22.01.2025

Audit

</>

Aave DIVA Wrapper

By 0xKoiner

Who is 0xKoiner

Hi, I'm **0xKoiner**, a developer and smart contract auditor based in sunny Barcelona. My journey in the crypto industry started back in 2016, and it's been an exciting ride ever since. With a background in Computer Science from the Technion (Israel Institute of Technology), I've built up a versatile skill set that spans Python development and smart contract security.

My Experience
Python Development
I have a solid background in Python, focusing on backend development, software automation, testing, and Web3 libraries. I enjoy building tools and solutions that help simplify complex problems, and I'm always looking for new ways to streamline workflows and processes.

Solidity & Smart Contract Development
I'm a junior Solidity developer with a growing passion for low-level aspects of Ethereum development, including OpCodes, Huff, and Assembly. I love diving into the details of smart contract design, aiming to write clean, efficient, and secure code. I'm still learning every day, but I enjoy the challenge of constantly improving.

Auditing & Security
I've completed several courses focused on smart contract audits and enjoy the process of analyzing code for potential vulnerabilities. My toolkit includes:

Foundry & HardHat: For testing and development.
Slither & Aderyn: For static analysis and finding common issues.
Certora: For formal verification to ensure contract safety.
I take a careful, methodical approach when auditing, trying to catch even the smallest issues that could become big problems. I believe every audit is a chance to learn and help make the Web3 space a bit safer.

My Approach
I'm motivated by a genuine interest in blockchain technology and its potential to bring about positive change. I'm not an expert in everything, but I'm always eager to learn and collaborate with others. If you're working on an interesting project or need a fresh set of eyes for an audit, I'd love to connect and see how I can help.

Let's build something great together!

Table of contents

Disclaimer

The OxKoiner team has made every effort to identify potential vulnerabilities within the time allocated for this audit. However, we do not assume responsibility for the findings or any issues that may arise after the audit. This security audit is not an endorsement of the project's business model, product, or team. The audit was time-limited and focused exclusively on assessing the security of the Solidity code implementation. It is strongly recommended that additional testing and security measures be conducted by the project team.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

Scope

```
All Contracts in `src` are in scope.

├── src
│   ├── AaveDIVAWrapper.sol
│   ├── AaveDIVAWrapperCore.sol
│   ├── WToken.sol
│   ├── interfaces
│   │   └── IAave.sol
│   │   └── IAaveDIVAWrapper.sol
│   │   └── IDIVA.sol
│   │   └── IWToken.sol
```

Protocol Summary

AaveDIVAWrapper is a smart contract that acts as a connector between DIVA Protocol and Aave V3, allowing assets deposited into DIVA Protocol pools to generate yield by supplying them on Aave V3.

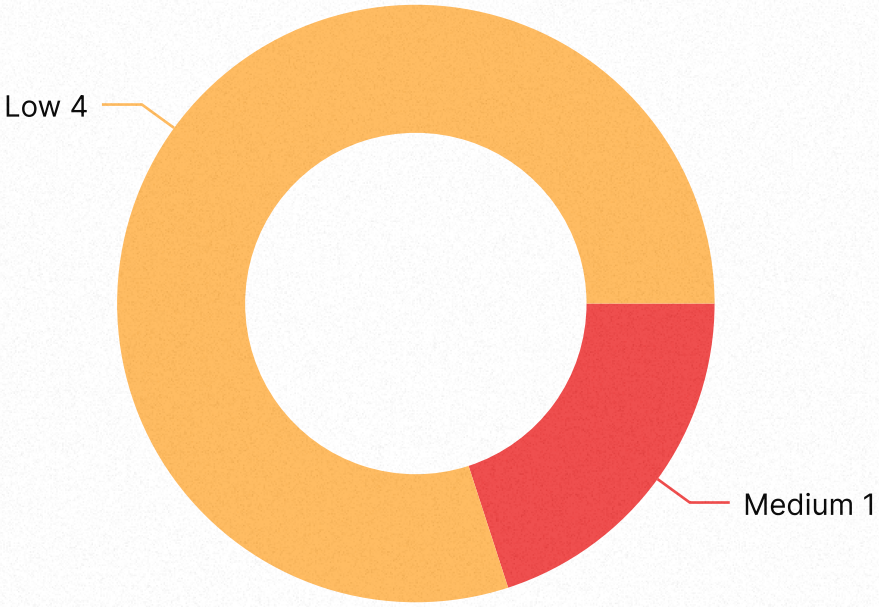
Roles

- **Buyer:** The purchaser of the nft fraction.
- **Seller:** The seller of the nft fraction

Compatibilities

- **Blockchains:**
  - Ethereum Mainnet
  - Arbitrum One
  - Gnosis Chain
  - Ethereum Sepolia (Testnet)
- **Tokens:**
  - Any ERC20 token supported by Aave V3, but mainly stablecoins like USDC, USDT are expected to be used for DIVA Donate.
  - ee-on-transfer and rebaseable tokens are NOT supported.

Report Findings



Medium Risk Findings

M-01. Inconsistent Owner Initialization Between `AaveDIVAWrapperCore.sol` and `WToken.sol`

Low Risk Findings

L-01. Unused Event PoolIssued in IDIVA.sol

L-02. Unused Event LiquidityAdded in IDIVA.sol

L-03. Bad Practice in Naming and Symbol Assignment for `WToken` in `AaveDIVAWrapperCore.sol`

L-04. `_owner` Variable Should Be Immutable in `WToken.sol`

[M-01]Medium Risk Findings

M-01. Inconsistent Owner Initialization Between `AaveDIVAWrapperCore.sol` and `WToken.sol`

Summary

The contracts AaveDIVAWrapperCore.sol and WToken.sol handle owner initialization inconsistently.
While AaveDIVAWrapperCore uses OpenZeppelin's Ownable2Step contract to initialize the owner, WToken manually sets an _owner variable in its constructor. This inconsistency creates confusion, increases the likelihood of errors, and leads to ambiguity regarding the ownership structure. Additionally, WToken.sol does not align with the OpenZeppelin ownership standard, which could hinder maintainability and interoperability.

Vulnerability Details

Type: Mismanagement of Ownership Initialization and Standard Deviation

Issue 1: Inconsistent Owner Initialization

Location:

- AaveDIVAWrapperCore.sol initializes ownership using OpenZeppelin's Ownable2Step:

```
@> constructor(address diva_, address aaveV3Pool_, address owner_)
  Ownable(owner_) {
```

- WToken.sol manually sets _owner in its constructor:

```
_owner = owner_;
```

Problem:

- The owner of WToken is the AaveDIVAWrapperCore contract itself:

```
WToken _wTokenContract = new WToken(..., address(this));
// wToken owner is the wrapper contract
```

However, the AaveDIVAWrapperCore relies on OpenZeppelin's ownership mechanisms, while WToken uses a custom _owner variable. This divergence creates inconsistency:

- Ownership logic across the system is unclear and difficult to track.
- Developers and auditors cannot rely on a uniform pattern.
- Future maintainers might misinterpret the ownership logic.

Issue 2: Non-Standard Ownership in WToken

The following tools and techniques were used to identify and analyze the issue:

- Static Code Review: Manual inspection of the _owner variable and related functions.
- Solidity Compiler (v0.8.26): Verified the impact of making _owner immutable.

Location: WToken.sol:

```
address private _owner; // Custom ownership logic
```

Problem:

- The _owner variable does not use OpenZeppelin's Ownable or Ownable2Step contracts, which are industry standards.
- This custom ownership logic lacks key OpenZeppelin features, such as ownership transfer mechanisms (transferOwnership or transferOwnershipWithTwoSteps).
- The lack of an ownership transfer function makes ownership immutable, which is a design decision that should be explicitly documented.

Impact

1. Functionality Impact:
- No direct security risk is posed, but the inconsistency can lead to developer confusion and implementation errors when extending the system.
2. Maintainability Impact:
- Custom ownership in WToken deviates from OpenZeppelin standards, reducing clarity and interoperability.
3. Risk of Mismanagement:
- If ownership behavior diverges between the contracts, it could lead to unexpected governance or operational issues in the future.

Tools Used

The following tools and techniques were used to identify and analyze the issue:

- Static Code Review: Manual inspection of ownership-related logic.
- Solidity Compiler (v0.8.26): Verified owner initialization and inheritance mechanisms.
- OpenZeppelin Documentation: Compared ownership patterns against established best practices.

Recommendations

Recommendation 1: Use OpenZeppelin's Ownership Standard for WToken

- Replace the custom _owner variable in WToken with OpenZeppelin's Ownable contract:

```
contract WToken is IWToken, ERC20, Ownable {
    uint8 private _decimals;

    constructor(string memory symbol_, uint8 decimals_, address owner_)
      ERC20(symbol_, symbol_) {
        _transferOwnership(owner_); // Use OpenZeppelin's ownership mechanism
        _decimals = decimals_;
    }
}
```

This ensures consistency in ownership management across all contracts.

Recommendations

Recommendation 2: Clearly Document Ownership Relationships

- Clarify that:
- The AaveDIVAWrapperCore contract acts as the owner of all WToken instances.
- Ownership of WToken is immutable after deployment.
- Include this information in the documentation or inline comments for future reference.

Recommendations

Recommendation 3: Consider Making Ownership Immutable in WToken

- If ownership of WToken is meant to be immutable, declare the owner as immutable:

```
address private immutable _owner;

constructor(string memory symbol_, uint8 decimals_, address owner_)
  ERC20(symbol_, symbol_) {
    _owner = owner_; // Immutable initialization
    _decimals = decimals_;
}

modifier onlyOwner() {
    require(msg.sender == _owner, "WToken: caller is not the owner");
    _;
}
```

Document the decision to use immutable ownership for transparency.

[L-01]Low Risk Findings

L-01. Unused Event PoolIssued in IDIVA.sol

Summary

The LiquidityAdded event is declared in the IDIVA.sol contract but is never used or emitted within the codebase. This creates unnecessary clutter in the contract and increases the risk of confusion during integration or debugging. Such unused elements contribute to technical debt and should be addressed for better maintainability and clarity.

Vulnerability Details

Type: Redundancy and Unused Code

- Location: IDIVA.sol, Line (exact line depends on your file version):

```
@> event LiquidityAdded(
      bytes32 indexed poolId,
      address indexed longRecipient,
      address indexed shortRecipient,
      uint256 collateralAmount
   );
```

- Issue:
  - The LiquidityAdded event is defined but not emitted or referenced anywhere in the contract.
  - It appears to be a placeholder or leftover from earlier development iterations but serves no active purpose in the current codebase.
- Problem:
  - Including unused events adds unnecessary complexity to the contract.
  - Misleads developers, auditors, or integrators into thinking this event has active relevance.
  - May slightly increase the compiled bytecode size, resulting in minor inefficiencies during deployment.

Impact

This issue has a Low severity, as it does not affect the functionality, security, or correctness of the contract. However, it introduces unnecessary complexity and should be resolved to ensure a clean and maintainable codebase.

Tools used

The following tools were utilized to identify and analyze the issue:

- Static Code Review: Manual inspection of the code.
- Solidity Compiler (v0.8.26): Verified the event declaration and potential usage in the bytecode.

Recommended

1. Remove the unused LiquidityAdded event:

```
// Remove this event declaration from `IDIVA.sol`
- event LiquidityAdded(
-     bytes32 indexed poolId,
-     address indexed longRecipient,
-     address indexed shortRecipient,
-     uint256 collateralAmount
- );
```

2. Audit Related Contracts: Ensure that other contracts in the system do not rely on this event. If this event was planned for future use, document its purpose clearly to avoid confusion.
3. Maintain Consistency: If a similar event is defined and used elsewhere, ensure a single, consistent declaration across the codebase.
4. Document Changes: Record the removal of this event and the reasons behind it in the repository's change log or documentation.

Recommended mitigation

You could implement a check to refund in case of overpayment:

```
+ uint256 totalRequiredAmount = order.price + sellerFee;
+ uint256 excessAmount = msg.value - totalRequiredAmount ;
+ if (excessAmount > 0) {
+   payable(msg.sender).transfer(excessAmount);
    // Refund excess Ether to the buyer
+ }
```

Alternatively, you can revert the transaction if an overpayment is detected:

```
+ error TokenDivider__ExcessivePaymentDetected();

+ if (msg.value > totalRequiredAmount) {
+   revert TokenDivider__ExcessivePaymentDetected();
+ }
```

The first solution (refund mechanism) ensures the user is refunded any excess funds, preventing financial loss. The second solution (revert mechanism) prevents overpayment from being accepted in the first place, ensuring that users cannot accidentally overpay.

[L-02]Low Risk Findings

L-02. Unused Event LiquidityAdded in IDIVA.sol

Summary

The IDIVA.sol contract includes an event declaration for PoolIssued that is unused throughout the codebase. Instead of using the declared event, another PoolIssued event is declared in the IAaveDIVAWrapper.sol contract, which creates redundancy and increases the risk of confusion or inconsistencies during integration or debugging.

Vulnerability Details

Type: Redundancy and Unused Code

- Location: IDIVA.sol, Line (exact line will depend on the file version provided):

```
@> event PoolIssued(
     bytes32 indexed poolId,
     address indexed longRecipient,
     address indexed shortRecipient,
     uint256 collateralAmount,
     address permissionedERC721Token
   );
```

- Issue: The PoolIssued event defined in IDIVA.sol is never emitted or used in the contract. Instead, the IAaveDIVAWrapper.sol file redefines and uses a simplified version of the PoolIssued event:

```
event PoolIssued(bytes32 indexed poolId);
```

- Problem:
  - Including unused events in the code adds unnecessary complexity.
  - It may lead to confusion for developers, auditors, or integrators, as they might assume that the PoolIssuedevent in IDIVA.sol is active and relevant.
  - It could increase gas costs slightly if compiled and deployed as part of the bytecode.

Impact

This issue has a Low severity, as it does not directly affect the functionality, security, or correctness of the contract. However, it introduces technical debt and can complicate future maintenance and debugging efforts.

Tools Used

- Static Code Review: Manual inspection of the code.
- Solidity Compiler (v0.8.26): Verified event declarations and potential usage in bytecode.

Recommendations

1. Remove the unused event declaration in IDIVA.sol:

```
// Remove this event declaration from `IDIVA.sol`
- event PoolIssued(
-     bytes32 indexed poolId,
-     address indexed longRecipient,
-     address indexed shortRecipient,
-     uint256 collateralAmount,
-     address permissionedERC721Token
- );
```

2. Standardize the event definition: Ensure that all contracts interacting with the IDIVA interface use a single, consistent declaration of the PoolIssued event. For instance, if the simpler version from IAaveDIVAWrapper.solis preferred, adopt it across the codebase:

```
```

3. Document Changes: Clearly document the reason for removing the unused event and the adoption of the standardized event in your repository or change log.
4. Audit Related Contracts: Check if other contracts in the codebase also declare similar unused or redundant events and follow the same cleanup process.

[L-03]Low Risk Findings

L-03. Bad Practice in Naming and Symbol Assignment for `WToken` in `AaveDIVAWrapperCore.sol`

Summary

The WToken constructor in AaveDIVAWrapperCore.sol initializes the token with the same name and symbol, which is considered a bad practice. Names and symbols are important identifiers for tokens and should provide clarity to users. Using identical values for both fields reduces the ability to distinguish between the token's full name and its shorthand symbol, potentially confusing users or integrators.

Vulnerability Details

Type: Poor Naming Convention / Bad Practice

- Location: AaveDIVAWrapperCore.sol, in the creation of the WToken instance:

```
WToken _wTokenContract = new WToken(
@>    string(abi.encodePacked("w", _collateralTokenContract.symbol())),
  // Symbol and Name will be the same
      _collateralTokenContract.decimals(),
      address(this) // wToken owner
  );
```

Issue:

- The constructor for WToken sets both the token's name and symbol to the same value, as seen in the WTokencontract:

```
@> constructor(string memory symbol_, uint8 decimals_, address owner_)
  ERC20(symbol_, symbol_) {
      _owner = owner_;
      _decimals = decimals_;
  }
```

- This practice reduces the clarity of the token's identity. Typically:
  - The name should be a descriptive, human-readable identifier (e.g., "Wrapped USDC").
  - The symbol should be a short, recognizable code (e.g., "wUSDC").
- Problem:
  - User Confusion: Identical names and symbols can confuse users when interacting with the token.
  - Industry Standard Violation: Most ERC-20 tokens in the ecosystem differentiate the name and symbol for clarity.
  - Missed Opportunities: The name field is valuable for describing the token's function or relationship to the collateral.

Impact

This issue has a Low severity level but affects usability and clarity. It does not directly compromise the security or functionality of the contract but goes against best practices and may create confusion for integrators and end-users.

Tools Used

The following tools and techniques were used to identify and analyze the issue:

- Static Code Review: Manual inspection of token creation logic.
- Solidity Compiler (v0.8.26): Verified constructor behavior and ERC-20 compliance.

Recommendations

1. Fetch and Use the Collateral Token Name: Instead of setting the name to match the symbol, use the collateral token's name() function to derive a more descriptive name for the WToken. For example:

```
WToken _wTokenContract = new WToken(
      string(abi.encodePacked(
        "w", _collateralTokenContract.symbol())), // Symbol
+     string(abi.encodePacked(
        "Wrapped ", _collateralTokenContract.name())), // Name
      _collateralTokenContract.decimals(),
      address(this) // Owner
  );
```

2. Update the Constructor of WToken: Modify the constructor to accept both the name and symbol as separate parameters:

```
constructor(
    string memory symbol_,
+   string memory name_,
    uint8 decimals_,
    address owner_
- ) ERC20(symbol_, symbol_)
+ ) ERC20(name_, symbol_) {
    _owner = owner_;
    _decimals = decimals_;
}
```

3. Document the Change: Clearly document the improved naming convention and the rationale behind the change for developers and auditors.
4. Test Changes Thoroughly: Ensure the changes are tested with a variety of collateral tokens to confirm compatibility.

[L-04]Low Risk Findings

L-04. `_owner` Variable Should Be Immutable in `WToken.sol`

Summary

The WToken constructor in AaveDIVAWrapperCore.sol initializes the token with the same name and symbol, which is considered a bad practice. Names and symbols are important identifiers for tokens and should provide clarity to users. Using identical values for both fields reduces the ability to distinguish between the token's full name and its shorthand symbol, potentially confusing users or integrators.

Vulnerability Details

Type: Poor Naming Convention / Bad Practice

- Location: AaveDIVAWrapperCore.sol, in the creation of the WToken instance:

```
WToken _wTokenContract = new WToken(
@>    string(abi.encodePacked("w", _collateralTokenContract.symbol())),
  // Symbol and Name will be the same
      _collateralTokenContract.decimals(),
      address(this) // wToken owner
  );
```

- Issue:
  - The constructor for WToken sets both the token's name and symbol to the same value, as seen in the WTokencontract:

```
@> constructor(string memory symbol_, uint8 decimals_, address owner_)
  ERC20(symbol_, symbol_) {
      _owner = owner_;
      _decimals = decimals_;
  }
```

- This practice reduces the clarity of the token's identity. Typically:
- The name should be a descriptive, human-readable identifier (e.g., "Wrapped USDC").
- The symbol should be a short, recognizable code (e.g., "wUSDC").
- Problem:
- User Confusion: Identical names and symbols can confuse users when interacting with the token.
- Industry Standard Violation: Most ERC-20 tokens in the ecosystem differentiate the name and symbol for clarity.
- Missed Opportunities: The name field is valuable for describing the token's function or relationship to the collateral.

Tools Used

The following tools and techniques were used to identify and analyze the issue:

- Static Code Review: Manual inspection of token creation logic.
- Solidity Compiler (v0.8.26): Verified constructor behavior and ERC-20 compliance.

Recommendations

1. Fetch and Use the Collateral Token Name: Instead of setting the name to match the symbol, use the collateral token's name() function to derive a more descriptive name for the WToken. For example:

```
WToken _wTokenContract = new WToken(
     string(abi.encodePacked(
       "w", _collateralTokenContract.symbol())), // Symbol
+    string(abi.encodePacked(
       "Wrapped ", _collateralTokenContract.name())), // Name
     _collateralTokenContract.decimals(),
     address(this) // Owner
 );
```

2. Update the Constructor of WToken: Modify the constructor to accept both the name and symbol as separate parameters:

```
constructor(
    string memory symbol_,
+   string memory name_,
    uint8 decimals_,
    address owner_
- ) ERC20(symbol_, symbol_)
+ ) ERC20(name_, symbol_) {
    _owner = owner_;
    _decimals = decimals_;
}
```

3. Document the Change: Clearly document the improved naming convention and the rationale behind the change for developers and auditors.
4. Test Changes Thoroughly: Ensure the changes are tested with a variety of collateral tokens to confirm compatibility.