Cyberscope

Audit Report

Tea-Fi

September 2024

Files ProxyTrade, SynthToken, SynthTokenFactory, TeaFiRelayer, TeaFiTrustedForwarder, Authorizable, Permitable, PermitPayable

Audited by © cyberscope

Table of Contents

Table of Contents **1**
Risk Classification **4**
Review **5**
Audit Scope 5
Audit Updates 5
Source Files 5
Overview **7**
ProxyTrade Contract 7
makePublicSwap Functionality 7
makePublicSwapWithPermit Functionality 7
makePublicSwapWithTwoPermits Functionality 7
General Functionalities 7
Authorities Functionalities 7
TeaFiRelayer Contract 8
relayCall Functionality 8
relayCallWithPermit Functionality 8
relayCallWithTwoPermits Functionality 8
General Functionalities 8
Authorities Functionalities 9
TeaFiTrustedForwarder Contract 9
execute Functionality 9
executeBatch Functionality 9
General Functionalities 9
Authorities Functionalities 9
SynthToken Contract 10
wrap Functionality 10
unwrap Functionality 10
General Functionalities 10
LiquidityBootstrapPoolFactory Contract 10
createSynthTokens Functionality 11
General Functionalities 11
Findings Breakdown **12**
Diagnostics **13**
PFV - Permit Front-running Vulnerability 15
Description 15
Recommendation 18
ACB - Amount Check Bypass 19
Description 19
Recommendation 19

ALM - Array Length Mismatch 20
Description 20
Recommendation 21
CO - Code Optimization 22
Description 22
Recommendation 24
CCR - Contract Centralization Risk 25
Description 25
Recommendation 28
DPI - Decimals Precision Inconsistency 29
Description 29
Recommendation 30
DTC - Duplicate Token Creation 32
Description 32
Recommendation 33
MC - Missing Check 34
Description 34
Recommendation 34
MEE - Missing Events Emission 35
Description 35
Recommendation 35
MTM - Missing Token-Asset Mapping 36
Description 36
Recommendation 37
MU - Modifiers Usage 38
Description 38
Recommendation 38
PTAI - Potential Transfer Amount Inconsistency 39
Description 39
Recommendation 39
PWD - Potential Wrap DOS 41
Description 41
Recommendation 41
L04 - Conformance to Solidity Naming Conventions 43
Description 43
Recommendation 43
L19 - Stable Compiler Version 43
Description 44
Recommendation 44
Functions Analysis **45**
Inheritance Graph **50**
Flow Graph **51**

Summary 52
Disclaimer 53
About Cyberscope 54

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
Critical	Highly Likely / High Impact
Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
Minor / Informative	Unlikely / Low to no Impact

Review

Audit Scope

The current contract heavily relies on the `trustedForwarder_` external contract, to perform crucial functionalities. While this dependency enables important functionality, any interactions with this external contract should be carefully reviewed and handled, as it is beyond the scope of this audit. The behavior and security of this external contract have not been assessed as part of this audit, and any interactions with it should be treated with caution to mitigate potential risks.

Audit Updates

Initial Audit	30 Sep 2024

Source Files

Filename	SHA256
TeaFiTrustedForwarder.sol	54ebec28b79279ede693eb65feee48b80401d1029299e0a76fd5c92856d1099b
TeaFiRelayer.sol	932addad1a35b1b46b550d553c48318d8dbd8fd4cc7d1e3146d1b34be43bc8a1
SynthTokenFactory.sol	97e4e5704f64eb79067a264b544746a2ca195dba06cb31e3d096e68e2ea02961
SynthToken.sol	a4f665599db4f11e1d71bc13f27cfa63466bf755f774647e40f4526e00ddd180
ProxyTrade.sol	3f7226f40a1df2c959b25a703b6461f102e0cf7a2dc5823902602346e824597d
solady/src/utils/LibString.sol	3606871f4e9afe0102bdf77afaebb0b601e65ad390f2bf5899bfaf59b7c287e5

interfaces/ZeroAddressError.sol	83642b852ae173732f849ec7dfe02b6ba5bf0fbc54f4571253c95628ae2cd1aa
interfaces/ITokenPermitSignatureDetails.sol	54f2b1f158566de51e0fbf6f86f47cc7745f375df8fbfba4ed35b7a41625128c
interfaces/ITeaFiRelayer.sol	30bd4c35e30c93148bd846b2092912e5df517e2de94cfb2de9c02865195e05b3
interfaces/ISynthToken.sol	c96da2ccf0070d7144c1ccac111481a615a06661b0bd8d789d0d6a4459e9a57c
interfaces/IProxyTrade.sol	657b3b875cf6e4a4b0bb1bb712b8c25db0c82d840cb86e5df6c664a31c40ec48
interfaces/IAllowanceTransfer.sol	53ef5d563cd769d51ac27a3517d0865f5b13646ba064f57f5e4b57f8398ea2d7
interfaces/IAggregationRouterV6.sol	368d7fca236de59037dae65e64b11b82049063f1745985c396b433e42cba5dd2
components/Permitable.sol	0dc8204811ab2e22cfeee6cdf312d1050ab028082440812bf9573b6203f5e401
components/PermitPayable.sol	f40dc08f17fdb2810c7f4a0e75b1e3c96b60813f003519a6ab3182f8e9f772f3
components/Authorizable.sol	b5ff523fa9e1e4ddbae19e6396173b12e92b6f1d2e2046cc59546d80fe852c75

Overview

ProxyTrade Contract

The `ProxyTrade` contract facilitates token swaps through the 1inch DEX aggregator. It serves as a proxy, allowing users to execute swaps while handling token approvals and payments. The contract also supports meta-transactions and integrates the Permit2 system for off-chain token authorization, enhancing security and flexibility.

`makePublicSwap` Functionality

This function enables users to perform token swaps by interacting with the 1inch aggregator. It first receives tokens from the user, checks and updates token allowances, and then executes the swap via 1inch, with the resulting tokens sent to the user's wallet.

`makePublicSwapWithPermit` Functionality

This function extends the swap process by including a token permit via the Permit2 system. After verifying the user's permit signature, it receives tokens, checks allowances, and executes the swap on 1inch, just like `makePublicSwap`.

`makePublicSwapWithTwoPermits` Functionality

In this method, both a token-specific permit and a Permit2 permit are processed. It first authorizes token transfers through these two permits, then proceeds with receiving tokens, checking allowances, and executing the swap on 1inch.

General Functionalities

The constructor initializes key components, including the trusted forwarder for meta-transactions and the 1inch aggregator address. It also sets up the Permit2 contract for token authorization. The contract overrides functions from `ERC2771Context` to support meta-transactions, enabling gasless transactions by allowing a forwarder to act on behalf of the user.

Authorities Functionalities

The contract incorporates functions from the `Permitable` contract to handle off-chain token approvals using the Permit2 system. These include making token permits and receiving payments based on the user's authorized allowances, ensuring efficient and secure token transfers.

TeaFiRelayer Contract

The `TeaFiRelayer` contract is responsible for managing gas payments and executing meta-transactions via a trusted forwarder. It uses the Permit2 system to handle token allowances for payments and allows operators to execute calls on behalf of users, with strict role-based access control. The contract ensures the secure and efficient relay of calls and payments through the trusted forwarder.

`relayCall` Functionality

This function allows operators to relay a meta-transaction on behalf of a user. It first processes the payment by receiving the required tokens, and then forwards the request to the trusted forwarder, which executes the transaction.

`relayCallWithPermit` Functionality

This function extends `relayCall` by adding support for Permit2. It accepts a permit signature, which allows the contract to process token payments without requiring on-chain approvals from the user. After verifying the signature, the payment is processed, and the call is relayed through the trusted forwarder.

`relayCallWithTwoPermits` Functionality

This function handles more complex payment scenarios where two permits are involved—one for the token and one for the Permit2 system. It first processes both permits, then receives the payment and executes the relayed transaction through the trusted forwarder.

General Functionalities

The constructor sets up critical components, including the trusted forwarder, multisig wallet, and token limits. Role-based access control is applied, where operators and token limit managers are granted specific roles to manage relays and token limits. The contract

overrides functions from `ERC2771Context` to support meta-transactions, enabling gasless calls on behalf of users.

Authorities Functionalities

The contract inherits from `Authorizable`, allowing it to manage operator and token limit manager roles. Operators can execute relayed calls, while token limit managers control the allowable token amounts for payments. This ensures secure and controlled access to the contract's critical functionalities.

TeaFiTrustedForwarder Contract

The `TeaFiTrustedForwarder` contract acts as the core forwarder for relaying meta-transactions. It implements role-based access control, where only authorized relayers can execute transactions on behalf of users. The contract ensures that only whitelisted relayer contracts can interact with it, adding an extra layer of security to the forwarding process.

`execute` Functionality

This function allows authorized relayers to execute a single meta-transaction on behalf of a user. The relayer submits the request, and the contract processes it securely by forwarding the call.

`executeBatch` Functionality

Similar to `execute`, this function enables relayers to execute multiple meta-transactions in a batch. This is useful for processing multiple operations in a single transaction, reducing gas costs and improving efficiency.

General Functionalities

The constructor grants admin and relayer roles, ensuring proper access control. The `setupRoles` function allows setting up the roles of multisig wallets and relayers, ensuring that only authorized entities can interact with the forwarder. The contract is initialized once, preventing reconfiguration after deployment.

Authorities Functionalities

As part of `AccessControl`, the contract manages roles such as `DEFAULT_ADMIN_ROLE` and `PROXY_ROLE`, allowing only trusted relayers to forward transactions. These roles are set during deployment, and any subsequent changes are restricted to ensure security.

SynthToken Contract

The `SynthToken` contract is an ERC20-compliant token designed to represent synthetic assets. It allows users to wrap and unwrap an underlying asset, effectively minting and burning synthetic tokens tied to that asset. The contract integrates with a trusted forwarder for meta-transactions and supports pausing functionalities for added control.

`wrap` Functionality

The `wrap` function enables users to convert their underlying assets into synthetic tokens. It transfers the underlying asset to the treasury, and in exchange, it mints an equivalent amount of synthetic tokens to the specified recipient.

`unwrap` Functionality

This function allows users to convert their synthetic tokens back into the underlying asset. It burns the synthetic tokens from the user and transfers the corresponding amount of the underlying asset from the treasury to the specified recipient.

General Functionalities

The constructor sets up the token's name, symbol, underlying asset, treasury, and trusted forwarder. It integrates with the `ERC2771Context` for meta-transaction support and includes role-based access for pausing and unpausing token activities. The contract can pause all token transfers, minting, and burning when triggered by the factory.

LiquidityBootstrapPoolFactory Contract

The `LiquidityBootstrapPoolFactory` is responsible for creating new synthetic tokens by wrapping underlying assets. It allows operators to create synthetic tokens based on user-defined token settings, such as the underlying asset and treasury address. It also manages token pause/unpause actions.

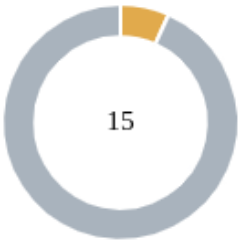
`createSynthTokens` Functionality

This function enables operators to create new synthetic tokens by specifying the token settings, including the underlying asset and treasury. Each token created is given a name and symbol derived from the underlying asset.

General Functionalities

The constructor assigns roles for token managers and operators while setting up the global treasury and trusted forwarder addresses. The contract includes functionality to update these settings and provides authority for token managers to pause or unpause synthetic tokens.

Findings Breakdown



Critical	0
Medium	1
Minor / Informative	14

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	1	0	0	0
Minor / Informative	14	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Medium	PFV	Permit Front-running Vulnerability	Unresolved
Minor / Informative	ACB	Amount Check Bypass	Unresolved
Minor / Informative	ALM	Array Length Mismatch	Unresolved
Minor / Informative	CO	Code Optimization	Unresolved
Minor / Informative	CCR	Contract Centralization Risk	Unresolved
Minor / Informative	DPI	Decimals Precision Inconsistency	Unresolved
Minor / Informative	DTC	Duplicate Token Creation	Unresolved
Minor / Informative	MC	Missing Check	Unresolved
Minor / Informative	MEE	Missing Events Emission	Unresolved
Minor / Informative	MTM	Missing Token-Asset Mapping	Unresolved
Minor / Informative	MU	Modifiers Usage	Unresolved
Minor / Informative	PTAI	Potential Transfer Amount Inconsistency	Unresolved
Minor / Informative	PWD	Potential Wrap DOS	Unresolved
Minor / Informative	L04	Conformance to Solidity Naming Conventions	Unresolved

●	L19	Stable Compiler Version	Unresolved

PFV - Permit Front-running Vulnerability

Criticality	Medium
Location	ProxyTrade.sol#L47 components/Permitable.sol#L28,50 components/PermitPayable.sol#L70,98
Status	Unresolved

Description

The `makePublicSwapWithTwoPermits` function is vulnerable to a front-running attack where a malicious actor can frontrun the transaction by calling the `permit` function before the intended user. Specifically, when a user attempts to perform a swap using two permits, an attacker could extract the signature parameters from the transaction and submit a direct `permit` transaction before the original `makePublicSwapWithTwoPermits` call is mined. As a result, the `makePublicSwapWithTwoPermits` function would revert, leading to a Denial of Service (DoS) for the user. This vulnerability is also present in related functions such as `_makeTokenPermit`, `_makePermit2`, `_receivePaymentWithPermit`, and `_receivePaymentWithTwoPermits`, which all execute a `permit` call unconditionally, causing reversion if frontrun.

```
    function makePublicSwapWithTwoPermits(
        OneInchSwap calldata swap,
        IAllowanceTransfer.PermitSingle calldata permitSingleStruct,
        bytes calldata permitSingleSignature,
        TokenPermitSignatureDetails calldata tokenPermitSignatureDetails
    ) external override {
        // make the token permit to Permit2 contract
        _makeTokenPermit(address(swap.desc.srcToken),
tokenPermitSignatureDetails);

        // make the permit2 to this contract
        _makePermit2(permitSingleStruct, permitSingleSignature);

        // receive tokens from the user
        _receivePayment(address(swap.desc.srcToken), swap.desc.amount);

        // approve the tokens to the oneInch contract
        _checkAllowance(swap.desc.srcToken, swap.desc.amount);

        // make the swap, in data the recepient is the user wallet
        IAggregationRouterV6(oneInch).swap(swap.executor, swap.desc,
swap.data);
    }
```

```
function _makeTokenPermit(
        address token,
        TokenPermitSignatureDetails calldata tokenPermitSignatureDetails
    ) internal {
        // make the first permit in token if it's first
        IERC20Permit(token).permit(
            _msgSender(),
            address(permit2),
            type(uint256).max, // max uint256
            tokenPermitSignatureDetails.deadline,
            tokenPermitSignatureDetails.v,
            tokenPermitSignatureDetails.r,
            tokenPermitSignatureDetails.s
        );
    }

    function _makePermit2(
        IAllowanceTransfer.PermitSingle calldata permitSingleStruct,
        bytes calldata signature
    ) internal {
        permit2.permit(_msgSender(), permitSingleStruct, signature);
    }
```

```
    function _receivePaymentWithTwoPermits(
        PaymentData calldata paymentData,
        IAllowanceTransfer.PermitSingle calldata permitSingleStruct,
        bytes calldata permitSingleSignature,
        TokenPermitSignatureDetails calldata tokenPermitSignatureDetails
    ) internal {
        if (paymentData.amount > 0) {
            // make the first permit in token if it's first
            IERC20Permit(paymentData.token).permit(
                paymentData.payer,
                address(permit2),
                type(uint256).max, // max uint256
                tokenPermitSignatureDetails.deadline,
                tokenPermitSignatureDetails.v,
                tokenPermitSignatureDetails.r,
                tokenPermitSignatureDetails.s
            );
            _receivePaymentWithPermit(paymentData, permitSingleStruct,
permitSingleSignature);
        }
    }


    function _receivePaymentWithPermit(
        PaymentData calldata paymentData,
        IAllowanceTransfer.PermitSingle calldata permitSingleStruct,
        bytes calldata signature
    ) internal {
        if (paymentData.amount > 0) {
            if (paymentData.amount > permitSingleStruct.details.amount)
revert NotEnoughAllowanceInPermit();
            permit2.permit(paymentData.payer, permitSingleStruct,
signature);
            _receivePayment(paymentData);
        }
    }
```

Recommendation

As a workaround, it is recommended to implement a check for sufficient allowance before calling the `permit` function in the `_makeTokenPermit` function to prevent front-running scenarios. A similar check should be added in `_makePermit2`, `_receivePaymentWithPermit`, and `_receivePaymentWithTwoPermits` to mitigate this vulnerability and avoid unexpected transaction reverts due to front-running attacks.

ACB - Amount Check Bypass

Criticality	Minor / Informative
Location	SynthToken.sol#L68
Status	Unresolved

Description

The contract contains a logic flaw in the `unwrap` function where the amount parameter can be set to the maximum `uint256` value (`type(uint256).max`). When this occurs, the amount is automatically set to the balance of the `msg.sender` . However, if the `msg.sender` has a zero balance, the function will bypass the initial check for zero amounts. This bypass can lead to unintended behavior, as the first check for a zero amount will not trigger a revert when it should, allowing transactions to proceed improperly.

```
   function unwrap(uint256 amount, address recipient) external override
{
        if (amount == 0) revert ZeroAmount();

        // this case is needed for private swaps
        if (amount == type(uint256).max) amount =
balanceOf(_msgSender());
        ...
        }
```

Recommendation

It is recommended to implement a check that validates the balance of the `msg.sender` after the assignment of the balance to the `amount` parameter. Specifically, if the `amount` is set to the maximum `uint256` , ensure that the `msg.sender` has a non-zero balance before allowing the function to continue. This will prevent the bypass of the initial zero-amount check and maintain the intended contract logic.

ALM - Array Length Mismatch

Criticality	Minor / Informative
Location	TeaFiRelayer.sol#L45 components/PermitPayable.sol#L55
Status	Unresolved

Description

The contract is designed to handle the process of elements from arrays through functions that accept multiple arrays as input parameters. These functions are intended to iterate over the arrays, processing elements from each array in a coordinated manner. However, there are no explicit checks to verify that the lengths of these input arrays are equal. This lack of validation could lead to scenarios where the arrays have differing lengths, potentially causing out-of-bounds access if the function attempts to process beyond the end of the shorter array. Such situations could result in unexpected behavior or errors during the contract's execution, compromising its reliability and security.

Specifically, the contract is missing a check to verify that the `tokens` and `limits` array have the same length.

```
    function changeTokenLimit(
        address[] calldata tokens,
        uint256[] calldata limits
    ) external override onlyRole(TOKEN_LIMIT_MANAGER_ROLE) {
        for (uint256 i = 0; i < tokens.length; ++i) {
            address token = tokens[i];
            uint256 limit = limits[i];
            if (token == address(0)) revert ZeroAddress();
            if (limit == 0) revert ZeroLimit();
            paymentTokenLimit[token] = limit;
        }
    }
```

```
    constructor(address _permit2, address _treasury, address[] memory
tokens, uint256[] memory limits) {
        ...
        uint256 length = tokens.length;
        for (uint256 i = 0; i < length; ++i) {
            if (tokens[i] == address(0)) revert ZeroAddress();
            if (limits[i] == 0) revert ZeroLimit();
            paymentTokenLimit[tokens[i]] = limits[i];
        }
    }
```

Recommendation

To mitigate this, it is recommended to incorporate a validation check at the beginning of the function that accepts multiple arrays to ensure that the lengths of these arrays are identical. This can be achieved by implementing a conditional statement that compares the lengths of the arrays, and reverts the transaction if the lengths do not match. Such a validation step will prevent out-of-bounds errors and ensure that elements from each array are processed in a paired and coordinated manner, thus preserving the integrity and intended functionality of the contract.

CO - Code Optimization

Criticality	Minor / Informative
Location	ProxyTrade.sol#L35,47,66
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

The contract is declaring multiple functions with similar code while it could reuse the existing function logic by calling them internally. Specifically, the contract could optimize by recalling one function inside the others where applicable, reducing redundancy and improving maintainability.

```
    function makePublicSwap(OneInchSwap calldata swap) external override
{
        // receive tokens from the user
        _receivePayment(address(swap.desc.srcToken), swap.desc.amount);

        // approve the tokens to the oneInch contract
        _checkAllowance(swap.desc.srcToken, swap.desc.amount);

        // make the swap, in data the recepient is the user wallet
        IAggregationRouterV6(oneInch).swap(swap.executor, swap.desc,
swap.data);
    }

    function makePublicSwapWithPermit(
        OneInchSwap calldata swap,
        IAllowanceTransfer.PermitSingle calldata permitSingleStruct,
        bytes calldata signature
    ) external override {
        // make the permit2 to this contract
        _makePermit2(permitSingleStruct, signature);

        // receive tokens from the user
        _receivePayment(address(swap.desc.srcToken), swap.desc.amount);

        // approve the tokens to the oneInch contract
        _checkAllowance(swap.desc.srcToken, swap.desc.amount);

        // make the swap, in data the recepient is the user wallet
        IAggregationRouterV6(oneInch).swap(swap.executor, swap.desc,
swap.data);
    }

    function makePublicSwapWithTwoPermits(
        OneInchSwap calldata swap,
        IAllowanceTransfer.PermitSingle calldata permitSingleStruct,
        bytes calldata permitSingleSignature,
        TokenPermitSignatureDetails calldata tokenPermitSignatureDetails
    ) external override {
        // make the token permit to Permit2 contract
        _makeTokenPermit(address(swap.desc.srcToken),
tokenPermitSignatureDetails);

        // make the permit2 to this contract
        _makePermit2(permitSingleStruct, permitSingleSignature);

        // receive tokens from the user
        _receivePayment(address(swap.desc.srcToken), swap.desc.amount);

        // approve the tokens to the oneInch contract
        _checkAllowance(swap.desc.srcToken, swap.desc.amount);
```

```
        // make the swap, in data the recepient is the user wallet
        IAggregationRouterV6(oneInch).swap(swap.executor, swap.desc,
swap.data);
    }
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it. It is recommended to refactor the contract to reuse the existing function logic by calling one function within another, optimizing the code and reducing repetition.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	SynthTokenFactory.sol#L104,136,147,158,169 TeaFiRelayer.sol#L45,59,71,85 TeaFiTrustedForwarder.sol#L35,57,66
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

Specifically, the following roles have significant authority over key contract functions:

- **OPERATOR_ROLE**: Has the authority to create new synthetic tokens with the provided settings and parameters, and execute relay calls.
- **DEFAULT_ADMIN_ROLE**: Has the authority to set the global treasury and the trusted forwarder address, and set up roles.
- **TOKEN_MANAGER**: Can pause and unpause transactions.
- **TOKEN_LIMIT_MANAGER_ROLE**: Can change token limits.
- **PROXY_ROLE**: Can execute single and multiple transactions via the relayer.

This concentration of power could lead to potential abuse or mismanagement if these roles are not properly decentralized or adequately secured.

```
    function createSynthTokens(
        TokenSettings[] memory args
    ) external onlyRole(OPERATOR_ROLE) returns (address[] memory) {
        address[] memory tokens = new address[](args.length);
        for (uint256 i = 0; i < args.length; ++i) {
            ...
            address token = address(
                // don't use create2 because of ERC2771Context
constructor by openzeppelin (even in upgradeable contracts)
                new SynthToken(name, symbol,
factorySettings.trustedForwarder, args[i].underlyingAsset, treasury)
            );
            tokens[i] = token;

            emit TokenCreated(token);
        }
        return tokens;
    }

    function setGlobalTreasury(address _newTreasury) external virtual
onlyRole(DEFAULT_ADMIN_ROLE) {
        ...
        factorySettings.globalTreasury = _newTreasury;

        emit GlobalTreasurySet(_newTreasury);
    }

    function setTrustedForwarder(address _newForwarder) external virtual
onlyRole(DEFAULT_ADMIN_ROLE) {
        ...
        factorySettings.trustedForwarder = _newForwarder;

        emit TrustedForwarderSet(_newForwarder);
    }

    function pauseTokens(address[] calldata tokens) external
onlyRole(TOKEN_MANAGER) {
        ...
            SynthToken(tokens[i]).pause();
        }
    }

    function unpauseTokens(address[] calldata tokens) external
onlyRole(TOKEN_MANAGER) {
        ...
            SynthToken(tokens[i]).unpause();
        }
    }
```

```
    function changeTokenLimit(
        address[] calldata tokens,
        uint256[] calldata limits
    ) external override onlyRole(TOKEN_LIMIT_MANAGER_ROLE) {
        for (uint256 i = 0; i < tokens.length; ++i) {
            ...
            paymentTokenLimit[token] = limit;
        }
    }

        ERC2771Forwarder.ForwardRequestData calldata request,
        PermitPayable.PaymentData calldata paymentData
    ) external override onlyRole(OPERATOR_ROLE) {
        // receive payment
        _receivePayment(paymentData);

        // execute the call
        trustedForwarder.execute(request);
    }

    function relayCallWithPermit(
        ERC2771Forwarder.ForwardRequestData calldata request,
        PermitPayable.PaymentData calldata paymentData,
        IAllowanceTransfer.PermitSingle calldata permitSignatureDetails,
        bytes calldata permitSingleSignature
    ) external override onlyRole(OPERATOR_ROLE) {
        ...
        trustedForwarder.execute(request);
    }

    ...
```

```
 function setupRoles(address multisigWallet, address[] memory relayers)
external onlyRole(DEFAULT_ADMIN_ROLE) {
        ...
            _grantRole(PROXY_ROLE, relayers[i]);
        }
        initialized = true;
    }


    function execute(ForwardRequestData calldata request) public payable
override onlyRole(PROXY_ROLE) {
        super.execute(request);
    }

    function executeBatch(
        ForwardRequestData[] calldata requests,
        address payable refundReceiver
    ) public payable override onlyRole(PROXY_ROLE) {
        super.executeBatch(requests, refundReceiver);
    }
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

DPI - Decimals Precision Inconsistency

Criticality	Minor / Informative
Location	SynthToken.sol#L51
Status	Unresolved

Description

The decimals field of a contract's ERC20 token can be used to specify the number of decimal places that the token uses. For example, if decimals are set to `8`, it means that the smallest unit of the token is `0.00000001`, and if decimals are set to `18`, it means that the smallest unit of the token is `0.000000000000000001`.

However, there is an inconsistency in the way that the decimals field is handled in some ERC20 contracts. The ERC20 specification does not specify how the decimals field should be implemented, and as a result, some contracts use different precision numbers.

This inconsistency can cause problems when interacting with these contracts, as it is not always clear how the decimals field should be interpreted. For example, if a contract expects the decimals field to be 18 digits, but the contract being interacted with uses 8 digits, the result of the interaction may not be what was expected.

Specifically, the contract is currently designed to transfer an amount of the `underlyingAsset` to the contract and mint the same amount of synthetic tokens to the recipient through the `wrap` function. However, this could lead to inconsistencies if the `underlyingAsset` and the minted synthetic token have different characteristics, such as differing decimal precision. In such cases, the function may result in an unintended amount being minted or transferred, which could cause discrepancies in the token balances or supply.

```
    function wrap(uint256 amount, address recipient) external override {
        if (amount == 0) revert ZeroAmount();

        // this case is needed for private swaps
        if (amount == type(uint256).max) amount =
IERC20(underlyingAsset).balanceOf(_msgSender());

        // transfer underlying asset to treasury
        SafeERC20.safeTransferFrom(IERC20(underlyingAsset),
_msgSender(), treasury, amount);

        // mint synth tokens to recipient
        // recipient is checked in ERC20._mint
        _mint(recipient, amount);

        emit Wrapped(_msgSender(), recipient, amount);
    }
```

Recommendation

To avoid these issues, it is important to carefully review the implementation of the decimals field of the underlying tokens. The team is advised to normalize each decimal to one single source of truth. A recommended way is to scale all the decimals to the greatest token's decimal. Hence, the contract will not lose precision in the calculations.

The following example depicts 3 tokens with different decimals precision.

ERC20	Decimals
Token 1	6
Token 2	9
Token 3	18

All the decimals could be normalized to 18 since it represents the ERC20 token with the greatest digits.

It is recommended to account for the characteristics of the `underlyingAsset` and the synthetic token, particularly the difference in decimal precision. The contract should handle the conversion between assets with differing decimals to ensure accurate transfers and

minting, preventing any discrepancies or potential imbalances caused by mismatched amounts. For instance, the contract could check that the `underlyingAsset` and the synthetic token have the same decimals to avoid any inconsistencies during the transfer and minting processes.

DTC - Duplicate Token Creation

Criticality	Minor / Informative
Location	SynthTokenFactory.sol#L104
Status	Unresolved

Description

The contract is missing a check in the `createSynthTokens` function to prevent the deployment of synthetic tokens with the same settings, specifically if identical arguments are provided in successive calls. As a result, each time the function is invoked with the same parameters, a new synthetic token contract will be created, even if a token with the same symbol or underlying asset already exists. This could lead to the unintentional creation of multiple tokens with the same underlying symbol, causing potential confusion or misuse.

```
  function createSynthTokens(
        TokenSettings[] memory args
    ) external onlyRole(OPERATOR_ROLE) returns (address[] memory) {
        address[] memory tokens = new address[](args.length);
        for (uint256 i = 0; i < args.length; ++i) {
            string memory underlyingSymbol =
IERC20Metadata(args[i].underlyingAsset).symbol();
            if (bytes(underlyingSymbol).length == 0) revert
InvalidSymbol();

            string memory name = abi.encodePacked("Tea-Wrapped ",
underlyingSymbol).toHexString();
            string memory symbol = abi.encodePacked("t",
underlyingSymbol).toHexString();

            ...

            address token = address(
                // don't use create2 because of ERC2771Context
constructor by openzeppelin (even in upgradeable contracts)
                new SynthToken(name, symbol,
factorySettings.trustedForwarder, args[i].underlyingAsset, treasury)
            );
            tokens[i] = token;

            emit TokenCreated(token);
        }
        return tokens;
    }
```

Recommendation

It is recommended to add a check in the `createSynthTokens` function to verify whether the provided arguments have already been used to deploy a synthetic token with the same symbol or underlying asset. If it is intended to prevent deploying multiple tokens with the same settings, the function should revert in such cases. This will ensure that synthetic tokens are only created once for each unique set of parameters, avoiding duplication and maintaining the integrity of token creation.

MC - Missing Check

Criticality	Minor / Informative
Location	TeaToken.sol#L24 SynthTokenFactory.sol#L71
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues.

Specifically, the contract is missing a check to verify that the addresses are not set to the zero address.

```
    constructor(
        ...
        address globalTreasury,
        address trustedForwarder,
        ...
    ) {
        ...
        factorySettings = FactorySettings(globalTreasury,
trustedForwarder);
        ...
    }
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	TeaFiRelayer.sol#L45
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
function changeTokenLimit(
    address[] calldata tokens,
    uint256[] calldata limits
) external override onlyRole(TOKEN_LIMIT_MANAGER_ROLE) {
    for (uint256 i = 0; i < tokens.length; ++i) {
        address token = tokens[i];
        uint256 limit = limits[i];
        if (token == address(0)) revert ZeroAddress();
        if (limit == 0) revert ZeroLimit();
        paymentTokenLimit[token] = limit;
    }
}
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

MTM - Missing Token-Asset Mapping

Criticality	Minor / Informative
Location	SynthTokenFactory.sol#L104
Status	Unresolved

Description

The contract lacks a mechanism to map synthetic token addresses to their underlying assets, making it difficult for users and administrators to track which assets correspond to specific synthetic tokens. Although the `SynthToken` contract stores this data, the factory contract does not maintain a public list or mapping for easy access. This oversight could lead to confusion when interacting with the contract. Implementing a mapping or public function to expose these relationships would improve transparency and usability.

```
contract LiquidityBootstrapPoolFactory is AccessControl {
    ...

    function createSynthTokens(
        TokenSettings[] memory args
    ) external onlyRole(OPERATOR_ROLE) returns (address[] memory) {
        address[] memory tokens = new address[](args.length);
        for (uint256 i = 0; i < args.length; ++i) {
            string memory underlyingSymbol =
IERC20Metadata(args[i].underlyingAsset).symbol();
            if (bytes(underlyingSymbol).length == 0) revert
InvalidSymbol();

            string memory name = abi.encodePacked("Tea-Wrapped ",
underlyingSymbol).toHexString();
            string memory symbol = abi.encodePacked("t",
underlyingSymbol).toHexString();

            address treasury = args[i].treasury == address(0) ?
factorySettings.globalTreasury : args[i].treasury;

            address token = address(
                // don't use create2 because of ERC2771Context
constructor by openzeppelin (even in upgradeable contracts)
                new SynthToken(name, symbol,
factorySettings.trustedForwarder, args[i].underlyingAsset, treasury)
            );
            tokens[i] = token;

            emit TokenCreated(token);
        }
        return tokens;
    }

    ...
}
```

Recommendation

To enhance transparency and usability, it is recommended to implement a mapping or public function within the factory contract that records and exposes the relationship between synthetic token addresses and their underlying assets. This would allow users and administrators to conveniently retrieve this information when needed.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	SynthToken.sol#L52,69
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
if (amount == 0) revert ZeroAmount();
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	SynthToken.sol#L58
Status	Unresolved

Description

The `safeTransferFrom` function IS used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

```
SafeERC20.safeTransferFrom(IERC20(underlyingAsset), _msgSender(),
treasury, amount);
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

```
 Actual Transferred Amount = Balance After Transfer - Balance
Before Transfer
```

PWD - Potential Wrap DOS

Criticality	Minor / Informative
Location	SynthToken.sol#L51
Status	Unresolved

Description

The contract is vulnerable to a front-running scenario when the `wrap` function is called with the maximum possible amount (`type(uint256).max`). In this case, the contract automatically sets the amount to the balance of the sender. However, if a malicious actor sends a small amount of the underlying asset to the sender before the transaction is mined, the balance of the sender will increase. If the sender has not provided an adequate allowance to the contract for this new balance, the transaction may revert due to insufficient allowance, potentially causing unexpected failures for users.

```
    function wrap(uint256 amount, address recipient) external override {
        if (amount == 0) revert ZeroAmount();

        // this case is needed for private swaps
        if (amount == type(uint256).max) amount =
IERC20(underlyingAsset).balanceOf(_msgSender());

        // transfer underlying asset to treasury
        SafeERC20.safeTransferFrom(IERC20(underlyingAsset),
_msgSender(), treasury, amount);

        // mint synth tokens to recipient
        // recipient is checked in ERC20._mint
        _mint(recipient, amount);

        emit Wrapped(_msgSender(), recipient, amount);
    }
```

Recommendation

It is recommended to include additional safeguards to ensure that the wrap function handles front-running scenarios gracefully, either by recalculating allowances before

transferring the underlying asset or by limiting the risk of reversion if the sender's balance changes unexpectedly.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	SynthTokenFactory.sol#L132,143
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
address _newTreasury
address _newForwarder
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	TeaFiTrustedForwarder.sol#L2 TeaFiRelayer.sol#L2 components/PermitPayable.sol#L2 components/Permitable.sol#L2 components/Authorizable.sol#L2
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.24;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
TeaFiTrustedForwarder	Implementation	ERC2771Forwarder, AccessControl, ZeroAddressError		
		Public	✓	ERC2771Forwarder
	setupRoles	External	✓	onlyRole
	execute	Public	Payable	onlyRole
	executeBatch	Public	Payable	onlyRole
	hashTypedDataV4	External		-
TeaFiRelayer	Implementation	PermitPayable, Authorizable, ITeaFiRelayer		
		Public	✓	PermitPayable Authorizable
	changeTokenLimit	External	✓	onlyRole
	relayCall	External	✓	onlyRole
	relayCallWithPermit	External	✓	onlyRole
	relayCallWithTwoPermits	External	✓	onlyRole
LiquidityBootstrapPoolFactory	Implementation	AccessControl		

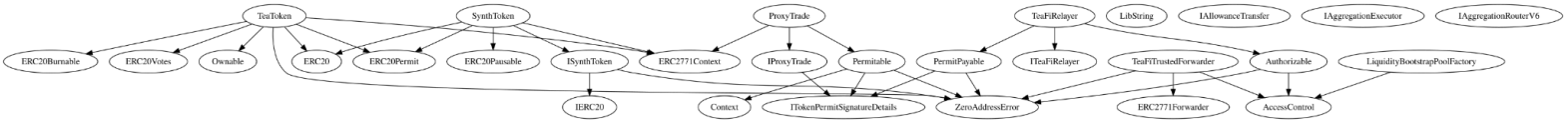
		Public	✓	-
	createSynthTokens	External	✓	onlyRole
	setGlobalTreasury	External	✓	onlyRole
	setTrustedForwarder	External	✓	onlyRole
	pauseTokens	External	✓	onlyRole
	unpauseTokens	External	✓	onlyRole
SynthToken	Implementation	ERC20, ISynthToken, ERC2771Context, ERC20Permit, ERC20Pausable		
		Public	✓	ERC20 ERC2771Context ERC20Permit
	wrap	External	✓	-
	unwrap	External	✓	-
	pause	External	✓	onlyFactory
	unpause	External	✓	onlyFactory
	_update	Internal	✓	
	_msgSender	Internal		
	_msgData	Internal		
	_contextSuffixLength	Internal		
	hashTypedDataV4	External		-
ProxyTrade	Implementation	ERC2771Context,		

		Permitable, IProxyTrade		
		Public	✓	ERC2771Context Permitable
	makePublicSwap	External	✓	-
	makePublicSwapWithPermit	External	✓	-
	makePublicSwapWithTwoPermits	External	✓	-
	_checkAllowance	Internal	✓	
	_msgSender	Internal		
	_msgData	Internal		
	_contextSuffixLength	Internal		
ITeaFiRelayer	Interface			
	trustedForwarder	External		-
	changeTokenLimit	External	✓	-
	relayCall	External	✓	-
	relayCallWithPermit	External	✓	-
	relayCallWithTwoPermits	External	✓	-
ISynthToken	Interface	IERC20, ZeroAddressError		
	underlyingAsset	External		-
	treasury	External		-
	factory	External		-
	wrap	External	✓	-

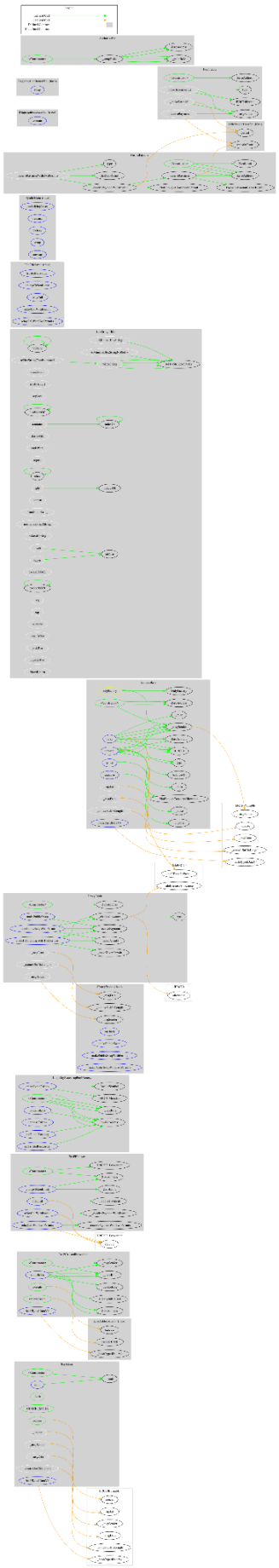
	unwrap	External	✓	-
IProxyTrade	Interface	ITokenPermit SignatureDet ails		
	oneInch	External		-
	makePublicSwap	External	✓	-
	makePublicSwapWithPermit	External	✓	-
	makePublicSwapWithTwoPermits	External	✓	-
Permitable	Implementation	ZeroAddress Error, ITokenPermit SignatureDet ails, Context		
		Public	✓	-
	_makeTokenPermit	Internal	✓	
	_makePermit2	Internal	✓	
	_receivePayment	Internal	✓	
PermitPayable	Implementation	ZeroAddress Error, ITokenPermit SignatureDet ails		
		Public	✓	-
	_receivePaymentWithTwoPermits	Internal	✓	
	_receivePaymentWithPermit	Internal	✓	
	_receivePayment	Internal	✓	

Authorizable	Implementation	AccessControl, ZeroAddressError		
		Public	✓	-
	_setupRoles	Internal	✓	

Inheritance Graph



Flow Graph



Summary

The Tea-Fi suite of contracts implements a comprehensive system for facilitating token swaps, synthetic asset creation, meta-transactions, and role-based access control. This audit investigates security vulnerabilities, business logic concerns, and potential improvements in the use of trusted forwarders, the Permit2 system, and role-based governance across the contracts.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

cyberscope.io