Composable Bridge

Security Assessment

August 12th, 2024 — Prepared by OtterSec

Akash Gurugunti	sud0u53r.ak@osec.io
Ajay Shankar Kunapareddy	d1r3wolf@osec.io
Robert Chen	r@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

Composable Finance engaged OtterSec to assess the `bridge-contract` program. This assessment was conducted between July 21st and August 8th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 14 findings throughout this audit engagement.

In particular, we identified several critical vulnerabilities, including the lack of checks to ensure that the staker has the authority to withdraw the specified amount, potentially allowing unauthorized withdrawals of all staked tokens (OS-CBG-ADV-00), and the missing validation for the token mint in the token bridging instruction, enabling transfers from any escrow account rather than verifying that the token mint matches the designated receipt token (OS-CBG-ADV-03). We also highlighted multiple high-risk issue, concerning the absence of validation of the bank account to verify if it matches the whitelisted bank addresses, allowing potential misuse by unauthorized banks (OS-CBG-ADV-03), and a missing check against the index of the Marginfi account in the deposit and withdraw instructions, enabling users to use a newly created Marginfi account with already existing mints (OS-CBG-ADV-04).

Furthermore, the rewards collection instruction deposits rewards into the rewards token account, with no mechanism to withdraw or manage these deposited rewards, essentially locking these funds (OS-CBG-ADV-05). Additionally, there is a mismatch in the deposit process, between the number of tokens deposited into the stake pool and the amount transferred from the staker's token account to the escrow token account (OS-CBG-SUG-00).

We also made recommendations for the removal of redundant and unutilized code for better maintainability and clarity (OS-CBG-SUG-02) and advised the implementation of proper validation (OS-CBG-SUG-03). We further suggested modifying the codebase for improved efficiency and mitigating potential security issues (OS-CBG-SUG-04).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/ComposableFi/bridge-contract. This audit was performed against commit d5534ec.

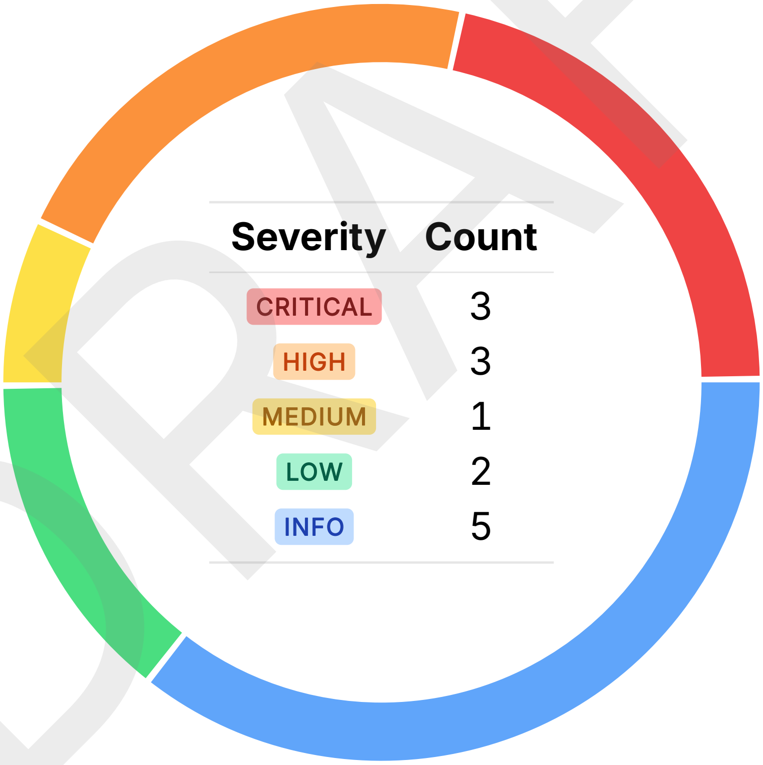
A brief description of the programs is as follows:

Name	Description
bridge-contract	The bridge program facilitates the transfer of assets between different blockchains. It integrates with external platforms for yield generation and restaking, allowing users to deposit, withdraw, and earn rewards on their assets. The program manages token whitelisting, MarginFi account creation, and administrative controls.

03 — Findings

Overall, we reported 14 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	3
HIGH	3
MEDIUM	1
LOW	2
INFO	5

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-CBG-ADV-00	CRITICAL	RESOLVED ✓	The `withdraw` instruction lacks checks to ensure that the staker has the authority to withdraw the specified amount, potentially allowing unauthorized withdrawals of all staked tokens.
OS-CBG-ADV-01	CRITICAL	RESOLVED ✓	`BridgeTokens` instruction lacks validation for `token_mint`, which allows transfers from any escrow account rather than verifying that `token_mint` matches the designated receipt token, and the `staker` account is not marked as `Signer`.
OS-CBG-ADV-02	CRITICAL	RESOLVED ✓	In the `deposit` instruction a staker may add an unbacked amount to their deposit when `deposit_sol` is false, as no tokens or lamports are actually transferred from the staker's account during the deposit process even though their `DepositState` is updated.
OS-CBG-ADV-03	HIGH	RESOLVED ✓	`deposit`, `withdraw`, and `collectRewards` instructions fail to validate that the `bank` account matches the `bank` address in `common_state.whitelisted_tokens` , which allows for potential misuse by passing unintended `banks`.

OS-CBG-ADV-04	HIGH	RESOLVED ✓	`deposit` and `withdraw` do not verify that the `marginfi_account_idx` is within the bounds of `common_state.marginfi_accounts`, potentially allowing the utilization of new Marginfi accounts with existing mints.
OS-CBG-ADV-05	HIGH	TODO	`collect_rewards` instruction deposits rewards into the `rewards_token_account`, but there is no mechanism to withdraw or manage these deposited rewards, essentially locking them in.
OS-CBG-ADV-06	MEDIUM	RESOLVED ✓	The `withdraw` instruction does not burn the receipt tokens withdrawn from the restaking program, which are meant to represent locked tokens in Marginfi, resulting in the accumulation of unburned receipt tokens.
OS-CBG-ADV-07	LOW	RESOLVED ✓	`bridge_tokens` will not execute as intended if `common_state.is_rollup_active` is false.
OS-CBG-ADV-08	LOW	RESOLVED ✓	The space calculation for `CommonState` in `UpdateTokenWhitelist` and `CreateMarginfiAccount` is inaccurate because it omits the 4-byte prefix required for storing vector lengths.

Unauthorized Withdrawals Of Staked Tokens CRITICAL OS-CBG-ADV-00

Description

The current implementation of the `withdraw` instruction does not verify if the amount being withdrawn is valid or authorized for the staker. As a result, a staker may withdraw an amount of tokens that exceeds their actual balance or claim tokens that belong to other stakers. Since there is no check to ensure that the withdrawn amount is within the staker's deposited balance, any staker may withdraw tokens staked by other users. This will result in significant security risks, including the unauthorized transfer of tokens and potential financial losses for other users.

>_ *instructions/withdraw.rs* RUST

```rust
pub fn withdraw<'a, 'info>(
    ctx: Context<'a, 'a, 'a, 'info, Withdraw<'info>>,
    amount: u64,
) -> Result<()> {
    let common_state = &mut ctx.accounts.common_state;
    let bump = common_state.bump;
    let seeds = [COMMON_SEED, core::slice::from_ref(&bump)];
    let seeds = seeds.as_ref();
    let signer_seeds = core::slice::from_ref(&seeds);
    [...]
}
```

Additionally, after the withdrawal, there is no mechanism in place to update or delete the staker's deposit record in the `deposits` state account. This implies that even after tokens are withdrawn, the state account may still reflect an outdated or incorrect balance.

Remediation

Ensure that the amount of tokens that need to be withdrawn are either in the staker's `deposit` state account or have been bridged back onto this chain by the staker to ensure the staker has authority over those tokens. Additionally, the deposit in the `deposits` state account should be deleted after the withdrawal.

Patch

Resolved in d5534ec.

Account Inconsistencies In Bridge Tokens Instruction CRITICAL OS-CBG-ADV-01

Description

In the `bridge_tokens` instruction, the `token_mint` is utilized to determine the type of token being transferred. However, the current implementation does not verify if `token_mint` is the correct mint address associated with the receipt token at the time of deposit. This lack of validation allows users to potentially transfer tokens from any escrow account, not just the intended one.

>_ instructions/bridge$_{t}okens.rs$ RUST

```rust
pub fn bridge_tokens<'a, 'info>(
    ctx: Context<'a, 'a, 'a, 'info, BridgeTokens<'info>>,
    deposit_index: u8,
) -> Result<()> {
    [...]
    let hashed_full_denom =
        lib::hash::CryptoHash::digest(ctx.accounts.token_mint.key().to_string().as_ref());
    let denom = ibc::apps::transfer::types::PrefixedDenom::from_str(
        &ctx.accounts.token_mint.key().to_string(),
    )
    .unwrap();
    let token = ibc::apps::transfer::types::Coin {
        denom,
        amount: deposit.amount.into(),
    };
    [...]
}
```

Additionally, the `staker` account in the `BridgeTokens` instruction is not explicitly marked as a `Signer`. Solana-IBC requires that the `staker` account must be a `Signer` to authorize the transfer of tokens. If the `staker` is not a `Signer`, the transaction will not be validated correctly. Furthermore, `user_receipt_escrow_account` is initialized with the `fee_payer` as its authority. However, it is more appropriate for the `staker` to be designated as the authority for this account.

Remediation

Store the `restake_receipt_token_mint` in `Deposit` and validate the `token_mint` against it. This ensures that each deposit record has a reference to the correct token mint that should be utilized. Also, declare the `staker` account as a `Signer` and modify the initialization of `user_receipt_escrow_account` so that the `staker` is set as the authority rather than the `fee_payer`.

Patch

Resolved in 337399d.

Unbacked Deposits In Stake Pool CRITICAL OS-CBG-ADV-02

Description

The vulnerability in the deposit instruction occurs when the deposit_sol flag is false. When deposit_sol is false, the code does not execute any logic to transfer tokens from the staker's account to the stake pool. Despite the lack of a token or lamport transfer, the code still adds the amount to the staker's DepositState. By incrementing DepositState, the contract records a deposit, even though no actual tokens or lamports were transferred from the staker.

>_ *instructions/deposit.rs* RUST

```rust
pub fn deposit<'a, 'info>(
    ctx: Context<'a, 'a, 'a, 'info, Deposit<'info>>,
    amount: u64,
    deposit_sol: bool,
) -> Result<()> {
    [...]
    if deposit_sol {
        if ctx.remaining_accounts.len() != LIQUID_STAKE_ACCOUNTS_LEN as usize {
            return Err(ErrorCode::InsufficientAccounts.into());
        }

        if ctx.remaining_accounts[5].key != &common_state.lst_delegation_mint {
            return Err(ErrorCode::InvalidMint.into());
        }
        [...]
    }
    [...]
}
```

Since the amount is added to the DepositState without any corresponding transfer of assets, a staker may potentially inflate their deposit balance, triggering the deposit instruction repeatedly with deposit_sol set to false, artificially increasing their stake in the system. Consequently, the staker may withdraw more than what they initially deposited, resulting in a loss of funds for the stake pool.

Remediation

Ensure that any amount added to the DepositState corresponds to an actual transfer of tokens or lamports from the user's account to the stake pool, preventing unbacked deposits.

Patch

Resolved in 435887a.

Absence Of Bank Account Validation HIGH OS-CBG-ADV-03

Description

There is a lack of validation for the `bank` account against the `whitelisted_tokens` field in `common_state` in the `deposit`, `withdraw`, and `collectRewards` instructions. Consequently, any malicious actor may utilize a different `bank` account that was not intended. Thus, rewards or deposits will be sent to this `bank` account, which might be different from the intended one

>_ *instructions/deposit.rs* RUST

```rust
#[derive(Accounts)]
pub struct Deposit<'info> {
    #[account(mut)]
    pub staker: Signer<'info>,
    #[account(mut, seeds = [COMMON_SEED], bump = common_state.bump)]
    pub common_state: Box<Account<'info, CommonState>>,
    pub token_mint: Box<Account<'info, Mint>>,
    [...]
    pub bank: UncheckedAccount<'info>,
    [...]
}
```

Remediation

Validate that the `bank` account is listed in `common_state.whitelisted_tokens` in the `deposit`, `withdraw`, and `collectRewards` instructions.

Patch

Resolved in bd8c15c.

Unverified Marginfi Account Indices HIGH OS-CBG-ADV-04

Description

In the `deposit` and `withdraw` instructions, the `marginfi_account_idx` is not validated against the index of the `marginfi_account` within the `common_state.marginfi_accounts`. The `marginfi_account_idx` is supposed to map a `mint` to a specific MarginFi account in the `common_state.marginfi_accounts` list. This mapping helps determine which MarginFi account handles operations related to the given `mint`. As a result, this oversight allows users to utilize a newly created Marginfi account with already existing mints. This misalignment will result in the incorrect allocation of funds.

>_ *instructions/deposit.rs* RUST

```rust
#[derive(Accounts)]
pub struct Deposit<'info> {
    #[account(mut)]
    pub staker: Signer<'info>,
    #[account(mut, seeds = [COMMON_SEED], bump = common_state.bump)]
    pub common_state: Box<Account<'info, CommonState>>,
    pub token_mint: Box<Account<'info, Mint>>,
    [...]
    // Marginfi accounts
    #[account(mut, constraint = common_state.marginfi_accounts.contains(marginfi_account.key))]
    /// CHECK: Address is checked above
    pub marginfi_account: UncheckedAccount<'info>,
    [...]
}
```

Additionally, when adding the check for `marginfi_account_idx`, it is also necessary to ensure that the length of the `whitelisted_token_mints` input parameter in the `initialize` instruction is less than or equal to 16. This is because if the length exceeds 16, some tokens may be incorrectly assigned to a Marginfi account that will not be able handle them due to capacity constraints.

Remediation

Ensure that `marginfi_account_idx` utilized in the `Deposit` and `Withdraw` instructions corresponds to an account listed in `common_state.marginfi_accounts`, and in the `initialize` instruction, add validation to ensure that the length of `whitelisted_token_mints` is `≤ 16`, aligning with the capacity of Marginfi accounts.

Patch

Resolved in 0227380.

Missing Rewards Withdrawal Functionality HIGH OS-CBG-ADV-05

Description

The `collect_rewards` instruction only handles the process of collecting rewards from a MarginFi lending account and depositing them into the `rewards_token_account`. There is no accompanying functionality to withdraw or manage these rewards once they are deposited. This implies that after the rewards are collected, they remain in the `rewards_token_account` without a defined way to be withdrawn or utilized. This will result in a buildup of rewards that are effectively locked in the account.

Remediation

Implement functionality that allows the withdrawal or transfer of the rewards from the `rewards_token_account`.

Failure To Burn Receipt Tokens MEDIUM OS-CBG-ADV-06

Description

In the `withdraw` instruction, tokens are withdrawn from the restaking program and deposited into the `restake_receipt_token_account`. These tokens are supposed to represent the user's stake or deposit in the restaking program. After the tokens are withdrawn from the restaking program, the code does not include a step to burn or destroy these receipt tokens. Burning these tokens is relevant because they should be invalidated once they are redeemed or unlocked from the restaking program.

>_ *instructions/withdraw.rs* RUST

```rust
pub fn withdraw<'a, 'info>(
    ctx: Context<'a, 'a, 'a, 'info, Withdraw<'info>>,
    amount: u64,
) -> Result<()> {
    [...]
    let accounts = restaking_v2_interface::instructions::WithdrawAccounts {
        staker: &common_state.to_account_info(),
        common_state: &ctx.accounts.restake_common_state.to_account_info(),
        token_mint: &ctx.accounts.restake_token_mint.to_account_info(),
        staker_token_account: &ctx.accounts.restake_receipt_token_account.to_account_info(),
        escrow_token_account: &ctx.accounts.restake_escrow_token_account.to_account_info(),
        receipt_token_mint: &ctx.accounts.restake_receipt_token_mint.to_account_info(),
        staker_receipt_token_account: &ctx
            .accounts
            .restake_staker_receipt_token_account
            .to_account_info(),
        token_program: &ctx.accounts.token_program,
        [...]
    };
    [...]
}
```

Remediation

Ensure that the receipt tokens in `restake_receipt_token_account` are burned after withdrawing tokens from the restaking program and before interacting with the MarginFi program.

Patch

Resolved in f49a0ad.

Absence Of Rollup Status Check LOW OS-CBG-ADV-07

Description

In the current implementation of the `bridge_tokens` instruction, there is no check to verify that the program is operating in an active rollup mode. If `bridge_tokens` is executed without checking if the rollup is active, it may result in unexpected errors, and the transaction would fail since, until the rollup is active, the receipt tokens will be stored in the contract, and they may be transferred only once the rollup is active.

Remediation

Add a check to verify that the rollup is active before proceeding with bridging tokens.

Patch

Resolved in ce06685.

Incorrect Space Calculation LOW OS-CBG-ADV-08

Description

The calculation of `required_space` in the `UpdateTokenWhitelist` and `CreateMarginfiAccount` instructions is currently incorrect. It is critical to ensure that the `common_state` account has sufficient space to store all its data. When storing a vector (`whitelisted_tokens` or `marginfi_accounts`), Solana's data structures require an additional four bytes to store the length of the vector. This is omitted in the space calculation of `required_space`, resulting in an underestimation of the required space for storing data about MarginFi accounts and whitelisted tokens. The anchor program structs also require 8 bytes for discriminator at the start of the storage slot. This should also be included in the storage space calculation.

```rust
let required_space = (MintWithBank::INIT_SPACE * (common_state.whitelisted_tokens.len()))
        + 32
        + 32
        + 32
        + (32 * (common_state.marginfi_accounts.len() + 1))
        + 32
        + 1 // since Pubkey has fixed size, an extra byte is used by `option`
        + 1;
```

Remediation

Include the 4-byte prefix that stores the length of the vector and the 8-bytes for the discriminator in the calculation of `required_space`. Additionally, move the `required_space` calculation to a function within `CommonState` to reduce redundancy.

Patch

Resolved in 0e8d9cc.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-CBG-SUG-00	The deposit process may result in a mismatch between the number of tokens deposited into the stake pool and the amount transferred from the staker's token account to the escrow token account.
OS-CBG-SUG-01	`deposit_sol` in the `deposit` instructions may fail on stake pools that require a `sol_deposit_authority` because it is passed as `None`.
OS-CBG-SUG-02	The codebase contains multiple cases of redundant and unutilized which should be removed for better maintainability and clarity.
OS-CBG-SUG-03	There are several instances where proper validation is not done, resulting in potential security issues.
OS-CBG-SUG-04	Recommendations for modifying the codebase to improve efficiency and mitigate potential security issues.

Token Amount Mismatch

OS-CBG-SUG-00

Description

In the `deposit` instruction, the `deposit_sol` operation deposits `SOL` into a stake pool and receives a certain amount of stake pool tokens in return. These tokens are received in an account with the mint `common_state.lst_delegation_mint`, not necessarily the same as the staker's token account. After the `SOL` deposit, the function transfers an amount of tokens from the staker's token account to the escrow token account. The amount transferred is based on the updated balance of the token account after the `SOL` deposit.

>_ *instructions/deposit.rs* RUST

```rust
pub fn deposit<'a, 'info>(
    ctx: Context<'a, 'a, 'a, 'info, Deposit<'info>>,
    amount: u64,
    deposit_sol: bool,
) -> Result<()> {
    [...]
    if deposit_sol {
        if ctx.remaining_accounts.len() != LIQUID_STAKE_ACCOUNTS_LEN as usize {
            return Err(ErrorCode::InsufficientAccounts.into());
        }

        if ctx.remaining_accounts[5].key != &common_state.lst_delegation_mint {
            return Err(ErrorCode::InvalidMint.into());
        }
        [...]
    }

    let transfer_ix = Transfer {
        from: ctx.accounts.staker_token_account.to_account_info(),
        to: ctx.accounts.escrow_token_account.to_account_info(),
        authority: ctx.accounts.staker.to_account_info(),
    };

    let cpi_ctx = CpiContext::new(ctx.accounts.token_program.to_account_info(), transfer_ix);
    [...]
}
```

However, the function does not ensure that the token account receiving the tokens (`lst_token_acc`) is the same as the staker's token account (`staker_token_account`). It also does not check that the token mint (`lst_delegation_mint`) for the received tokens is the same as the token mint (`token_mint`) utilized for the staker's token account. Consequently, the number of tokens transferred may not align with what was actually deposited into the stake pool.

Remediation

Add checks to ensure that `lst_token_acc` matches `staker_token_account` to confirm that tokens are deposited and received as expected. Also, verify that the token mint of the tokens received (`lst_delegation_mint`) is the same as the token mint of the staker's token account (`token_mint`).

Inconsistencies In Deposit Instruction OS-CBG-SUG-01

Description

In the `deposit` instruction, the `if deposit_sol` block interacts with the stake pool program to handle `SOL` deposits. In the stake pool program, `sol_deposit_authority` is an optional authority required for depositing `SOL` into the stake pool. Some stake pools are configured to require this authority for processing `SOL` deposits. `deposit_sol` passes `sol_deposit_authority` as `None`. If the stake pool being interacted with requires a `sol_deposit_authority`, this absence will result in the deposit failing.

>_ *instructions/deposit.rs* RUST

```rust
pub fn deposit<'a, 'info>(
    ctx: Context<'a, 'a, 'a, 'info, Deposit<'info>>,
    amount: u64,
    deposit_sol: bool,
) -> Result<()> {
    [...]
    if deposit_sol {
        let accounts = vec![
            [...]
            token_acc.clone(),
            ctx.remaining_accounts[4].to_account_info(),
            token_acc.clone(), // Same account as receiver token account
            ctx.remaining_accounts[5].to_account_info(),
            ctx.accounts.system_program.to_account_info(),
            ctx.accounts.token_program.to_account_info(),
            ctx.remaining_accounts[6].to_account_info(),
        ];
        [...]
    }
    [...]
}
```

Stake pools that require `sol_deposit_authority` have specific permissions and controls around who may deposit `SOL`. Passing `None` implies that no authority is provided, which is incompatible with stake pools expecting authority. When `sol_deposit_authority` is expected but not provided, the instruction will not be processed, resulting in a failed deposit operation, preventing the deposit of `SOL` into the pool.

Furthermore, the accounts vector includes `ctx.remaining_accounts[6]`, which is passed to `invoke_signed` for executing the `deposit` instruction. Since `sol_deposit_authority` is passed as `None` in `deposit_sol`, the presence of `ctx.remaining_accounts[6]` in the accounts vector is redundant. This account is not utilized in the actual deposit operation and is not necessary for executing the instruction.

Remediation

Determine if the stake pool requires `sol_deposit_authority`. If it does, pass the appropriate authority to `deposit_sol`. Also, ensure that only the necessary accounts are included in the `accounts` vector for the `deposit` instruction.

Patch

Resolved in 0227380.

Removal Of Unused And Redundant Code

OS-CBG-SUG-02

Description

1. In the `withdraw` instruction, after creating the instruction and generating the `account_infos` list, the code pushes the `bank` account and `oracle_key` account again into `ix.accounts`. Since the `bank` account is already included in `accounts.into()`, adding it again will result in duplication in the list of accounts, and thus this operation should be removed.
2. Prior to the initialization of `whitelisted_tokens`, check for duplicates to prevent the addition of similar tokens to the list. Similarly, perform this check while updating the list. Searching for a specific token in a list with duplicates will result in unintended issues and data inconsistencies.
3. In the `updates_state` instruction, the initialization of `marginfi_acc_idx` with `current_whitelisted_tokens_len / MAXIMUM_LENDING_ACC_BALANCES` should be removed as it is unnecessary because it is overwritten in the subsequent loop.

   >_ instructions/update_state.rs RUST

```rust
pub fn update_token_whitelist(
    ctx: Context<UpdateState>,
    new_tokens: Vec<MintWithBankPayload>,
) -> Result<()> {
    [...]
    let mut marginfi_acc_idx =
        current_whitelisted_tokens_len / MAXIMUM_LENDING_ACC_BALANCES as usize;
    [...]
    let token_mints_with_idx = new_tokens
        .iter()
        .enumerate()
        .map(|(index, mint)| {
            marginfi_acc_idx =
                (current_whitelisted_tokens_len + index) / MAXIMUM_LENDING_ACC_BALANCES as
                 ↪  usize;
            MintWithBank {
                mint: mint.mint,
                bank: mint.bank,
                marginfi_account_idx: marginfi_acc_idx as u8,
            }
        })
        .collect::<Vec<MintWithBank>>();
    [...]
}
```

4. In `deposit` instruction, `spl_stake_pool::instruction::deposit_sol` `CPI` requires only the `lamports_from_account` (which is the staker in this case) as a signer. The code unnecessarily includes the `common_state` account as a signer in the `CPI` call, which should be removed.

Remediation

Remove the redundant and unutilized code.

Patch

1. Issue #2 resolved in ac248d9.
2. Issue #3 resolved in 9685968.
3. Issue #4 resolved in aa3765f.

Missing Validation Logic

OS-CBG-SUG-03

Description

1. `collect_reward` utilizes `token_mint` to initialize the `rewards_token_account` instead of `emission_mint`. The MarginFi program expects the rewards to be transferred to a token account associated with the `emission_mint`. utilizing a different mint for the `rewards_token_account` would violate this expectation and result in the `CPI` call failing.

   >_ *instructions/update_state.rs* RUST

   ```rust
   pub fn collect_rewards(ctx: Context<CollectRewards>) -> Result<()> {
       [...]
       let accounts =
           ↪  marginfi_interface::instructions::LendingAccountWithdrawEmissionsAccounts {
           marginfi_group: &ctx.accounts.marginfi_group.to_account_info(),
           marginfi_account: &ctx.accounts.marginfi_account.to_account_info(),
           signer: &ctx.accounts.common_state.to_account_info(),
           bank: &ctx.accounts.bank.to_account_info(),
           emissions_mint: &ctx.accounts.emissions_mint.to_account_info(),
           emissions_auth: &ctx.accounts.emissions_auth.to_account_info(),
           emissions_vault: &ctx.accounts.emissions_vault.to_account_info(),
           destination_account: &ctx.accounts.rewards_token_account.to_account_info(),
           token_program: &ctx.accounts.token_program.to_account_info(),
       };
       [...]
   }
   ```

2. `update_token_whitelist` and `create_marginfi_account` unconditionally transfer the difference between the new and old rent requirements to the `common_state` account. This may result in transferring more lamports than necessary, especially if the account already holds more lamports than the calculated minimum balance, resulting in a waste of funds.
3. In the `withdraw` instruction, it should be verified that the `token_mint` associated with the withdrawal request is whitelisted before proceeding with the withdrawal process, to prevent unauthorized token transfers.

Remediation

1. Utilize `emission_mint` instead of `token_mint` in `rewards_token_account` checks.
2. Compare the required lamports with the existing lamports in the `common_state` account. Only the difference between the two amounts should be transferred.
3. Implement a check to verify if the `token_mint` is present in the `whitelisted_tokens` array.

Patch

1. Issue #1 resolved in 26add09.

Code Refactoring OS-CBG-SUG-04

1. Utilize `concatenated_token_list.len` instead of recalculating the length multiple times in `update_token_whitelist`.
2. In the current design of `update_token_whitelist`, deleting whitelisted tokens is not permitted. Thus, implementing a pause flag that allows tokens to be temporarily deactivated without removing them from the whitelist may be beneficial. This will allow for safer management of tokens in various scenarios, such as security or operational issues.
3. The constant is set to six currently, whereas only five accounts are utilized. It would be appropriate to update the constant accordingly.
4. The `Deposit` instruction may be optimized for improved redability by utilizing `Option` accounts for `deposit_state` and `restake_user_receipt_token_account` instead of implementing custom checks and parsing functionality.

Remediation

Modify the code as mentioned above.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.