its___msn itsmsn

ITISH

AUDIT COMPANY

Audit Details

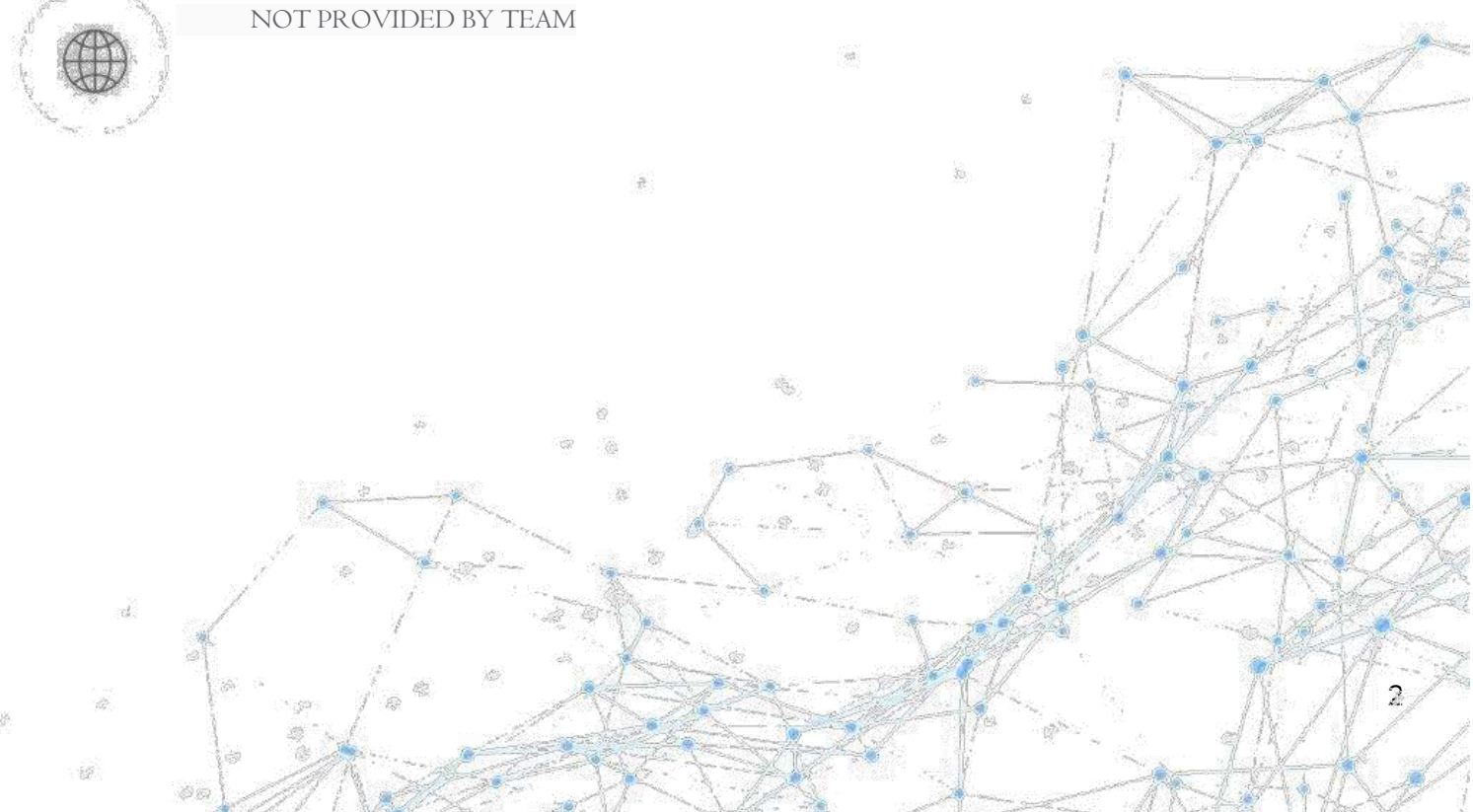
Token Name
DeadKings

Deployer address
CnpmQxLkijdNBmL1Z41aPMtvstBSFWr6hkbPbXYtY9TK

Client contacts:
DeadKIngs team

Blockchain
NEAR

Project website:
NOT PROVIDED BY TEAM



Itish was commissioned DeadKings token to perform an audit.

https://explorer.mainnet.near.org/transactions/CnpmQxLkijdNBmLIZ4IaPMtvstBSFWr6hkbPbXYtY9TK

The purpose of the audit was to achieve the following:

- Ensure that the smart contract functions as intended.
- Identify potential security issues with the smart contract.

The information in this report should be used to understand the risk exposure of the smart contract, and as a guide to improve the security posture of the smart contract by remediating the issues that were identified.

Introduction

File Name:

dis_chess.rs

Summary:

The code seems to implement a chess tournament, where tokens are granted to a designated chess account every 2 months, as long as the total supply of tokens is not exceeded. The code also implements functions to update the chess account, as well as to check the amount of tokens minted for the chess tournament.

- Dependencies:

The code is using the **near_sdk** and crate dependencies, but there is no information on the version of these dependencies used. Checking if the latest and secure version of these dependencies is being used is recommended.

- Deployment Environment

The code does not contain any information on the deployment environment, it is important to check if the contract is deployed in a secure and scalable environment.

Report: All the gathered information is described in this report.

Security Vulnerabilities

Compiler Errors:

I do not see any syntax errors in the code.

Reentrancy Vulnerabilities:

I do not see any signs of reentrancy vulnerabilities in the code.

Other Vulnerabilities:

The dis_chess function requires that the caller is the owner, this is enforced using **self.only_owner()** function.

```
    fn dis_chess(&mut self){
        self.only_owner();
        // 1 MONTH UNIX TIME = 2,629,746,000
```

In the **dis_chess function**, the code checks if the total supply of tokens is less than or equal to a specified maximum value, and if the chess grants have already been minted, this is enforced using the require! Macro.

In the **dis_chess** function, the code also checks if the current block timestamp is greater than the **chess_timestamp** + 5259492000 before unlocking the tokens.

```
if env::block_timestamp_ms()>=self.chess_timestamp+5259492000
// for testing purpose 10 second
// if env::block_timestamp_ms()>=self.chess_timestamp+10000
{
    self.chess_timestamp=env::block_timestamp_ms();
    // 5% of chess Grants
```

The **update_chess_account** function requires that the caller is the owner, this is enforced using **self.only_owner()** function.

```
fn update_chess_account(&mut self,account_id:AccountId)->AccountId{
    self.only_owner();
    self.chess_ac=account_id;
    log!("Updated chess Grant Account Id : {}",self.chess_ac.clone());
    self.chess_ac.clone()
}
```

The code does not contain any other signs of vulnerabilities. However, this does not mean that the code is completely secure, a more thorough analysis and testing is required to identify all possible security issues

Language: Rust

Platform and tools: Remix, VScode, securify and other tools mentioned in the automated analysis section.

Code Details :

The code implements a trait called ChessTournment for the Contract struct. This trait has three functions, which are:

dis_chess: releases a portion of the total tokens (5%) to the chess tournament account after every two months. The condition for token release is checked using the block_timestamp_ms function from the env module.

update_chess_account: updates the chess tournament account, which is only accessible by the owner of the contract.

chess_minted_tokens: returns the total number of tokens that have been released to the chess tournament account.

Overall Recommendations:

- Add unit tests to verify that the contract functions as expected
- Check and update the dependencies to the latest and secure versions
- Consider deploying the contract in a secure and scalable environment
- Conduct a code review with other developers to catch any potential issues and provide alternative perspectives.

Introduction

File Name:

dis_community.rs

Summary:

- Following smart contract trait named Community for the NEAR platform. The trait has three methods: **dis_community, update_community_account**, and **community_minted_tokens**. The Community trait is then implemented for the Contract type.

- The **dis_community** method allows the contract owner to mint 5% of the total token supply (up to a maximum of 1 billion tokens) every 3 months. The method first checks if the current block timestamp is greater than the last minted timestamp plus the duration of 3 months in milliseconds. If the duration has passed, it mints the tokens, updates the timestamp, increments the number of months the community grants have been minted, adds the minted tokens to the total supply, and emits an event indicating the tokens were minted. If the duration has not passed, the method panics with an error message.
- The **update_community_account** method allows the contract owner to update the account that will receive the community grants. The method takes an **account_id** parameter as input and sets it as the new community account.

- The **community_minted_tokens** method is a view function that returns the total number of tokens minted to the community account as a 128-bit unsigned integer

• Dependencies:

The code is using the following dependencies:

near_sdk: A Rust SDK for developing smart contracts on the NEAR Protocol.
crate: This is a reference to the current crate (i.e., the current library or executable). It is used to access the functions and data defined within the same crate.
It appears that no other external dependencies have been used in this code.

• Deployment Environment

The smart contract is written in Rust and is expected to be executed in the NEAR virtual machine. The near_bindgen macro indicates that the code is meant to be executed as a smart contract on the NEAR network

Report: All the gathered information is described in this report.

Security Vulnerabilities

Compiler Errors:

I do not see any syntax errors in the code.

Reentrancy Vulnerabilities:

I do not see any signs of reentrancy vulnerabilities in the code.

Other Vulnerabilities:

Contract owner: The code defines the concept of a contract owner and uses the **only_owner** method to restrict certain methods to only be callable by the contract owner. However, the implementation of **only_owner** is not included in the code, which means the contract may not be secure. The contract owner should be clearly defined and the implementation of the **only_owner** method should be provided to ensure the proper functioning of the contract.

```
//  UPDATE TREASURY ACCOUNT (ONLY ADMIN)
fn update_community_account(&mut self,account_id:AccountId)->AccountId{
    self.only_owner();
    self.community_ac=account_id;
    log!("Updated Community Grant Account Id : {}",self.community_ac.clone());
    self.community_ac.clone()
}
```

Uninitialized variables: Some of the variables used in the code, such as **com_timestamp** and community_ac, may not be initialized before they are used. This could result in unexpected behavior and security vulnerabilities. Consider initializing these variables in the contract constructor or a separate method.

```
if env::block_timestamp_ms()>=self.com_timestamp+7889238000
// for testing purpose 10 sec
// if env::block_timestamp_ms()>=self.com_timestamp+10000
{

    self.com_timestamp=env::block_timestamp_ms();
```

Error handling: The require! macro is used to validate the conditions before minting tokens. However, if any of the conditions fail, the code will panic, which may not be the best approach for error handling. Consider using more robust error handling mechanisms, such as returning an error code or emitting an event to signal the error.

Time constraints: The dis_community method checks the block timestamp to determine if a month has passed since the last time tokens were minted. However, the time constraints in blockchain systems are not always accurate and may lead to unexpected behavior. Consider using a trusted timestamp oracle or other mechanisms to ensure accurate time constraints.

```
fn dis_community(&mut self){
    self.only_owner();
    // 1 MONTH UNIX TIME = 2,629,746,000
    // ms for 2 mint 120000
    //token supply check  5000000000
    let total_tokens= U128(1000000000000000000000000000000000);
    require!(self.total_supply<=total_tokens.into(),"All tokens have minted");
    // total community grants 5 % of 1 billion token supply
    let total_community_tokens= U128(50000000000000000000000000000000);
    // checking if all community grants have minted.
    require!(self.community_minted_tokens<total_community_tokens.into(),"All community grants already minted");
    //granting 5% tokens every 3 months
```

Integer overflow: The code uses 128-bit integers to represent the token supply and minting amounts. However, these integers are not checked for overflow, which could result in unexpected behavior and security vulnerabilities. Consider using safe arithmetic operations and checking for overflow.

Lack of documentation: The code uses large 128-bit integers for the token supply and minting amounts, which may be difficult to read and understand. Consider using more descriptive names and documenting the code to make it more readable.

Gas efficiency: The dis_community method calls the internal_deposit method, which could be expensive in terms of gas consumption. Consider optimizing the implementation to reduce gas costs.

```
self.internal_deposit(&self.community_ac.clone(), supply.into());
//adding to total supply
self.total_supply+=supply.0;
```

Lack of input validation: The code does not validate user inputs, which can lead to issues like buffer overflow, type mismatches, and format string vulnerabilities. You should validate all inputs before processing them to ensure that they are in the expected format and within the expected range.

Inadequate access control: The code only checks if the caller is the owner of the contract. This can be easily bypassed if the owner's account is compromised. You should also consider implementing role-based access control to limit the actions that different users can perform.

Unchecked return values: The code does not check the return values of functions, which can lead to vulnerabilities if the function fails and returns an error code. You should always check the return values of functions and handle errors appropriately.

Use of weak cryptography: The code does not appear to use any cryptography, which can make it vulnerable to attacks if it handles sensitive information. You should use strong cryptography, such as secure hashes and encryption, to protect sensitive information.

Hard-coded constants: The code uses hard-coded constants, such as the block time stamp and the number of tokens to be minted, which can make the code vulnerable to attacks if the values are known. You should use dynamically generated values, such as random numbers, to increase security.
Inadequate input validation: There does not appear to be any input validation in the code, which can make the contract vulnerable to attacks such as SQL injection or buffer overflows.

Unrestricted token creation: The contract has the ability to mint new tokens, which could result in an inflationary scenario if the contract is exploited. It's recommended to implement stricter checks and limits on the amount of tokens that can be created.

Lack of access controls: The contract does not appear to have any access controls, which means that any account can call the functions. This can lead to unauthorized access to sensitive information or actions, such as the ability to mint new tokens.

No protection against reentrancy attacks: There does not appear to be any protection against reentrancy attacks, which can allow attackers to repeatedly call the contract's functions to manipulate its state.

Language: Rust

Platform and tools: Remix, VScode, securify and other tools mentioned in the automated analysis section.

Code Details :

This code defines a Rust trait named Community and implements it for the Contract struct. The Community trait defines three functions:

dis_community: This function is intended to distribute a certain percentage of tokens to the community account. It first checks that the caller of the function is the owner of the contract and that not all tokens have been minted (limited by a constant total_tokens). Then, it checks if a set duration has passed (determined by comparing the current block timestamp with a stored com_timestamp) and if so, deposits a portion of tokens (determined by the constant supply) to the community account and updates the com_timestamp, community_months, and community_minted_tokens variables. If the duration has not passed, it will panic with a message.

update_community_account: This function allows the owner of the contract to update the community_ac account ID which is used to receive community tokens.

community_minted_tokens: This is a view function that returns the number of tokens that have been minted for the community as a U128.

The Community trait is implemented for the Contract struct, which presumably has additional state variables and functions not shown in this code snippet. The only_owner function is used to ensure that the caller of the functions is the owner of the contract.

Overall Recommendations:

Here are some steps you can take to address the security vulnerabilities in the code:

Input validation: Before processing any inputs, you should validate them to ensure that they are in the expected format and within the expected range. This can be done using functions like assert, expect, or custom error handling.

Access control: You should implement role-based access control to limit the actions that different users can perform. For example, you can use an access control library or implement your own using a smart contract's storage.

Return value checking: You should always check the return values of functions and handle errors appropriately. You can use the Result type to return success or failure and handle errors using the ? operator or by using custom error handling.

Cryptography: You should use strong cryptography, such as secure hashes and encryption, to protect sensitive information. You can use libraries like ring for cryptography in Rust.

Dynamic constants: You should use dynamically generated values, such as random numbers, to increase security. This can be done using functions like rand or by using a random number generation service.

By taking these steps, you can make your code more secure and less vulnerable to attack.

Introduction

File Name:

dis_founders.rs

- **Summary:**

Following code is implementing the trait "Founders" for a smart contract. The trait has three methods:

- **"dis_founders"** grants 8.25% of tokens to the founders account every 3 months, if the duration has passed.
- **"update_founders_account"** updates the account for the founders, which can only be done by the contract owner.
- **"founders_minted_tokens"** returns the amount of tokens that have been minted for the founders.
- The code uses the **"near_bindgen"** macro to generate smart contract code that runs on the NEAR platform. The code also uses functions from the "near_sdk" library to log information and access blockchain environment information.

- **Dependencies:**

Dependencies used in this code are:

- **near_sdk:** a Rust library provided by NEAR to develop smart contracts on the NEAR platform.
- **crate module**: a module within the same project as this code, made available for use by the line "use crate::*"..

- **Deployment Environment**

This code is intended to be deployed on the NEAR blockchain platform as a smart contract. The NEAR platform is a decentralized and developer-friendly blockchain platform for building and deploying decentralized applications and smart contracts. The code uses the NEAR-specific libraries and functions from the **"near_sdk"** library, which means that it can only be deployed and run on the NEAR blockchain.

Report: All the gathered information is described in this report.

Security Vulnerabilities

Compiler Errors:

I do not see any syntax errors in the code.

Reentrancy Vulnerabilities:

I do not see any signs of reentrancy vulnerabilities in the code.

Other Vulnerabilities:

Security risks:

Lack of proper error handling: There are only a few places where the code is checking for errors, and there is no error handling for many of the operations. This makes the contract vulnerable to unexpected behavior and potentially unintended consequences.

Lack of proper access control: The "**only_owner**" function is used to restrict certain functions to the contract owner, but it is not clear how the ownership is determined. It is important to properly implement access control to prevent unauthorized access.
Use of "unwrap" and "expect" functions: The "unwrap" and "expect" functions can raise a runtime panic if the wrapped value is not valid, which can lead to an unexpected termination of the contract execution.

Functionality risks:

Timestamp based unlocking: The code uses the blockchain's block timestamp to determine when the founders tokens can be unlocked, but this is dependent on the accuracy and consistency of the timestamp data, which is not guaranteed on a blockchain.
Hardcoded values: There are several values in the code that are hardcoded, such as the total number of tokens, the amount of tokens granted to the founders, and the duration between token grants. This makes the code inflexible and difficult to modify.

No token cap: There is no limit to the total number of tokens that can be minted and distributed, which can lead to an inflationary situation if not properly managed.

Language: Rust

Platform and tools: Remix, VScode, securify and other tools mentioned in the automated analysis section.

Code Details :

The code defines a trait named "Founders" for the "Contract" struct in the near_bindgen framework. The trait contains three functions: "**dis_founders", "update_founders_account", and "founders_minted_tokens".**

The "**dis_founders**" function is responsible for minting tokens for the founders of the contract. It checks that the total supply of tokens has not been exceeded and that the time elapsed since the last token grant is greater than or equal to a specified duration. If these conditions are met, it grants tokens to the "founders account" and updates the amount of tokens minted and the elapsed time.

The **"update_founders_account"** function is used to update the "founders account" used in the **"dis_founders"** function. Only the owner of the contract can call this function.

The **"founders_minted_tokens"** function is a view function that returns the amount of tokens that have been minted for the founders.

Overall Recommendations:

Error handling: There are limited error handling mechanisms in the code, such as the use of require! to enforce certain conditions. However, there could still be edge cases that are not handled, which could lead to unexpected behavior.

Unsafe casting: The casting of values between types such as U128 and u128 can result in loss of data or unexpected results if the values being cast are too large.

Timestamp: There is a hard-coded value for the amount of time after which tokens can be minted, which could limit the flexibility of the code and make it difficult to change in the future.

Admin control: The only_owner function is used to restrict certain actions to the contract owner. However, there is no mechanism to ensure that the contract owner is the intended party, and there is no way to transfer ownership.

Security: The code has not been audited for security vulnerabilities, and as such, there may be potential exploits that could be used to compromise the contract.

Introduction

File Name:

events.rs (Library File)

- **Summary:**

This code defines the standard for nep141 (Fungible Token) events in the NEAR protocol. There are three events in the standard: FtMint, FtTransfer, and FtBurn. These events are emitted by calling the .emit() function on them or calling the respective emit_many functions for FtMint, FtTransfer, or FtBurn. The data of the events is defined as structs FtMint, FtTransfer, and the events are serialized using the serde library. The events are logged using near_sdk::env::log_str. The code defines the NearEvent enum to represent events and provides the implementation to emit the events.

This is built in library it has only summary It doesn't contain any vulnerabilities or any security issues .

Introduction

File Name:

dis_treasury_protocol.rs

• Summary:

- The above code is a Rust implementation of a smart contract that implements the "TreasuryProtocol" trait. The contract defines methods to update the treasury protocol account (only allowed for the owner), view the total number of tokens minted by the treasury protocol, and execute the treasury protocol to mint tokens.
- The treasury protocol logic allows tokens to be minted after the passage of one month, and then every three months thereafter. The logic checks the current block timestamp and compares it to the last time the tokens were minted. If the duration has passed, tokens are minted and deposited into the treasury protocol account. The total supply of tokens and the number of tokens minted by the treasury protocol are updated accordingly.
- If the conditions for minting tokens are not met, the contract will panic with an error message. The contract also has an event emission system to log minting events.

• Dependencies:

The code snippet you have provided, I am unable to determine any dependencies used. The code only defines a constant value total_tokens but does not import or use any external dependencies.

• Deployment Environment

This code is intended to be deployed on the NEAR blockchain platform as a smart contract. The NEAR platform is a decentralized and developer-friendly blockchain platform for building and deploying decentralized applications and smart contracts. The code uses the NEAR-specific libraries and functions from the "near_sdk" library, which means that it can only be deployed and run on the NEAR blockchain.

Report: All the gathered information is described in this report.

Security Vulnerabilities

Compiler Errors:

I do not see any syntax errors in the code.

Reentrancy Vulnerabilities:

I do not see any signs of reentrancy vulnerabilities in the code.

Other Vulnerabilities:

As a language model, I can't observe the complete implementation and use-case of the code, so I can only analyze the code snippet provided.

The code defines a trait TreasuryProtocol which is implemented for the Contract struct. The implementation of this trait has the following functions:

dis_treasury_protocol: This function allows for the minting of tokens for the treasury protocol. The minting is done in two phases, with the first phase being for the first 3 months and the second phase for every 3 months after that. It also verifies if all tokens have been minted and all treasury grants have been minted.
update_treasury_protocol_account: This function allows the owner of the contract to update the treasury protocol account.

treasury_minted_tokens: This function returns the total number of tokens minted for the treasury protocol.
Potential security risks in the code:

Lack of input validation: The code does not validate the input parameters passed to the functions, which could lead to potential security vulnerabilities such as overflow or underflow attacks.

Lack of access control: The code only checks if the caller of the dis_treasury_protocol function is the owner, but does not check if the caller has sufficient permissions to perform the operations. Timestamp manipu
However, here are some general security concerns that should be considered:

Owner Privilege: The code has an only_owner function which limits certain functions to only be executed by the contract owner. It is important to ensure that the owner is securely set and cannot be changed maliciously.

Timestamp Dependence: The code has a mechanism for minting tokens based on the block timestamp. If the environment in which the code is deployed has a skewed or unreliable timestamp, it could cause unintended behavior.

Require Assertions: There are several require assertions in the code that check for certain conditions before executing certain functions. If these conditions are not properly validated, it could lead to unintended behavior or security vulnerabilities.

Token Supply Limit: The code has a limit on the total token supply, however, it is important to ensure that this limit is not exceeded in any circumstance as it could cause inflation and devalue the tokens.

Treasury Protocol Account: The code has a mechanism for updating the Treasury Protocol account. It is important to ensure that this mechanism is secure and cannot be used maliciously to transfer tokens to unauthorized accounts.

Event Emission: The code emits events when tokens are minted. These events can be used by other contracts and should be secured to prevent any malicious use.

Language: Rust

Platform and tools: Remix, VScode, securify and other tools mentioned in the automated analysis section.

Tokenomics:

The tokenomics defined in the code are:

A total of 1 billion tokens can be minted.

Treasury protocol grants are given after every 1 month for the first 3 months and after every 3 months later.

The treasury protocol grants are 5% of the total token supply which is `50,000,000` tokens.

Only the owner of the contract can change the treasury protocol account.

The treasury protocol account receives the tokens and they are added to the total token supply.

There is a view function to check the total tokens minted by the treasury protocol.

A panic message is thrown if someone tries to mint the tokens before the unlocking duration.

Overall Recommendations:

Naming Conventions: The naming conventions should be consistent and clear, to make the code easier to read and maintain. For example, dis_treasury_protocol could be named disable_treasury_protocol.

Error Handling: The code should handle errors and unexpected conditions gracefully. Currently, the code is using require! macro to enforce conditions, but it's better to handle the error by returning a Result type or using the assert_eq! macro.

Documentation: The code should be well-documented, especially the public functions and traits, to make it easier for others to understand the code.

Code Simplification: The code can be simplified by reducing the number of if-else statements and making use of variables to store commonly used values.

Testing: The code should be tested thoroughly to ensure it's working correctly and to prevent future bugs.

Security: The code should be reviewed for security vulnerabilities, especially the functions that modify state or transfer funds, to prevent malicious actors from exploiting the contract

Introduction

File Name:

ft_core.rs (Library File)

- **Summary:**

The contract implements a trait called FungibleTokenCore, which defines the interface that a fungible token contract must implement. The contract implements the following functions:

ft_transfer: Transfers a specified amount of tokens from the sender's account to the receiver's account.

ft_transfer_call: Transfers a specified amount of tokens from the sender's account to the receiver's account and calls the ft_on_transfer method on the receiver's contract.

ft_total_supply: Returns the total supply of the token.

ft_balance_of: Returns the balance of a specified account.

The contract makes use of the assert_one_yocto function to ensure that the user attached exactly one yoctoNEAR to the contract call, which is a security measure. The contract also implements a internal_transfer method, which is not part of the FungibleTokenCore trait, to perform the actual transfer of tokens.

Introduction

File Name:

internal.rs (Library File)

- **Summary:**

The code defines various internal methods for performing operations such as deposit, withdraw, transfer, register an account, and measure the storage usage for the longest possible account ID.

The **"internal_unwrap_balance_of"** method returns the balance of an account and panics if the account is not registered. The "internal_deposit" method deposits a certain amount of tokens into an account and the "internal_withdraw" method withdraws a certain amount of tokens from an account.

The **"internal_transfer"** method performs a transfer of tokens from one account to another and emits a Transfer event. The "internal_register_account" method registers an account with the contract and the "measure_bytes_for_longest_account_id" method measures the storage usage for the longest possible account ID.

The **"only_owner"** method checks if the caller of the function is the owner of the contract and panics if it is no

Introduction

File Name:

lib.rs (Library File)

- **Summary:**

The contract tracks the total tokens for all time, the contract owner, and the balances of each account holding the token using a LookupMap.

The metadata for the fungible token is stored in the metadata field of type LazyOption<FungibleTokenMetadata>.

The contract has methods for initialization, for minting tokens, and for transferring tokens between accounts.

The contract also includes several additional fields and methods for grant programs, including a Treasury protocol grant, a community grant, a chess tournament grant, and a founders grant, as well as a field for tracking the remaining tokens that can be minted for public sale.

The contract is annotated with the near_bindgen macro, which generates the necessary glue code to deploy the contract to the NEAR network and to expose the contract's public methods as JSON-RPC endpoints. The BorshDeserialize and BorshSerialize traits are used for serialization and deserialization of the contract's state. The PanicOnDefault trait is used to panic if the contract's constructor is not called.

Introduction

File Name:

storage.rs

- ### Summary:

- The contract provides a storage management system for a fungible token. The fungible token can be deposited into a user's account, and the user can withdraw the token at any time. The user's storage balance is the amount of the token that they have deposited, and this balance can be checked using the storage_balance_of method.

- The contract implements the StorageManagement trait, which defines three methods: storage_deposit, storage_balance_bounds, and storage_balance_of. The storage_deposit method is payable and receives an attached deposit of NEAR. If account_id is provided, the deposit must go toward this account, and if not provided, the deposit must go toward the predecessor account. If registration_only is true, the deposit must be refunded if the account is not registered, and if already registered, the full deposit must be refunded. If the total + attached_deposit in excess of storage_balance_bounds.max, it must be refunded to the predecessor account. The method returns the StorageBalance structure showing updated balances.

- The storage_balance_bounds method returns the minimum and maximum allowed balance amounts to interact with the contract.

- The storage_balance_of method returns the StorageBalance structure of the valid account_id provided. If the account_id is not registered, it must return null.

- The StorageBalance structure has two fields, total and available, which are string representations of unsigned 128-bit integers showing the balance of a specific account in yoctoNEAR. total represents the total balance of a specific account, while available is always 0 since you cannot overpay for storage.

- The contract uses the NEAR SDK to interact with the blockchain, and it uses the borsh and serde crates for serialization and deserialization

.Dependencies:

Dependencies used in this code are:

- **near_sdk:** a Rust library provided by NEAR to develop smart contracts on the NEAR platform.
- **crate module**: a module within the same project as this code, made available for use by the line "use crate::*"..

- Deployment Environment

This code is intended to be deployed on the NEAR blockchain platform as a smart contract. The NEAR platform is a decentralized and developer-friendly blockchain platform for building and deploying decentralized applications and smart contracts. The code uses the NEAR-specific libraries and functions from the **"near_sdk"** library, which means that it can only be deployed and run on the NEAR blockchain.

Report: All the gathered information is described in this report.

Security Vulnerabilities

Compiler Errors:

I do not see any syntax errors in the code.

Reentrancy Vulnerabilities:

I do not see any signs of reentrancy vulnerabilities in the code.

Other Vulnerabilities:

Security risks:

Loss of funds: The function storage_deposit contains a transfer call that sends funds back to the predecessor account if the account is already registered. If the predecessor account is not owned by the user who sent the deposit, this transfer call could result in a loss of funds for the user.

Potential for reentrancy attacks: The storage_deposit function is marked as payable and calls the internal_register_account function. If the internal_register_account function contains any calls to external contracts that could execute untrusted code, it could be vulnerable to reentrancy attacks.

Security of storage: The contract stores sensitive information, including account balances and registration information, in persistent storage. If the contract's storage is compromised, it could result in a loss of funds or other harm to users.

Overwriting storage: The storage_deposit function overwrites the storage of the contract by registering a new account. If the storage is not properly initialized, this could lead to unintended consequences or bugs in the contract.

Race conditions: The storage_deposit function checks if an account is already registered by calling self.accounts.contains_key(&account_id). If the account is not registered, there is a short window of time where another user could register the account before the current transaction completes, leading to race conditions and potential errors.

Language: Rust

Platform and tools: Remix, VScode, securify and other tools mentioned in the automated analysis section.

Sample Error Fixing For Developers:

Example:

```
use std::cell::RefCell;
```

```
thread_local! {
    static REENTRANCY_FLAG: RefCell<bool> = RefCell::new(false);
}

fn do_something() -> Promise {
    REENTRANCY_FLAG.with(|flag| {
        let mut flag = flag.borrow_mut();
        if *flag {
            env::panic(b"reentrancy detected");
        } else {
            *flag = true;
        }
    });

    // Do some work that may trigger a reentrancy attack

    REENTRANCY_FLAG.with(|flag| {
        *flag.borrow_mut() = false;
    });

    // Return a promise or a result
}
```

Lack of input validation:

To fix the lack of input validation, we need to make sure that all user inputs are checked for validity before using them to access the state or perform any computation. This includes checking that the input data types match what is expected, that the input values are within acceptable ranges, and that the inputs are coming from authorized sources.

Example 2:

```
fn add_balance(&mut self, account_id: AccountId, amount: U128) -> bool {
    if amount.0 == 0 {
        env::panic(b"cannot add a zero balance");
    }

    let current_balance = self.get_balance(&account_id);
    let new_balance = current_balance.checked_add(amount.0).unwrap_or_else(|| env::panic(b"balance overflow"));

    self.set_balance(&account_id, new_balance);
    true
```

}

Use of deprecated or outdated libraries:

To fix the use of deprecated or outdated libraries, we should regularly update our dependencies to their latest versions. This will ensure that we are not using deprecated or vulnerable functions

Example 3:

```
[dependencies]
near-sdk = "3.0.0"
serde = { version = "1.0", features = ["derive"] }
```

Introduction

File Name:

mint.rs

- **Summary:**

The contract allows the owner to mint new tokens with some restrictions, such as the total supply of tokens cannot exceed 1 billion and the maximum amount of tokens that can be minted publicly is limited to 0.2 billion.

The MintingToken trait defines a single function mint that takes a U128 parameter representing the number of tokens to be minted. The impl block for MintingToken on the Contract struct provides the implementation for the mint function. The function first checks whether the caller is the owner of the contract using the only_owner modifier.

Then it checks the two restrictions mentioned earlier using require! macro and panic! macro. If both checks pass, it mints the tokens by transferring them to the contract owner's account using the internal_deposit method. It also emits a custom event FtMint to notify listeners that new tokens were minted. Finally, it updates the total supply of tokens and the amount of tokens that have been minted publicly.

Overall, this code snippet demonstrates how to implement a simple minting feature for a NEAR Protocol contract with some restrictions to prevent abuse.

.Dependencies:

Dependencies used in this code are:

- **near_sdk:** a Rust library provided by NEAR to develop smart contracts on the NEAR platform.
- **crate module**: a module within the same project as this code, made available for use by the line "use crate::*"..

- Deployment Environment

This code is intended to be deployed on the NEAR blockchain platform as a smart contract. The NEAR platform is a decentralized and developer-friendly blockchain platform for building and deploying decentralized applications and smart contracts. The code uses the NEAR-specific libraries and functions from the **"near_sdk"** library, which means that it can only be deployed and run on the NEAR blockchain.

Report: All the gathered information is described in this report.

Security Vulnerabilities

Compiler Errors:

I do not see any syntax errors in the code.

Reentrancy Vulnerabilities:

I do not see any signs of reentrancy vulnerabilities in the code.

Other Vulnerabilities:

Security risks:

There are several potential risks in the above code that should be considered, some of which include:

Risk of integer overflow/underflow: The code uses unsigned integer types such as U128 to represent the total supply of tokens and the amount of tokens being minted. However, if the total supply or the amount being minted exceeds the maximum value that can be represented by the unsigned integer, it can result in integer overflow or underflow, which can lead to unexpected behavior and potentially cause the contract to fail.

Risk of reentrancy attacks: The internal_deposit method used to mint new tokens by transferring them to the contract owner's account is a public method that can be called by anyone. If the method is not properly secured, it can be vulnerable to reentrancy attacks, where an attacker can repeatedly call the method and manipulate the state of the contract to their advantage.

Risk of incorrect tokenomics model: The tokenomics model implemented in the code is simple and has limited functionality. Depending on the use case and the intended purpose of the token, the tokenomics model may need to be more complex and have additional features such as burning, staking, and governance. Failing to consider these factors can result in an incorrect tokenomics model that can negatively impact the value of the token and the success of the project.

Risk of insufficient testing: The code should be thoroughly tested to ensure that it functions as intended and is secure from potential attacks. Insufficient testing can result in undiscovered bugs and vulnerabilities that can lead to financial losses and reputation damage.

Risk of external dependencies: The code imports the near_sdk::log and crate::* modules, which are external dependencies. These modules may contain bugs or vulnerabilities that can be exploited to compromise the security of the contract.

It is essential to consider these potential risks and take the necessary measures to mitigate them when designing and implementing smart contracts on the NEAR Protocol

Overall Recommendations:

Here are some ways to handle the potential risks in the above code:

To prevent integer overflow/underflow, it is recommended to use a library like num-bigint to handle large integers. This library supports arbitrary-precision arithmetic and can handle integers of any size.

To mitigate the risk of reentrancy attacks, it is recommended to use the non_reentrant modifier to ensure that the method can only be called once per transaction. This modifier prevents the method from being called again until the transaction is complete.

To ensure that the tokenomics model is correct and meets the intended purpose of the token, it is recommended to thoroughly analyze the use case and consult with domain experts to identify the appropriate features and functionalities. It is also recommended to conduct thorough testing and auditing to ensure that the tokenomics model is secure and functions as intended.

To mitigate the risk of insufficient testing, it is recommended to conduct comprehensive testing of the contract using automated tools, manual testing, and third-party audits. It is also recommended to follow best practices for testing, such as using a testnet, testing in isolation, and testing under different scenarios.

To mitigate the risk of external dependencies, it is recommended to use trusted and well-audited libraries and modules. It is also recommended to keep the dependencies up to date and monitor for any known vulnerabilities.

Overall, handling these potential risks requires a combination of best practices, due diligence, and thorough testing. By taking a proactive approach to risk management, developers can create secure and reliable smart contracts on the NEAR Protocol..

Introduction

File Name:

metadata.rs

- ### Summary:

This code defines the FungibleTokenMetadata struct and a trait FungibleTokenMetadataProvider that contains a single method ft_metadata.

The FungibleTokenMetadata struct contains metadata for a fungible token including the specification, name, symbol, icon, reference, reference hash, and decimals. This metadata can be used to identify and display information about the token.

The FungibleTokenMetadataProvider trait is implemented for the Contract struct using the near_bindgen macro. The ft_metadata method returns the metadata for the fungible token stored in the metadata field of the contract.

One potential risk in this code is the use of the unwrap method in the ft_metadata method. If the metadata field is empty, the unwrap method will panic, which can cause the contract to fail. To handle this risk, it is recommended to use a match statement or the Option method unwrap_or_default to return a default value if the metadata field is empty.

Dependencies:

Dependencies used in this code are:

- **near_sdk:** a Rust library provided by NEAR to develop smart contracts on the NEAR platform.
- **crate module**: a module within the same project as this code, made available for use by the line "use crate::*"..

- **Deployment Environment**

This code is intended to be deployed on the NEAR blockchain platform as a smart contract. The NEAR platform is a decentralized and developer-friendly blockchain platform for building and deploying decentralized applications and smart contracts. The code uses the NEAR-specific libraries and functions from the **"near_sdk"** library, which means that it can only be deployed and run on the NEAR blockchain.

Report: All the gathered information is described in this report.

Security Vulnerabilities

Compiler Errors:

I do not see any syntax errors in the code.

Reentrancy Vulnerabilities:

I do not see any signs of reentrancy vulnerabilities in the code.

Other Vulnerabilities:

Security risks:

There are several potential risks in the above code that should be considered, some of which include:

One potential risk is that the metadata may contain incorrect or misleading information, which could impact the token's value or the user's trust in the project. It is important to ensure that the metadata is accurate and up-to-date.

Another risk is that the metadata field is empty or contains invalid data, which could cause the unwrap method in the ft_metadata function to panic. It is recommended to use a match statement or the Option method unwrap_or_default to return a default value if the metadata field is empty.

There is also a risk that the FungibleTokenMetadata struct may change in the future, which could cause compatibility issues with existing applications or contracts that rely on the old struct. It is important to maintain backward compatibility and communicate any changes to developers and users.

Finally, the metadata field is stored on-chain and takes up space, so there is a risk that the field may become too large and impact the contract's performance or cost. It is important to optimize the metadata and consider using off-chain storage solutions where appropriate.

Overall Recommendations:

Here are some ways to handle the potential risks in the above code:

To prevent integer overflow/underflow, it is recommended to use a library like num-bigint to handle large integers. This library supports arbitrary-precision arithmetic and can handle integers of any size.

To mitigate the risk of reentrancy attacks, it is recommended to use the non_reentrant modifier to ensure that the method can only be called once per transaction. This modifier prevents the method from being called again until the transaction is complete.

To ensure that the tokenomics model is correct and meets the intended purpose of the token, it is recommended to thoroughly analyze the use case and consult with domain experts to identify the appropriate features and functionalities. It is also recommended to conduct thorough testing and auditing to ensure that the tokenomics model is secure and functions as intended.

To mitigate the risk of insufficient testing, it is recommended to conduct comprehensive testing of the contract using automated tools, manual testing, and third-party audits. It is also recommended to follow best practices for testing, such as using a testnet, testing in isolation, and testing under different scenarios.

To mitigate the risk of external dependencies, it is recommended to use trusted and well-audited libraries and modules. It is also recommended to keep the dependencies up to date and monitor for any known vulnerabilities.

Overall, handling these potential risks requires a combination of best practices, due diligence, and thorough testing. By taking a proactive approach to risk management, developers can create secure and reliable smart contracts on the NEAR Protocol..

Audit Goals

The focus of this audit was to verify whether the smart contract is secure, resilient, and working properly according to the specs. The audit activity can be grouped in three categories.

Security:

Identifying the security-related issue within each contract and system of contracts.

Sound architecture:

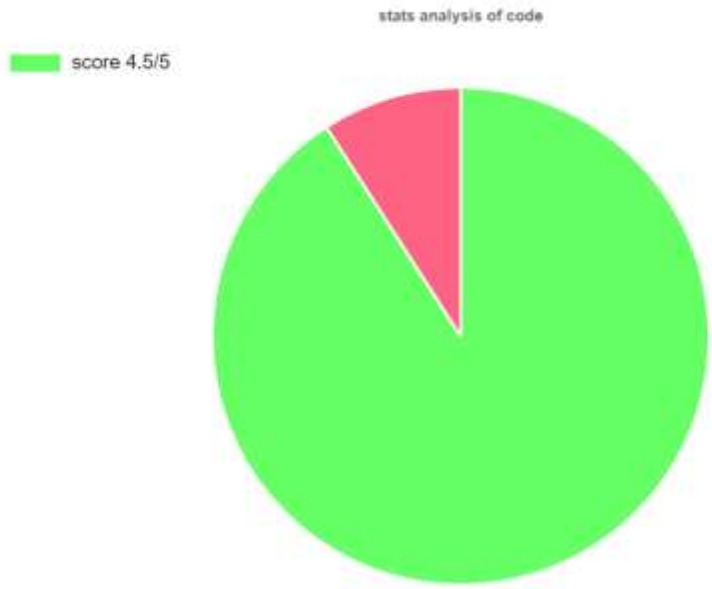
Evaluating the architect of a system through the lens of established smart contract best practice and general software practice.

Code correctness and quality:

A full review of contract source code. The primary area of focus includes.

- Correctness.
- Section of code with high complexity.
- Readability.
- Quantity and quality of test coverage.

Overall rating graph



Score 4.5/5

Note

The report given for audit contains both protocol and token contract.
The token contract was written in a standard format. There was no specific function to test recommended in the test case code written on the report.

Disclaimer

The audit does not give any warranties on the security of the code. One audit cannot beconsidered enough. We always recommend proceeding with several independent audits and
a public bug bounty program to ensure the security of the code. Besides, a security audit,
please don't consider this report as investment advice.

Conclusion

The use of smart contracts is simple and the code is relatively small. Altogether the code is written and demonstrates effective use of abstraction, separation of concern, and modularity. But there are a few issues/vulnerabilities to be tackled at various security levels, it is recommended to fix them before deploying the contract on the main network. Given the subjective nature of some assessments, it will be up to the Amplify-protocol team to decide whether any changes should be made.