Genopets Staking Program Audit Report

20.06.2022

—

SolShield

Introduction

SolShield conducted a full security audit and vulnerability analysis on the Genopets Staking Program. The audit process took approximately ~2 weeks to complete starting from April 27th and ending on May 13th. This report briefly covers the program's workflow along with a short description of the vulnerabilities discovered by the SolShield team.

Overall, we classified 7 brackets of bugs which the multiple issues identified by our team fall into. Gladly, all of them have since been patched by the Genopets development team.

Overview

The Genopets Staking is a recently launched incentive program to reward the GENE token holders __ the governance token of the Genoverse. The team has made careful design decisions and considerations to reward early adopters, incentivize long term staking and enabling users to enjoy the staking rewards without inflating the GENE supply.

At a high-level, the program is inspired by the Illuvium staking. Rewards are associated with each staking pool whereby to encourage early staking, a decaying factor is applied to exponentially accumulate lesser rewards over time. Moreover, the longer a staker chooses to lock their stake, a higher reward bonus is used for their reward calculation, hence incentivizing extended staking. The most notable component of the architecture is the double-token mechanism. Stakers can claim their rewards in the form of an instantly redeemable synthetic token called sGENE instead of waiting for GENE unlocks. The sGENE token can be used for in-game purposes, however it has minimal liquidity and cannot be swapped for GENE after being claimed. This preserves GENE scarcity all in while ensuring that users can enjoy its utility and their rewards via the synthetic token without waiting for the default 1 year lockout.

For more details about the staking program, please refer to the Genopets litepaper.

Account Structure

In this section, we take a closer look at the program's implementation in Solana's programming model. Here we briefly explain the adopted account model and structure.

- **Stake Master:**

  The Stake Master is a singleton account which holds general data and parameters like the program authority, start and end dates, epoch times, etc. The account is a

PDA with deterministic seeds and once initialized cannot be re-initialized later. There are multiple instructions to modify the various parameters set in this account, however, they are safely limited to be only accessible by the master authority enforced through Anchor framework macros and attributes.

- **Staking Pool:**

  Accounts that hold the data and parameters about each staking pool. Also PDAs, these accounts can only be created by the master authority above. At this time, there is only one staking pool accepting GENE tokens, however there will be more as the staking program evolves.

  Of notable parameters saved in this account is the pool weight, pool token and pool unlock dates.

- **Staker:**

  The program creates exactly one staker account per user holding universal data about the stake/withdraw parameters and yield rewards of the user. A PDA with the user's public key as part of the seeds, the account holds information like total rewards earned by the user, the rewards distributed thus far and most importantly an index parameter determining the seed input for the deposit accounts explained below.

- **Deposit:**

  The deposit accounts are responsible to keep track of user's funds locked in program controlled token accounts for either staked or reward GENE. A deposit account is created every time a user stakes tokens or claims yield rewards. Please note that the yield deposit account is only necessary for when yield is claimed as GENE because sGENE yields are immediately redeemable and can be utilized for in-game purposes.

  There is a one-to-many relationship between the staker and the deposit accounts. To keep track of the number of deposit accounts, as stated above, an index is kept in the staker account whereby it's used as part of the PDA seed and subsequently incremented each time a deposit account is initialized.

  These accounts play a crucial role in the implementation of the program as they keep important information such as the type and amount of locked tokens and unlock dates.

Reward Mechanism

The amount of yield distributed among all stakers across all staking pools is controlled by an accumulating parameter (for more info re. Illuvium). This parameter is updated with a synchronization method. that fetches the current rewards rate which in turn is controlled by a decaying factor. This factor decreases the emission of GENEs after each epoch following the formula below:

$$R = R_0 * decay\_factor^{(epoch - 1)}$$

With the current decay factor set to be 0.97 and the epoch of 14 days, this means the reward rate is exponentially decreased by 3% every 2 weeks.

In addition to the decaying rewards, a reward bonus is associated to incentivize long term staking. The mechanism is pretty much similar to the decaying factor but with an exponential base of 5. The formula is as follows:

$$reward\_bonus = 5^{\frac{locked\_months}{1\ year}}$$

This means for example that locking tokens for 1 year gives you a bonus of 5x over locking tokens for 1 month.

As we see later, instead of using exponentials and exposing the program to the risk of falling into floating-point pitfalls, our team proposed a static solution by sampling the exponentials at discrete points in time. This was possible because the staking duration for genopets is currently bounded between 1 and 24 months, hence, all the calculations could be bounded to that interval.

Methodology

After the initial contact from the Genopets team, we held an online session to go through the logic and the code structure. The complexity of the implementation was assessed to be unnecessarily high, therefore and thanks to the effort of the core developers from Genopets, a code revamp was done which took ~2 weeks to complete.

After that, we started to do extensive code analysis. The staking program makes extensive and spot-on use of PDAs to manage program associated data. The SolShield team also took extra care to confirm the program is resilient against classic Solana program attacks such as account re-initialization and substitution, missing authority and signer checks and token account confusions.

Instances of these primitive classes of vulnerabilities were discovered which we will explain later. In the next step, to guarantee the implementation follows the intended program specification, our lead auditor had multiple 1-on-1 sessions with the lead developer of the program, where we inspected the data flow through program logic ensuring correct behavior.

Then, as per SolShield promise, our team deployed the program on devnet and ran intense fuzzy and penetration tests, hitting the program with custom transactions with randomly generated data and different types of accounts to uncover any residual attack vector that might put the program in danger.

Lastly, we reported all the bugs and discoveries to the Genopets team with suggestions on how to resolve and mitigate the issues. The developers were swift in releasing patches to address the vulnerabilities we pointed out. The final code was scanned yet once again as a clean up review to ensure the validity of the fixes and that no new vulnerabilities were introduced in the process.

Findings & Mitigations

In this section, we enumerate and categorize the discovered vulnerabilities and give a short description of the security implications and their resolutions.

Arithmetic Computation Bugs

1. **Avoiding floating-point arithmetic**

   The original code of the program used rust powf function for the exponential computations needed for both the decaying rewards and the reward bonus. These functions take inputs as floating point numbers which are highly recommended to be avoided.

   Since the staking period is limited between 1-24 months, we noticed that only discrete points on the exponential curve are used during the life of the program, namely, 12 points for user reward bonus and 24 points for decaying rewards. Hence, we recommended to the team to use sampled points in the integer format with fixed-point arithmetic to avoid the whole floating-point logic and terminate the gateway to a major class of bugs altogether.

2. **Bit-extended fixed-point arithmetic**

   To perform the multiplication/division on the integer numbers resulting from above, we utilized a similar mechanism to how the basis point calculation is performed in Defi or NFT royalties. The numbers are bit-extended from u64 to u128 for the multiplication and the decimal point is recovered by a division to a proper divisor e.g. 10000. Combined with the previous trick, this completes the exponential aversion.

3. **Opt-in for checked operations & simplified expressions**

   There were multiple instances where bare +/- rust operations were used for addition/summation. By using the conventional checked operation, the code is now certain to be safe from common overflow and underflow vulnerabilities.

   Also per our discussion with Genopets team, we made a great deal of simplification to the composition of the mathematical expressions and the sanity checks. For example, computing the min and max duration for a stake deposit was originally carried out multiple times with intermediate results not kept for future calculations.

Account Validation Bugs

4. **Missing constraints for deposit accounts**

   As mentioned before, the deposit accounts are responsible for holding data regarding both user's locked staked and reward GENE tokens. They are used extensively in the program especially in all 3 instructions of Stake, Withdraw and Claim. A close inspection of the validations enforced on these accounts through anchor constraints, revealed account substitution and re-initialization attacks vectors.

   This code section shows the use of deposit accounts in the Withdraw instruction. An attacker could pass the same newly created user_re_deposit account as user_deposit and in the absence of proper checks put in later by the fixes, claim the rewards pre-maturely. This pattern was observed especially in the first draft of the code where upon our recommendation, the Genopets developers paid extra attention to make sure these confusions are avoided entirely.

```
#[account(
 init_if_needed,
 seeds = [
     DEPOSIT_TAG.as_ref(),
     user.key().as_ref(),
     &staker.current_deposit_index.to_le_bytes()
 ],
 bump,
 payer = user,
 space = Deposit::LEN
)]
pub user_re_deposit: Account<'info, Deposit>,

#[account(mut, constraint = user_deposit.user == user.key(), close = user)]
pub user_deposit: Account<'info, Deposit>,
```

5. **Insufficient checks on Associated Token Accounts (ATA)**

   Throughout the code, there were numerous occurrences of using token accounts assumed to be associated token accounts but the actual PDA checks for the ATA were not performed. We suggested using the convenient assert_is_ata function from the Metaplex Program Library to enforce the ATA conditions.

Miscellaneous & Logical Bugs

6. **Inability to claim GENE rewards**

   The withdraw function is both used to get back the GENE rewards and the initial staked GENE. In order to do so, the program makes use of binary conditionals to determine the parameters passed to the Transfer cross program invocation into the Token Program. In our review, we observed that the transfer authority was constantly passed as the staker account which means only staked GENE tokens were redeemable. This can be seen in the code segment below taken from the faulty code.

```
let signer_seeds: &[&[u8]] = if self.user_deposit.is_yield {
   rewarder_seeds
} else {
   staker_seeds
};

let from_account = if self.user_deposit.is_yield {
   self.ata_gene_rewarder.to_account_info()
} else {
   self.ata_vault.to_account_info()
};

…

cpi_accounts = Transfer {
   from: from_account,
   to: self.ata_user.to_account_info(),
   authority: self.staker.to_account_info(),
};
```

   The fix should, similar to the other variables, check if the deposit account is a yield account or not and assign the correct authority accordingly.

7. **Kill switch**

   With the use of the anchor attribute (access_control), we built in a kill switch over all the instructions which could be triggered by the master authority. With the purpose of halting the system in case a future incident happens, this prevents disputable upgrades to prevent losses and can eventually be easily removed once the code has proved to be secure and robust in the production environment for a while.