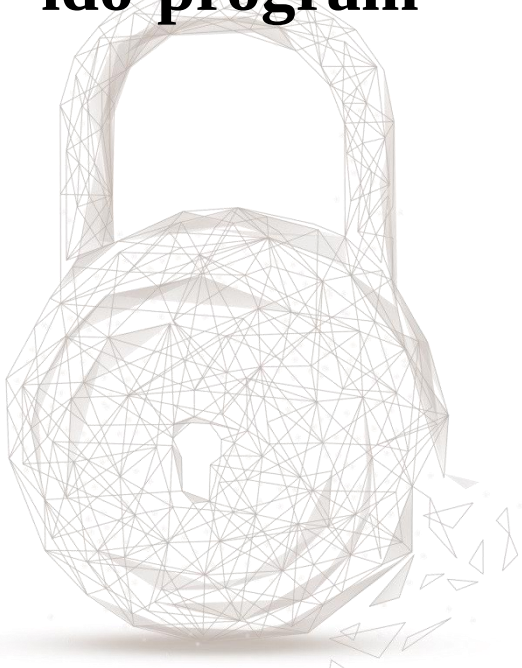
Smart Contract Audit Report

for

ido-program



Audit Number: 202111181835

Contract Name: ido

Deployment Platform: Solana

Audit contract link: https://github.com/tribeland/ido-program

Initial commit: 088040e28beba6e53908cf58b19b04bdd53b1c85

Final commit: e2dd9fe9767cbdb576a6d8003ec52230044da04f

Audit Start Date: 2021.11.01

Audit Completion Date: 2021.11.18

Audit Result: Pass

Audit Team: Beosin Technology Co. Ltd.

Audit Results Overview

Beosin Technology has used several methods including Formal Verification, Static Analysis, Typical Case Testing and Manual Review to audit three major aspects of ido smart contract, including Coding Conventions, General Vulnerability and Business Security. **After auditing, the ido smart contract was found to have 5 risk items: 2 high-risks, 2 medium-risks and 1 low-risk. As of the completion of the audit, all risk items have been fixed or properly handled. The overall result of the ido smart contract is Pass.** The following is the detailed audit information for this project.

Index	Risk items	Risk level	Status
ido-1	Any user can call the initialization function *initialize_pool* of the program at will to destroy the stored data of the corresponding accounts	High	Fixed
ido-2	Program owner can withdraw deposit tokens and share tokens from the contract at will	High	Fixed
ido-3	The administrator can update the ido data after the start of the ido (before the end_grace_phase)	Medium	Fixed
ido-4	When updating the ido data with the *update_pool* function, the amount of share tokens is not filled and removed based on the new *total_supply*.	Medium	Fixed
Ido-5	The share token calculation may be distorted due to token decimals issues	Low	Fixed

Table 1. Key Audit Findings

Risk Descriptions and Fix Results:

[ido-1 High] Multiple initialization of pool

Description: The *initialize_pool* function in the program is used to initialize the program. However, since there is no call limit, any user can call the *initialize_pool* function again to modify the pool data in the program even after ido has started.

Fix recommendations: It is recommended to limit the *initialize_pool* function to be called only once, or only by the specified account.

Fix results: Fixed. In the new version, PROGRAM_SEED has been specified and the ido account will be initialized in the *initialize_pool* function, which ensures that the function can only be called once.

[ido-2 High] Program owner can withdraw deposit tokens and share tokens from the Contract at will

Description: As shown in the figure below, the administrator can call the *owner_withdraw_deposit* and *owner_withdraw_share* functions at any time to withdraw the deposit tokens and share tokens from the contract without restriction.

```
pub fn owner_withdraw_deposit(
    ctx: Context<OwnerWithdrawDepositRequest>
) -> ProgramResult {
    let seed = ctx.accounts.ido_account.seed.as_ref();
    let seeds = &[
        seed.trim_ascii_whitespace(),
        &[ctx.accounts.ido_account.bumps.ido_account]
    ];
    let signer = &[&seeds[..]];

    let cpi_accounts = Transfer {
        from: ctx.accounts.pool_deposit_account.to_account_info(),
        to: ctx.accounts.owner_deposit_account.to_account_info(),
        authority: ctx.accounts.ido_account.to_account_info(),
    };
    let cpi_program = ctx.accounts.token_program.to_account_info();
    let cpi_ctx = CpiContext::new_with_signer(cpi_program, cpi_accounts, signer);
    token::transfer(cpi_ctx, ctx.accounts.pool_deposit_account.amount)?;

    Ok(())
}
```

Figure 1 source code of *owner_withdraw_deposit* function(before fixed)

```rust
pub fn owner_withdraw_share(
    ctx: Context<OwnerWithdrawShareRequest>
) -> ProgramResult {
    let seed = ctx.accounts.ido_account.seed.as_ref();
    let seeds = &[
        seed.trim_ascii_whitespace(),
        &[ctx.accounts.ido_account.bumps.ido_account]
    ];
    let signer = &[&seeds[..]];

    let cpi_accounts = Transfer {
        from: ctx.accounts.pool_share_account.to_account_info(),
        to: ctx.accounts.owner_share_account.to_account_info(),
        authority: ctx.accounts.ido_account.to_account_info(),
    };
    let cpi_program = ctx.accounts.token_program.to_account_info();
    let cpi_ctx = CpiContext::new_with_signer(cpi_program, cpi_accounts, signer);
    token::transfer(cpi_ctx, ctx.accounts.pool_share_account.amount)?;

    Ok(())
}
```

Figure 2 source code of owner_withdraw_deposit function(before fixed)

Fix recommendations: Limit these two functions to be called after the user redeem is completed.

Fix results: Fixed.

```rust
impl<'info> OwnerWithdrawDepositRequest<'info> {
    fn allow_withdraw(
        ido_account: &ProgramAccount<'info, IdoAccount>,
        clock: &Sysvar<'info, Clock>,
    ) -> ProgramResult {
        if clock.unix_timestamp <= ido_account.ido_times.end_redeem_phase {
            return Err(ErrorCode::WithdrawNotAllowed.into());
        }

        Ok(())
    }
}
```

Figure 3 source code of *allow_withdraw* function of *owner_withdraw_deposit* (fixed)

```rust
impl<'info> OwnerWithdrawShareRequest<'info> {
    fn allow_withdraw(
        ido_account: &ProgramAccount<'info, IdoAccount>,
        clock: &Sysvar<'info, Clock>,
    ) -> ProgramResult {
        if clock.unix_timestamp <= ido_account.ido_times.end_redeem_phase {
            return Err(ErrorCode::WithdrawNotAllowed.into());
        }

        Ok(())
    }
}
```

Figure 4 source code of *allow_withdraw* function of *owner_withdraw_share* (fixed)

[ido-3 medium] Program owner can update ido data after the start of sale

Description: As shown in the figure below, the program restricts the call time of *update_pool* function to before end_grace_phase, and the time period when the user performs the stake is between start_sale_phase and end_sale_phase. Then it means the administrator can modify the pool data after the user performs the stake.

```
impl<'info> UpdateRequest<'info> {
    fn allow_update(
        ido_account: &ProgramAccount<'info, IdoAccount>,
        clock: &Sysvar<'info, Clock>
    ) -> ProgramResult {
        if
            clock.unix_timestamp > ido_account.ido_times.end_grace_phase
        {
            return Err(ErrorCode::UpdateNotAllowed.into());
        }

        Ok(())
    }
}
```

Figure 5 source code of *allow_update* function(before fixed)

Fix recommendations: Limit the effective call time of *update_pool* function before *start_sale_phase*.

Fix results: Fixed.

```
impl<'info> UpdateRequest<'info> {
    fn allow_update(
        ido_account: &ProgramAccount<'info, IdoAccount>,
        clock: &Sysvar<'info, Clock>,
    ) -> ProgramResult {
        if clock.unix_timestamp >= ido_account.ido_times.start_sale_phase {
            return Err(ErrorCode::UpdateNotAllowed.into());
        }

        Ok(())
    }
}
```

Figure 6 source code of *allow_update* function(fixed)

[ido-4 medium] When updating the pool, the share tokens in the program are not updated according to the new *total_supply* value

Description: As shown in the figure below, the *update_pool* function does not perform the filling and removal of the number of tokens after updating the pool's *total_supply*. If the total_supply becomes larger, then it may lead to the contract not having enough share tokens on the *pool_share_account*, resulting in the user not being able to redeem the share tokens normally.

```rust
pub fn update_pool(
    ctx: Context<UpdateRequest>,
    ido_times: IdoTimes,
    min_deposit_amount: u64,
    total_supply: u64
) -> ProgramResult {
    let ido_account = &mut ctx.accounts.ido_account;

    ido_account.ido_times = ido_times;
    ido_account.min_deposit_amount = min_deposit_amount;
    ido_account.total_supply = total_supply;

    Ok(())
}
```

Figure 7 source code of *update_pool* function(before fixed)

Fix recommendations: After the program updates the total_supply, it chooses to add or remove share tokens to the pool according to the difference between before and after, so that the actual tokens owned by the contract are consistent with the total_supply record.

Fix results: Fixed.

```rust
pub fn update_pool(
    ctx: Context<UpdateRequest>,
    ido_times: IdoTimes,
    min_deposit_amount: u64,
    total_supply: u64,
) -> ProgramResult {
    let ido_account_info = ctx.accounts.ido_account.to_account_info();
    let ido_account = &mut ctx.accounts.ido_account;

    if total_supply > ido_account.total_supply {
        let cpi_accounts = Transfer {
            from: ctx.accounts.owner_share_account.to_account_info(),
            to: ctx.accounts.pool_share_account.to_account_info(),
            authority: ctx.accounts.owner.to_account_info(),
        };
        let cpi_program = ctx.accounts.token_program.to_account_info();
        let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);
        token::transfer(cpi_ctx, total_supply.checked_sub(ido_account.total_supply).ok_or(ErrorCode::AmountCalculationFailure)?)?;
    }
    else if total_supply < ido_account.total_supply {
        let seeds = &[
            PROGRAM_SEED,
            &[ido_account.bumps.ido_account],
        ];
        let signer = &[&seeds[..]];

        let cpi_accounts = Transfer {
            from: ctx.accounts.pool_share_account.to_account_info(),
            to: ctx.accounts.owner_share_account.to_account_info(),
            authority: ido_account_info,
        };
        let cpi_program = ctx.accounts.token_program.to_account_info();
        let cpi_ctx = CpiContext::new_with_signer(cpi_program, cpi_accounts, signer);
        token::transfer(cpi_ctx, ido_account.total_supply.checked_sub(total_supply).ok_or(ErrorCode::AmountCalculationFailure)?)?;
    }

    ido_account.ido_times = ido_times;
    ido_account.min_deposit_amount = min_deposit_amount;
    ido_account.total_supply = total_supply;

    Ok(())
}
```

Figure 8 source code of *update_pool* function(fixed)

[ido-5 Low] The share token calculation may be distorted due to token decimals issues

Description: As shown in the figure below, in redeem, the price of share tokens is calculated based on the ratio of the number of share tokens to the number of deposit tokens, but the token decimals is not taken into account here, which may make the calculated price distorted.

```
pub fn redeem(ctx: Context<RedeemRequest>) -> ProgramResult {
    let member = &mut ctx.accounts.member;
    let ido_account = &mut ctx.accounts.ido_account;

    let price = (ido_account.total_supply as u128).checked_div(ido_account.total_deposited as u128).ok_or(ErrorCode::AmountCalculationFailure)?;
    member.share = (member.deposit as u128).checked_mul(price).ok_or(ErrorCode::AmountCalculationFailure)? as u64;

    if member.share > 0 {
        let seed = ctx.accounts.ido_account.seed.as_ref();
        let seeds = &[
            seed.trim_ascii_whitespace(),
            &[ctx.accounts.ido_account.bumps.ido_account]
        ];
        let signer = &[&seeds[..]];

        let cpi_accounts = Transfer {
```

Figure 9 source code of *redeem* function(before fixed)

Fix recommendations: Modify the calculation here to "multiply before divide".

Fix results: Fixed. As shown in the figure below, the precision_factor has been added in the new version, which can avoid the above problem.

```
pub fn redeem(ctx: Context<RedeemRequest>) -> ProgramResult {
    let member = &mut ctx.accounts.member;
    let ido_account = &mut ctx.accounts.ido_account;

    let max_price = ido_account.max_price as u128;
    let price = (ido_account.total_deposited as u128)
        .checked_mul(ido_account.precision_factor as u128)
        .ok_or(ErrorCode::AmountCalculationFailure)?
        .checked_div(ido_account.total_supply as u128)
        .ok_or(ErrorCode::AmountCalculationFailure)?;

    if price > 0 {
        member.share = (member.deposit as u128)
            .checked_mul(ido_account.total_supply as u128)
            .ok_or(ErrorCode::AmountCalculationFailure)?
            .checked_div(ido_account.total_deposited as u128)
            .ok_or(ErrorCode::AmountCalculationFailure)? as u64;

        if price > max_price {
            member.accepted_deposit = (member.share as u128)
                .checked_mul(max_price as u128)
                .ok_or(ErrorCode::AmountCalculationFailure)?
                .checked_div(ido_account.precision_factor as u128)
                .ok_or(ErrorCode::AmountCalculationFailure)? as u64;

            let overflow_deposit = member.deposit.checked_sub(member.accepted_deposit).ok_or(ErrorCode::AmountCalculationFailure)?;

            if overflow_deposit > 0 {
```

Figure 10 source code of redeem function(fixed)

Other Audit Items Descriptions

1. ido mode

This project implements a type of ido, in which users can deposit tokens into the contract for a specified period of time, and receive the corresponding share tokens according to the percentage of stake volume when the redeem period comes.

In this mode, if the contract receives deposit tokens greater than a predetermined maximum (price > max_price), then the contract will refund the user's excess staking tokens based on max_price after issuing share tokens on a percentage basis.

2. The project owner redeems the unclaimed tokens

This ido item sets up several phases (set up before the start of the sale): sale_phase, grace_phase and redeem_phase. Users can stake in the sale_phase, revoke staking in the sale_phase and grace_phase, and redeem their share tokens at the redeem_phase. However, if the user does not redeem tokens at the end of the redeem_phase, then the tokens can be redeemed by the project party.

3. Solana feature description

Since the contract on the Solana chain supports upgrades, the results of this audit are only for the contract of the specified version of the report.

Appendix 1 Description of Vulnerability Level

Vulnerability Level	Description	Example
Critical	Vulnerabilities that lead to the complete destruction of the project and cannot be recovered. It is strongly recommended to fix.	Malicious tampering of core contract privileges and theft of contract assets.
High	Vulnerabilities that lead to major abnormalities in the operation of the contract due to contract operation errors. It is strongly recommended to fix.	Unstandardized docking of the USDT interface, causing the user's assets to be unable to withdraw.
Medium	Vulnerabilities that cause the contract operation result to be inconsistent with the design but will not harm the core business. It is recommended to fix.	The rewards that users received do not match expectations.
Low	Vulnerabilities that have no impact on the operation of the contract, but there are potential security risks, which may affect other functions. The project party needs to confirm and determine whether the fix is needed according to the business scenario as appropriate.	Inaccurate annual interest rate data queries.
Info	There is no impact on the normal operation of the contract, but improvements are still recommended to comply with widely accepted common project specifications.	It is needed to trigger corresponding events after modifying the core configuration.

Appendix 2 Description of Audit Categories

No.	Categories	Subitems
1	Coding Conventions	Basic grammatical security
		Deprecated Items
		Redundant Code
		Transaction fee
2	General Vulnerability	Integer Overflow/Underflow
		Reentrancy
		Pseudo-random Number Generator (PRNG)
		DoS (Denial of Service)
		Function Call Permissions
		Account signature check
		Arithmetic accuracy vulnerability
		External call audit
		Replay Attack
		Mint forgery attack
3	Business Security	Business Logics
		Business Implementations

1. Coding Conventions

1.1 Basic grammatical security

Solana currently supports the development of smart contracts in RUST and C languages, so it is important to first ensure the security of their underlying grammar before securing smart contracts.

1.2 Deprecated Items

The Solana smart contract development language is in rapid iteration. Some keywords have been deprecated by newer versions of the compiler, contract developers should not use the keywords that have been deprecated by the current compiler version.

1.3 Redundant Code

Redundant code in smart contracts can reduce code readability and may require more fee for contract deployment. It is recommended to eliminate redundant code.

1.4 Transaction fee

The smart contract virtual machine needs gas to execute the contract code. When the fee is insufficient, the code execution will throw an exception and cancel all state changes.

2. General Vulnerability

2.1 Integer overflow

Integer overflow is a security problem in many languages, and they are especially dangerous in smart contracts. Overflow conditions can lead to incorrect results, especially if its possible results are not expected, which may affect the reliability and safety of the program.

2.2 Reentrancy

The reentrancy vulnerability is the most typical smart contract vulnerability. Similarly, in the Solana contract, the target contract of CPI should be strictly controlled to avoid security problems caused by reentrancy.

2.3 Pseudo-random Number Generator (PRNG)

Random numbers may be used in smart contracts. Solana will also use block information as a random factor to generate, but such use is insecure. As these random numbers are to some extent predictable or collidable.

2.4 DoS(Denial of Service)

DoS, or Denial of Service, can prevent the target from providing normal services. Due to the immutability of smart contracts, this type of attack can make it impossible to ever restore the contract to its normal working state.

2.5 Function Call Permissions

If smart contracts have high-privilege functions, such as coin minting, self-destruction, change owner, etc., permission restrictions on function calls are required to avoid security problems caused by permission leakage.

2.6 Account signature check

In Solana, the user can provide any account when invoking a smart contract, so if the function corresponding to the contract does not check the account signature, this will lead to unpredictable security problems.

2.7 Arithmetic accuracy vulnerability

In Solana's smart contracts, mathematical calculations are inevitable, especially when tokens are involved, which may lead to distorted results if the mathematical operations are performed directly without considering the decimals of the tokens.

2.8 External call audit

When making cross-program invokes (CPI), please make sure to confirm the relevant parameters and authorization, and check the corresponding invoke results.

2.9 Replay Attack

A replay attack means that if two contracts use the same code implementation, and the identity authentication is in the transmission of parameters, the transaction information can be replayed to the other contract to execute the transaction when the user executes a transaction to one contract.

2.10 Mint forgery attack

If the smart contract involves tokens, make sure that the tokens passed in by the user are the tokens specified in the contract when performing related operations to avoid forging mint.

Appendix 3 Disclaimer

This report is made in response to the project code. No description, expression or wording in this report shall be construed as an endorsement, affirmation or confirmation of the project. This audit is only applied to the type of auditing specified in this report and the scope of given in the results table. Other unknown security vulnerabilities are beyond auditing responsibility. Beosin Technology only issues this report based on the attacks or vulnerabilities that already existed or occurred before the issuance of this report. For the emergence of new attacks or vulnerabilities that exist or occur in the future, Beosin Technology lacks the capability to judge its possible impact on the security status of smart contracts, thus taking no responsibility for them. The security audit analysis and other contents of this report are based solely on the documents and materials that the contract provider has provided to Beosin Technology before the issuance of this report, and the contract provider warrants that there are no missing, tampered, deleted; if the documents and materials provided by the contract provider are missing, tampered, deleted, concealed or reflected in a situation that is inconsistent with the actual situation, or if the documents and materials provided are changed after the issuance of this report, Beosin Technology assumes no responsibility for the resulting loss or adverse effects. The audit report issued by Beosin Technology is based on the documents and materials provided by the contract provider, and relies on the technology currently possessed by Beosin. Due to the technical limitations of any organization, this report conducted by Beosin still has the possibility that the entire risk cannot be completely detected. Beosin disclaims any liability for the resulting losses.

The final interpretation of this statement belongs to Beosin Technology.

Appendix 4 About Beosin

BEOSIN is a leading global blockchain security company dedicated to the construction of blockchain security ecology, with team members coming from professors, post-docs, PhDs from renowned universities and elites from head Internet enterprises who have been engaged in information security industry for many years. BEOSIN has established in-depth cooperation with more than 100 global blockchain head enterprises; and has provided security audit and defense deployment services for more than 1,000 smart contracts, more than 50 blockchain platforms and landing application systems, and nearly 100 digital financial enterprises worldwide. Relying on technical advantages, BEOSIN has applied for nearly 50 software invention patents and copyrights.

BEOSIN

Blockchain Security

Twitter

https://twitter.com/Beosin_com