BLOCKAPEX

SMART CONTRACT SECURITY

V 1.0

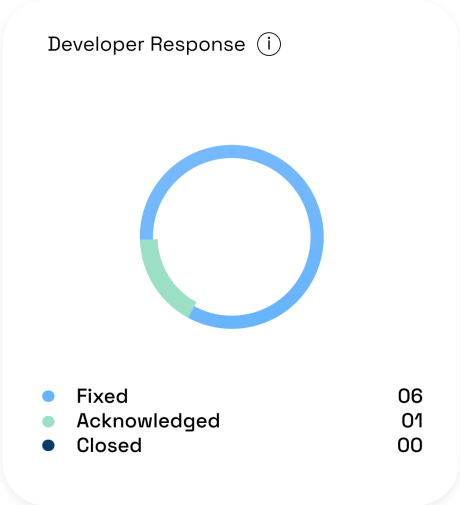
DATE: 19th SEP 2024

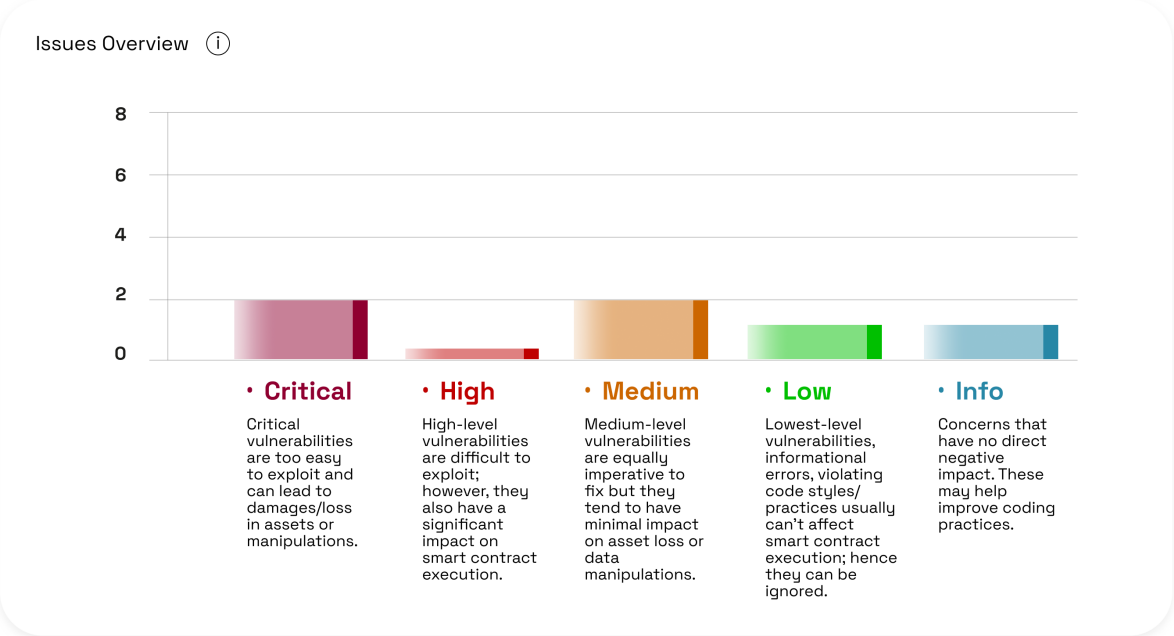
PREPARED FOR: STAKERA

Contents

1 Executive Summary

Stakera engaged BlockApex for a security review of the Stakera Lottery Program. A team of three security researchers reviewed the source code of the Lottery Program for 10-days of effort. The details of project scope, complexity and coverage are provided in the the subsequent sections of this report.





1.1 Scope

1.1.1 In Scope

Overview of the Stakera Lottery:

The **Stakera Lottery Protocol** is a decentralized lottery system on the Solana blockchain. It manages both small and large lottery events, allowing users to deposit funds into a pool, with winners selected through a random process using Switchboard oracles. The protocol is entirely on-chain, managing deposits, withdrawals, and yield distributions. It uses verifiable randomness to ensure fairness and includes “depeg protection” to safeguard participants against price fluctuations during withdrawals. This audit focuses on the protocol’s on-chain operations, particularly randomness handling, deposit and withdrawal processes, and yield distribution integrity.

Files in Scope:

- TheMauss/stakera_contract/contract.rs - Private Repo

Initial Audit Commit Hash: adbc65a8e7f43252dea649730a09288a9f86b9c1

Final Audit Commit Hash: 4c89ec0d2c80adbc47df058531564ff6eaf837c3

Note: Some part of the codebase was shared after the commencement of the audit.

1.1.2 Out of Scope

Any features or functionalities not explicitly mentioned in the “In Scope” section are considered outside the scope of this security review. Similarly, any off-chain, auto-generated, or client-side portions of the codebase as well as deployment and upgrade scripts are not included in the scope of this engagement.

1.2 Methodology

The codebase was audited using a filtered audit technique. A band of three (3) auditors scanned the codebase in an iterative process for a time spanning 10 days. Starting with the recon phase, a basic understanding of the code was developed, and the auditors worked on developing presumptions for the developed codebase and the relevant documentation/ whitepaper. Furthermore, the audit moved on with the manual code reviews to find logical flaws in the codebase complemented with code optimizations, software, and security design patterns, code styles and best practices

1.3 Status Description

Acknowledged: The issue has been recognized and is under review. It indicates that the relevant team is aware of the problem and is actively considering the next steps or solutions.

Fixed: The issue has been addressed and resolved. Necessary actions or corrections have been implemented to eliminate the vulnerability or problem.

Closed: This status signifies that the issue has been thoroughly evaluated and acknowledged by the development team. While no immediate action is being taken.

Partially Fixed:: The issue has multiple implications, some of which have been addressed and fixed but the rest still require action from the developer.

Temporary Fixed: The issue has been fixed for short term as a working solution, however, the long term permanent fix is under consideration and does not require immediate action.

1.4 Architecture Overview

The protocol's architecture leverages several key modules to ensure secure and transparent operations, including a custom lottery system, integration with external oracles for price feeds, and interactions with the Whirlpool liquidity pools for token swaps. The system is built using the Anchor framework, which provides a robust structure for managing Solana programs.

Core Components:

- **Lottery System:** The lottery system is the heart of the Stakera protocol. It handles user deposits, determines eligible participants, and conducts lottery draws based on randomness sourced from an external provider. It also manages the distribution of rewards to winners. The system supports both small and large lotteries, each with distinct rules and timelines.
- **Whirlpool Integration:** This component allows the protocol to perform token swaps between SOL and INF tokens. When a user deposits SOL, the system converts it to INF through the Whirlpool liquidity pools. The protocol ensures that these swaps are conducted under specific conditions to prevent slippage and depegging. The integration also supports managing associated accounts and tokens required for these operations.
- **Oracle Integration:** The protocol relies on oracles to fetch real-time price data for SOL and INF tokens. This data is used to validate swap operations and to ensure the accuracy and fairness of the lottery. The oracles provide price feeds that are crucial for maintaining the integrity of the system, particularly in operations involving token valuation.
- **Participant Management:** This component tracks all users who interact with the protocol. It records their deposits, monitors their participation in lotteries, and ensures that only eligible users are included in the draws. The system also handles withdrawals, making sure that all transactions are processed securely and correctly.

Operational Flow:

- **Initialization:** The protocol begins with the initialization of the lottery system, setting up necessary accounts, and establishing initial parameters like the total deposits and lottery timing.
- **Deposits and Swaps:** Users interact with the system by depositing SOL tokens, which are then converted to INF tokens via the Whirlpool pools. The protocol ensures that these swaps adhere to predefined limits to prevent losses due to market volatility.
- **Lottery Draws:** The protocol conducts both small and large lottery draws at scheduled intervals. The randomness required for these draws is sourced from an external randomness provider, ensuring unpredictability and fairness.

- **Rewards Distribution:** After each lottery draw, the protocol distributes rewards to the winners. The distribution process takes into account the user's deposit size and the randomness of the draw.
- **Withdrawals:** Users can withdraw their deposits or winnings at any time, with the system ensuring that all transactions are executed correctly and securely.

External Dependencies:

- **DEx (Orca Whirlpool):** Using Orca's Whirlpool DEx the stakera lottery protocol stakes the user's deposits to the INF/WSOL LST-pool. This DEx serves as swapping engine for users to swap their WSOL for INF and then to be staked in the Infinite's LST Pools.
- **Oracle (Pyth):** The protocol depends on external oracles for accurate and timely price data. The oracle provides price feeds for SOL and INF tokens, which are integral to the system's operations.
- **Randomness Provider (Switchboard):** The randomness for lottery draws is provided by an external randomness account, ensuring that the lottery outcomes are fair and unbiased.

1.5 Project Goals

To provide security assessment for the Stakera Lottery Protocol, we seek to answer the following list of non-exhaustive questions;

- Are winner selection and yield distribution logically sound?
- Do calculations and logic for small and big lottery yields align with the lottery rules?
- Are there checks to ensure consistency between committed randomness and distribution?
- Is the logic for ratio loss and withdrawals clear and consistent?
- How does committing randomness affect the fairness and trust in the lottery?
- Are participants treated fairly during winner selection, and are safeguards in place to protect them from excessive losses?
- How does withdrawing team yield affect the overall pool and participants' stakes?
- How are participants who do not win managed, and what is their experience throughout the lottery process?
- How does the timing of operations like deposits, withdrawals, and yield distribution impact lottery participation?
- Are there specific time windows for these operations, and how are they enforced?
- Are there timing constraints unique to small and big lottery yields, and what are the consequences of missing these windows?
- Are reward calculations and distributions transparent?
- Does the process for deposits, withdrawals, and yield distribution align with the contract's design goals?
- Are there any specific functions or operations that are intentionally disabled or restricted during the delay period? If so, what are they, and why are they restricted?
- Does the contract handle all possible scenarios where a delay might impact functionality? For instance, what happens if a delay overlaps with the expected time for a lottery draw?
- How does the delay affect the random number generation and its subsequent use in determining lottery outcomes? Is there a risk that the delay could skew the randomness or fairness of the lottery?
- Are there safeguards against replay attacks, storage manipulation, or unauthorized access?

1.6 Security Considerations

- **Trust Boundaries:**
  - **User to Protocol:** Trust that the protocol correctly manages deposits, conducts lotteries fairly, and processes withdrawals securely.
  - **Protocol to Oracles:** Trust that oracles provide accurate and timely price data, as any discrepancies could lead to incorrect swap operations or unfair lottery outcomes.
  - **Protocol to Randomness Provider:** Trust that the randomness used for lottery draws is truly random and free from manipulation.
- **Potential Points of Weakness:**
  - **Randomness Sourcing:** If the randomness provider is compromised, lottery draws could be unfairly manipulated.
  - **Oracle Data Integrity:** Inaccurate or delayed price feeds could result in incorrect token swaps or unfair reward distributions.
  - **User Input Validation:** Malicious input from users could exploit vulnerabilities in deposit, swap, or withdrawal operations.

1.7 Summary of Findings Identified

S.No	Severity	Findings	Status
#1	CRITICAL	Potential Randomness Slot Mismatch Leads to DoS	FIXED
#2	CRITICAL	Participant Limit Exploit Leads to Denial of Service (DOS)	FIXED
#3	MEDIUM	Inconsistent Contract States Due to Residual Token Balances	FIXED
#4	MEDIUM	Insufficient Randomness Validation and Potential Expiry Vulnerability	FIXED
#5	LOW	Modulo Bias Vulnerability in Lottery Selection Process	FIXED
#6	INFO	Redundant Check on Small Lottery Distribution	FIXED
#7	UNDETERMIN-ED	Potential Insolvency Due to Adverse Market Conditions	ACKNOWLEDGED

2 Findings and Risk Analysis

2.1 Potential Randomness Slot Mismatch Leads to DoS

Severity: Critical

Status: Fixed

Description The `commit_randomness` function for both the small and big lotteries requires a function argument of type PubKey, named as `randomness_account` (a randomness feed from the switchboard on-demand program) and from this variable the function later stores the `randomness_account.seed_slot` attribute in respective variables for committed slot named `small_randomness_account` and `big_randomness_account`. However, there is no explicit check ensuring that the randomness accounts used for both cases should be different.

In case, the same `randomness_account` is used for both lotteries, the `big_randomness_account` will be overwritten when the small lottery commits later, leading to a `RandomnessSlotMismatch` error during the big lottery distribution phase.

Impact This issue can lead to a complete failure of the big lottery distribution process, rendering the protocol unusable for its intended purpose. If a randomness mismatch occurs, the big lottery cannot distribute rewards to participants.

Proof of Concept :

Vulnerable Flow:

1. The signer commits randomness for the big lottery with `randomness_data.seed_slot = 50` and stores it in `lottery_account.big_randomness_account`.
2. The signer then commits randomness for the small lottery with `randomness_data.seed_slot = 60`, but unintentionally uses the same `randomness_account`. This overwrites the `big_randomness_account` .
3. When the big lottery tries to distribute, it checks the randomness slot:

```
        if randomness_data.seed_slot != lottery_account.big_commit_slot {
            return err!(CustomError::RandomnessSlotMismatch);
        }
```

Since the seed slot now corresponds to the small lottery (60 instead of 50), this results in a `RandomnessSlotMismatch` error, preventing the big lottery from being executed, effectively causing a Denial of Service.

Code Affected:

```
         pub fn commit_randomness(
            ctx: Context&lt;CommitRandomness&gt;,
            randomness_account: Pubkey,
            small_lottery: bool,
        ) -&gt; Result&lt;()&gt; {
            let clock = Clock::get()?;
            let current_time = clock.unix_timestamp;
            let lottery_account = &amp;mut ctx.accounts.lottery_account;
            let randomness_data =
                RandomnessAccountData::parse(ctx.accounts.randomness_account_data.data.borrow
                    ())
                    .unwrap();
            if randomness_data.seed_slot != clock.slot - 1 {
                msg!(&quot;seed_slot: {}&quot;, randomness_data.seed_slot);
                msg!(&quot;slot: {}&quot;, clock.slot);
                return err!(CustomError::RandomnessAlreadyRevealed);
            }

            // add checks
            if small_lottery == true {

                ...

                // Commit the current slot and randomness account
                lottery_account.small_commit_slot = randomness_data.seed_slot;
                lottery_account.small_randomness_account = randomness_account;


            } else {
                ...


                // Commit the current slot and randomness account
                lottery_account.big_commit_slot = randomness_data.seed_slot;
                lottery_account.big_randomness_account = randomness_account;
            }
            ...
        }
```

Recommendation Introduce a check within the `commit_randomness` method to ensure that the randomness account used for the small lottery is different from the one used for the big lottery. For example, implement a validation step such as:

```
        if lottery_account.small_randomness_account == lottery_account.big_randomness_account
            {
            return err!(CustomError::RandomnessAccountConflict);
        }
```

This will prevent the same randomness account from being used in both lotteries, ensuring the integrity of both lottery distributions.

2.2 Participant Limit Exploit Leads to Denial of Service (DOS)

Severity: Critical

Status: Fixed

Description The deposit function in the lottery system allows users to participate by making a deposit. The lottery account initially allocates `4096 bytes` of `space`, allowing a maximum of approximately `73` participants. Each participant's data consumes `56 bytes`. If this limit is reached, no new participants can be added without manual reallocation of the `lottery account`'s `space`.

A malicious user can exploit this by creating multiple accounts with small deposits (just above `0`) to fill up the lottery account. Since the deposit function does not restrict small-value deposits, the malicious user can create many participants, eventually preventing legitimate users from participating and causing a Denial of Service (DoS) for the lottery system.

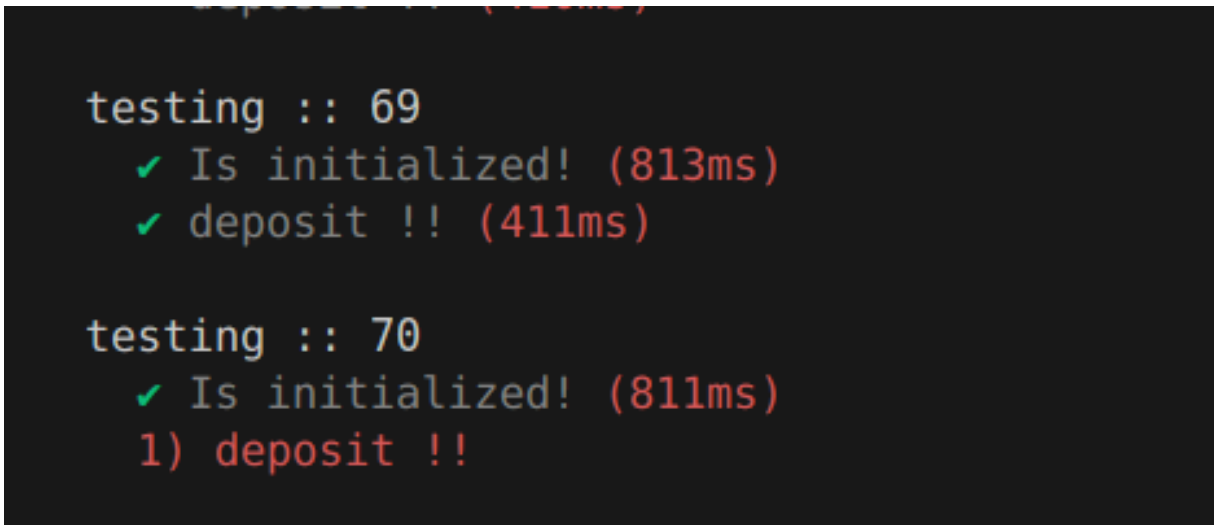
Impact This issue can cause a Denial of Service (DoS) by preventing new participants from joining the lottery once the account space limit is reached. A malicious user can exploit this by creating multiple low-value deposits, filling the participant list with dummy accounts and effectively locking out legitimate participants.

Proof of Concept :

Here is a test demonstrating the issue:

```
          it(&quot;Is initialized!&quot;,  async() =&gt; {
           const sig =  await programProvider.connection.requestAirdrop(user.publicKey, 1000 *
                (10 ** 9));
           await programProvider.connection.confirmTransaction(sig,&quot;confirmed&quot;);
           const tx = await program.methods.initialize().accounts({
             user: user.publicKey
           }).signers([user]).rpc({skipPreflight:true});
            await programProvider.connection.confirmTransaction(tx, &quot;confirmed&quot;);
        });
          it(&quot;deposit !!&quot;, async() =&gt; {
              const tx = await program.methods.deposit(new anchor.BN(1)).accounts({
                 user: user.publicKey
              }).signers([user]).rpc({skipPreflight: true});
              await programProvider.connection.confirmTransaction(tx);
              const [lotteryAccountaddress, a] = anchor.web3.PublicKey.findProgramAddressSync(
                 [Buffer.from(&quot;lottery&quot;)], program.programId
              )
              // const participant = await program.account.lotteryAccount.fetch(
                   lotteryAccountaddress);
              // console.log(&quot;participants :: &quot;, participant.participants);
          })
        });
        }
        const sleep = (ms: number) =&gt; new Promise(resolve =&gt; setTimeout(resolve, ms));
        // Run the test suite 31 times with a 1-second delay between each run
        (async () =&gt; {
          for (let i = 0; i &lt; 73; i++) {
            createTestSuite(i);
            await sleep(1000); // Wait for 1 second
          }
```

```
        })();
```



This test demonstrates that the system fails after `69-73` deposits due to `space` limitations in the lottery account.

Attack Scenario:

- A malicious user creates multiple accounts and deposits small amounts, just above `0 lamports`.
- The deposit function accepts these deposits, adding new participants to the lottery account.
- As the lottery account reaches its `space` limit (approximately `73 participants`), the account needs reallocation.
- The lottery system cannot accept more participants until manual reallocation is performed, causing disruption to the lottery operations.

Recommendation Enforce a minimum deposit threshold to prevent spam deposits with negligible amounts.

2.3 Inconsistent Contract States Due to Residual Token Balances

Severity: Medium

Status: Fixed

Description In the `deposit` function, users specify a `SOL amount` that they wish to deposit. This amount is first transferred into `wSOL` and then swapped for `INF` tokens. The function tracks the `wSOL` and `INF` amounts using `initial_balance_a` and `initial_balance_b` before the swap, and `final_balance_a` and `final_balance_b` after the swap. A fee is deducted from the initial deposit amount (not from the swapped amount), which is calculated and stored in `deposit_sub_orca_fee` and this variable has been potentially stored in `participant.pending_deposit`. However, if not all `wSOL` is converted to `INF` (e.g., `0.1 wSOL` remains unconverted in `final_balance_a`), the remaining `wSOL` is returned to the user at the end of the function, potentially leading to a discrepancy between the actual assets held by the contract and the amount recorded as deposited.

Impact This misalignment could lead to incorrect tracking of user deposits within the contract, as the contract might record a deposit of `10 SOL` while only actually holding the equivalent of `9 SOL` in `INF` tokens. This discrepancy can affect the overall liquidity management of the contract and might lead to errors in user balances, ultimately resulting in an insolvent state for the lottery smart contract.

Proof of Concept :

Code effected:

```
        let deposit_sub_orca_fee = amount
                            .checked_mul(9999)
                            .ok_or(CustomError::ArithmeticError)?
                            .checked_div(10000)
                            .ok_or(CustomError::ArithmeticError)?
                            .checked_mul(9999)
                            .ok_or(CustomError::ArithmeticError)?
                            .checked_div(10000)
                            .ok_or(CustomError::ArithmeticError)?;

          participant.pending_deposit = participant
                            .pending_deposit
                            .checked_add(deposit_sub_orca_fee)
                            .ok_or(CustomError::ArithmeticError)?;
```

Recommendation To address this issue, the calculation of `deposit_sub_orca_fee` should be based on the actual swapped amount rather than the initially specified deposit amount from the parameter.

2.4 Insufficient Randomness Validation and Potential Expiry Vulnerability

Severity: Medium

Status: Fixed

Description The function retrieves the randomness value via `randomness_data.get_value(&clock).unwrap()`. However, there is no robust validation to ensure the randomness value retrieved is of sufficient quality or has not expired. Specifically, the randomness value could be trivially small, such as [0x0, 0x0, 0x0, 0x1] or other low values, which may introduce predictable results or allow an attacker to exploit the randomness source. The methods `distribute_big_lottery_yield` and `distribute_small_lottery_yield` functions do not verify if the randomness has expired or if it was generated within a valid time window. Also, the check only ensures that the length of `random_value` is greater than or equal to `4 bytes`, which may not be enough to prevent potential manipulation.

Impact If randomness is not highly robust, the trust in the lottery system could be severely affected, resulting in decreased user engagement and potential reputational damage. This issue althoiugh doesnot compromise the randomness source directly or indirectly but always leaves a window of predictability in the winner selection process, ultimately undermining fairness.

Proof of Concept :

For example, with a random value of `[0x00, 0x00, 0x00, 0x01]`, the calculated `random_num` would be extremely small, reducing the randomness scope and making it easier for an attacker to predict the winner.

Vulnerable Code:

```
        if random_value.len() &lt; 4 {
            return err!(CustomError::InsufficientRandomness);
        }
```

Recommendation :

1. Introduce stricter validation checks for randomness values. Ensure that the value of the randomness is sufficiently large and that it falls within an acceptable range. Reject small or predictable values (e.g., values like 0x01, 0x02, etc.).
2. Add a check to ensure the randomness was generated within a valid timeframe and has not expired.

2.5 Modulo Bias Vulnerability in Lottery Selection Process

Severity: Low

Status: Fixed

Description The `select_winner` function, responsible for selecting a winner in the lottery, utilizes a random value to determine the winning participant based on their total deposits. However, due to the modulo operation `(random_num % total_deposits)`, a bias is introduced toward participants who control a significant portion of the total deposits.

In the current implementation, an attacker can skew the randomness by making large deposits, significantly increasing their chances of winning. This is especially critical in low-frequency lotteries where the attacker can monitor total deposits and strategically deposit enough to manipulate the outcome.

Impact The vulnerability allows an attacker with significant capital to monopolize the outcome of the lottery by manipulating the randomness. If unaddressed, the issue undermines the fairness of the lottery and diminishes user trust in the protocol, leading to a loss of users and negative publicity.

Note: The likelihood of such a scenario is strictly dependant on a couple of factors listed as below: 1. Max caps for a deposit being 10 SOL, currently. 2. Max participants due to limited space, 73 only.

Proof of Concept :

An attacker deposits via `65` accounts, each contributing `10 SOL`, into a lottery pool with `5` initial participants holding random small deposits (e.g., 2, 6, 9, 1, 5 SOL). The attacker's total contribution becomes `650 SOL`, compared to the `23 SOL` contributed by the initial depositors.

Given the random value `[0x12, 0x34, 0x56, 0x78]`, the random_num is calculated as `305419896`. The modulo operation scales this number to the range `[0, total_deposits)`:

```
        random_point = 305419896 % 673 = 462
```

The attacker's deposit occupies intervals between `23` and `673` in the `total_deposit` range. This ensures a `96.58%` chance that one of their accounts will win, leaving the original participants with only a `3.42%` chance to succeed.

Recommendation Implement **rejection sampling** to mitigate the modulo bias. This technique ensures that random numbers falling outside a uniformly distributed range are rejected and a new random number is generated.

```
    // Helper function to select a winner based on the random value with rejection sampling
    fn select_winner(
        random_value: &amp;[u8],
        offset: usize,
        intervals: &amp;Vec&lt;(Pubkey, u64)&gt;,
        total_deposits: u64,
    ) -&gt; Pubkey {
        // Convert the random_value slice starting at the offset into a single large integer
```

```
        let mut random_num = 0u128;

        ...

        // Calculate the range that would cause modulo bias
        let bias_threshold = (u128::MAX - (u128::MAX % total_deposits as u128));

        // Implement rejection sampling
        while random_num &gt;= bias_threshold {
            // If the random number is within the bias range, regenerate it
            msg!(&quot;Rejection sampling: Regenerating random_num to avoid bias&quot;);
            random_num = 0;
            for i in 0..random_value.len() {
                let byte = random_value[(offset + i) % random_value.len()];
                random_num = (random_num &lt;&lt; 8) | byte as u128;
            }
        }

        // Scale the large random number to the range [0, total_deposits)
        let random_point = (random_num % total_deposits as u128) as u64;

        ...
    }
```

2.6 Redundant Check on Small Lottery Distribution

Severity: Info

Status: Fixed

Description In the `distribute_small_lottery_yield` function, there is a check placed after the small lottery yield has been distributed which is as follows;

```
        small_lottery_to_big &gt;= 4
```

The check is ineffective because an earlier condition `(small_lottery_to_big == 4)` already ensures that no further small lotteries can occur once the threshold has been met. As a result, this additional check serves no functional purpose and introduces unnecessary complexity into the codebase.

Recommendation Remove the redundant check if `lottery_account.small_lottery_to_big >= 4` to improve the clarity of the code.

2.7 Potential Insolvency Due to Adverse Market Conditions

Severity: *Undetermined*

Status: Acknowledged

Description: The withdrawal function is designed to swap back user-deposited tokens (in this case, `INF`) back into `SOL` but based on current market rates when a user requests a withdrawal. This process depends heavily on the preservation of favorable exchange ratios in the underlying liquidity pool (e.g., `ORCA`). If `SOL` becomes significantly more valuable against `INF` the move `INF` will consume to retrieve the `SOL` back now this becomes problematic. This imbalance can deplete the pool's `SOL` reserves, leading to a situation where users cannot withdraw their expected `SOL` amounts due to insufficient liquidity.

Impact:

This issue can lead to severe financial loss for users who cannot withdraw their funds as expected. It also poses a risk to the platform as it fails to guarantee the liquidity necessary for users to reclaim their deposited values, especially during volatile market conditions.

Recommendation:

Implement dynamic withdrawal based on current liquidity and token swap rates within the pool. This way users will be able to withdraw a fair amount according to the market price and current value of SOL swap back from INF. These limits should be clearly communicated to the users and adjusted in real-time to prevent depletion of liquidity.

Disclaimer:

The code provided by the client with the purpose of security review have been thoroughly analyzed in compliance with the industrial best practices till date, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered, and should not be interpreted as an influence, on the potential economics of the token (if any), its sale, or any other aspect of the project that contributes to the protocol's public marketing.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract or the blockchain can have vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the programmable code and only the programmable code, we note, as being within the scope of our review within this report. The language itself remains under development and is subject to unknown risks and flaws.

This security review cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While BlockApex has done their best in conducting the analysis and producing this report, it is important to note that one should not rely on this report only - we recommend proceeding with several independent code security reviews and a public bug bounty program to ensure the security of the system.