Cyberscope

Audit Report

TheTrumpToken

September 2024

Repository	https://github.com/KM-TrumpToken/Smart-Contracts/tree/dev/asnan
Commit	778441c2ea487f5a472a68767e344d865492a391
Audited by	© cyberscope

Table of Contents

Risk Classification

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

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

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

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

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

Review

Repository	https://github.com/KM-TrumpToken/Smart-Contracts/tree/dev/asnan
Commit	778441c2ea487f5a472a68767e344d865492a391
Network	SOL

Audit Updates

Initial Audit	12 Sept 2024

Source Files

Filename	SHA256
programs/trump_token/src/lib.rs	d8d293cf1a8ccb57c3d9f8c68b531488587b2a949f0f3f7e9f84d591baf32c3f

Overview

TheTrumpToken’s smart contract is designed to facilitate an Initial Coin Offering (ICO) on the Solana blockchain using the Anchor framework. The contract allows users to participate in the ICO by purchasing tokens with either SOL, USDT, or USDC. It implements functions for transferring tokens from the admin's account to a program's token account (ATA) to manage the ICO, as well as for users to deposit funds and receive tokens in return.

Key functionalities include token purchasing mechanisms for multiple assets (SOL, USDT, USDC) using real-time price feeds from the Pyth network. The contract also supports the handling of administrative operations such as the withdrawal of remaining tokens after the ICO, updating ICO-related data like the end time or token price, and managing key accounts for administrators and managers.

Additionally, the contract makes use of several token-related operations, including splitting funds between administrative and funding accounts and transferring ICO tokens to users after purchase. It incorporates security checks to ensure that only the authorized accounts can execute specific functions, such as token withdrawals and updates to ICO data. Overall, the contract is designed to facilitate an ICO process while ensuring token distribution to users based on the prevailing market prices of SOL, USDT, and USDC.

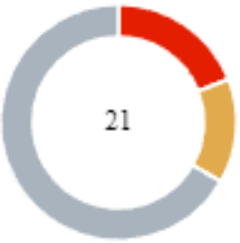
Additional Note

The configuration of the token involved in this ICO is considered out of scope. It is important to note that tokens can have a mint authority, which grants the ability to mint new tokens, potentially increasing the token supply. Additionally, there may be a freeze authority associated with the token. This authority can restrict certain wallet addresses from sending or transacting with their tokens, effectively freezing their assets. These features should be carefully reviewed and understood, as they can significantly impact the token's functionality and security.

Contract Readability Comment

The audit scope is to check for security vulnerabilities, validate the business logic, and propose potential optimizations. The contract does not adhere to key principles of Solana and Rust development regarding resource efficiency, code readability, and proper data structures. Given these issues, it cannot be considered production-ready. The development team is strongly advised to re-evaluate the business logic and follow Solana and Rust best practices. It is recommended to optimize resource usage, simplify function definitions, and use descriptive variable names to enhance auditability, efficiency, and security.

Findings Breakdown



Critical	4
Medium	3
Minor / Informative	14

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	4	0	0	0
Medium	3	0	0	0
Minor / Informative	14	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	ISF	Incorrect Scaling Factor	Unresolved
●	MOC	Misconfigured Ownership Checks	Unresolved
●	PAR	Potential Arbitrage Risk	Unresolved
●	PCPL	Price Calculation Precision Loss	Unresolved
●	IWM	Incomplete Withdrawal Mechanism	Unresolved
●	ISTR	Inconsistent Sold Tokens Reset	Unresolved
●	MAC	Missing Admin Check	Unresolved
●	PCR	Program Centralization Risk	Unresolved
●	ESA	Excessive Space Allocation	Unresolved
●	HTD	Hardcoded Token Decimals	Unresolved
●	ID	Inconsistent Documentation	Unresolved
●	IPC	Inconsistent Permission Checks	Unresolved
●	IEH	Incorrect Error Handling	Unresolved
●	IOA	Inefficient Operator Assignment	Unresolved

●	IDP	Integer Division Precision	Unresolved
●	MNV	Misleading Naming Variable	Unresolved
●	PIR	Potential Insolvency Risk	Unresolved
●	POCR	Potential Ownership Checks Redundancy	Unresolved
●	UA	Unused Assignment	Unresolved
●	UPV	Unused Phase Variable	Unresolved
●	VAUD	Vulnerabilities and Unmaintained Dependencies	Unresolved

ISF - Incorrect Scaling Factor

Criticality	Critical
Location	lib.rs#L34,148
Status	Unresolved

Description

The contract defines a scaling constant `SCALE` with the value `1_000_00`, which appears to be a misplacement of underscores that also results in an incomplete value. The correct scaling factor should be `1_000_000` to account for the correct number of decimal places in the token calculations. This scaling factor is used in the `buy_with_sol` function to calculate the amount of tokens the user should receive based on the price of SOL and the ICO token price. Due to the incorrect value of SCALE, the calculation results in an amount that is 10 times smaller than expected. Consequently, users will receive significantly fewer tokens than intended when purchasing tokens with SOL. This results in incorrect token distribution and impacts user trust and participation in the ICO.

```
pub const SCALE: u64 = 1_000_00;

let amount_in_usdt = display_price * SCALE;
```

Recommendation

The scaling factor should be reviewed and updated to accurately reflect the intended precision for token calculations. Specifically, the value of `SCALE` should be adjusted to ensure the correct number of tokens is distributed to users based on the SOL price and ICO token price.

MOC - Misconfigured Ownership Checks

Criticality	Critical
Location	lib.rs#L367
Status	Unresolved

Description

The `withdraw` function in the contract enforces that both the admin and the manager must match the values stored in the contract's state, meaning the caller must be both the admin and the manager at the same time. This creates an unrealistic condition where the function becomes inaccessible unless the admin and manager are the same person. This restricts the flexibility and usability of the function, as it doesn't account for scenarios where either the admin or the manager should be able to execute the withdrawal. By requiring both roles to match the caller, the current implementation introduces a significant limitation, potentially blocking legitimate users from performing the withdrawal.

```
pub fn withdraw(
        ctx: Context<WithDraw>,
        _ico_ata_for_ico_program_bump: u8,
        token_amount: u64,
    ) -> ProgramResult {
        if ctx.accounts.data.admin != ctx.accounts.admin.key()
{
            return Err(ProgramError::IllegalOwner);
        }
        if ctx.accounts.data.manager !=
ctx.accounts.manager.key() {
            return Err(ProgramError::IllegalOwner);
        }

        let ico_mint_address = ctx.accounts.ico_mint.key();
        let seeds = &[ico_mint_address.as_ref(),
&[_ico_ata_for_ico_program_bump]];
        let signer = [&seeds[..]];
        let cpi_ctx = CpiContext::new_with_signer(
            ctx.accounts.token_program.to_account_info(),
            token::Transfer {
                from:
ctx.accounts.ico_ata_for_ico_program.to_account_info(),
                to:
ctx.accounts.ico_ata_for_user.to_account_info(),
                authority:
ctx.accounts.ico_ata_for_ico_program.to_account_info(),
            },
            &signer,
        );
        token::transfer(cpi_ctx, token_amount)?;
        Ok(())
    }
```

Recommendation

It is recommended to revise the ownership checks in the withdraw function to allow either the admin or the manager to perform the action, depending on the project's intended authorization structure. This will ensure that the function can be accessed by the correct entities and avoid unnecessary restrictions on its use.

PAR - Potential Arbitrage Risk

Criticality	Critical
Location	lib.rs#L122,216,292
Status	Unresolved

Description

The contract allows users to buy tokens at a price set by the admin, and these tokens are immediately transferred to the user upon purchase. While the token may not initially have a market price at the time of the ICO launch, an arbitrage risk could arise if liquidity is later added to external markets (such as decentralized exchanges) where the token could be traded. If the ICO price is lower than the token's price in these external markets, users could exploit the price discrepancy by purchasing tokens cheaply from the ICO and selling them at a higher price elsewhere.

```
pub fn buy_with_sol(
        ctx: Context<BuyWithSol>,
        _ico_ata_for_ico_program_bump: u8,
        sol_amount: u64
    ) -> ProgramResult {
    ...

 pub fn buy_with_usdt(
        ctx: Context<BuyWithUsdt>,
        _ico_ata_for_ico_program_bump: u8,
        usdt_amount: u64,
    ) -> ProgramResult {
    ...

pub fn buy_with_usdc(
    ctx: Context<BuyWithUsdc>,
    _ico_ata_for_ico_program_bump: u8,
    usdc_amount: u64,
) -> ProgramResult {
    ...
```

Recommendation

To mitigate this arbitrage risk, additional functionality should be implemented to control the immediate transfer of tokens. A potential solution is to introduce a `claim` function, allowing users to claim their tokens only after the ICO's end time. This would delay token distribution and reduce the risk of arbitrage by ensuring tokens cannot be immediately traded on external markets. The system should also keep track of each user's purchased tokens and allow them to claim these tokens only after the ICO concludes. However, introducing such a delay might also introduce a risk of insolvency if the admin withdraws tokens from the contract. To address this, safeguards should be implemented to prevent the admin from withdrawing more tokens than are needed to fulfill user claims. This will ensure that the contract remains solvent and can distribute tokens to users as promised.

PCPL - Price Calculation Precision Loss

Criticality	Critical
Location	lib.rs#L143
Status	Unresolved

Description

In the `buy_with_sol` function, the contract retrieves the price of SOL/USD from an oracle and converts it to a usable format by dividing the raw price value by a power of 10 based on the exponent provided by the oracle. However, the calculation uses integer division, which discards any fractional values and leads to the loss of precision in the price. As a result, when the price contains decimals, such as 123.46, only the integer part, 123, is retained, which could lead to inaccurate calculations. This can potentially result in users receiving an incorrect amount of tokens during purchases, particularly in cases where precise pricing is critical.

```
let display_price = u64::try_from(price.price).unwrap()
            / 10u64.pow(u32::try_from(-price.exponent).unwrap());
```

Recommendation

It is recommended to handle the price conversion in a way that preserves decimal precision. This can be done by ensuring that fractional values are correctly accounted for in the price calculations. This will improve the accuracy of the token distribution and ensure the integrity of the pricing logic throughout the contract.

IWM - Incomplete Withdrawal Mechanism

Criticality	Medium
Location	lib.rs#L367
Status	Unresolved

Description

The `withdraw` function allows the admin and manager to withdraw tokens from the contract, but there are issues in its current implementation. Firstly, there is no check to ensure that this function is only called after the ICO has ended. Without verifying that the current time is past the ICO's designated end time, tokens can be withdrawn prematurely, potentially undermining the integrity of the ICO process. Additionally, the function does not update important state variables such as the total token supply or the amount of tokens sold. This can create a situation where the contract holds an incorrect view of the available token balance, leading to discrepancies between the actual token holdings and what the contract records.

```
pub fn withdraw(
        ctx: Context<WithDraw>,
        _ico_ata_for_ico_program_bump: u8,
        token_amount: u64,
    ) -> ProgramResult {
        if ctx.accounts.data.admin != ctx.accounts.admin.key()
{
            return Err(ProgramError::IllegalOwner);
        }
        if ctx.accounts.data.manager !=
ctx.accounts.manager.key() {
            return Err(ProgramError::IllegalOwner);
        }

        let ico_mint_address = ctx.accounts.ico_mint.key();
        let seeds = &[ico_mint_address.as_ref(),
&[_ico_ata_for_ico_program_bump]];
        let signer = [&seeds[..]];
        let cpi_ctx = CpiContext::new_with_signer(
            ctx.accounts.token_program.to_account_info(),
            token::Transfer {
                from:
ctx.accounts.ico_ata_for_ico_program.to_account_info(),
                to:
ctx.accounts.ico_ata_for_user.to_account_info(),
                authority:
ctx.accounts.ico_ata_for_ico_program.to_account_info(),
            },
            &signer,
        );
        token::transfer(cpi_ctx, token_amount)?;
        Ok(())
    }
```

Recommendation

It is recommended to implement a check that ensures the withdrawal can only occur after the ICO end time. This helps maintain the fairness of the ICO and prevents premature token withdrawals. Furthermore, the contract should update the relevant state variables when tokens are withdrawn, ensuring that the contract accurately reflects the token balances at all times. By doing so, the contract will maintain consistency between the state variables and the actual token balances, preventing any potential exploitation or incorrect accounting during or after the ICO.

ISTR - Inconsistent Sold Tokens Reset

Criticality	Medium
Location	lib.rs#L400
Status	Unresolved

Description

The `update_data` function appears to reset the `amount_sold` variable to zero while updating other important ICO parameters, such as the token price in USDT and the ICO phase. This behavior suggests that the function is intended to change or reset phases of the ICO. However, the contract lacks explicit functionality for managing different ICO phases, and the reset of the `amount_sold` variable does not align with any clear phase-transition logic. Without a proper phase-handling mechanism, resetting `amount_sold` can lead to inaccurate reporting of token sales and create a discrepancy between the actual token supply and the contract's records. This might cause issues such as selling more tokens than are available or incorrectly reflecting the progress of the ICO.

```
pub fn update_data(ctx: Context<UpdateData>, total_amount:
u64,end_time: i64, usdt_price: u64,phase: u64) -> ProgramResult
{
        if ctx.accounts.data.manager !=
*ctx.accounts.manager.key {
            return Err(ProgramError::IncorrectProgramId);
        }
        let data = &mut ctx.accounts.data;

        data.end_time = end_time;
        data.usdt = usdt_price;
        data.total_amount = data.total_amount + total_amount -
data.amount_sold;
        data.amount_sold = 0;
        data.phase_id = phase;
        msg!("update ico time {}, {} and USDT/ICO {}",
total_amount,end_time, usdt_price);
        Ok(())
    }
```

Recommendation

It is recommended to introduce a clear mechanism for managing phases in the ICO, ensuring that token sales data is correctly tracked across phases. The reset of `amount_sold` should only occur if it is part of a well-defined phase transition process. Additionally, the contract should maintain a consistent record of the total tokens sold throughout all phases of the ICO to prevent any misrepresentation of sales data. Lastly, the `end_time` variable should be managed carefully. The next phase should take place after the `end_time` of the previous phase, and checks should be implemented that the new phase's `end_time` is after the previous ones.

MAC - Missing Admin Check

Criticality	Medium
Location	lib.rs#L44
Status	Unresolved

Description

The `create_ico_ata` function lacks a proper authorization check, allowing any external entity to call it and initialize key aspects of the contract. This function is critical as it sets important variables such as the token amounts, end time, and admin addresses. Without a proper check to restrict who can invoke this function, an attacker could potentially call this function first after deployment, effectively taking control of the contract. This exposes the project to a significant risk of ownership theft, where a malicious actor could seize control of the ICO setup.

```
pub fn create_ico_ata(
        ctx: Context<CreateIcoATA>,
        ico_amount: u64,
        end_time:i64,
        usdt_price: u64,
        usdt_ata_for_admin: Pubkey,
        manager: Pubkey,
        phase: u64,
        usdc_ata_for_admin: Pubkey,
        funding_account: Pubkey,
        usdt_ata_for_funding_account: Pubkey,
        usdc_ata_for_funding_account: Pubkey
    ) -> Result<()> {
        msg!("create program ATA for hold ICO");
        // transfer ICO admin to program ata
        let cpi_ctx = CpiContext::new(
            ctx.accounts.token_program.to_account_info(),
            token::Transfer {
                from:
ctx.accounts.ico_ata_for_admin.to_account_info(),
                to:
ctx.accounts.ico_ata_for_ico_program.to_account_info(),
                authority:
ctx.accounts.admin.to_account_info(),
            },
        );
        token::transfer(cpi_ctx, ico_amount)?;
        msg!("send {} ICO to program ATA.", ico_amount);


        let data = &mut ctx.accounts.data;
        data.end_time = end_time;
        data.usdt = usdt_price;
        data.total_amount = ico_amount;
        data.amount_sold = 0;
        data.admin = *ctx.accounts.admin.key;
        data.manager = manager;
        data.usdt_ata_for_admin = usdt_ata_for_admin;
        data.usdc_ata_for_admin = usdc_ata_for_admin;
        data.phase_id = phase;
        data.funding_account = funding_account;
        data.usdt_ata_for_funding_account =
usdt_ata_for_funding_account;
        data.usdc_ata_for_funding_account =
usdc_ata_for_funding_account;
        msg!("save data in program PDA.");
        Ok(())
    }
```

Recommendation

To address this issue, it is crucial to implement a proper authorization mechanism to ensure that only the intended administrator can invoke the `create_ico_ata` function. A solution could involve hardcoding an initial admin address that is checked before allowing the function to proceed. This would prevent any unauthorized entities from front-running the legitimate owner during contract initialization. Properly securing the initialization function is essential to prevent unauthorized access and control over the contract.

PCR - Program Centralization Risk

Criticality	Minor / Informative
Location	lib.rs#L44,400,416
Status	Unresolved

Description

The program's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion. Specifically, the program's functionality and behavior are heavily dependent on external parameters or configurations. These functions must be executed by a specific authorized account to set and update critical parameters within the protocol.

```
pub fn create_ico_ata(
        ctx: Context<CreateIcoATA>,
        ico_amount: u64,
        end_time:i64,
        usdt_price: u64,
        usdt_ata_for_admin: Pubkey,
        manager: Pubkey,
        phase: u64,
        usdc_ata_for_admin: Pubkey,
        funding_account: Pubkey,
        usdt_ata_for_funding_account: Pubkey,
        usdc_ata_for_funding_account: Pubkey
    ) -> Result<()> {
    ...

pub fn update_data(ctx: Context<UpdateData>, total_amount:
u64,end_time: i64, usdt_price: u64,phase: u64) -> ProgramResult
{
        if ctx.accounts.data.manager !=
*ctx.accounts.manager.key {
            return Err(ProgramError::IncorrectProgramId);
        }
        let data = &mut ctx.accounts.data;

        data.end_time = end_time;
        data.usdt = usdt_price;
        data.total_amount = data.total_amount + total_amount -
data.amount_sold;
        data.amount_sold = 0;
        data.phase_id = phase;
        msg!("update ico time {}, {} and USDT/ICO {}",
total_amount,end_time, usdt_price);
        Ok(())
    }


    pub fn update_admin(ctx: Context<UpdateAdmin>,
usdt_ata_for_admin:Pubkey,new_admin: Pubkey,
new_manager:Pubkey, usdc_ata_for_admin:Pubkey) -> ProgramResult
{
        // if ctx.accounts.data.manager !=
*ctx.accounts.manager.key {
        //     return Err(ProgramError::IncorrectProgramId);
        // }
        if ctx.accounts.data.admin != *ctx.accounts.admin.key {
            return Err(ProgramError::IncorrectProgramId);
        }
        let data = &mut ctx.accounts.data;
```

```
        data.admin = new_admin;
        data.usdt_ata_for_admin = usdt_ata_for_admin;
        data.usdc_ata_for_admin = usdc_ata_for_admin;
        data.manager = new_manager;


        Ok(())
    }
```

Recommendation

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

IEH - Incorrect Error Handling

Criticality	Minor / Informative
Location	lib.rs#L138,221,297,409,428
Status	Unresolved

Description

Across the contract, multiple functions use incorrect error codes when validating certain conditions. For example, in the `buy_with_sol` , `buy_with_usdt` , and `buy_with_usdc` functions, there is a check to ensure that users cannot purchase tokens after the ICO has ended. However, these functions return different error codes for the same condition. The `buy_with_sol` function returns an InvalidArgument error when a user attempts to buy tokens after the ICO ends, while the `buy_with_usdt` and `buy_with_usdc` functions return an `IncorrectProgramId` error for the same situation. In addition to the buy functions, similar issues are present in other parts of the contract, such as the `update_data` function, where the use of `IncorrectProgramId` to validate permissions is not appropriate. The current errors do not adequately reflect the issues being checked and may cause confusion for developers managing and handling errors across the contract. The lack of consistency in error handling and the use of incorrect error codes can lead to unclear or misleading error messages, making debugging and error management more challenging. Errors like `InvalidArgument` and `IncorrectProgramId` do not accurately describe the conditions they are meant to validate, such as the end of an ICO or permission-related checks.

```
if data.end_time < Clock::get()?.unix_timestamp {
    return Err(ProgramError::InvalidArgument);
}

if ctx.accounts.data.end_time < Clock::get()?.unix_timestamp {
    return Err(ProgramError::IncorrectProgramId);
}

if ctx.accounts.data.manager != *ctx.accounts.manager.key {
    return Err(ProgramError::IncorrectProgramId);
}

if ctx.accounts.data.admin != *ctx.accounts.admin.key {
    return Err(ProgramError::IncorrectProgramId);
}
```

Recommendation

To ensure clarity and uniformity, it is recommended that the error handling across the contract be standardized. Custom errors should be introduced to more clearly describe the issues being checked. For example, a custom error indicating that the ICO has ended should be used consistently across all buy functions (`buy_with_sol` , `buy_with_usdt` , `buy_with_usdc`). Similarly, error codes should be carefully selected in other parts of the contract, such as in the `update_data` function, to ensure they accurately represent the condition being checked (e.g., permission-related checks). This approach will improve error clarity, ease debugging, and ensure consistent error handling throughout the codebase.

ESA - Excessive Space Allocation

Criticality	Minor / Informative
Location	lib.rs#L458
Status	Unresolved

Description

The program allocates more space than is necessary for the `data` account. Specifically, `data` is allocated 9000 bytes which exceeds the actual storage requirements for the data these accounts are intended to hold. Over-allocation of space leads to increased costs for account creation, as well as inefficient use of blockchain storage resources. Such practices can result in unnecessarily high transaction fees.

```
#[account(init, payer=admin, space=9000, seeds=[b"data",
admin.key().as_ref()], bump)]
pub data: Account<'info, Data>,
```

Recommendation

It is recommended to review and adjust the space allocations for both `data` account to closely match its actual data storage needs.

HTD - Hardcoded Token Decimals

Criticality	Minor / Informative
Location	lib.rs#L259,332
Status	Unresolved

Description

The contract currently uses hardcoded values to handle token decimals in several places, particularly when calculating token amounts. For example, the number `100000000` is used to represent the token's 8 decimal places. While this approach may work for tokens with fixed decimals, it reduces the contract's flexibility and maintainability. If the token's decimal places change in the future or if the contract is applied to other tokens with different decimals, this hardcoded value could lead to incorrect calculations. Relying on hardcoded values can also make the code harder to read and maintain, as it obscures the logic behind the calculations.

```
let mut ico_amount = 0;
ico_amount = (usdt_amount * 100000000) / ctx.accounts.data.usdt;

let mut ico_amount = 0;
ico_amount = (usdc_amount * 100000000) / ctx.accounts.data.usdt;
```

Recommendation

It is recommended to replace the hardcoded decimal values with a more dynamic approach that retrieves the token's decimals directly from the token's mint account. This ensures that the correct multiplier is always applied based on the token's actual decimal places. Additionally, removing hardcoded values enhances code readability and reduces the potential for errors.

ID - Inconsistent Documentation

Criticality	Minor / Informative
Location	lib.rs#L289,591
Status	Unresolved

Description

There are two instances in the contract where the documentation incorrectly refers to USDT instead of USDC. The first instance is in the comment above the `buy_with_usdc` function, which describes it as the `buy_with_usdt` function. The second instance is above the BuyWithUsdc struct, which is also labeled as a USDT-related struct. These errors in documentation can lead to confusion during development, testing, and auditing processes, especially when handling different stablecoins such as USDT and USDC.

```
/*
    ============================================================
        buy_with_usdt function use BuyWithUsdc struct
    ============================================================
*/
pub fn buy_with_usdc(
    ctx: Context<BuyWithUsdc>,

/*
    ------------------------------------------------------------
        BuyWithUsdt struct for buy_with_usdt function
    ------------------------------------------------------------
*/
#[derive(Accounts)]
#[instruction(_ico_ata_for_ico_program_bump: u8)]
pub struct BuyWithUsdc<'info> {
```

Recommendation

Ensure that all comments and documentation accurately reflect the associated functionality and structs. Specifically, the documentation above the `buy_with_usdc` function should be corrected to reflect that it handles purchases made with USDC, not USDT. Similarly, the documentation for the `BuyWithUsdc` struct should also be updated to indicate that it is related to USDC purchases. It is advisable to perform a thorough review of the entire codebase to ensure consistency and clarity in all comments and documentation.

IPC - Inconsistent Permission Checks

Criticality	Minor / Informative
Location	lib.rs#L98,142,376,379,409
Status	Unresolved

Description

The contract includes multiple instances of checks comparing account keys for authorization purposes, such as verifying the admin and manager keys. However, there is inconsistency in how these keys are accessed in different functions. In some cases, the `.key()` method is used to access the public key, while in other instances, direct dereferencing of the key field is applied. This inconsistency can lead to confusion and reduces the overall readability of the code. It is essential to maintain a uniform pattern when accessing keys to ensure clarity and to prevent potential misunderstandings in how accounts are validated throughout the contract.

```
if ctx.accounts.data.admin != *ctx.accounts.admin.key {
    return Err(ProgramError::IncorrectProgramId);
}

if ctx.accounts.data.admin != ctx.accounts.admin.key() {
    return Err(ProgramError::IllegalOwner);
}

if ctx.accounts.data.manager != ctx.accounts.manager.key() {
    return Err(ProgramError::IllegalOwner);
}

if ctx.accounts.data.manager != *ctx.accounts.manager.key {
    return Err(ProgramError::IncorrectProgramId);
}
```

Recommendation

It is recommended to standardize the key access approach across the entire codebase. The most commonly used pattern should be applied consistently for better readability and maintainability. This helps avoid confusion and reduces the likelihood of errors. Specifically, using the `.key()` method where applicable is preferable, as it is cleaner and aligns with general coding practices in Solana programs.

IOA - Inefficient Operator Assignment

Criticality	Minor / Informative
Location	lib.rs#L203,281,356
Status	Unresolved

Description

A manual implementation of an assign operation is used to update the state variable `amount_sold` . Instead of using a shorthand operation that directly increments the variable, the contract repeatedly applies a more verbose and less efficient method. This practice increases the complexity of the code unnecessarily and goes against best practices in Rust development.

```
data.amount_sold = data.amount_sold + ico_amount;
```

Recommendation

The contract should be updated to use the appropriate shorthand assignment operators where applicable. This improves readability, reduces the chance for mistakes in arithmetic operations, and aligns with Rust's best practices.

IDP - Integer Division Precision

Criticality	Minor / Informative
Location	lib.rs#L156,231,306
Status	Unresolved

Description

The calculation for distributing token amounts between multiple accounts involves dividing an integer value, such as `usdc_amount`, by 2. However, integer division in Rust discards any fractional remainder, which can lead to a loss of precision. For example, if the value is odd, the division truncates the fractional part, potentially resulting in a discrepancy where the total amount distributed is less than the original value. This may cause an unintended loss of tokens and create imbalance in the token distribution, especially in cases where high precision is important, such as in financial transactions.

```
let funding_amount = sol_amount / 2;

let amount_share = usdt_amount/2;

let amount_share = usdc_amount / 2;
```

Recommendation

To prevent this issue, the contract should implement a mechanism that ensures the full value is distributed accurately. This can be achieved by explicitly accounting for any remainder when performing such division, so that no portion of the original token amount is lost during the transfer process.

MNV - Misleading Naming Variable

Criticality	Minor / Informative
Location	lib.rs#L48,73,340
Status	Unresolved

Description

The variable `usdt_price`, defined in the `create_ico_ata` function and stored in the data structure, is intended to represent the price of the ICO token in USD. However, it is misleadingly named as `usdt_price`, which implies that it is only applicable to USDT transactions. In reality, this value is used across different functions, including `buy_with_usdc`, where the token price is calculated using USDC. This inconsistency in naming could lead to confusion for developers and auditors, as the variable does not strictly relate to USDT but is instead a generic USD price. As a result, this naming convention increases the risk of incorrect assumptions about the functionality.

```
usdt_price: u64,

data.usdt = usdt_price;

ico_amount = (usdc_amount * 100000000) /
ctx.accounts.data.usdt;
```

Recommendation

To prevent confusion and ensure clarity in the codebase, the variable `usdt_price` should be renamed to `usd_price` or a similarly appropriate name that reflects its usage across multiple token types. This change will make it clear that the value represents the price of the ICO token in USD, regardless of whether the purchase is made using USDT or USDC. Clear variable naming is crucial for maintainability and understanding.

PIR - Potential Insolvency Risk

Criticality	Minor / Informative
Location	lib.rs#L400
Status	Unresolved

Description

The `update_data` function allows the manager to modify key parameters of the ICO, including the total amount of tokens, without performing any token transfers. This function directly updates the `total_amount` by adding the provided input to the current total. However, no corresponding transfer of tokens occurs to reflect this increase in the program's actual token holdings. As a result, the smart contract may report a higher token balance than what it actually holds. This discrepancy could lead to insolvency, where the contract believes it has more tokens than it actually possesses. This issue may result in the contract failing to fulfill token purchases or withdrawals once the real tokens run out, causing potential losses or unmet expectations for users.

```
pub fn update_data(ctx: Context<UpdateData>, total_amount:
u64,end_time: i64, usdt_price: u64,phase: u64) -> ProgramResult
{
        if ctx.accounts.data.manager !=
*ctx.accounts.manager.key {
            return Err(ProgramError::IncorrectProgramId);
        }
        let data = &mut ctx.accounts.data;

        data.end_time = end_time;
        data.usdt = usdt_price;
        data.total_amount = data.total_amount + total_amount -
data.amount_sold;
        data.amount_sold = 0;
        data.phase_id = phase;
        msg!("update ico time {}, {} and USDT/ICO {}",
total_amount,end_time, usdt_price);
        Ok(())
    }
```

Recommendation

To avoid this inconsistency and the risk of insolvency, the contract should ensure that any increase in the `total_amount` is accompanied by a corresponding token transfer into the program's associated token account. This would ensure that the `total_amount` always accurately reflects the actual token balance held by the contract.

POCR - Potential Ownership Checks Redundancy

Criticality	Minor / Informative
Location	lib.rs#L141
Status	Unresolved

Description

The `buy_with_sol` function currently performs checks to ensure that the admin and funding_account provided by the user match the values stored in the program's state. While these checks may serve a purpose, they seem redundant because the relevant account information already exists within the program’s state. Users should not need to supply these values during the function call, as this could lead to unnecessary duplication without enhancing security. In contrast, the `buy_with_usdt` and `buy_with_usdc` functions handle account validation without relying on user-supplied values for these critical accounts, creating a potential inconsistency in how authorization is enforced across the contract.

```
pub fn buy_with_sol(
        ctx: Context<BuyWithSol>,
        _ico_ata_for_ico_program_bump: u8,
        sol_amount: u64
    ) -> ProgramResult {

        // transfer sol from user to admin
        let data = &mut ctx.accounts.data;

        let price_update = &mut ctx.accounts.price_feed;
        let price = price_update.get_price_no_older_than(
            &Clock::get()?,
            MAXIMUM_AGE,
            &get_feed_id_from_hex(FEED_ID).map_err(|_err|
ProgramError::Custom(1))?,
        ).map_err(|_err| ProgramError::Custom(1))?;
        // let current_timestamp1 =
Clock::get()?.unix_timestamp;
        if data.end_time < Clock::get()?.unix_timestamp {
            return Err(ProgramError::InvalidArgument);
        }
        if data.admin != ctx.accounts.admin.key() ||
data.funding_account != ctx.accounts.funding_account.key() {
            return Err(ProgramError::IllegalOwner);
        }

    ...
```

Recommendation

We recommend re-evaluating the purpose of these checks in the `buy_with_sol` function. If authorization of specific accounts, such as `admin` and `funding_account`, is necessary, the contract should ensure these checks are implemented robustly and consistently. Consider removing the user-supplied account inputs and instead rely solely on the program's state to perform the necessary validations. This will simplify the function and bring it in line with the approach taken in the `buy_with_usdt` and `buy_with_usdc` functions, ensuring a consistent and secure validation process across all purchase mechanisms.

UA - Unused Assignment

Criticality	Minor / Informative
Location	lib.rs#L260,335
Status	Unresolved

Description

In the contract, there is an unused assignment of the `ico_amount` variable in two instances. The value assigned to `ico_amount` is not used before being overwritten. Such assignments can lead to unnecessary computations and potential confusion during contract development, as it suggests that a part of the code is redundant.

```
let mut ico_amount =0;
    ico_amount = (usdt_amount * 100000000)/
ctx.accounts.data.usdt;

let mut ico_amount =0;
    ico_amount = (usdc_amount * 100000000) /
ctx.accounts.data.usdt;
```

Recommendation

Review the logic surrounding the ico_amount variable to determine if the initial assignment is necessary. If the initial value of `ico_amount` is not needed, consider removing the redundant assignment. Alternatively, if the initial value serves a purpose, ensure it is properly used before any subsequent overwriting. Addressing this issue will improve the contract's efficiency.

UPV - Unused Phase Variable

Criticality	Minor / Informative
Location	lib.rs#51,80,693
Status	Unresolved

Description

The code defines a `phase` and `phase_id` variable, which are typical in ICOs that involve multiple phases with varying conditions such as pricing, token availability, or incentives for early participants. However, despite the presence of these variables, they are not utilized anywhere in the contract logic to enforce specific behavior based on the phase of the ICO. The code does not implement any functionality that adjusts the token sale conditions depending on the current phase, making the `phase` and `phase_id` effectively redundant.

```
phase: u64

data.phase_id = phase;

pub struct Data {
    pub phase_id: u64,
```

Recommendation

It is recommended to either implement proper logic that adjusts the ICO behavior according to different phases (e.g., changing token prices, limiting token amounts per phase, etc.) or remove the `phase` and `phase_id` variables if they are not intended to play an active role in the ICO. This would simplify the contract and reduce any potential misunderstandings or expectations regarding phase-based functionality.

VAUD - Vulnerabilities and Unmaintained Dependencies

Criticality	Minor / Informative
Status	Unresolved

Description

The project relies on several third-party crates that contain known security vulnerabilities or have been marked as unmaintained. These vulnerabilities could expose the project to security risks, including timing attacks, denial of service, infinite loops, and configuration corruption. Additionally, using unmaintained dependencies can introduce further risks, as they may no longer receive security patches or updates. These issues affect critical crates within the dependency tree, potentially impacting the overall security and stability of the project. Such issues can be identified by running a tool like `cargo audit`, which helps detect vulnerabilities in dependencies.

Recommendation

Review all identified vulnerabilities and unmaintained dependencies in the project's third-party crates. It is recommended to update the affected crates to the latest secure versions where possible, or consider alternative libraries that are actively maintained. Regularly monitoring and auditing third-party dependencies with tools such as `cargo audit` is crucial to ensure the project's security and integrity, reducing exposure to potential risks.

Summary

TheTrumpToken implements an ICO mechanism for the project’s token. This audit investigates security issues, business logic concerns, and potential improvements.

Disclaimer

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

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

About Cyberscope

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

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

The Cyberscope team

cyberscope.io