HXRO Staking Program Audit Report

2.11.2022

—

SolShield

Introduction

SolShield conducted a full security audit and vulnerability analysis on the HXRO Staking Program. The audit took approximately ~4 weeks to complete starting from August 15th and ending on September 15th. This report briefly covers the program's workflow along with a short description of our general findings.

Overall, we are glad to confirm that no critical vulnerabilities were found in the code. Apart from a few issues that have since been communicated to the team, the program operates within the boundaries of its specification.

Overview

The HXRO Staking is an incentive program to reward HXRO token holders with the value accrued from the trading fees collected on the platform. By staking their HXRO, users lock away their tokens in pool vaults which in turn makes them eligible to earn rewards generated by the network. The team makes many careful considerations to bring the economic value of the protocol to the HXRO token without significantly diluting its circulating supply.

To this end, the program is especially designed to encourage users toward long-term staking. There are two mechanisms in place for this purpose. First, a staking weight is assigned to each stake that is directly proportional to the stake's duration. Secondly, the rewards paid out in HXRO token are locked for an amount of time inversely proportional to the stake's duration. This helps mitigate inflation as new tokens issued as reward cannot be used immediately unless the initial stake is taken out of circulation and locked in network vaults for a prolonged period.

In addition, a novel technique is used to further retain the value of the HXRO token within the protocol. Instead of using HXRO itself as the emission token, a designated esHXRO token is created which is backed 1:1 by HXRO. The difference is esHXRO can only be redeemed for native HXRO at a rate of 1/365 per day. However, esHXRO can be used in limited scenarios defined by the network like staking. Therefore, users are incentivized to re-stake their esHXRO tokens back into the vaults to collect more rewards instead of converting to HXRO. This approach results in a self-sustained tokenomics where there won't be an over-issuance of unlocked HXRO tokens, hence, gives users competitive long-term staking returns.

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. This program has a clever architecture with a unique property that we call account efficiency i.e. it uses a very small number of account types to support complex staking activities. Concretely, there are only two account types in total, but the program can accept many tokens as both stake and reward tokens which is quite remarkable. We explain this in the following.

- **Stake Pool:**

  The Stake Pool is a singleton account holding the essential data that determines the overall staking parameters. The account has two vectors of custom structs named **Vault** and **StakeReward**. Each struct in the Vault vector keeps a token account to receive user stakes in the form of its corresponding token mint. For example, to accept native HXRO as a staking token, a new Vault struct needs to be created with a token account of HXRO mint and the Stake Pool account as the owner. Similarly, each StakeReward struct contains information about the token account that the rewards are paid out from. The ability to add many Vault and StakeRewards to the Stake Pool account, allows for multiple stake and reward tokens to be accepted by the same pool. This is the main reason for the account efficiency mentioned above.

  We note that only the authority of the Stake Pool can add new structs to these vectors which prevents malicious peddling of the account by anyone other than the network operators.

  The code snippet below shows the structure of the Stake Pool account.

```
#[account]
#[derive(Default, Debug)]
pub struct StakePool {
   pub authority: Pubkey,

   pub effective_total_staked: u128,

   pub vaults: Vec<Vault>,

   pub stake_rewards: Vec<StakeReward>,
}
```

- **Stake Rewards:**

  These accounts are PDAs created each time a user deposits a stake. The account stores the stake pool address, the user token account used to transfer the stake from, the pool token account which is the vault account receiving the stake and the staker address itself. Other important parameters such as the stake duration and the deposit date are also kept here.

  The most important data saved in this account is a struct called **StakeRewardState** that decides the stake state which can be one of *Staking* or *Withdrawn*. In the *Staking* state, the struct holds parameters such as stake amount, weighted stake amount and a vector of pool unit shares that are used to calculate the reward. We elaborate on this in more detail in the next section. Upon user stake withdrawal, if there are locked rewards left to be claimed, the account is set to the *Withdrawn* state. The struct in this case stores a vector of the rewards that can be claimed in the future when they are unlocked. However, if all the rewards have been sent out already, the account is closed.

  This design, although intricate, is very smart and allows for simultaneous accounting of the many reward tokens emitted by the Stake Pool.

Reward Mechanism

The amount of yield that can be distributed among all participants is controlled by a cumulative parameter called unit shares that is held in the StakeReward struct of the Stake Pool account. Concretely, denote the total stake amount from all deposits in the pool at time *ti* as $Stake_{ti}$. Following a distribution of rewards R_{ti} at this time, each unit of stake is entitled to earn:

$$u_{ti} = \frac{R_{ti}}{Stake_{ti}}$$

Summing for all the reward distribution events at all times upto t, we get

$$U_t = \sum_{ti=0}^{t} u_{ti}$$

Which is the cumulative unit share that the program keeps track of. Upon user deposit of stake *s* at time *t1*, a Stake Rewards account is created that records the current cumulative

unit share U_{t1} in the *Staking* struct. When at a later time *t2* the user claims their rewards, the program computes their share of the rewards as

$$r = s * (U_{t2} - U_{t1})$$

Where U_{t2} is retrieved from the Stake Pool account that has been constantly updated with each reward distribution in the interval between t2 and t1. This process, borrowed from the work in [1], is extremely time and memory efficient.

As mentioned in the introduction, the program incentivizes long term staking in two ways. First, the program assigns a boosted weight to stakes that are locked for more than 1 year. This weight increases linearly with the duration as follows

$$weight = 2 + 2 * \frac{duration - 1\ year}{3\ year - 1\ year}$$

This means that the user's stake has an effective multiple of 2.x (if locked for 1 year) upto 4.x (if locked for 3 years). Secondly, rewards are unlocked sooner the longer the tokens are staked for. The program enforces this inverse relation as

$$unlock\_timestamp = deposit\_timestamp + 365\ days * \frac{3\ year - duration}{3\ year - 7\ days}$$

Hence, users can instantly claim their rewards if they stake for the maximum duration of 3 years whereas their rewards are locked for 1 year if they opt-in for the minimum 7 days.

As an extra measure, the network has implemented an escrow program that allows it to administer an escrowed version of HXRO token called esHXRO. Rewards paid in esHXRO are frozen by the escrow program. Although esHXRO is redeemable for HXRO at a rate of 1/365 per day, it can also be re-staked into the pool. As users are incentivized to do so and earn rewards in the form of stablecoins (network fees in USDC), the HXRO token is further kept from being inflated by the staking rewards. The escrow program is audited by SolShield as well and we will dive into its internals in a separate audit report that we will publish soon.

Methodology

After the initial contact from the HXRO team, we did a quick read through of the source code to evaluate the scope of the work and recognize early potential footguns. Due to the complex nature of the staking algorithm, we were in constant contact with the development team to realize the specification in detail and that the program implements it accordingly.

After that, we started to do extensive code analysis. As mentioned in the account structure section, the program has minimal account types from which only one is PDA. Therefore, it enjoys a highly reduced attack vector with regard to account re-initialization and substitution. We then took extra care to confirm the program is resilient against classic Solana attacks such as authority/signer mischecks and token account confusions.

Instances of potential vulnerabilities were discovered that we communicated during online sessions to the HXRO developers. However, we are happy to announce that no critical or loss of funds bugs were discovered which bears a huge kudos to the team at HXRO. There were however some minor issues and improvements that we will describe in the following section.

Next, 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 account types to uncover any residual vulnerabilities that might put the program in danger. The program did not exhibit any unexpected or erroneous executions.

Finally, we note that it is critically important to set up and initialize the Stake Pool accounts, the associated stake/reward token mints and their multisig authorities correctly. As a result, we have been continuously supporting the team with administering the programs and ensuring that required modifications are reviewed by our team before final deployment to avoid introducing any new vulnerabilities in this process.

Findings & Recommendations

In this section, we enumerate some of the minor findings and issues we discovered and explain their implications and resolutions.

1. **Unchecked arithmetic operations**

   There are occurrences of vanilla Rust additions and subtractions in the source code. For example, the following code snippet shows the reward timestamp calculation. We suggest always using checked operations for extra protection.

```
pub fn reward_timestamp(duration: u64) -> Result<u64> {
   // locked for 1 year at 7 days
   // instant 'unlock' at 3 years
   let start = 7 * SECONDS_IN_DAY;
   let end = 1095 * SECONDS_IN_DAY;

   (365 * SECONDS_IN_DAY)
       .checked_mul((end - start) - (duration - start))
       .ok_or(ErrorCode::ArithmeticOverflow)?
       .checked_div(end - start)
       .ok_or(ErrorCode::ArithmeticOverflow.into())
}
```

2. **Non-deterministic nonce for PDA seeds**

   The Stake Rewards account created by user deposit instruction, has the following seeds

```
seeds = [
   staker.key().as_ref(),
   stake_pool.key().as_ref(),
   nonce.to_le_bytes().as_ref(),
]
```

   Currently, the deposit timestamp is used as the nonce. This causes an indexing overhead where the nonce has to be stored for future reference. In addition, nonce reuse is possible that could mess up the indexer data and cause invalid front-end statistics. We recommend storing the nonce in the Stake Rewards account and also keep an incrementally deterministic nonce in the Stake Pool account to avoid these problems.

3. **The stake multiplier is preserved even after the stake duration has passed**

   This is a unique corner case that we uncovered which stands to be more of an acute observation than a bug. It is better explained with an example. Assume user A deposits a stake with a 1 year duration. The program assigns a 2.x multiplier to their stake effectively doubling their rewards. After the 1 year period has passed, if user A doesn't withdraw their tokens, the stake maintains the 2.x boost even though it's not locked anymore. This might be unfair to other users at that point of time as they have to lock their stake for 1 year to achieve the same multiple.

   During our discussions with the team, it was decided that this is acceptable since user A has after all "served their time". Regardless, the team is willing to reevaluate and change this if needed based on feedback from the community.

References

[1] B. Batog, L. Boca and N. Johnson, Scalable Reward Distribution on the Ethereum Blockchain, http://batog.info/papers/scalable-reward-distribution.pdf, 2018