Cyberscope

Audit Report

Tea-Fi

December 2024

Audited by © cyberscope

Table of Contents

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 Updates

Initial Audit	19 Dec 2024

Source Files

Filename	SHA256
SynthStaking.sol	efb974cc50dd3637a77d6a85c12119146d67cce3248cffadbad2ad5e2e9d3543
interfaces/ZeroAddressError.sol	96a30953a9b30293688e8d159a503b6e6763013274b295a5d946b775b7dd9a6b
interfaces/ITokenPermitSignatureDetails.sol	4621780d35e831ef8270085cababbe682705812b793555a09909cf02b63a8b1e
interfaces/ISynthStaking.sol	9a7daf722c0fc3aacebc0928c89a99b35897ffa40abf41ee447d630d88820d24
interfaces/IAllowanceTransfer.sol	e6d68e521724e091a35046f89b912f9be8dcbe1ddd3e21bfac2ef9c6f8f5f655
components/Storage.sol	26e35e0715fccb20430249d28a302537e4bc17475837c16a17db771e6288d740
components/StakingService.sol	a44324df89fe288f289e11a808ccf50f7629dc67edd018e3bb1c176b14e2c03a
components/RewardsService.sol	eec465f5afd0130d1f8cffeaad5c95d530bef2c37aa2b31f445393f707148539
components/Permitable.sol	2127a07cc9361af368175faa3c38ee4f107b975e6fdb33cc907ae09de8b01227

components/EIP712Service.sol	c7ca4c7a44843449dce0b27643c4a767a4 314053c065fd9475ab1b34ae61c3d2

Overview

The provided smart contracts constitute a comprehensive staking system named SynthStaking. This system enables users to stake ERC20 tokens, manage their allowances through signature-based permissions, and earn rewards based on their staked amounts. The architecture is modular, comprising several interfaces and abstract contracts that define distinct functionalities such as permission handling, reward distribution, and state management.

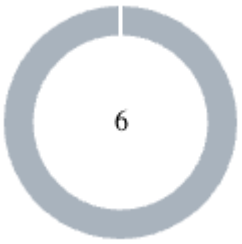
At the core of the system is the SynthStaking contract, which inherits from both StakingService and RewardsService. The StakingService manages the core staking and unstaking operations, ensuring that users can deposit and withdraw their tokens securely while adhering to predefined cooldown periods. It leverages the Permitable contract to handle token permissions via the Permit2 protocol, allowing users to approve token transfers through off-chain signatures. This enhances the user experience by reducing the number of on-chain transactions required for staking.

The RewardsService is responsible for the distribution and management of staking rewards. It utilizes the EIP712Service for secure and standardized signature verification, ensuring that reward claims are authenticated and authorized properly. The contract maintains meticulous records of rewards and implements cooldown mechanisms to mitigate abuse and ensure fair distribution among participants.

Security is a paramount consideration across all contracts. The system employs robust access control mechanisms through role-based permissions (ADMIN_ROLE and OPERATOR_ROLE) to restrict sensitive operations to authorized entities. Additionally, the contracts incorporate error handling for various edge cases, such as zero address validations and invalid signature detections, thereby enhancing the overall resilience of the staking platform.

In summary, the SynthStaking smart contracts provide a secure, efficient, and user-friendly framework for staking ERC20 tokens. By integrating advanced permission schemes, meticulous state management, and comprehensive reward distribution mechanisms, the system ensures a reliable and scalable solution for decentralized staking applications.

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	6

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	6	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	CR	Code Repetition	Unresolved
	TSI	Tokens Sufficiency Insurance	Unresolved
	CCR	Contract Centralization Risk	Unresolved
	DCI	Duplicate Contract Import	Unresolved
	MMN	Misleading Method Naming	Unresolved
	L15	Local Scope Variable Shadowing	Unresolved

CR - Code Repetition

Criticality	Minor / Informative
Location	components/RewardsService.sol#L64 components/StakingService.sol#L124
Status	Unresolved

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

```
function getWithdrawableRewardIds(address staker) external view returns
(WithdrawableIds[] memory ids) {
    uint256 length = stakerInfo[staker].rewardIds.length();
    ids = new WithdrawableIds[](length);

    for (uint256 i = 0; i < length; ++i) {
        uint256 rewardId = stakerInfo[staker].rewardIds.at(i);
        ids[i] = WithdrawableIds(rewardId, block.timestamp >
rewardRecords[rewardId].endCooldownTimestamp);
    }
}
function getWithdrawableStakeIds(address staker) external view returns
(WithdrawableIds[] memory ids) {
    uint256 length = stakerInfo[staker].unstakeIds.length();
    ids = new WithdrawableIds[](length);

    for (uint256 i = 0; i < length; ++i) {
        uint256 rewardId = stakerInfo[staker].unstakeIds.at(i);
        ids[i] = WithdrawableIds(rewardId, block.timestamp >
unstakeRecords[rewardId].endCooldownTimestamp);
    }
}
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	components/RewardsService.sol#L75
Status	Unresolved

Description

The tokens are not held within the contract itself. Instead, the contract is designed to provide the tokens from an external administrator. While external administration can provide flexibility, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks.

```
teaToken.safeTransferFrom(treasury, to, amount);
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to hold the tokens within the contract itself. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	components/Storage.sol#L192,201
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.

For instance, the `DEFAULT_ADMIN_ROLE` has the authority to set the `unstakeCooldownDuration` and `rewardCooldownDuration` to a very large value. As a result, stakers will not be able to withdraw their unstaked tokens and their rewards respectively.

```
function setUnstakeCooldownDuration(uint256 duration) external
onlyRole(DEFAULT_ADMIN_ROLE) {
    _setUnstakeCooldownDuration(duration);
}
function setRewardCooldownDuration(uint256 duration) external
onlyRole(DEFAULT_ADMIN_ROLE) {
    _setRewardCooldownDuration(duration);
}
function _verifyCollectRewardSignature(
    address from,
    CollectRewardParam calldata param
) internal returns (bool result, string memory errorReason) {
    bytes memory encodedData = abi.encode(
        COLLECT_REWARD_TYPEHASH,
        param.staker,
        param.rewardAmount,
        param.nonce,
        param.deadline
    );
    return
        _verifySignature(encodedData, from, param.operator, param.nonce,
param.deadline, param.v, param.r, param.s);
}
```

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.

DCI - Duplicate Contract Import

Criticality	Minor / Informative
Location	SynthStaking.sol#L4,8
Status	Unresolved

Description

In the SynthStaking contract, the ERC2771Context from OpenZeppelin is imported twice. This redundant import can lead to several issues. Duplicate imports can unnecessarily inflate the contract's bytecode size, leading to higher deployment and execution costs. Additionally, multiple imports of the same dependency can complicate the contract's dependency graph, making it harder to manage and update in the future.

```
import {ERC2771Context} from "@openzeppelin/contracts/metatx/ERC2771Context.sol";
import {ERC2771Context, Context} from
"@openzeppelin/contracts/metatx/ERC2771Context.sol";
```

Recommendation

To mitigate this issue and optimize the contract, it is recommended to eliminate one of the ERC2771Context import statements. Ensure that only a single import exists for ERC2771Context. After making this change, the team should thoroughly test the contract to ensure that the removal of the duplicate import does not affect its functionality. Implementing this recommendation will reduce the contract size and simplify future maintenance.

MMN - Misleading Method Naming

Criticality	Minor / Informative
Location	components/Storage.sol#L111
Status	Unresolved

Description

Methods can have misleading names if their names do not accurately reflect the functionality they contain or the purpose they serve. The contract uses some method names that are too generic or do not clearly convey the underneath functionality. Misleading method names can lead to confusion, making the code more difficult to read and understand.

For instance, the `validUint256` method asserts a given unsigned integer (uint256) is not zero. However, zero is a valid unsigned integer. As a result, the method name could lead to confusion.

```
modifier validUint256(uint256 value) {
    if (value == 0) revert ZeroUint256();
    _;
}
```

Recommendation

It's always a good practice for the contract to contain method names that are specific and descriptive. The team is advised to keep in mind the readability of the code.

L15 - Local Scope Variable Shadowing

Criticality	Minor / Informative
Location	components/StakingService.sol#L56,165
Status	Unresolved

Description

Local scope variable shadowing occurs when a local variable with the same name as a variable in an outer scope is declared within a function or code block. When this happens, the local variable "shadows" the outer variable, meaning that it takes precedence over the outer variable within the scope in which it is declared.

```
StakerInfo storage stakerInfo = stakerInfo[staker]
```

Recommendation

It's important to be aware of shadowing when working with local variables, as it can lead to confusion and unintended consequences if not used correctly. It's generally a good idea to choose unique names for local variables to avoid shadowing outer variables and causing confusion.

Functions Analysis

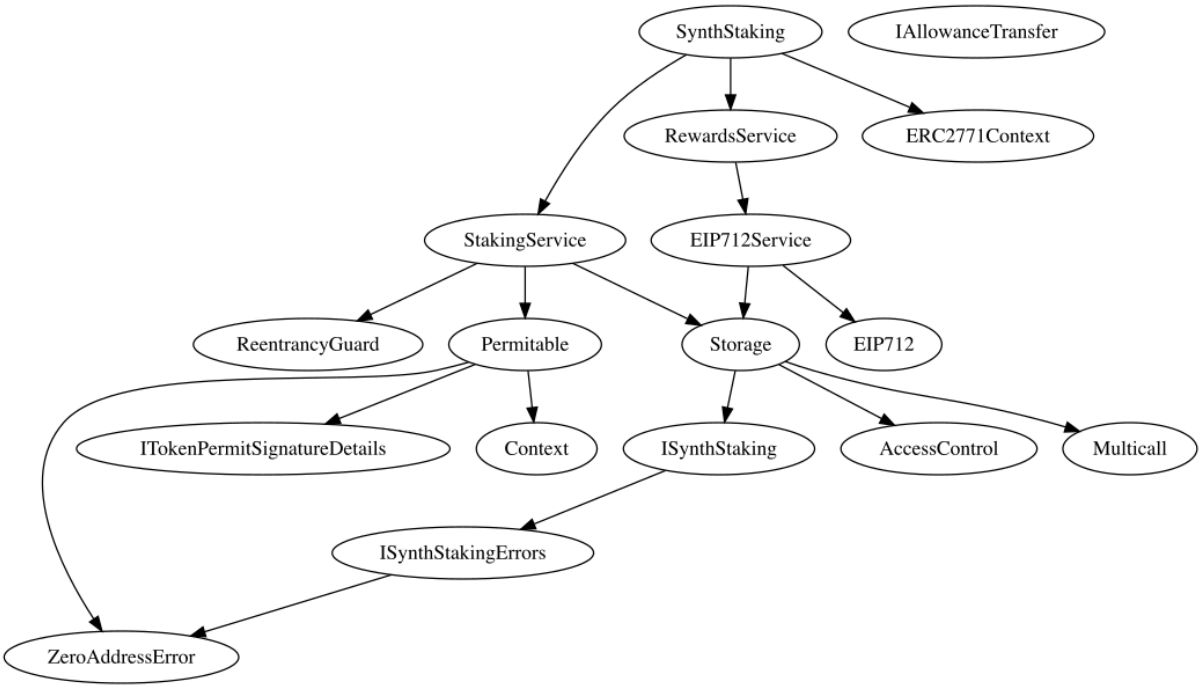
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
SynthStaking	Implementation	StakingService, RewardsService, ERC2771Context		
		Public	✓	validAddress StoragePermitable ERC2771Context EIP712
	emergencyWithdrawErc20	External	✓	validAddress onlyRole
	emergencyWithdrawEth	External	✓	validAddress onlyRole
	_msgSender	Internal		
	_msgData	Internal		
	_contextSuffixLength	Internal		
ISynthStakingErrors	Interface	ZeroAddressError		
ISynthStaking	Interface	ISynthStakingErrors		
	stake	External	✓	-
	stake	External	✓	-
	stake	External	✓	-
	unstake	External	✓	-

	withdrawStake	External	✓	-
	collectReward	External	✓	-
	withdrawReward	External	✓	-
	getPoolInfo	External		-
	getStakerInfo	External		-
Storage	Implementation	ISynthStaking, AccessControl, Multicall		
		Public	✓	validAddress
	createPool	External	✓	onlyRole validAddress
	setUnstakeCooldownDuration	External	✓	onlyRole
	setRewardCooldownDuration	External	✓	onlyRole
	setTreasury	External	✓	onlyRole
	setTotalAllocation	External	✓	onlyRole
	getPoolInfo	External		-
	getStakerInfo	External		-
	getTotalStakedByStakerByPool	External		-
	_setupRoles	Private	✓	
	_setUnstakeCooldownDuration	Private	✓	validUint256
	_setRewardCooldownDuration	Private	✓	validUint256
	_setTreasury	Private	✓	validAddress
	_setTotalAllocation	Private	✓	validUint256
	_grantRoleWallet	Private	✓	validAddress

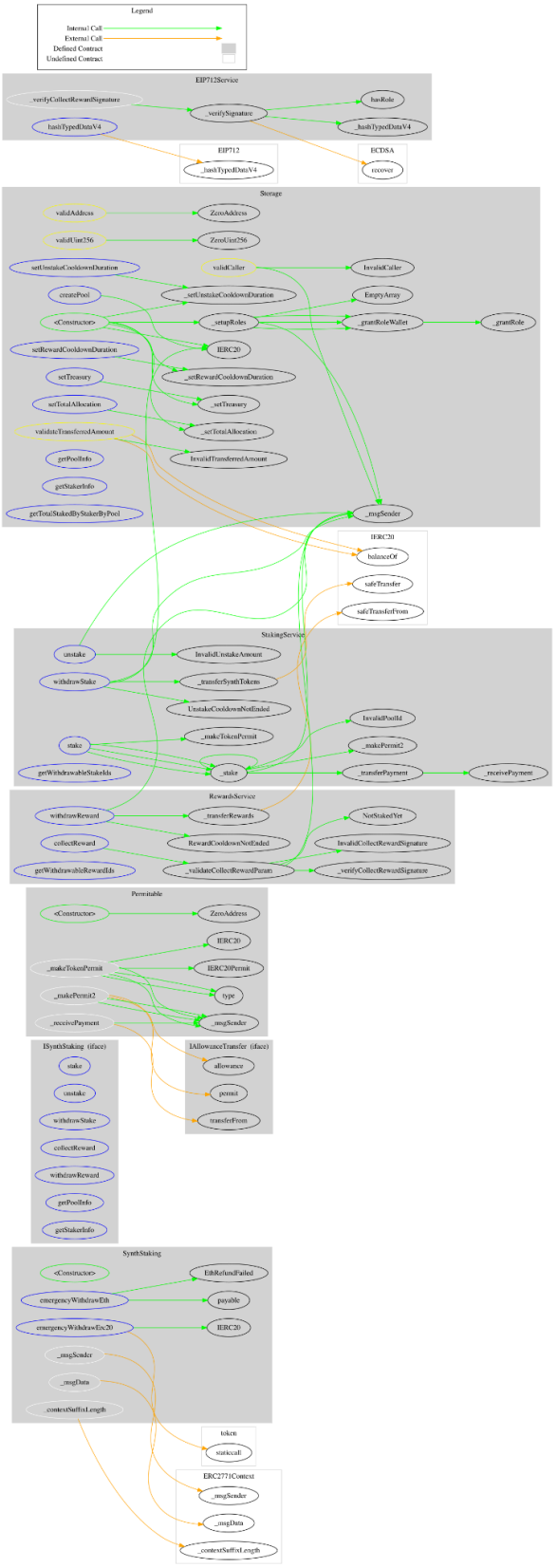
StakingService	Implementation	Storage, Permitable, ReentrancyGuard		
	stake	External	✓	-
	stake	External	✓	-
	stake	External	✓	-
	unstake	External	✓	validCaller validUint256
	withdrawStake	External	✓	validCaller
	getWithdrawableStakeIds	External		-
	_stake	Internal	✓	
	_stake	Internal	✓	validUint256 nonReentrant
	_transferPayment	Private	✓	validateTransferredAmount
	_transferSynthTokens	Private	✓	validateTransferredAmount
RewardsService	Implementation	EIP712Service		
	collectReward	External	✓	-
	withdrawReward	External	✓	validCaller
	getWithdrawableRewardIds	External		-
	_transferRewards	Private	✓	validateTransferredAmount
	_validateCollectRewardParam	Private	✓	validAddress validUint256 validCaller

Permitable	Implementation	ZeroAddress Error, ITokenPermit SignatureDet ails, Context		
		Public	✓	-
	_makeTokenPermit	Internal	✓	
	_makePermit2	Internal	✓	
	_receivePayment	Internal	✓	
EIP712Service	Implementation	Storage, EIP712		
	_verifyCollectRewardSignature	Internal	✓	
	_verifySignature	Internal	✓	
	hashTypedDataV4	External		-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
EIP712Service
_verifyCollectRewardSignature
hashTypedDataV4
_verifySignature
hasRole
_hashTypedDataV4
EIP712
_hashTypedDataV4
ECDSA
recover
Storage
validAddress
ZeroAddress
validUint256
ZeroUint256
setUnstakeCooldownDuration
validCaller
InvalidCaller
createPool
_setUnstakeCooldownDuration
EmptyArray
<Constructor>
_setupRoles
_grantRoleWallet
_grantRole
IERC20
setRewardCooldownDuration
_setRewardCooldownDuration
setTreasury
_setTreasury
setTotalAllocation
_setTotalAllocation
validateTransferredAmount
InvalidTransferredAmount
getPoolInfo
getStakerInfo
getTotalStakedByStakerByPool
_msgSender
IERC20
balanceOf
safeTransfer
safeTransferFrom
StakingService
unstake
InvalidUnstakeAmount
withdrawStake
_transferSynthTokens
UnstakeCooldownNotEnded
InvalidPoolId
stake
_makeTokenPermit
_makePermit2
getWithdrawableStakeIds
_stake
_transferPayment
_receivePayment
RewardsService
withdrawReward
_transferRewards
NotStakedYet
collectReward
RewardCooldownNotEnded
InvalidCollectRewardSignature
getWithdrawableRewardIds
_validateCollectRewardParam
_verifyCollectRewardSignature
Permitable
<Constructor>
ZeroAddress
IERC20
_makeTokenPermit
IERC20Permit
_makePermit2
type
_receivePayment
_msgSender
ISynthStaking (iface)
stake
unstake
withdrawStake
collectReward
withdrawReward
getPoolInfo
getStakerInfo
IAllowanceTransfer (iface)
allowance
permit
transferFrom
SynthStaking
<Constructor>
EthRefundFailed
emergencyWithdrawEth
payable
emergencyWithdrawErc20
IERC20
_msgSender
_msgData
_contextSuffixLength
token
staticcall
ERC2771Context
_msgSender
_msgData
_contextSuffixLength

Summary

Tea-Fi contracts implement a staking and rewards mechanism. This audit investigates security issues, business logic concerns, and potential improvements.

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