Cyberscope

A TAC Security Company

Audit Report
PIKAMOON

July 2025

Repository : https://github.com/orbit-cosmos/Bridge

Commit : a9b546e777ad5dd05d0846435bf8d90bef56cdf4

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/orbit-cosmos/Bridge
Commit	a9b546e777ad5dd05d0846435bf8d90bef56cdf4

Audit Updates

Initial Audit	23 Jul 2025

Source Files

Filename	SHA256
lib.rs	8dc02a84e4f26d0d5bc12aa47773af563bd5ad10e224f0ce08b8d29069bb632d

Overview

The `lock_tokens_program` is a Solana smart contract built using the Anchor framework, designed to facilitate a token locking and distribution system. It allows users to deposit tokens into a vault during a specified window, which can later be swapped for payout tokens at a `1:1` ratio after a claim phase is opened by the contract owner. The program enforces strict access controls, time-based deposit and claim windows, and includes safety features like minimum deposit amounts, decimal validation, and emergency pause functionality. The contract supports secure token transfers, burns unclaimed tokens post-claim window, and emits events for transparency. It is designed for secure and controlled token swaps with robust error handling and state management.

Admin Functionality

The contract provides several owner-only functions to manage the token locking process. The `initialize` instruction sets up the `LockerConfig` with the deposit mint, deposit window, and claim window duration, restricted to an authorized initializer. The `set_payout_mint` instruction, callable only after the deposit window ends, assigns the payout mint and vault, ensuring decimal compatibility with the deposit mint to prevent mismatches. The `fund_payout_vault` instruction allows the owner to fund the payout vault with tokens matching the total deposited amount, while `open_claims` enables the claim phase post-deposit window. The `burn_unclaimed` function allows the owner to burn any unclaimed tokens after the claim window expires. Additionally, `pause` and `unpause` functions provide emergency control to halt or resume contract operations, ensuring governance flexibility. All admin actions are protected by ownership checks and time-based constraints.

Deposit

Users can deposit tokens into the contract's vault during the active deposit window using the `deposit` instruction. Deposits must meet a minimum amount (100 tokens with 8 decimals) to prevent dust attacks and cannot exceed the maximum total deposit limit (50 billion tokens with 8 decimals). The contract tracks each user's deposits in a `UserDeposit` account, updates the total deposited amount in `LockerConfig`, and transfers tokens to the deposit vault using the SPL Token program. Deposits are blocked if the claim phase is active or the contract is paused, ensuring controlled deposit periods.

Fund Payout Vault

The `fund_payout_vault` instruction enables the contract owner to fund the payout vault with payout tokens matching the total deposited amount, ensuring a `1:1` swap ratio. This can only occur after the deposit window ends and the payout mint is set, with checks to prevent re-funding and ensure the correct mint is used. The tokens are transferred from the owner's associated token account to the payout vault, marking the vault as funded and enabling claims.

Open Claims

The `open_claims` instruction, restricted to the owner, activates the claim phase after the deposit window closes and the payout vault is funded. It sets the claim window duration based on the configured minutes, allowing users to claim their payout tokens within this period. The instruction ensures the contract is not paused and the payout mint is set, maintaining secure state transitions.

Claim

Users can claim their payout tokens during the claim window using the `claim` instruction. The contract verifies the user is the original depositor, the claim phase is active, the payout vault is funded, and the claim window has not expired. It transfers the user's deposited amount (in payout tokens) from the payout vault to the user's associated token account, updates the total claimed amount, and closes the `UserDeposit` account to reclaim rent. The instruction uses PDA signer seeds for secure token transfers and includes checks to prevent over-claiming or reentrancy attacks.

Burn Unclaimed

After the claim window expires, the owner can call `burn_unclaimed` to burn any remaining tokens in the payout vault. This instruction ensures the claim window is closed, the contract is not paused, and the payout mint is set. It uses the SPL Token program's `burn` function with PDA signer seeds to destroy unclaimed tokens, ensuring no tokens remain locked in the vault indefinitely.

Pause and Unpause

The `pause` and `unpause` instructions allow the owner to halt or resume contract operations in emergencies. Pausing prevents deposits, claims, funding, mint setting, or burning, while unpausing restores normal functionality. Both actions are restricted to the owner and include checks to avoid redundant state changes, ensuring secure and controlled contract management.

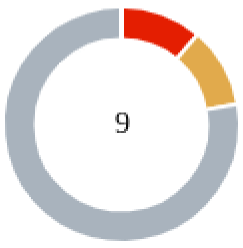
Reward Distribution Mechanism

The lock_tokens_program does not implement a traditional reward distribution system but instead facilitates a `1:1` token swap mechanism. Users deposit tokens into a vault, and after the deposit window closes, the owner funds the payout vault with an equivalent amount of payout tokens. During the claim phase, users receive payout tokens proportional to their deposited amount, with no additional rewards or weighting applied. The system ensures fairness by enforcing a direct `1:1` correspondence between deposited and claimed tokens, validated through strict accounting and vault balance checks.

Summary

The `lock_tokens_program` provides a secure, owner-managed system for locking tokens and distributing payout tokens on a `1:1` basis, with robust safeguards like time-based windows, minimum deposit thresholds, and emergency pause functionality. This audit examines the contract's security, business logic, and potential optimizations to ensure it is production-ready, focusing on correctness, efficiency, and resilience against common vulnerabilities.

Findings Breakdown



●	Critical	1
●	Medium	1
●	Minor / Informative	7

	Severity	Unresolved	Acknowledged	Resolved	Other
●	Critical	1	0	0	0
●	Medium	1	0	0	0
●	Minor / Informative	7	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	PGA	Potential Griefing Attack	Unresolved
●	IOI	Inconsistent Owner Initialization	Unresolved
●	CR	Code Repetition	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	DBPI	Deposit Before Payout Initialization	Unresolved
●	FDP	Fixed Decimal Precision	Unresolved
●	FER	Fixed Exchange Ratio	Unresolved
●	IDT	Inaccessible Deposited Tokens	Unresolved
●	TSI	Tokens Sufficiency Insurance	Unresolved

PGA - Potential Griefing Attack

Criticality	Critical
Location	lib.rs#L471
Status	Unresolved

Description

The contract includes functionality designed to enforce specific conditions on transactions. However, this design is vulnerable to griefing attacks, where malicious actors can exploit the contract's logic to interfere with legitimate user operations.

In this case, the contract implements checks that prevent the `payout_vault` from being funded if it holds a non-zero balance. Nevertheless the account is initialized in a seperate instance during a call of the `set_payout_mint` method. As a result a malicious user could transfer tokens to this account preventing the admin from finalizing the funding process. This could lead to inconsistencies such as locked funds for the users.

```
#[account(
mut,
seeds = [b"payout-vault", config.key().as_ref()],
bump,
constraint = payout_vault.amount == 0 @ LockerError::PayoutAlreadyFunded,
constraint = payout_vault.owner == config.key() @ LockerError::InvalidVault,
constraint = payout_vault.mint == config.payout_mint @
LockerError::WrongMint,
)]
pub payout_vault: Account<'info, TokenAccount>,
```

Recommendation

The team is advised to review the described mechanism to ensure that all legitimate operations are processed as intended. This will help maintain the integrity of user activities and strengthen trust in the system. Specifically, aforementioned checks may be redundant resulting in exposure to potential risks.

IOI - Inconsistent Owner Initialization

Criticality	Medium
Location	lib.rs#L351
Status	Unresolved

Description

The contract implements the `Initialize` instruction to allow users to create a new config account. It validates that the provided account is a PDA of the signer. However, it does not verify that the owner property of the `LockerConfig` matches the signer. If these two owner properties differ, the user may be unable to proceed with the contract's execution flow, such as setting a payout configuration, as the expected seed will differ from that of the actual signer.

```
seeds = [b"locker-config", owner.key().as_ref()]
```

```
seeds = [b"locker-config", config.owner.as_ref()],
bump = config.bump,
constraint = config.owner == owner.key() @
LockerError::Unauthorized,
```

Recommendation

The team is advised to implement constraints in the initialization process to ensure the owner property of the `LockerConfig` corresponds to the transaction signer. Specifically, the team could utilize the `has_one` instruction provided by the Anchor library to enforce this validation.

CR - Code Repetition

Criticality	Minor / Informative
Location	lib.rs#L313,328
Status	Unresolved

Description

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

```
pub fn pause(ctx: Context<EmergencyPause>) -> Result<()> {
let cfg = &mut ctx.accounts.config;
require!(!cfg.paused, LockerError::AlreadyPaused);

cfg.paused = true;

emit!(PauseEvent {
owner: cfg.owner,
timestamp: Clock::get()?.unix_timestamp,
});

Ok(())
}

pub fn unpause(ctx: Context<EmergencyPause>) -> Result<()> {
let cfg = &mut ctx.accounts.config;
require!(cfg.paused, LockerError::NotPaused);

cfg.paused = false;

emit!(UnpauseEvent {
owner: cfg.owner,
timestamp: Clock::get()?.unix_timestamp,
});

Ok(())
}
```

Recommendation

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

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	lib.rs#L58,146,179,276,313,328
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

```
pub fn set_payout_mint(ctx: Context<SetPayoutMint>) -> Result<()> {...}
 pub fn fund_payout_vault(ctx: Context<FundPayoutVault>) -> Result<()>
{...}
 pub fn open_claims(ctx: Context<OpenClaims>) -> Result<()> {...}
 pub fn burn_unclaimed(ctx: Context<BurnUnclaimed>) -> Result<()> {...}
 pub fn pause(ctx: Context<EmergencyPause>) -> Result<()> {...}
 pub fn unpause(ctx: Context<EmergencyPause>) -> Result<()> {...}
```

Recommendation

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

DBPI - Deposit Before Payout Initialization

Criticality	Minor / Informative
Location	lib.rs#L58
Status	Unresolved

Description

The contract operates under the assumption that users deposit tokens prior to the configuration for a payout. If a payout mint token is not set, the assets of users may remain locked in the contract.

```
pub fn set_payout_mint(ctx: Context<SetPayoutMint>) -> Result<()> {
...
}
```

Recommendation

The team is advised to revise the protocol to ensure necessary configurations are provided before user deposits. This would enhance consistency and user trust. Alternatively, the team should consider implementing a refund mechanism in accordance with the recommendation of the finding `IDT` .

FDP - Fixed Decimal Precision

Criticality	Minor / Informative
Location	lib.rs#L7,8
Status	Unresolved

Description

The contract defines `MAX_TOTAL_DEPOSITS` and `MIN_DEPOSIT_AMOUNT` assuming a fixed decimal precision of 8 decimal points, as indicated by the provided comments. However, the decimal precision of deposited tokens may vary. This discrepancy could cause the actual deposited amounts to differ from the expected amounts.

```
const MAX_TOTAL_DEPOSITS: u64 = 5_000_000_000_000_000_000; // 50B tokens with
8 decimals
const MIN_DEPOSIT_AMOUNT: u64 = 10_000_000_000; // 100 tokens with 8 decimals
```

Recommendation

The contract should handle decimal precision gracefully. Specifically, it could utilize the `.decimals` method of the provided token mint to query the actual decimal points, ensuring accurate handling of token deposits.

FER - Fixed Exchange Ratio

Criticality	Minor / Informative
Location	lib.rs#L252
Status	Unresolved

Description

The contract facilitates a 1:1 token exchange ratio. This design lacks flexibility and discourages the use of assets with differing financial values. Additionally, it could lead to misconfigurations and potential loss of funds if tokens with varying values are provided.

```
token::transfer(
CpiContext::new_with_signer(
ctx.accounts.token_program.to_account_info(),
    Transfer {
        from: ctx.accounts.payout_vault.to_account_info(),
        to: ctx.accounts.user_payout_ata.to_account_info(),
        authority: ctx.accounts.config.to_account_info(),
        },
    signer,
    ),
claim_amount,
)?;
```

Recommendation

The team should consider scenarios where the underlying tokens have different monetary values. In such cases, it is advisable to monitor the exchange ratio using a decentralized oracle to ensure consistent fund valuation.

IDT - Inaccessible Deposited Tokens

Criticality	Minor / Informative
Location	lib.rs#L84
Status	Unresolved

Description

Tokens deposited into the contract cannot be accessed or withdrawn. Users can deposit tokens using the `deposit` method, receiving payout tokens once the `claim` method is enabled. However, the inability to withdraw deposited tokens may not align with the intended business logic.

```
token::transfer(
    CpiContext::new(
        ctx.accounts.token_program.to_account_info(),
        Transfer {
            from: ctx.accounts.user_deposit_ata.to_account_info(),
            to: ctx.accounts.deposit_vault.to_account_info(),
            authority: ctx.accounts.user.to_account_info(),
        },
    ),
    amount,
)?;
```

Recommendation

The team should review the implementation to ensure it aligns with the intended business design. If the underlying tokens hold monetary value, it may be of interest to retain access to these assets.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	lib.rs#L146
Status	Unresolved

Description

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

```
pub fn fund_payout_vault(ctx: Context<FundPayoutVault>) -> Result<()>
{
...
token::transfer(
CpiContext::new(
    ctx.accounts.token_program.to_account_info(),
    Transfer {
        from: ctx.accounts.owner_payout_ata.to_account_info(),
        to: ctx.accounts.payout_vault.to_account_info(),
        authority: ctx.accounts.owner.to_account_info(),
    },
),
    amount,
)?;
}
```

Recommendation

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

Summary

The PIKAMOON contract implements a bridge and locker mechanism. This audit examines security issues, business logic concerns, and potential improvements. PIKAMOON is an engaging project with a friendly and growing community. The smart contract analysis revealed no compiler errors but identified one critical concern. The team is advised to address these considerations to enhance overall consistency and security.

Disclaimer

About Cyberscope

Cyberscope is a TAC 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.

A ***TAC Security*** *Company*

The Cyberscope team

cyberscope.io