Cyberscope

Audit Report

Fluxtra

March 2025

Repository https://github.com/neonswapfi/lsd-contracts

Commit 5caff13770b37d738be27539fc70875e9aa0ba75

Audited by © cyberscope

Table of Contents

Table of Contents 1
Risk Classification 2
Review 3
Audit Updates 3
Source Files 3
Overview 5
Fluxtra Hub Contract 5
Stake Token Contract 6
Findings Breakdown 7
Diagnostics 8
CCR - Contract Centralization Risk 9
Description 9
Recommendation 10
GER - Gas Exhaustion Risk 11
Description 11
Recommendation 12
ICT - Inefficient Collection Traversal 13
Description 13
Recommendation 13
MRA - Misleading Rebalance Access 14
Description 14
Recommendation 15
MEH - Missing Error Handling 16
Description 16
Recommendation 17
MEE - Missing Events Emission 18
Description 18
Recommendation 20
MSV - Missing Swap Validation 21
Description 21
Recommendation 22
MVO - Multiple Vote Overrides 23
Description 23
Recommendation 25
Summary 26
Disclaimer 27
About Cyberscope 28

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

Repository	https://github.com/neonswapfi/lsd-contracts
Commit	5caff13770b37d738be27539fc70875e9aa0ba75

Audit Updates

Initial Audit	14 Mar 2025

Source Files

Filename	SHA256
protos/mod.rs	2c9fdd48e9ebec0b41430e7cc4e784c138a85e04f96e39cd2e47ae6e571c0ba9
protos/proto.rs	89d48ff41fb120dde48fa48073dd3c4075cf4c5856c37ee26818052be26bc1e2
types/gauges.rs	2320b8eab07bcd15a8e53e5340cb82c1452ef37f0eb55cfcc32ab4b6f54086c4
types/coins.rs	2c2286998d5c2c04b49c7fa79cf40a3d8727270395f6d1179c7cb43765a8e9b3
types/mod.rs	c83ab8ccc4791b74b19ea6a01b33f4f9eb7a7d53a98824c161abab78f2c0e7b2
types/staking.rs	0955bf977a943b04c3eaa938757a87678f66e4bcbf2852ffb1b3e993b068dbcd
types/keys.rs	44eda63ad456c232ada60aecdc7e1ee96114c9b0bc3ba032d4695fb4b2d1ca4d
execute.rs	e7593be4314e180b3d55a5a14e8f08976e89df2505942abecdaef9983c70b796
math.rs	58cc63353d6be2901d1482fd911354cdca046ae425799fb938dd71eb45e82564
constants.rs	f80719b620570e0c3e7f557ee97dc33c447a261735d784ff14a05a9cfab9651b
queries.rs	606c8b108d923274e8b3f10f0373f07a1605382abd4733e91ce6bb6cfe68da0a

error.rs	d5ee79644b89529444372d25bba0c888d6743e5b76e23e0121c565898b7d9da5
gov.rs	00dbdda5ecc2f91892df6df3faab6de89e9c959859a69e6355a60ea54b3516e8
lib.rs	722306f304de746f36fe36c553c34aeba18582e686c71bcb3944b6b7b2af4de4
contract.rs	a69437d33223a2b04306655a29ca5823ed2a1411f87cd1b580d45bde97d81793
helpers.rs	0f70d39403524ad2adc93c39dfdeb382240f633a759bdd75c1340391b44ed6ee
claim.rs	7bf04504f2f42783a60e0e837b02d03d185f4560edcda4b90ad00e86e32e0e4e
state.rs	5c1ada11c19ea4dc7fec678236d2c4f356c33e2df59e4998112ff2e16aef33be
token/src/lib.rs	a6b9958d0241d0d8817e4032ef73f62c3cd94d2328cfff155dccf724575d51d9

Overview

Fluxtra Hub Contract

The Fluxtra staking hub contract is a sophisticated component of a liquid staking protocol within the Cosmos ecosystem, designed to facilitate staking of native tokens while providing users with liquid staking tokens in return. Its primary function is to allow users to deposit a native token, referred to as utoken , and receive a corresponding liquid token, ustake , which represents their staked position. This enables users to maintain liquidity while earning staking rewards, which are reinvested to increase the value of ustake over time. The contract delegates these tokens to a set of validators, either uniformly or based on a gauge-based strategy, ensuring balanced distribution and adaptability to governance preferences. Users can later request to unbond their ustake , triggering a batched unbonding process that adheres to the Cosmos unbonding period, after which they reclaim their utoken adjusted for rewards or potential losses.

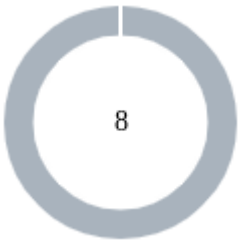
The contract supports a robust reward management system where staking rewards from validators are periodically harvested and reinvested after deducting a protocol fee, enhancing the overall value locked in the system. It maintains transparency through detailed queries, allowing users to monitor configuration settings, current state (like total value locked and exchange rates), delegation targets, unbonding batches, and historical performance metrics such as APR. Governance features include the ability to update validators, adjust delegation strategies, and facilitate voting via an operator, providing flexibility and community control. The unbonding process is carefully managed in batches, with reconciliation mechanisms to handle discrepancies like slashing, ensuring fairness in distribution among users.

Security and maintenance are prioritized with functionalities for rebalancing delegations, tuning strategies based on external gauge data, and handling edge cases like validator removal or slashing losses. The contract’s design emphasizes automation, fairness, and user empowerment, making it a central hub for staking operations within a decentralized finance framework.

Stake Token Contract

The Stake Token contract is a CW20-based token designed as the liquid staking token (ustake) for the Fluxtra Staking Hub within the Cosmos ecosystem, enabling users to stake native tokens and receive a transferable representation of their staked assets. Its primary purpose is to provide liquidity to stakers, allowing them to trade or utilize ustake in DeFi while the hub manages staking and reward reinvestment. The contract restricts minting and burning to the staking hub, ensuring that token supply aligns with staked value, and disables third-party burns to prevent price manipulation, enhancing security and stability. It supports standard token operations like transfers and balance queries, integrating seamlessly with the hub to facilitate staking, unbonding, and reward distribution workflows.

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	8

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	0	8	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	CCR	Contract Centralization Risk	Acknowledged
	GER	Gas Exhaustion Risk	Acknowledged
	ICT	Inefficient Collection Traversal	Acknowledged
	MRA	Misleading Rebalance Access	Acknowledged
	MEH	Missing Error Handling	Acknowledged
	MEE	Missing Events Emission	Acknowledged
	MSV	Missing Swap Validation	Acknowledged
	MVO	Multiple Vote Overrides	Acknowledged

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	execute.rs#L829 vote.rs#L13,33
Status	Acknowledged

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.

```
pub fn rebalance(
    deps: DepsMut,
    env: Env,
    sender: Addr,
    min_redelegation: Option<Uint128>,
) -> ContractResult {
...
```

```
pub fn vote(
    deps: DepsMut,
    _env: Env,
    info: MessageInfo,
    proposal_id: u64,
    vote: cosmwasm_std::VoteOption,
) -> ContractResult {
    ...
}

pub fn vote_weighted(
    deps: DepsMut,
    _env: Env,
    info: MessageInfo,
    proposal_id: u64,
    votes: Vec<(Decimal, cosmwasm_std::VoteOption)>,
) -> ContractResult {
...
}
```

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.

GER - Gas Exhaustion Risk

Criticality	Minor / Informative
Location	excecute.rs#L718
Status	Acknowledged

Description

The contract is prone to potential gas exhaustion or memory overflow in the `withdraw_unbonded` function, which processes all unbond requests for a user in a single transaction. The function iterates over all unbond requests stored for a user, performing state updates and calculations for each request. If a user accumulates a large number of unbond requests (e.g., hundreds), the transaction may exceed the block gas limit or WASM memory constraints, causing it to fail. This could prevent users from claiming their unbonded tokens, effectively locking their funds until the issue is resolved, as there is no pagination or limit on the number of requests processed per transaction.

```
pub fn withdraw_unbonded(deps: DepsMut, env: Env, user: Addr, receiver:
Addr) -> ContractResult {
    let state = State::default();
    let stake_token = state.stake_token.load(deps.storage)?;
    let current_time = env.block.time.seconds();

    ...
    let requests = state
        .unbond_requests
        .idx
        .user
        .prefix(user.to_string())
        .range(deps.storage, None, None, Order::Ascending)
        .map(|item| {
            let (_, v) = item?;
            Ok(v)
        })
        .collect::<StdResult<Vec<_>>>()?;
```

Recommendation

It is recommended to implement pagination or a maximum limit on the number of unbond requests processed in a single `withdraw_unbonded` transaction. This can be achieved by allowing users to specify a range of request IDs or a maximum number of requests to process, ensuring that transactions remain within safe gas and memory limits. Additionally, providing a mechanism to query the number of unbond requests for a user can help users anticipate and manage their withdrawals.

ICT - Inefficient Collection Traversal

Criticality	Minor / Informative
Location	coins.rs#L25
Status	Acknowledged

Description

The contract is designed to manage a collection of `coin` objects and includes a method to locate a specific coin by its denomination. This method iterates over the entire collection, cloning each coin object before searching for a match, which results in unnecessary memory allocation and increased computational overhead. In a blockchain environment where gas costs are tied to resource usage, this inefficiency can lead to higher transaction costs and reduced performance, particularly when the collection grows large. The approach unnecessarily duplicates every item in the collection, even though only one item (if any) needs to be returned, amplifying the resource footprint without functional benefit.

```
    pub fn find(&self, denom: &str) -> Coin {
        self.0
            .iter()
            .cloned()
            .find(|coin| coin.denom == denom)
            .unwrap_or_else(|| Coin::new(0, denom))
    }
```

Recommendation

It is recommended to optimize the method by first searching the collection for the matching coin using a reference-based approach, then cloning only the found item if it exists. This reduces the number of cloning operations from the entire collection to just the single matched element, minimizing memory usage and gas costs. The default case for when no match is found should also be handled efficiently, ensuring the method remains functional while improving overall performance. This optimization aligns with best practices for resource-constrained environments like smart contracts, enhancing scalability and cost-effectiveness.

MRA - Misleading Rebalance Access

Criticality	Minor / Informative
Location	excecute.rs#L164,821
Status	Acknowledged

Description

The contract is prone to confusion due to a misleading comment in the `bond` function documentation, which states that `anyone can invoke ExecuteMsg::Rebalance to balance the delegations.` In reality, the `rebalance` function restricts access to the contract owner only, as enforced by the `state.assert_owner check`. This discrepancy between the documentation and implementation can mislead users or developers into believing that rebalancing is a permissionless operation, potentially leading to incorrect assumptions about the contract's behavior or attempts to invoke the function by unauthorized parties, resulting in failed transactions.

```
/// (e.g. when a single user makes a very big deposit), anyone can invoke
`ExecuteMsg::Rebalance`
/// to balance the delegations.
pub fn bond(
...

pub fn rebalance(
    deps: DepsMut,
    env: Env,
    sender: Addr,
    min_redelegation: Option<Uint128>,
) -> ContractResult {
    let state = State::default();
    let stake_token = state.stake_token.load(deps.storage)?;
    state.assert_owner(deps.storage, &sender)?;
    ...
    }
```

Recommendation

It is recommended to update the comment above the `bond` function to accurately reflect the access control of the `rebalance` function, explicitly stating that only the contract owner can invoke `ExecuteMsg::Rebalance`. Additionally, ensuring that all public-facing documentation and inline comments align with the actual implementation will improve clarity and prevent confusion for users and developers interacting with the contract.

MEH - Missing Error Handling

Criticality	Minor / Informative
Location	math.rs#L30,47
Status	Acknowledged

Description

The contract is susceptible to arithmetic errors in the `compute_mint_amount` and `compute_unbond_amount` functions due to the absence of proper error handling for arithmetic operations. These functions perform multiplication and division using multiply_ratio without using checked arithmetic operations (`checked_mul` , `checked_div`), which can lead to unexpected behavior such as integer overflows or division-by-zero errors. For example, in `compute_mint_amount` , if `utoken_bonded` is zero, the function avoids division-by-zero by returning `utoken_to_bond` , but in other cases, unchecked division and multiplication may cause runtime panics or incorrect results. Similarly, in `compute_unbond_amount` , while the function assumes `ustake_supply` is non-zero, there is no explicit check to ensure this, and unchecked arithmetic operations is not handled with proper error handling.

```
pub(crate) fn compute_mint_amount(
    ustake_supply: Uint128,
    utoken_to_bond: Uint128,
    current_delegations: &[Delegation],
) -> Uint128 {
    let utoken_bonded: u128 = current_delegations.iter().map(|d|
d.amount).sum();
    if utoken_bonded == 0 {
        utoken_to_bond
    } else {
        ustake_supply.multiply_ratio(utoken_to_bond, utoken_bonded)
    }
}

pub(crate) fn compute_unbond_amount(
    ustake_supply: Uint128,
    ustake_to_burn: Uint128,
    current_delegations: &[Delegation],
) -> Uint128 {
    let utoken_bonded: u128 = current_delegations.iter().map(|d|
d.amount).sum();
    Uint128::new(utoken_bonded).multiply_ratio(ustake_to_burn,
ustake_supply)
}
```

Recommendation

It is recommended to implement checked arithmetic operations using methods like `checked_mul` and `checked_div` from the `Uint128` type in both the `compute_mint_amount` and `compute_unbond_amount` functions. This ensures that arithmetic overflows and division-by-zero scenarios are explicitly handled, either by returning an error (`StdResult`) or by gracefully handling edge cases. Additionally, adding explicit validation for inputs (e.g., ensuring `ustake_supply` is non-zero in `compute_unbond_amount`) will enhance the robustness of the contract and prevent runtime errors.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	token/src/lib.rs#L27 excecute.rs#L951,969,990
Status	Acknowledged

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.

The contract delegates to `cw20_execute` for burning, which reduces the sender's balance and total supply but does not explicitly require or enforce event emission. While `cw20_execute` may emit events in the base implementation, the custom logic here doesn't guarantee visibility of burn actions in the transaction log, especially since burns are critical for unbonding in the staking hub context.

Additionaly the contract is lacking transparency in state changes due to missing detailed event emissions in governance functions such as `update_config`, `accept_ownership, and` transfer_ownership `. The` update_config `function updates key contract parameters but only emits a basic` action `attribute without detailed events listing the updated values. Similarly, the` transfer_ownership` function proposes a new owner but does not emit a detailed event with the proposed new owner's address. This lack of detailed event emissions hinders off-chain monitoring, as external systems cannot easily track or verify changes to the contract's configuration or ownership state, potentially reducing trust and traceability.

```
ExecuteMsg::Burn { amount } => {
    assert_minter(deps.storage, &info.sender)?;
    let mut response = cw20_execute(deps, env, info, msg)?;
    response = response.add_attribute("action",
"burn").add_attribute("amount", amount.to_string());
    Ok(response)
}
```

```
pub fn transfer_ownership(deps: DepsMut, sender: Addr, new_owner: String)
-> ContractResult {
    let state = State::default();

    state.assert_owner(deps.storage, &sender)?;
    state.new_owner.save(deps.storage,
&deps.api.addr_validate(&new_owner)?)?;

    Ok(Response::new().add_attribute("action",
"fluxtra/transfer_ownership"))
}

pub fn accept_ownership(deps: DepsMut, sender: Addr) -> ContractResult {
    let state = State::default();
    ...
    }

pub fn update_config(
    deps: DepsMut,
    sender: Addr,
    protocol_fee_contract: Option<String>,
    protocol_reward_fee: Option<Decimal>,
    operator: Option<String>,
    stages_preset: Option<Vec<Vec<(Addr, AssetInfo)>>>,
    allow_donations: Option<bool>,
    delegation_strategy: Option<DelegationStrategy>,
    vote_operator: Option<String>,
) -> ContractResult {
    let state = State::default();
    ...
```

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.

MSV - Missing Swap Validation

Criticality	Minor / Informative
Location	execute.rs#L990
Status	Acknowledged

Description

The contract is prone to misconfigurations in the `update_config` function due to inadequate validation of the `stages_preset` parameter, which represents a vector of swap stage configurations. This parameter, consisting of nested vectors of address and asset information pairs, is saved directly to storage without ensuring that the addresses are valid swap router contracts or that the vector is free of duplicate entries. The existing validation only prevents the inclusion of the contract's native token or staking token, but it does not verify the functional correctness or uniqueness of the swap stage entries. This could allow an owner to inadvertently or maliciously configure invalid or redundant swap routes, potentially disrupting the contract's swapping logic, leading to inefficient execution, or enabling exploitation through unexpected behavior.

```
#[allow(clippy::too_many_arguments)]
pub fn update_config(
    deps: DepsMut,
    sender: Addr,
    protocol_fee_contract: Option<String>,
    protocol_reward_fee: Option<Decimal>,
    operator: Option<String>,
    stages_preset: Option<Vec<Vec<(Addr, AssetInfo)>>>,
    allow_donations: Option<bool>,
    delegation_strategy: Option<DelegationStrategy>,
    vote_operator: Option<String>,
) -> ContractResult {
    ...
    }
```

Recommendation

It is recommended to enhance the validation of the `stages_preset` parameter in the `update_config` function by implementing comprehensive checks. Specifically, ensure that each address in the configuration is validated as a legitimate swap router contract by querying its contract type or interface compatibility. Additionally, introduce a deduplication mechanism to eliminate duplicate swap stage entries, ensuring the configuration remains concise and intentional. These measures will strengthen the contract's robustness, prevent misconfigurations, and maintain the integrity of the swapping process. This finding captures the essence of the validation and deduplication issues identified in the `update_config` function while adhering to the requested format and avoiding code snippets.

MVO - Multiple Vote Overrides

Criticality	Minor / Informative
Location	gov.rs#L13,33
Status	Acknowledged

Description

The contract is designed to allow the vote operator to submit votes on governance proposals multiple times through its voting functions, without restrictions on the number of submissions. In the context of the underlying blockchain’s governance system, this means that each new vote submitted by the operator overrides any previous votes cast by the contract for the same proposal. Only the last valid vote submitted before the voting period ends is ultimately counted, which is a standard behavior of the governance module. While this does not introduce a direct vulnerability, it could lead to confusion or unintended outcomes if the operator submits conflicting votes unintentionally or if external systems monitoring the contract’s voting activity assume earlier votes are final. This flexibility might also obscure the contract’s voting intent until the voting period concludes.

```
pub fn vote(
    deps: DepsMut,
    _env: Env,
    info: MessageInfo,
    proposal_id: u64,
    vote: cosmwasm_std::VoteOption,
) -> ContractResult {
    let state = State::default();
    state.assert_vote_operator(deps.storage, &info.sender)?;

    let event = Event::new("fluxtra/voted").add_attribute("prop",
proposal_id.to_string());

    let vote = CosmosMsg::Gov(GovMsg::Vote {
        proposal_id,
        vote,
    });


Ok(Response::new().add_message(vote).add_event(event).add_attribute("acti
on", "fluxtra/vote"))
}

pub fn vote_weighted(
    deps: DepsMut,
    _env: Env,
    info: MessageInfo,
    proposal_id: u64,
    votes: Vec<(Decimal, cosmwasm_std::VoteOption)>,
) -> ContractResult {
    let state = State::default();
    state.assert_vote_operator(deps.storage, &info.sender)?;

    let event =
Event::new("fluxtra/voted_weighted").add_attribute("prop",
proposal_id.to_string());

    let vote = MsgVoteWeighted {
        proposal_id,
        voter: _env.contract.address.to_string(),
        options: votes
            .into_iter()
            .map(|vote| WeightedVoteOption {
                special_fields: SpecialFields::default(),
                option: match vote.1 {
                    cosmwasm_std::VoteOption::Yes =>
VoteOption::VOTE_OPTION_YES.into(),
                    cosmwasm_std::VoteOption::No =>
VoteOption::VOTE_OPTION_NO.into(),
```

```
                    cosmwasm_std::VoteOption::Abstain =>
VoteOption::VOTE_OPTION_ABSTAIN.into(),
                    cosmwasm_std::VoteOption::NoWithVeto => {
                        VoteOption::VOTE_OPTION_NO_WITH_VETO.into()
                    },
                },
                weight: vote.0.numerator().to_string(),
            })
            .collect_vec(),
        special_fields: SpecialFields::default(),
    };

    let vote = vote.to_cosmos_msg();

    Ok(Response::new()
        .add_message(vote)
        .add_event(event)
        .add_attribute("action", "fluxtra/vote_weighted"))
}
```

Recommendation

It is recommended to consider if this is the intended functionality; otherwise, consider preventing votes for proposals that have already been made by tracking submitted votes in the contract's state and rejecting subsequent attempts for the same proposal. This could enhance clarity and enforce a single-vote policy if multiple overrides are not desired, aligning the contract's behavior more closely with specific operational expectations. Additionally, is recommended to document this behavior clearly in the contract's documentation to ensure users and operators understand that votes can be overridden by subsequent submissions, with only the final vote counting.

Summary

Fluxtra contract implements a liquid staking mechanism. This audit investigates security issues, business logic concerns, and potential improvements in the Fluxtra Hub and Stake Token contracts, enabling native token bonding, reward reinvestment, and batched unbonding within the Cosmos ecosystem.

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