Null Return

Launchnodes

SMART CONTRACT AUDIT REPORT

Produced for **Launchnodes**
by **Null Return**

Date: **November 2025**

PUBLIC

Contents

Prepared by

Project Manager: Alex Demidov

Auditors: Andy Cho, Alexander Mazaletskiy

Introduction

This document presents the results of a smart contract audit conducted for the Stablecoin For Impact (SFI) protocol. The purpose of the audit was to evaluate the security and correctness of SFI's smart contracts prior to deployment. Special attention was given to identifying vulnerabilities that may lead to loss of funds, compatibility issues with tokens of varying decimal precision, and logical errors in yield distribution and withdrawal operations.

About Stablecoin For Impact

Stablecoin For Impact (SFI) enables anyone to use stablecoins (USDT, USDC, and DAI) to generate secure, sustainable yield through Aave's collateralized supply markets, and donate that yield to organizations driving real-world social impact. This model empowers individuals and legal entities to contribute to poverty alleviation, climate action, and other global causes, without giving up their original capital. Key features include:

- **Yield Donation Model**: Users deposit stablecoins and direct a percentage (or all) of the generated yield to charitable organizations while retaining their principal.
- **Aave Integration**: Leverages Aave's proven DeFi lending protocol for secure yield generation through collateralized supply markets.
- **Multi-Token Support**: Supports USDT, USDC, DAI, and other stablecoins with varying decimal precision (2-18 decimals).
- **Transparent Impact Tracking**: Builds a marketplace of credible, data-driven organizations working on poverty reduction, climate action, and other pressing challenges.
- **Low-Risk Contributions**: Donors retain their original capital, creating a sustainable, long-term financing tool for social impact.

The protocol is designed to demonstrate that DeFi tools can help address the world's biggest challenges while preserving user funds and trust, challenging the perception that crypto is merely speculative.

Project Overview

Stablecoin For Impact is designed as a purpose-driven DeFi platform that transforms stablecoin holdings into a force for social good. By depositing stablecoins into Aave's lending pools, users generate yield that is automatically distributed to verified charitable organizations based on user-defined percentages. The platform maintains full capital preservation—users never lose their principal and can withdraw at any time.

The architecture combines on-chain smart contracts with Aave protocol integration, ensuring robust security for deposits, yield calculation, and distribution. The protocol implements a share-based system for tracking user deposits and NGO allocations, with dynamic calculations accounting for accrued interest from Aave.

Scope

This audit covered the core components of the Stablecoin For Impact protocol, with a focus on the correctness, safety, and robustness of its smart contracts. The analysis was based on the source code provided in the Launchnodes-Ltd/stablecoin-impact repository at the specified commits.

Repository: https://github.com/Launchnodes-Ltd/stablecoin-impact

Initial Audit Commit: e7437fd63fa2c13866e7162a99fe5efb8844fb93

Fixes Commit: fa3b273e07be0ca4538f3d8fb8ba6b7e42c7e853

Audit Target: Solidity smart contracts implementing core deposit logic, Aave integration, yield distribution, withdrawal mechanisms, and token validation.

Files Audited: - stablecoin-impact/src/AaveImpact.sol - stablecoin-impact/src/AaveImpactFactory.sol

Methodology

The goal of this audit was to identify potential vulnerabilities, logical errors, and deviations from best practices in the Stablecoin For Impact smart contracts. Our process followed a structured and layered approach, including:

1. **Manual Code Review** Each file in scope was reviewed manually by experienced security auditors to analyze:
    - Correctness of deposit and withdrawal logic
    - Aave protocol integration security
    - Token decimal compatibility across different stablecoins
    - Share calculation and yield distribution mechanisms
    - Access control and ownership management
2. **Compliance Verification** The contracts were checked for conformance with relevant Ethereum standards, including:
    - ERC20 token interactions and decimal handling
    - Aave V3 protocol integration patterns
    - OpenZeppelin upgradeable contract standards
3. **Static Analysis & Tooling** We used automated tools to detect:
    - Known vulnerability patterns (e.g., reentrancy, arithmetic issues)
    - Storage layout inefficiencies
    - Unused code and declarations
    - Gas optimization opportunities
4. **Threat Modeling** We examined critical assumptions and attack surfaces relevant to SFI's architecture, including:
    - Token decimal compatibility issues (2-18 decimals)
    - Aave protocol dependency risks
    - Minimum deposit and withdrawal validation
    - Share calculation edge cases with small amounts
    - Zero address and null value validations
5. **Issue Reporting & Validation** All issues were documented with severity levels (Medium, Low, Informational). After reporting, the SFI team responded and applied fixes, which we then re-reviewed and validated.

Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Findings

In this section, we document the security issues identified during the audit of Stablecoin For Impact, along with their severity levels, status, and relevant recommendations. Each finding includes a description of the issue, potential impact, and the client's response with implemented fixes.

ID	Title	Severity	Status
M-1	Missing Aave Protocol Token Validation in Factory	Medium	Resolved
M-2	Zero NGO Assets Due to Missing Decimal-Aware Minimum Deposit Validation	Medium	Resolved
M-3	Hardcoded WITHDRAW_GAP Incompatible with Low-Decimal Tokens	Medium	Resolved
M-4	Hardcoded MIN_WITHDRAWAL_AMOUNT Blocks Legitimate Withdrawals	Medium	Resolved
L-1	Single-Step Ownership Transfer Risk	Low	Resolved
I-1	Redundant Storage Reads	Informational	Resolved
I-2	Unused Custom Error Declarations	Informational	Resolved
I-3	Unnecessary Inheritance of ERC721HolderUpgradeable and Unused ReentrancyGuardUpgradeable	Informational	Resolved
I-4	Inefficient Struct Storage Layout (Unpacked Variables)	Informational	Resolved
I-5	Duplicate Error Definitions	Informational	Resolved

Medium

[M-1] Missing Aave Protocol Token Validation in Factory

- **Status:** Resolved
- **Initial Commit:** `e7437fd63fa2c13866e7162a99fe5efb8844fb93`
- **Fix Commit:** `fa3b273e07be0ca4538f3d8fb8ba6b7e42c7e853`
- **Location:** `AaveImpactFactory.sol:addValidDepositToken()`

- **Impact:** The factory allows adding arbitrary token addresses without verifying Aave protocol support, leading to:
  - Failed deposits when users supply unsupported tokens
  - Locked user funds if tokens transfer but cannot be supplied to Aave
  - Contract malfunction from Aave reverts on unsupported assets
- **Description:** The `addValidDepositToken()` function lacks validation that the token is an active reserve in Aave with a corresponding aToken address.

```
function addValidDepositToken(address _newTokenAddress, address
 ↪  _wrappedTokenAddress) public onlyOwner {
    if (_newTokenAddress == address(0)) revert NullAddress();
    if (_wrappedTokenAddress == address(0)) revert NullAddress();
    if (validDepositTokenAddresses[_newTokenAddress] != address(0)) {
        revert AlreadyExist();
    }
    validDepositTokenAddresses[_newTokenAddress] = _wrappedTokenAddress;
}
```

- **Recommendation:** Integrate Aave's `ProtocolDataProvider` to validate token support before whitelisting:

```
import {IProtocolDataProvider} from
 ↪  "@aave/core-v3/contracts/interfaces/IProtocolDataProvider.sol";

IProtocolDataProvider public immutable protocolDataProvider;

constructor(
    address _aaveSC,
    address _ngoImplementationAddress,
    address _owner,
    address _dataProvider
) Ownable(_owner) {
    if (_dataProvider == address(0)) revert NullAddress();
    protocolDataProvider = IProtocolDataProvider(_dataProvider);
    aaveSC = _aaveSC;
    ngoImplementationAddress = _ngoImplementationAddress;
}

function addValidDepositToken(
    address _newTokenAddress,
    address _wrappedTokenAddress
) public onlyOwner {
    if (_newTokenAddress == address(0)) revert NullAddress();
    if (_wrappedTokenAddress == address(0)) revert NullAddress();

    // Step 1: Validate aToken mapping
    (address aTokenAddress, , ) =
 ↪  protocolDataProvider.getReserveTokensAddresses(_newTokenAddress);
    if (aTokenAddress != _wrappedTokenAddress) {
        revert InvalidATokenMapping();
    }

    if (validDepositTokenAddresses[_newTokenAddress] != address(0)) {
```

```
            revert AlreadyExist();
        }

        // Step 2: Get reserve configuration data
        (
            uint256 decimals,
            uint256 ltv,
            uint256 liquidationThreshold,
            uint256 liquidationBonus,
            uint256 reserveFactor,
            bool usageAsCollateralEnabled,
            bool borrowingEnabled,
            bool stableBorrowRateEnabled,
            bool isActive,
            bool isFrozen
        ) = protocolDataProvider.getReserveConfigurationData(_newTokenAddress);

        // Step 3: Validate reserve is active
        if (!isActive) {
            revert ReserveNotActive();
        }

        // Step 4: Validate reserve is not frozen
        if (isFrozen) {
            revert ReserveFrozen();
        }

        validDepositTokenAddresses[_newTokenAddress] = _wrappedTokenAddress;
        emit ValidDepositTokenAdded(_newTokenAddress, _wrappedTokenAddress);
    }
```

- **Client Response:** Added integration with `poolDataProvider` to validate supported tokens before adding them to the whitelist.

[M-2] Zero NGO Assets Due to Missing Decimal-Aware Minimum Deposit Validation

- **Status:** Resolved
- **Initial Commit:** `616cf776b252171ae36b1e9bfe0e0e8a7712adbe`
- **Fix Commit:** `fa3b273e07be0ca4538f3d8fb8ba6b7e42c7e853`
- **Location:** `AaveImpact.sol:assetsCalculation()`, line ~467
- **Impact:** When users deposit with minimum percentage (100 = 1%) and small amounts, the NGO assets calculation can result in zero, leading to division by zero in subsequent calculations and incorrect share distribution. The lack of decimal-aware minimum deposit validation allows this scenario across tokens with different decimal precision.
- **Description:** The contract lacks minimum deposit validation that accounts for token decimals. This allows users to deposit amounts that, when multiplied by the minimum NGO percentage, produce zero NGO assets due to integer division truncation. The problem varies by token decimals:

Example scenarios:

- **GUSD (2 decimals):** User deposits 50 units (0.50 GUSD) at 1% → `_ngoAssets = 50 * 100 / 10000 = 0` → **Division by zero**
- **USDC (6 decimals):** User deposits 5000 units (0.005 USDC) at 1% → `_ngoAssets = 5000 * 100 / 10000 = 50` → Works but very small
- **DAI (18 decimals):** User deposits 50 units (0.00000000000000005 DAI) at 1% → `_ngoAssets = 50 * 100 / 10000 = 0` → **Division by zero**

```
function assetsCalculation(uint256 _amount, uint16 _percent, address
 ↪  _tokenAddress) private {
    uint256 _ngoAssets = _amount.mulDiv(_percent, PERCENT_DIVIDER);
    // @audit Can be 0 for small amounts: 50 * 100 / 10000 = 0

    _assets[msg.sender][_tokenAddress][_depositId] = _amount - _ngoAssets;

    if (_totalShares[_tokenAddress] == 0) {
        _ngoShare = _ngoAssets;
        _userShares[msg.sender][_tokenAddress][_depositId] =
 ↪  _assets[msg.sender][_tokenAddress][_depositId]
            .mulDiv(_ngoShare, _ngoAssets); // @audit Division by zero if
             ↪  _ngoAssets = 0
    }
    // ...
}
```

- **Recommendation:** Implement decimal-aware minimum deposit validation that ensures non-zero NGO assets for all token types:

```
import {IERC20Metadata} from
 ↪  "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol";

// Add custom error
error InvalidDepositAmount();
error NGOAssetsZero();

/**
 * @notice Calculate minimum deposit amount based on token decimals
 * @dev Ensures at least 1.00 tokens deposit, guaranteeing min 0.01 tokens
 ↪  for NGO at 1%
 * @param _tokenAddress Address of the deposit token
 * @return Minimum deposit amount in token's smallest unit
 */
function getMinDepositAmount(address _tokenAddress) internal view returns
 ↪  (uint256) {
    uint256 decimals = IERC20Metadata(_tokenAddress).decimals();

    // Minimum 1.00 tokens for all token types
    // This ensures: 1.00 * 1% = 0.01 tokens minimum for NGO (non-zero)
    return 10 ** decimals; // 1.00 tokens scaled to decimals
}

function deposit(
    address _tokenAddress,
```

```
    uint256 _tokenAmount,
    uint16 _ngoPercent
)
    public
    notFinished
    validDeposit(_ngoPercent)
    onlyExistingToken(_tokenAddress)
    notBanned
{
    // Validate minimum deposit amount based on token decimals
    uint256 minDeposit = getMinDepositAmount(_tokenAddress);
    if (_tokenAmount < minDeposit) {
        revert InvalidDepositAmount();
    }

    address wrappedTokenAddress =
↪   factory.validDepositTokenAddresses(_tokenAddress);

    IERC20(_tokenAddress).safeTransferFrom(msg.sender, address(this),
     ↪  _tokenAmount);
    IERC20(_tokenAddress).forceApprove(address(aavePool), _tokenAmount);

    _balanceBeforeDeposit[_tokenAddress] =
↪   IERC20(wrappedTokenAddress).balanceOf(address(this));

    emit Deposit(
        _depositId,
        msg.sender,
        _tokenAmount,
        _ngoPercent,
        address(this),
        block.timestamp,
        block.number,
        _tokenAddress
    );

    aavePool.supply(_tokenAddress, _tokenAmount, address(this), 0);
    uint256 _balanceAfter =
↪   IERC20(wrappedTokenAddress).balanceOf(address(this));
    uint256 _balanceInWrappedTokens = _balanceAfter -
↪   _balanceBeforeDeposit[_tokenAddress];

    assetsCalculation(_balanceInWrappedTokens, _ngoPercent, _tokenAddress);
    _depositId++;
}
```

- **Client Response:** Implemented new internal function `getMinDepositAmount` to validate tokens with all decimals.

[M-3] Hardcoded WITHDRAW_GAP Incompatible with Low-Decimal Tokens

- **Status:** Resolved
- **Initial Commit:** `e7437fd63fa2c13866e7162a99fe5efb8844fb93`
- **Fix Commit:** `fa3b273e07be0ca4538f3d8fb8ba6b7e42c7e853`
- **Location:** `AaveImpact.sol:withdraw()`, lines 347-349
- **Impact:** The hardcoded `WITHDRAW_GAP = 100` assumes 18-decimal tokens but causes incorrect behavior for tokens with fewer decimals, potentially forcing users to withdraw their entire balance when they intended a partial withdrawal.
- **Description:** For tokens with low decimals (e.g., GUSD with 2 decimals), the gap of 100 units represents 1.00 tokens, which may be significantly higher than intended for dust prevention.

```
uint8 constant WITHDRAW_GAP = 100;

function withdraw(uint256 _amount, uint256 _id, address _tokenAddress)
 ↪  public {
    uint256 _userBalance = getUserBalance(msg.sender, _id, _tokenAddress);

    if (_userBalance - _amount < WITHDRAW_GAP) {
        _amount = _userBalance; // Forces full withdrawal
    }
    // ...
}
```

  Example: Token with 2 decimals, user balance 12.00 tokens (1200), wants to withdraw 11.50 (1150):

  - Remaining: 1200 - 1150 = 50 < 100
  - Result: Forces withdrawal of entire 12.00 tokens instead of leaving 0.50

- **Recommendation:** Implement dynamic gap calculation based on token decimals:

```
import {IERC20Metadata} from
 ↪  "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol";

function getWithdrawGap(address _tokenAddress) internal view returns
 ↪  (uint256) {
    uint256 decimals = IERC20Metadata(_tokenAddress).decimals();

    // Set gap to approximately 0.01 tokens for all token types
    if (decimals <= 2) {
        return 1; // 0.01 tokens for 2 decimals
    } else {
        return 10 ** (decimals - 2); // 0.01 tokens scaled to decimals
    }
}

function withdraw(
    uint256 _amount,
    uint256 _id,
    address _tokenAddress
) public validWithdrawalAmount(_amount) onlyExistingToken(_tokenAddress) {
    uint256 _userBalance = getUserBalance(msg.sender, _id, _tokenAddress);
```

```
    if (_amount > _userBalance) {
        revert RequestAmountTooLarge(_amount);
    }

    uint256 gap = getWithdrawGap(_tokenAddress);
    if (_userBalance - _amount < gap) {
        _amount = _userBalance;
    }

    withdrawCalculation(_amount, _id, _userBalance, _tokenAddress);

    emit Withdraw(
        msg.sender,
        address(this),
        _amount,
        block.timestamp,
        block.number,
        _id,
        _tokenAddress
    );

    uint256 withdrawnAmount = aavePool.withdraw(_tokenAddress, _amount,
↪   address(this));

    if (withdrawnAmount != _amount) {
        revert AaveWithdrawnAmountMismatch(_amount, withdrawnAmount);
    }

    IERC20(_tokenAddress).safeTransfer(msg.sender, _amount);
}
```

- **Client Response:** Implemented new internal function `getWithdrawGap`. Removed unnecessary constant `WITHDRAW_GAP`.

[M-4] Hardcoded MIN_WITHDRAWAL_AMOUNT Blocks Legitimate Withdrawals

- **Status:** Resolved
- **Initial Commit:** `e7437fd63fa2c13866e7162a99fe5efb8844fb93`
- **Fix Commit:** `fa3b273e07be0ca4538f3d8fb8ba6b7e42c7e853`
- **Location:** `AaveImpact.sol:validWithdrawalAmount` modifier
- **Impact:** The hardcoded `MIN_WITHDRAWAL_AMOUNT = 100` blocks legitimate withdrawals for low-decimal tokens and can permanently lock user funds if their remaining balance falls below this threshold.
- **Description:** For tokens with 2 decimals (e.g., GUSD), MIN_WITHDRAWAL_AMOUNT = 100 represents 1.00 tokens, blocking all withdrawals under $1 and potentially locking small remaining balances.

```
uint8 constant MIN_WITHDRAWAL_AMOUNT = 100;

modifier validWithdrawalAmount(uint256 _amount) {
    if (_amount < MIN_WITHDRAWAL_AMOUNT) {
        revert InvalidWithdrawalAmount();
    }
    _;
}
```

Critical Example: User has 50 units (0.50 tokens with 2 decimals) remaining. They cannot withdraw because 50 < 100, permanently locking their funds.

- **Recommendation:** Implement dynamic minimum withdrawal based on token decimals and allow full balance withdrawals:

```
import {IERC20Metadata} from
 ↪  "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol";

function getMinWithdrawalAmount(address _tokenAddress) internal view returns
 ↪  (uint256) {
    uint256 decimals = IERC20Metadata(_tokenAddress).decimals();

    // Set minimum to 0.01 tokens for all token types
    if (decimals == 0) {
        return 1; // 1 whole token for tokens without decimals
    } else if (decimals == 1) {
        return 1; // 0.1 tokens
    } else {
        return 10 ** (decimals - 2); // 0.01 tokens
    }
}

function withdraw(
    uint256 _amount,
    uint256 _id,
    address _tokenAddress
) public onlyExistingToken(_tokenAddress) {
    uint256 _userBalance = getUserBalance(msg.sender, _id, _tokenAddress);

    if (_amount > _userBalance) {
        revert RequestAmountTooLarge(_amount);
    }

    // Allow full balance withdrawal OR minimum amount
    uint256 minAmount = getMinWithdrawalAmount(_tokenAddress);
    if (_amount != _userBalance && _amount < minAmount) {
        revert InvalidWithdrawalAmount();
    }

    uint256 gap = getWithdrawGap(_tokenAddress);
    if (_userBalance - _amount < gap) {
        _amount = _userBalance;
    }
```

```
    withdrawCalculation(_amount, _id, _userBalance, _tokenAddress);

    emit Withdraw(
        msg.sender,
        address(this),
        _amount,
        block.timestamp,
        block.number,
        _id,
        _tokenAddress
    );

    uint256 withdrawnAmount = aavePool.withdraw(_tokenAddress, _amount,
↪   address(this));

    if (withdrawnAmount != _amount) {
        revert AaveWithdrawnAmountMismatch(_amount, withdrawnAmount);
    }

    IERC20(_tokenAddress).safeTransfer(msg.sender, _amount);
}
```

- **Client Response:** Implemented new internal function `getMinWithdrawAmount`. Removed unnecessary constant `MIN_WITHDRAWAL_AMOUNT`.

Low

[L-1] Single-Step Ownership Transfer Risk

- **Status:** Resolved
- **Initial Commit:** `e7437fd63fa2c13866e7162a99fe5efb8844fb93`
- **Fix Commit:** `fa3b273e07be0ca4538f3d8fb8ba6b7e42c7e853`
- **Location:** `AaveImpact.sol`, `AaveImpactFactory.sol`
- **Impact:** Using `OwnableUpgradeable` instead of `Ownable2StepUpgradeable` creates risk of accidentally transferring ownership to an incorrect address, permanently losing admin control.
- **Description:** Single-step ownership transfer allows immediate transfer without confirmation from the new owner, increasing risk of typos or malicious input.
- **Recommendation:** Upgrade to two-step ownership transfer pattern:

```
// In AaveImpact.sol
import {Ownable2StepUpgradeable} from "@openzeppelin/contracts-
↪   upgradeable/access/Ownable2StepUpgradeable.sol";

contract AaveImpact is
    Initializable,
    ReentrancyGuardUpgradeable,
    UUPSUpgradeable,
```

```
    Ownable2StepUpgradeable
{
    function initialize(
        address _aavePoolAddress,
        address _rewardOwnerAddress,
        address _owner,
        address _oracle,
        address _factoryAddress
    ) public initializer {
        __UUPSUpgradeable_init();
        __Ownable2Step_init();
        __ReentrancyGuard_init();
        _transferOwnership(_owner);

        // ... rest of initialization
    }
}

// In AaveImpactFactory.sol
import {Ownable2Step} from
 ↪  "@openzeppelin/contracts/access/Ownable2Step.sol";

contract AaveImpactFactory is Ownable2Step {
    constructor(
        address _aaveSC,
        address _ngoImplementationAddress,
        address _owner
    ) Ownable(_owner) {
        // ... initialization
    }
}
```

- **Client Response:** Added integration with `Ownable2Step` on `AaveImpact` and `AaveImpactFactory`.

Informational

[I-1] Redundant Storage Reads

- **Status:** Resolved
- **Fix Commit:** `fa3b273e07be0ca4538f3d8fb8ba6b7e42c7e853`
- **Location:** `AaveImpact.sol#637` `AaveImpact.sol#760`
- **Impact:** Repeated storage reads of the same unmodified state variable unnecessarily increase gas consumption.
- **Description:** The following functions read storage values multiple times:
  - `handleNGOShareDistribution` → `_pendingNGORewards[_tokenAddress]`
  - `assetsCalculation` → `_depositId`

- **Recommendation:** Cache the storage variable in a local (memory) variable at the beginning of the function to eliminate redundant SLOAD operations:

```
function handleNGOShareDistribution(address _tokenAddress) public onlyOracle
 ↪  {
    uint256 cachedPendingReward = _pendingNGORewards[_tokenAddress];
    // ... use cachedPendingReward
}

function assetsCalculation(
    uint256 _amount,
    uint16 _percent,
    address _tokenAddress
) private {
    uint256 depositId = _depositId;
    // ... use depositId
}
```

- **Client Response:** Added new variables to `handleNGOShareDistribution` and `assetsCalculation`.

[I-2] Unused Custom Error Declarations

- **Status:** Resolved
- **Fix Commit:** `fa3b273e07be0ca4538f3d8fb8ba6b7e42c7e853`
- **Location:** AaveImpact.sol#174 AaveImpact.sol#184 AaveImpact.sol#189 AaveImpact.sol#199 AaveImpact.sol#214
- **Impact:** Unused custom error declarations contribute unnecessary bytecode to the contract, increasing deployment gas cost, overall contract size, and codebase complexity.
- **Description:** The contract declares some custom errors, but these are never used in the contract.
- **Recommendation:** Remove all unused custom error declarations to reduce contract size and deployment cost:

```
// Remove these if not used:
// error RequestAmountTooSmall(uint256 _amount);
// error InvalidRequestIdForUser(address _claimer, uint256 _requestId);
// error NotFinalizedStatus();
// error TokenNotAllowed();
// error ZeroAmount();
```

- **Client Response:** Removed unused errors: `RequestAmountTooSmall`, `InvalidRequestIdForUser`, `NotFinalizedStatus`, `TokenNotAllowed()`, `ZeroAmount()`.

[I-3] Unnecessary Inheritance of `ERC721HolderUpgradeable` and Unused `ReentrancyGuardUpgradeable`

- **Status:** Resolved

- **Fix Commit:** `fa3b273e07be0ca4538f3d8fb8ba6b7e42c7e853`
- **Location:** `AaveImpact.sol:22`, `AaveImpact.sol:23`
- **Impact:** The inclusion of `ERC721HolderUpgradeable` and `ReentrancyGuardUpgradeable` adds unnecessary bytecode and increases deployment gas cost. Additionally, `ReentrancyGuardUpgradeable` is initialized but the `nonReentrant` modifier is never used in any function, providing no actual protection while consuming gas.
- **Description:** The contract inherits from two upgradeable utilities that serve no purpose:
  1. **`ERC721HolderUpgradeable`**: Implements `onERC721Received` to safely receive ERC721 tokens via `safeTransferFrom`, but the contract never receives, stores, or interacts with NFTs.
  2. **`ReentrancyGuardUpgradeable`**: Initialized in the `initialize()` function via `__ReentrancyGuard_init()`, but the `nonReentrant` modifier is not applied to any external/public functions in the contract.

```
// Current inheritance
contract AaveImpact is
    Initializable,
    ReentrancyGuardUpgradeable,  // @audit Initialized but never used
    UUPSUpgradeable,
    OwnableUpgradeable,
    ERC721HolderUpgradeable  // @audit Not needed - no NFT functionality
{
    // ...
}

function initialize(
    address _aavePoolAddress,
    address _rewardOwnerAddress,
    address _owner,
    address _oracle,
    address _factoryAddress
) public initializer {
    __ERC721Holder_init();  // @audit Unnecessary initialization
    __UUPSUpgradeable_init();
    __Ownable_init(_owner);
    __ReentrancyGuard_init();  // @audit Initialized but nonReentrant
     ↪  modifier never used
    // ...
}
```

- **Recommendation:** Apply the `nonReentrant` modifier to sensitive functions and remove `ERC721HolderUpgradeab`

```
// Remove import:
// import {ERC721HolderUpgradeable} from "@openzeppelin/contracts-
 ↪  upgradeable/token/ERC721/utils/ERC721HolderUpgradeable.sol";

contract AaveImpact is
    Initializable,
    ReentrancyGuardUpgradeable,
    UUPSUpgradeable,
    OwnableUpgradeable
{
```

```
    // ... contract code
}

function initialize(
    address _aavePoolAddress,
    address _rewardOwnerAddress,
    address _owner,
    address _oracle,
    address _factoryAddress
) public initializer {
    // Remove: __ERC721Holder_init();

    __UUPSUpgradeable_init();
    __Ownable_init(_owner);
    __ReentrancyGuard_init();

    // ... rest of initialization
}

// Add nonReentrant to sensitive functions
function deposit(
    address _tokenAddress,
    uint256 _tokenAmount,
    uint16 _ngoPercent
)
    public
    nonReentrant
    notFinished
    validDeposit(_ngoPercent)
    onlyExistingToken(_tokenAddress)
    notBanned
{
    // ... function body
}

function withdraw(
    uint256 _amount,
    uint256 _id,
    address _tokenAddress
)
    public
    nonReentrant
    onlyExistingToken(_tokenAddress)
{
    // ... function body
}

function handleNGOShareDistribution(address _tokenAddress)
    public
    nonReentrant
    onlyOracle
{
    // ... function body
```

```
}
```

- **Client Response:** Removed `ERC721HolderUpgradeable` from `AaveImpact`. Added `nonReentrant` modifier to sensitive functions.

[I-4] Inefficient Struct Storage Layout (Unpacked Variables)

- **Status:** Resolved
- **Fix Commit:** `fa3b273e07be0ca4538f3d8fb8ba6b7e42c7e853`
- **Location:** `AaveImpact.sol#33`
- **Impact:** Inefficient packing increases storage slot usage, leading to higher gas costs and cumulative gas waste.
- **Description:** The contract defines a `UserDepositInfo` struct that stores multiple variables, but the variables are not packed efficiently according to EVM storage rules.
- **Recommendation:** Reorder struct fields from largest to smallest and group small types together to maximize packing:

```
struct UserDepositInfo {
    uint256 amount;
    uint256 startDate;
    address tokenAddress;
    uint16 percent;
}
```

- **Client Response:** Reordered struct fields.

[I-5] Duplicate Error Definitions

- **Status:** Resolved
- **Fix Commit:** `fa3b273e07be0ca4538f3d8fb8ba6b7e42c7e853`
- **Location:** `AaveImpact.sol#167`, `AaveImpact.sol#172`
- **Impact:** Two similar error definitions cause code confusion and reduced maintainability.
- **Description:** The contract defines two errors that appear to serve the same purpose:
  - `InvalidWithdrawalAmount()` - Used in `validWithdrawalAmount` modifier
  - `InvalidWithdrawAmount()` - Used in `withdrawCalculation()` when ratio is zero
- **Recommendation:** Consolidate into a single, more descriptive error:

```
// Remove:
// error InvalidWithdrawalAmount();
// error InvalidWithdrawAmount();
```

```
// Replace with:
error InvalidWithdrawalAmount(string reason);
```

- **Client Response:** Added one error `InvalidWithdrawalAmount(string reason)`.

Severity Levels

Each issue is categorized according to its severity. Here's how each severity level is defined:

- **High**: Critical vulnerabilities that may result in significant loss of funds, unauthorized control, or failure of core zapping and wallet functionality.
- **Medium**: Issues that could potentially cause financial loss or disrupt functionality but are less immediate or impactful than high-severity issues.
- **Low**: Minor issues that have no immediate risk but may affect user experience or the maintainability of the code.
- **Informational**: Non-critical issues such as minor optimizations or suggestions for best practices.

Disclaimer

This audit is not a guarantee of the absence of vulnerabilities or bugs. Security audits are time-boxed and can only identify issues present at the time of the audit. Audits complement other security measures but cannot replace comprehensive testing, code reviews, and ongoing security monitoring. We recommend that multiple audits be conducted, and a bug bounty program be established for continued testing. This report does not provide any investment advice, nor does it guarantee the performance or compliance of the audited project.

Recommendations

Based on our audit of the Stablecoin For Impact smart contracts, we provide the following general recommendations to enhance the security and maintainability of the project:

1. **Regular Independent Audits** While this audit covers the core components of the codebase, we strongly recommend conducting additional independent audits at key stages of the project's lifecycle, especially before adding support for new token types or making significant protocol changes.
2. **Continuous Monitoring and Bug Bounty Program** We recommend the implementation of a public bug bounty program to encourage third-party security researchers to identify and report vulnerabilities in deposit operations, withdrawal mechanisms, or yield distribution logic. Ongoing monitoring of smart contract behavior in production is also essential.
3. **Comprehensive Unit Testing** Expanding test coverage, particularly for edge cases with different token decimals (2-18 decimals), minimum deposit/withdrawal amounts, and share calculations with small values, can help identify potential vulnerabilities. Ensure that tests are continuously integrated into the development process.
4. **Enhanced Access Control Management** Carefully review and limit the privileges of administrator roles, particularly those managing the token whitelist and oracle functions. The implemented two-step ownership transfer pattern significantly improves security in this area.

5. **Token Compatibility Testing** Regularly test the protocol with various stablecoins having different decimal configurations (GUSD with 2 decimals, USDC with 6 decimals, DAI with 18 decimals) to ensure the dynamic minimum deposit and withdrawal calculations work correctly across all supported tokens.
6. **Aave Protocol Integration Monitoring** Stay informed of changes to the Aave V3 protocol that could affect token support, reserve status, or yield generation. The implemented Aave token validation significantly improves safety in this area.
7. **User Education** Provide clear documentation about minimum deposit amounts for different tokens, withdrawal mechanisms, and the share-based yield distribution system to ensure users understand how their funds are managed and how yields are calculated.

About Null Return

Null Return is a Web3 security firm specializing in smart contract audits, formal verification, architecture consulting, and pre-deployment reviews. We work closely with builders to uncover and help resolve vulnerabilities before they become problems — combining deep technical knowledge with a hands-on, product-aware approach. We take pride in being fast, transparent, and highly responsive throughout the audit process.

Contact us:

- Website: nullreturn.io
- Telegram: @nullreturn_io
- Twitter: @nullreturn_io
- LinkedIn: Null Return