessia Credit is a private credit protocol enabling users to deposit capital into 90-day yield-generating cycles managed by Tauri. The system implements a batched vault approach where 6 staggered vaults start 15 days apart, ensuring continuous entry points and seamless cycle transitions. Users earn yield based on a fixed rate that accrues per block, with flexible withdrawal options and oracle-based pricing.

Disclaimer

The Chain Defenders team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the implementation of the contracts.

Risk Classification

Impact/Likelihood	High	Medium	Low
High	High	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Audit Details

Commit Hash

8c165a5947dc26f427d17a13501769cb3a8c43a6

Scope

Id	Files in scope
1	./src/cycle-vaults/Configurator.sol
2	./src/cycle-vaults/Minter.sol
3	./src/cycle-vaults/Oracle.sol
4	./src/cycle-vaults/ReceiptToken.sol
5	./src/cycle-vaults/Redeemer.sol

Roles

Id	Roles
1	Owner
2	User

Executive Summary

Issues Found

Severity	Count	Description
High	0	Critical vulnerabilities
Medium	1	Significant risks
Low	7	Minor issues with low impact
Informational	2	Best practices or suggestions
Gas	1	Optimization opportunities

Issues Statuses

Id	Title	Status
M-01	‘minWithdrawal‘ Changes Can Lead To Locked User Funds	Acknowledged
L-01	Cycles And Deposit Windows Can Be Wrongfully Mutated	Acknowledged
L-02	‘minDeposit‘ Can Be Modified During A Deposit Window	Acknowledged
L-03	Missing Zero Address Checks And No ‘Ownable2Step‘ Used	Acknowledged
L-04	Cycles Can Be Prolonged, Shortened Or Disabled	Acknowledged
L-05	Price Configuration Changes During Active Cycles Lead To Changing Withdrawal Amounts	Acknowledged
L-06	Centralization Risk With How Ending A Cycle Works Currently	Acknowledged
L-07	Reusing The Same Receipt Token With Different Deposit Tokens With Different Decimals Will Lead To Share Calculation Inconsistencies	Acknowledged
I-01	Minter And Redeemer Can Be Potentially Pointing To Different Configurators Than The Chosen One	Acknowledged
I-02	Missing Token Whitelisting Will Lead To Confusing Errors If Users Use Unsupported Tokens	Acknowledged
G-01	Unnecessary Storage Field Leading To Higher Gas Costs On Read And Write	Acknowledged

Security Grade

Based on the audit findings and code quality, the overall protocol security grade is 90.00 / 100.00. This reflects an excellent security posture with almost no room for improvement.

Findings

Medium

Mid 01 `minWithdrawal` Changes Can Lead To Locked User Funds

Location

Configurator.sol

Description

The `minWithdrawal` parameter in the `GlobalConfig` struct is mutable and can be increased at any time via `Configurator::setGlobalConfig`. If `minWithdrawal` is increased during an active cycle (or at its end), users who deposited funds under the previous threshold may find themselves unable

to withdraw because their withdrawal amount now falls below the new, higher minimum withdrawal requirement. This creates a situation where user funds are effectively locked in the contract.

Additionally, if `minDeposit` and `minWithdrawal` are not properly aligned, users can be allowed to deposit amounts that are subsequently impossible to withdraw - for example, if `minDeposit = 100` but `minWithdrawal = 500`, any user depositing between such tokens that in the end result in a withdrawal amount between 0 and 499 will be permanently unable to withdraw their funds.

Proof of Concept

Let's have the following situations:

Scenario 1: Mid-Cycle Increase

1. Initial configuration: `minWithdrawal = 100 USDC`
2. User deposits `150 USDC` during the deposit window
3. Owner calls `setGlobalConfig` increasing `minWithdrawal` to `500 USDC` mid-cycle
4. User attempts to request a withdrawal but the transaction reverts because it's less than 500
5. User's funds are locked, unable to execute their withdrawal strategy

Scenario 2: Misaligned Parameters

1. Configuration: `minDeposit = 100 USDC`, `minWithdrawal = 500 USDC`
2. User deposits `100 USDC` (meets `minDeposit` but will not meet `minWithdrawal`)
3. When the user later attempts to withdraw, the transaction reverts
4. User's funds are permanently locked - they can never withdraw

Recommendation

Add validation in `setGlobalConfig` to block changes to `minWithdrawal` while any cycle is active for any token. Add a validation rule to prevent the misalignment scenario entirely. Clearly document to administrators that `minDeposit` and `minWithdrawal` must be carefully aligned, with `minDeposit ⩽ minWithdrawal`, and that increasing `minWithdrawal` mid-cycle can lock user funds.

Status

Acknowledged

Low

Low 01 Cycles And Deposit Windows Can Be Wrongfully Mutated

Location

Configurator.sol

Description

The `cycleDuration` and `depositWindow` fields in the `GlobalConfig` struct are mutable state variables that can be modified at any time via the `Configurator::setGlobalConfig` function. These parameters control critical timing for the cycle system:

- `cycleDuration`: Determines the length of each deposit/withdrawal cycle (e.g., 90 days)
- `depositWindow`: Determines when deposits are accepted relative to the cycle start (e.g., 15 days before)

The vulnerability arises when these parameters are changed while an active cycle is underway. This allows the owner to:

- Prolong or shorten ongoing cycles: Users expecting their funds to unlock at a specific time may find the cycle extended or abruptly shortened
- Alter deposit windows mid-cycle: Users planning timed deposits could be denied access or unexpectedly granted/denied access to the deposit window
- Cause value loss: Unpredictable timing changes can result in users missing deposit/withdrawal windows, locking their capital longer than expected or preventing them from executing their withdrawal strategy

Users have no recourse once these parameters are changed.

Recommendation

Restrict modifications to `cycleDuration` to only occur when no active cycles are underway and to `depositWindow` to only occur when no active deposit windows are underway.

Status

Acknowledged

Low 02 `minDeposit` Can Be Modified During A Deposit Window

Location

Configurator.sol

Description

The `minDeposit` parameter in the `GlobalConfig` struct is a mutable state variable that can be modified at any time via the `Configurator` :: `setGlobalConfig` function. This parameter sets the minimum deposit amount required from users.

The vulnerability arises when `minDeposit` is increased during an active deposit window. Users who have planned or automated deposits based on the current `minDeposit` will have their transactions rejected if the owner increases this value mid-window. This is particularly impactful for:

- Users with automated deposit scripts or bots that submit transactions at predetermined times
- Users who calculated their deposit size based on the current minimum threshold
- Users expecting to deposit during a specific deposit window

When a transaction fails due to an increased `minDeposit`, the user misses their intended deposit opportunity and must wait for the next cycle, resulting in lost investment time and potential opportunity costs.

Recommendation

Choose one of the following recommendations:

1. Prevent changes during active cycles (preferred): Add validation in `setGlobalConfig` to prohibit modifying `minDeposit` while any cycle is active:
2. Implement a timelock: Introduce a delay mechanism before `minDeposit` changes take effect, giving users advance notice to adjust their automation or deposit plans accordingly.

Status

Acknowledged

Low 03 Missing Zero Address Checks And No `Ownable2Step` Used

Location

Configurator.sol

Minter.sol

Redeemer.sol

Oracle.sol

ReceiptToken.sol

Description

Configurator.sol

The `Configurator` constructor accepts `_owner` and `_depositToken` parameters without validating that they are not zero addresses. If either parameter is passed as `address(0)`:

- `_owner = address(0)`: The contract becomes unowned since Solmate's `Owned` contract doesn't validate zero addresses. This makes the contract permanently unmanageable—no one can call `onlyOwner` functions like `setGlobalConfig` or `setCycleStart`.
- `_depositToken = address(0)`: The immutable `depositToken` is set to a zero address. Any external call to `depositToken` methods (e.g., in `Minter` or `Redeemer` when calling `safeTransferFrom`) will fail, breaking core functionality.

Minter.sol

The `Minter` constructor accepts `_owner` and `_configurator` parameters without validation. If either is a zero address:

- `_owner = address(0)`: The contract has no owner, making all `onlyOwner` functions permanently inaccessible. No future configurations or emergency functions can be called.
- `_configurator = address(0)`: The immutable `configurator` reference points to the zero address. When `_deposit` calls `configurator.getGlobalConfig` or other configurator methods, the transaction reverts.

Redeemer.sol

The `Redeemer` constructor accepts `_owner` and `_configurator` parameters without validation. The same issues apply as in `Minter.sol`:

- `_owner = address(0)`: No owner for the contract, making `endCycle` permanently inaccessible. Cycle processing cannot occur, trapping users' withdrawal requests in a pending state forever.
- `_configurator = address(0)`: Calls to `configurator.getGlobalConfig` in `_requestWithdrawal` and `endCycle` revert, breaking both withdrawal requests and cycle processing.

Oracle.sol

The `Oracle` constructor accepts `_owner` without validation. If `_owner = address(0)`, the contract has no owner, making `setPriceConfig` and price override functions permanently inaccessible. Without the ability to set or update prices, the entire pricing mechanism fails, breaking deposit/withdrawal calculations throughout the system.

ReceiptToken.sol

The `ReceiptToken` constructor accepts `_owner` without validation. If `_owner = address(0)`, the token has no owner, making `setMinter` and `setBurner` permanently inaccessible. Minters and burners cannot be set, meaning tokens cannot be minted or burned, breaking the entire minting and redemption workflow.

Recommendation

Consider migrating from Solmate's `Owned` to OpenZeppelin's `Ownable2Step`, which: - Includes built-in zero address validation

- Implements a two-step ownership transfer (propose -> accept) preventing accidental ownership loss
- Is battle-tested and widely adopted in production systems

Also, validate all address fields to not be equal to `address(0)`.

Status

Acknowledged

Low 04 Cycles Can Be Prolonged, Shortened Or Disabled

Location

Configurator.sol

Description

The `Configurator::setCycleStart` function allows the owner to modify the cycle start timestamp for any token without any restrictions. There is no validation to prevent changing the cycle start while a cycle is actively in progress (i.e., when `block.timestamp` is between the cycle start and cycle end).

If `setCycleStart` is called while a cycle is underway, the owner can:

- Shorten the cycle: Set a new `cycleStart` that is further in the past, causing the current cycle to end prematurely. Users expecting their deposits to be locked until the previous cycle end may find their funds suddenly available for withdrawal before they anticipated.
- Prolong the cycle: Set a new `cycleStart` closer to the current time, effectively extending the current cycle end time. Users expecting access to their funds at a specific time find the cycle extended, locking their capital longer than planned.
- Deactivate the cycle: Set `cycleStart` to 0, which deactivates the token entirely (the contract checks `if (start == 0) revert TokenNotActive()`). This would prevent new deposits and trap ongoing withdrawal requests in the system.

All these actions directly harm users by disrupting the timing guarantees they relied upon when depositing funds. There is no way for users to anticipate or respond to such changes, and the changes take immediate effect.

Recommendation

Add a validation check in `setCycleStart` to prevent modifications when a cycle is currently active:

```
function setCycleStart(
    address token,
    uint256 _cycleStart
) external onlyOwner {
    uint256 currentStart = cycleStart[token];

    // Prevent changing cycle start if cycle is already active
    if (currentStart ≠ 0 && block.timestamp ≥ currentStart) {
        uint256 cycleEnd = currentStart + globalConfig.cycleDuration;
        require(
            block.timestamp ≥ cycleEnd,
            "Cannot change cycle start while cycle is active"
        );
    }

    cycleStart[token] = _cycleStart;
```

```
17      emit CycleStartSet(token, _cycleStart);
18  }
```

This check ensures that cycle start can only be changed before the cycle begins (`block.timestamp < currentStart`) or after the current cycle has completely ended (`block.timestamp ⩾ cycleEnd`) but not during an active cycle.

Status

Acknowledged

Low 05 Price Configuration Changes During Active Cycles Lead To Changing Withdrawal Amounts

Location

Oracle.sol

Description

The `Oracle :: setPriceConfig` function allows the owner to modify the price calculation parameters (`index`, `ratePerSecond`, and `lastUpdatedTimestamp`) at any time without restrictions. These parameters directly affect the price calculation used throughout the system:

```
price = config.index + (timeElapsed * config.ratePerSecond);
```

Because price is mutable and can be changed mid-cycle, it creates inconsistencies:

Impact on Deposits (`Minter`):

- Users deposit at a specific price to receive a calculated number of shares: `shares = (amount * depositTokenDecimals) / price`
- If `setPriceConfig` is called during the deposit window, new depositors will receive different share amounts for the same deposit, creating unfairness
- Early depositors lock in one price while later depositors in the same cycle get a different price

Impact on Withdrawal Requests (`Redeemer`):

- When users request withdrawals, the share value is calculated as: `sharesValue = (shares * price) / depositTokenDecimals`

- If the price config is changed mid-cycle, the value of pending withdrawal requests changes retroactively
- A request that met the `minWithdrawal` requirement may no longer meet it after a price config change
- Users cannot predict the final redemption value of their withdrawal request

Example Scenario:

1. Cycle starts with `index = 100`, `ratePerSecond = 1`
2. User deposits 1000 tokens at price 100, receives 100 shares
3. After 50 seconds, price = 100 + (50 * 1) = 150
4. Owner calls `setPriceConfig` with `index = 200`, `ratePerSecond = 0.5`
5. Now the price at the same timestamp would be calculated differently: 200 + (50 * 0.5) = 225
6. When the cycle ends and shares are redeemed, the new price formula applies
7. The user's 100 shares may now be worth a drastically different amount than expected

Recommendation

Add validation to `setPriceConfig` to prevent changes while any cycle is active. This requires checking if there are any active cycles across all tokens:

```
function setPriceConfig(
    address base,
    address quote,
    uint256 ratePerSecond,
    uint256 index,
    uint256 startTimestamp
) external onlyOwner {
    // Check that no cycles are currently active for this token pair
    // (This requires access to Configurator to check cycle status)
    require(
        !hasCycleActive(base, quote),
        "Cannot modify price config during active cycles"
    );

    priceConfigs[base][quote] = PriceConfig({
        ratePerSecond: ratePerSecond,
        index: index,
        lastUpdatedTimestamp: startTimestamp
    });

```

```
21         emit PriceConfigSet(base, quote, ratePerSecond, index, startTimestamp);
22     }
```

Status

Acknowledged

Low 06 Centralization Risk With How Ending A Cycle Works Currently

Location

Redeemer.sol

Description

The `Redeemer` :: `endCycle` function is restricted to `onlyOwner` and is a critical operation required to process all pending withdrawal requests at the end of each cycle. If the owner fails to call this function - whether due to negligence, system outage, or malicious intent - users become unable to withdraw their funds.

How the Vulnerability Manifests:

1. User Deposits and Requests Withdrawal: A user deposits into a cycle and later requests a regular withdrawal, locking their receipt tokens in the `Redeemer` contract.
2. Cycle Ends: The cycle end timestamp arrives, but the owner does not call `endCycle`.
3. Funds Are Locked: The user's receipt tokens remain locked in the contract with status `Requested`. The `fulfill` function only processes requests with status `Pending`, so the user cannot claim their funds.
4. No Escape Mechanism: There is no way for the user to:
    - Cancel their withdrawal request and retrieve their receipt tokens
    - Withdraw directly
    - Force the cycle to process
    - Escalate the issue through a timelock or alternative mechanism
5. Permanent Lock: If the owner never calls `endCycle`, the user's funds are permanently locked, becoming inaccessible.

Additional Risk - No Cancellation Path: Users who request regular withdrawals have no mechanism to cancel:

```
if (requestType == RequestType.Regular) {
    // Receipt tokens are transferred to this contract and locked
    ReceiptToken(token).transferFrom(msg.sender, address(this), shares);
    // No way to retrieve them if endCycle is never called
}
```

In contrast, early withdrawal requests immediately set the expected amount and burn tokens, allowing users to proceed to fulfillment independently. Regular withdrawal users have no such option.

Recommendation

Option 1: Decentralize Cycle Processing Allow anyone (not just the owner) to call `endCycle`, removing the single point of failure:

```
function endCycle(
    address token,
    uint256 cycleNumber,
    uint256[] calldata requestIds
) external {  // Remove onlyOwner
    // Verify cycle has ended
    uint256 cycleEnd = configurator.getCycleEnd(token, cycleNumber);
    if (block.timestamp < cycleEnd) revert CycleNotEnded();

    // ... rest of function
}
```

This allows anyone (users, keepers, automated bots) to trigger the critical operation.

Option 2: Implement a Keeper Network Use a decentralized keeper system (e.g., Gelato Network, Chainlink Automation, or custom incentivized keepers) to automatically call `endCycle` when cycles complete.

Option 3: Implement Withdrawal Request Cancellation Allow users to cancel regular withdrawal requests within a grace period, retrieving their locked receipt tokens.

Implement Option 1 (decentralized processing) as the primary fix, combined with Option 3 (cancellation mechanism) to provide users with an escape path if cycle processing is delayed. This eliminates the single point of failure while respecting the normal withdrawal flow.

Status

Acknowledged

Low 07 Reusing The Same Receipt Token With Different Deposit Tokens With Different Decimals Will Lead To Share Calculation Inconsistencies

Location

Minter.sol

Description

The share calculation in `Minter::_deposit` uses the formula:

```
uint256 shares = (amount * (10 ** depositToken.decimals())) / price;
```

This formula is tied to the decimals of the deposit token via `depositToken.decimals`. If the same receipt token is deployed and used across multiple `Minter` instances, each with a different `Configurator` pointing to a different deposit token with different decimals, the share minting becomes inconsistent.

Example Scenario:

Minter A (Configurator A):

- Deposit token: USDC (6 decimals)
- Receipt token: LiquidityToken
- Share calculation: `(amount * 10^6) / price`

Minter B (Configurator B):

- Deposit token: DAI (18 decimals)
- Receipt token: LiquidityToken (same token, reused)
- Share calculation: `(amount * 10^18) / price`

The Problem: If a user deposits 1000 USDC through Minter A at price 100:

```
shares = (1000 * 10^6) / 100 = 10,000,000 LiquidityTokens
```

If another user deposits 1000 DAI through Minter B at the same price 100:

```
shares = (1000 * 10^18) / 100 = 10,000,000,000,000,000,000 LiquidityTokens
```

Both users deposited equivalent value (1000 of their respective tokens), but received vastly different share amounts. The receipt token loses its value consistency.

Impact on Redemption: When redeeming in `Redeemer`, the inverse calculation is:

```
uint256 sharesValue = (shares * price) / (10 ** depositToken.decimals());
```

If a receipt token was minted under Minter A (using USDC decimals) but redemption is processed through a Redeemer connected to a Configurator with DAI as the deposit token, the valuation breaks:

- Shares minted: based on USDC's 6 decimals
- Shares redeemed: calculated using DAI's 18 decimals
- Result: Incorrect withdrawal amounts (either over/underpayment)

Additional Complexity:

The `ReceiptToken` contract has no awareness of which deposit token configuration was used when shares were minted. There's no tracking mechanism to verify that shares are being redeemed under the same configuration they were created under.

Recommendation

Document and enforce that each receipt token can only be used with a single `Minter` instance, which must always use the same deposit token. Include this requirement in deployment guidelines:

```
/// @notice The receipt token must be used with a single Minter instance.
/// Do not reuse the same receipt token across multiple Minters with different deposit tokens.
/// Doing so will cause incorrect share calculations due to decimal misalignment.
contract Minter is Owned {
    // ...
}
```

Add a comment in `_deposit`:

```
// Calculate shares
// NOTE: This formula is calibrated for the specific depositToken configured in the Configurator.
// Do NOT reuse this receipt token with a different Minter/Configurator pair with a different deposit token.
// Different deposit token decimals will result in incorrect share valuations.
uint256 shares = (amount * (10 ** depositToken.decimals())) / price;
```

Status

Acknowledged

Informational

Info 01 Minter And Redeemer Can Be Potentially Pointing To Different Configurators Than The Chosen One

Location

Configurator.sol

Description

The `setGlobalConfig` function in `Configurator` accepts a `GlobalConfig` struct containing addresses for the `minter`, `redeemer`, and `oracle` contracts. However, there is no validation to ensure that these contract instances are actually configured to use the current `Configurator` instance.

If a configuration is set where the `minter` or `redeemer` addresses point to contracts that reference a different `Configurator` instance, the system becomes incoherent:

- The primary `Configurator` is updated with new configuration parameters (e.g., `minDeposit`, `cycleDuration`, `depositWindow`)
- The `Minter` contract reads from its configured `Configurator` (which is different)
- The `Redeemer` contract reads from its configured `Configurator` (which may also be different)
- All three may be reading different, potentially stale or conflicting configurations

Recommendation

Add validation in `setGlobalConfig` to verify that the `minter` and `redeemer` contracts passed in the configuration are actually configured to use the current `Configurator` instance:

```
function setGlobalConfig(GlobalConfig calldata _config) external onlyOwner {
    // Verify that minter and redeemer point back to this configurator
    require(
        Minter(_config.minter).configurator() == address(this),
        "Minter does not point to this Configurator"
    );
    require(
        Redeemer(_config.redeemer).configurator() == address(this),
        "Redeemer does not point to this Configurator"
    );

    globalConfig = _config;
```

```
        emit GlobalConfigUpdated(_config);
    }
```

Status

Acknowledged

Info 02 Missing Token Whitelisting Will Lead To Confusing Errors If Users Use Unsupported Tokens

Location

Redeemer.sol

Description

The `_requestWithdrawal` function in `Redeemer` accepts any token address without validating whether that token is actually supported by the system. When a user calls `Redeemer::requestWithdrawal` or `Redeemer::requestEarlyWithdrawal` with an unsupported token, the function attempts to fetch its price from the Oracle:

```
uint256 sharesValue = (shares * oracle.getPrice(token, address(depositToken))) / (10 **
    depositToken.decimals());
```

If the price configuration for that token pair has not been set in the Oracle contract, the call reverts with a generic `PriceConfigNotSet` error. This error provides no context about:

- Which specific token caused the failure
- What tokens are actually supported by the system
- How to find the list of whitelisted tokens
- Whether the token address was misspelled or incorrect

A user attempting to withdraw with the wrong token address receives a cryptic error message that doesn't help them understand the problem or correct it. This creates friction and confusion, especially in a system where multiple tokens may be supported.

Recommendation

Implement a token whitelist on the `Redeemer` contract and validate tokens before attempting to fetch prices:

```solidity
/*//////////////////////////////////////////////////////////////
                              STORAGE
//////////////////////////////////////////////////////////////*/

mapping(address => bool) public whitelistedTokens;

/*//////////////////////////////////////////////////////////////
                          ADMIN FUNCTIONS
//////////////////////////////////////////////////////////////*/

function setTokenWhitelist(address token, bool isWhitelisted) external onlyOwner {
    whitelistedTokens[token] = isWhitelisted;
}

/*//////////////////////////////////////////////////////////////
                        WITHDRAWAL REQUEST
//////////////////////////////////////////////////////////////*/

function _requestWithdrawal(
    address token,
    uint256 shares,
    address to,
    RequestType requestType,
    Status status
) internal returns (uint256 requestId) {
    // Validate token is whitelisted
    require(whitelistedTokens[token], "UnsupportedToken");

    Configurator.GlobalConfig memory config = configurator
        .getGlobalConfig();
    ERC20 depositToken = configurator.depositToken();

    // Get current price to check minimum
    Oracle oracle = Oracle(config.oracle);
    uint256 sharesValue = (shares *
        oracle.getPrice(token, address(depositToken))) /
        (10 ** depositToken.decimals());

    if (sharesValue < config.minWithdrawal) revert BelowMinWithdrawal();

    // ... rest of function
}
```

Status

Acknowledged

Gas

Gas 01 Unnecessary Storage Field Leading To Higher Gas Costs On Read And Write

Location

Configurator.sol

Description

The `cycleDuration` field in the `GlobalConfig` struct is stored as a mutable state variable, but according to a comment, it is always intended to be 90 days in seconds. Storing this value in mutable storage introduces unnecessary gas costs:

- SSTORE overhead: Writing to `globalConfig.cycleDuration` in `setGlobalConfig` consumes gas for storage updates
- SLOAD overhead: Reading from `globalConfig.cycleDuration` in functions like `getCurrentCycle`, `getCycleStart`, `getCycleEnd` incurs storage read costs

Recommendation

Replace `cycleDuration` with an immutable constant defined at the contract level:

```
uint256 public constant CYCLE_DURATION = 90 days;
```

Then:

1. Remove `cycleDuration` from the `GlobalConfig` struct
2. Update all references to `globalConfig.cycleDuration` throughout the contract to use `CYCLE_DURATION` instead
3. Remove `cycleDuration` initialization from the `setGlobalConfig` function

This approach reduces gas consumption on both writes and reads while making the immutable nature of the value explicit in the code.

Status

Acknowledged